Lecture Schedule

subject	topics	reading
Preliminaries	Analyzing algorithms	1-32
”	Asymptotic notation	32-37
”	Recurrence relations	53-64
Sorting	Heapsort	140-150
”	Quicksort	153-167
”	Linear Sorting	172-182
Searching	Data structures	200-215
”	Binary search trees	244-245
”	Red-Black trees:insertion	262-272
“	Red-Black trees:deletion	272-277
MIDTERM 1		
Comb. Search	Backtracking	
”	Elements of dynamic programming	301-314
”	Examples of dynamic programming	314-323
Graph Algorithms	Data structures for graphs	465-477
”	Breadth/depth-first search	477-483
”	Topological Sort/Connectivity	485-493
”	Minimum Spanning Trees	498-510
”	Single-source shortest paths	514-532
”	All-pairs shortest paths	550-563
MIDTERM 2		
Intractability	P and NP	916-928
”	NP-completeness	929-939
”	NP-completeness proofs	939-951
”	Further reductions	951-960
”	Approximation algorithms	964-974
”	Set cover / knapsack heuristics	974-983
FINAL EXAM		

What Is An Algorithm?

Algorithms are the ideas behind computer programs.

An algorithm is the thing which stays the same whether the program is in Pascal running on a Cray in New York or is in BASIC running on a Macintosh in Kathmandu!

To be interesting, an algorithm has to solve a general, specified problem. An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

Example: Sorting

Input: A sequence of N numbers $a_1 ... a_n$

Output: the permutation (reordering) of the input sequence such as $a_1 \leq a_2 \ldots \leq a_n$.

We seek algorithms which are *correct* and *efficient.*

Correctness

For any algorithm, we must prove that it *always* returns the desired output for all legal instances of the problem.

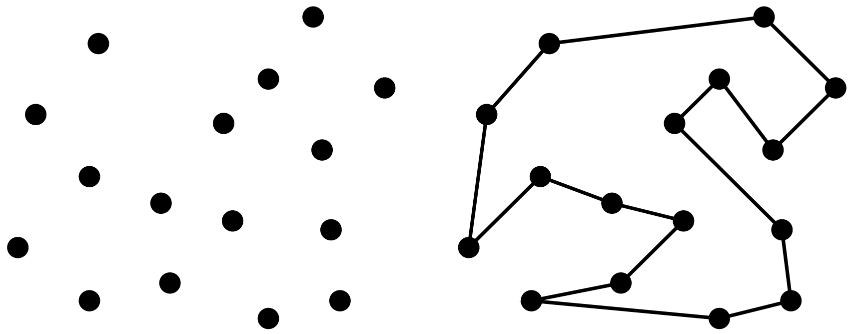
For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.

Correctness is Not Obvious!

The following problem arises often in manufacturing and transportation testing applications.

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the first contact point, then visits the second point, third, and so forth until the job is done.

Since robots are expensive, we need to find the order which minimizes the time (ie. travel distance) it takes to assemble the circuit board.



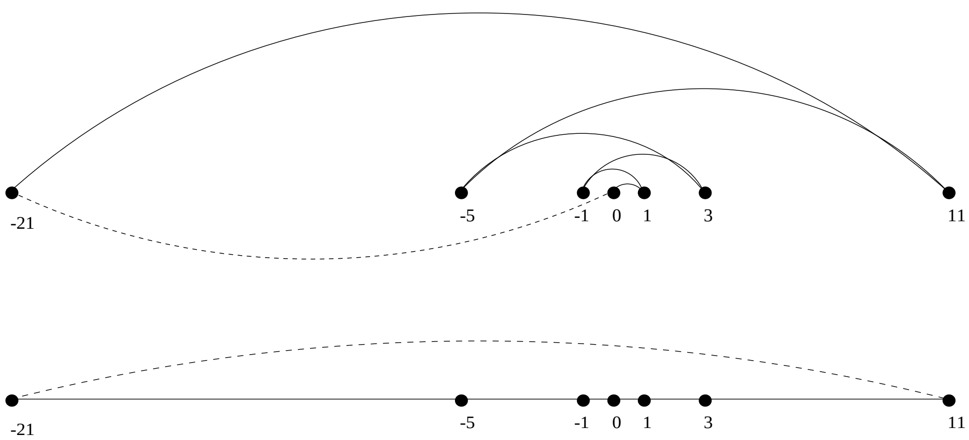
You are given the job to program the robot arm. Give me an algorithm to find the best tour!

Nearest Neighbor Tour

A very popular solution starts at some point p_0 and then walks to its nearest neighbor p_1 first, then repeats from p_1, etc. until done.

Pick and visit an initial point p_0
$p = p_0$
$i = 0$
While there are still unvisited points
    $i = i + 1$
    Let p_i be the closest unvisited point to p_{i-1}
    Visit p_i
Return to p_0 from p_i

This algorithm is simple to understand and implement and very efficient. However, it is **not correct!**



Always starting from the leftmost point or any other point will not fix the problem.

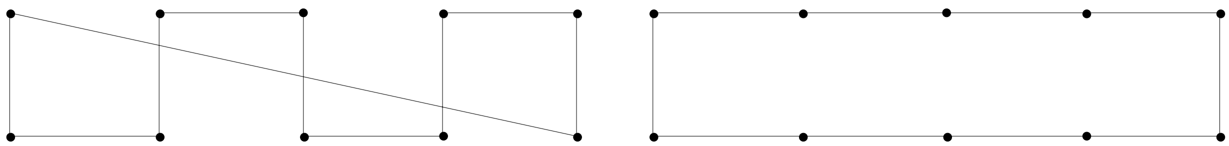
Closest Pair Tour

Always walking to the closest point is too restrictive, since that point might trap us into making moves we don't want.

Another idea would be to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed, until we have a single chain with all the points in it.

Let n be the number of points in the set
$d = \infty$
For $i = 1$ to $n - 1$ do
    For each pair of endpoints (x, y) of partial paths
        If $dist(x, y) \leq d$ then
            $x_m = x$, $y_m = y$, $d = dist(x, y)$
    Connect (x_m, y_m) by an edge
Connect the two endpoints by an edge.

Although it works correctly on the previous example, other data causes trouble:



This algorithm is **not correct**!

A Correct Algorithm

We could try all possible orderings of the points, then select the ordering which minimizes the total length:

$d = \infty$
For each of the $n!$ permutations Π_i of the n points
  If $(cost(\Pi_i) \leq d)$ then
    $d = cost(\Pi_i)$ and $P_{min} = \Pi_i$
Return P_{min}

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour.

Because it trys all $n!$ permutations, it is extremely slow, much too slow to use when there are more than 10-20 points.

No efficient, correct algorithm exists for the *traveling salesman problem*, as we will see later.

Efficiency

"Why not just use a supercomputer?"

Supercomputers are for people too rich and too stupid to design efficient algorithms!

A faster algorithm running on a slower computer will *always* win for sufficiently large instances, as we shall see.

Usually, problems don't have to get that large before the faster algorithm wins.

Expressing Algorithms

We need some way to express the sequence of steps comprising an algorithm.

In order of increasing precision, we have English, pseudocode, and real programming languages. Unfortunately, ease of expression moves in the reverse order.

I prefer to describe the *ideas* of an algorithm in English, moving to pseudocode to clarify sufficiently tricky details of the algorithm.

The RAM Model

Algorithms are the *only* important, durable, and original part of computer science *because* they can be studied in a machine and language independent way.

The reason is that we will do all our design and analysis for the RAM model of computation:

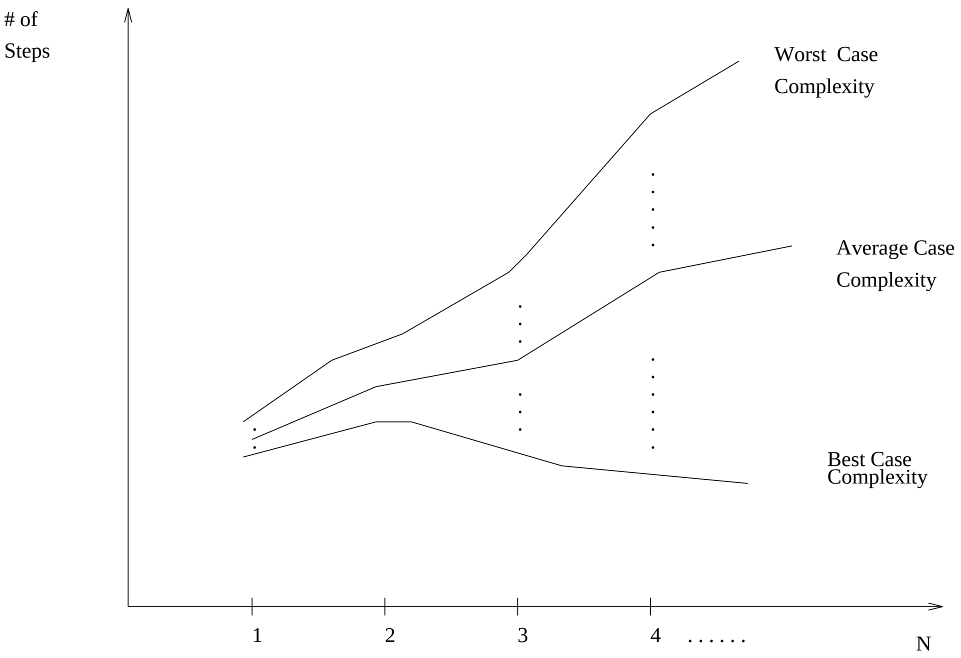
- Each "simple" operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine. We do not want "sort" to be a single step operation.
- Each memory access takes exactly 1 step.

We measure the run time of an algorithm by counting the number of steps.

This model is useful and accurate in the same sense as the flat-earth model (which *is* useful)!

Best, Worst, and Average-Case

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.



The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function – time vs. size!

Insertion Sort

One way to sort an array of n elements is to start with a_n empty list, then successively insert new elements in the proper position:

$$a_1 \leq a_2 \leq \ldots \leq a_k \mid a_{k+1} \ldots a_n$$

At each stage, the inserted element leaves a sorted list, and after n insertions contains exactly the right elements. Thus the algorithm must be correct.

But how *efficient* is it?

Note that the run time changes with the permutation instance! (even for a fixed size problem)

How does insertion sort do on sorted permutations?

How about unsorted permutations?

Exact Analysis of Insertion Sort

Count the number of times each line of pseudocode will be executed.

Line	InsertionSort(A)	#Inst.	#Exec.
1	for j:=2 to len. of A do	c1	n
2	key:=A[j]	c2	n-1
3	/* put A[j] into A[1..j-1] */	c3=0	/
4	i:=j-1	c4	n-1
5	while $i > 0 \& A[1] > key$ do	c5	tj
6	A[i+1]:= A[i]	c6	
7	i := i-1	c7	
8	A[i+1]:=key	c8	n-1

The **for** statement is executed $(n-1)+1$ times (why?)
Within the **for** statement, "key:=A[j]" is executed n-1 times.
Steps 5, 6, 7 are harder to count.
Let $t_j = 1+$ the number of elements that have to be slide right to insert the jth item.
Step 5 is executed $t_2 + t_3 + ... + t_n$ times.
Step 6 is $t_{2-1} + t_{3-1} + ... + t_{n-1}$.

Add up the executed instructions for all pseudocode lines to get the run-time of the algorithm:

$c_1 * n + c_2(n-1) + c_4(n-1) +$ $c_5 \sum_{j=2}^{n} t_j +$ $c_6 \sum_{j=2}^{n}(t_j - 1)$ $+ c_7 \sum_{j=2}^{n}(t_j - 1) + c_8$

What are the $t_j's$? They depend on the particular input.

Best Case

If it's already sorted, all t_j's are 1.

Hence, the best case time is

$$c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) = Cn + D$$

where C and D are constants.

Worst Case

If the input is sorted in *descending* order, we will have to slide *all* of the already-sorted elements, so $t_j = j$, and step 5 is executed

$$\sum_{j=2}^{n} j = (n^2 + n)/2 - 1$$

Problem 1.2-6: How can we modify almost any algorithm to have a good best-case running time?

To improve the best case, all we have to do it to be able to solve one instance of each size efficiently. We could modify our algorithm to first test whether the input is the special instance we know how to solve, and then output the canned answer.

For sorting, we can check if the values are already ordered, and if so output them. For the traveling salesman, we can check if the points lie on a line, and if so output the points in that order.

The supercomputer people pull this trick on the linpack benchmarks!

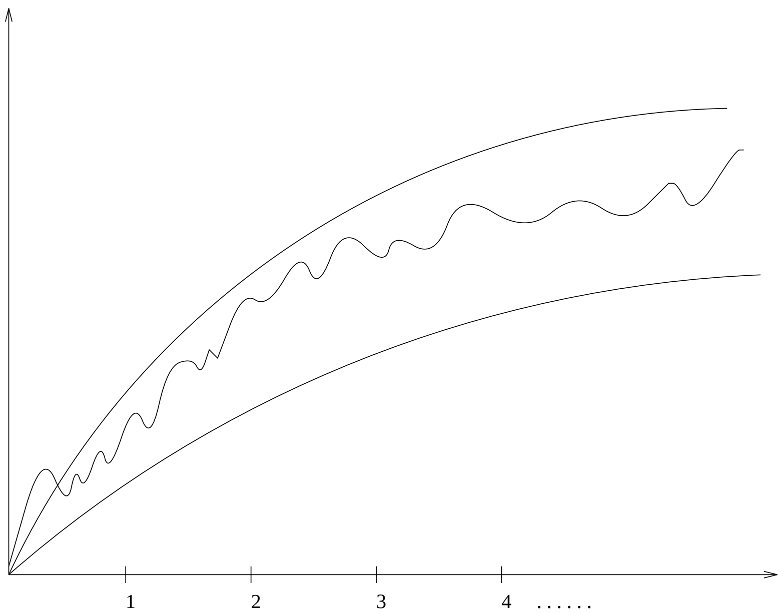
Because it is so easy to cheat with the best case running time, we usually don't rely too much about it.

Because it is usually very hard to compute the average running time, since we must somehow average over all the instances, we usually strive to analyze the worst case running time.

The worst case is usually fairly easy to analyze and often close to the average or real running time.

Exact Analysis is Hard!

We have agreed that the best, worst, and average case complexity of an algorithm is a numerical function of the size of the instances.



However, it is difficult to work with exactly because it is typically very complicated!

Thus it is usually cleaner and easier to talk about *upper and lower bounds* of the function.

This is where the dreaded big O notation comes in!

Since running our algorithm on a machine which is twice as fast will effect the running times by a multiplicative constant of 2 - we are going to have to ignore constant factors anyway.

Names of Bounding Functions

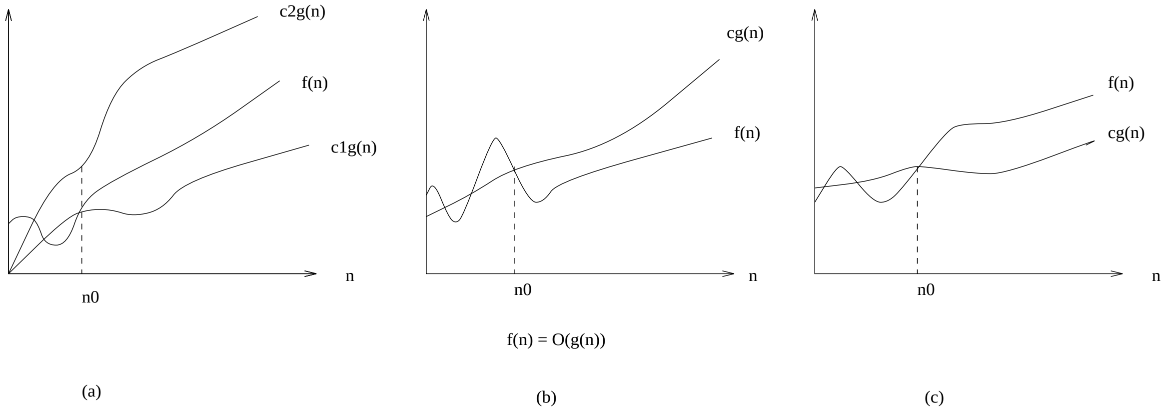
Now that we have clearly defined the complexity functions we are talking about, we can talk about upper and lower bounds on it:

- $g(n) = O(f(n))$ means $C \times f(n)$ is an *upper bound* on $g(n)$.
- $g(n) = \Omega(f(n))$ means $C \times f(n)$ is a *lower bound* on $g(n)$.
- $g(n) = \Theta(f(n))$ means $C_1 \times f(n)$ is an upper bound on $g(n)$ and $C_2 \times f(n)$ is a lower bound on $g(n)$.

Got it? C, C_1, and C_2 are all constants independent of n.

All of these definitions imply a constant n_0 *beyond which* they are satisfied. We do not care about small values of n.

O, Ω, and Θ



The value of n_0 shown is the minimum possible value; any greater value would also work.

(a) $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive.

(b) $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

(c) $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

Asymptotic notation (O, Θ, Ω) are as well as we can practically deal with complexity functions.

What does all this mean?

$$\begin{aligned}
3n^2 - 100n + 6 &= O(n^2) \textit{ because } 3n^2 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &= O(n^3) \textit{ because } .01n^3 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq O(n) \textit{ because } c \cdot n < 3n^2 \;\; \textit{when} \;\; n > c
\end{aligned}$$

$$\begin{aligned}
3n^2 - 100n + 6 &= \Omega(n^2) \textit{ because } 2.99n^2 < 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq \Omega(n^3) \textit{ because } 3n^2 - 100n + 6 < n^3 \\
3n^2 - 100n + 6 &= \Omega(n) \textit{ because } 10^{10^{10}} n < 3n^2 - 100 + 6
\end{aligned}$$

$$\begin{aligned}
3n^2 - 100n + 6 &= \Theta(n^2) \textit{ because } O \;\; \textit{and} \;\; \Omega \\
3n^2 - 100n + 6 &\neq \Theta(n^3) \textit{ because } O \;\; \textit{only} \\
3n^2 - 100n + 6 &\neq \Theta(n) \textit{ because } \Omega \;\; \textit{only}
\end{aligned}$$

Think of the equality as meaning *in the set of functions.*

Note that time complexity is every bit as well defined a function as $\sin(x)$ or you bank account as a function of time.

Testing Dominance

$f(n)$ dominates $g(n)$ if $\lim_{n\rightarrow\infty} g(n)/f(n) = 0$, which is the same as saying $g(n) = o(f(n))$.

Note the little-oh – it means "grows strictly slower than".

Knowing the dominance relation between common functions is important because we want algorithms whose time complexity is as low as possible in the hierarchy. If $f(n)$ dominates $g(n)$, f is much larger (ie. slower) than g.

- n^a dominates n^b if $a > b$ since

$$\lim_{n\rightarrow\infty} n^b/n^a = n^{b-a} \rightarrow 0$$

- $n^a + o(n^a)$ doesn't dominate n^a since

$$\lim_{n\rightarrow\infty} n^a/(n^a + o(n^a)) \rightarrow 1$$

Complexity	10	20	30	40	50	60
n	0.00001 sec	0.00002 sec	0.00003 sec	0.00004 sec	0.00005 sec	0.00006 sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec	0.025 sec	0.036 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec	0.216 sec
n^5	0.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	0.001 sec	1.0 sec	17.9 min	12.7 days	35.7 years	366 cent
3^n	0.59 sec	58 min	6.5 years	3855 cent	2×10^8 cent	1.3×10^{13} cent

Logarithms

It is important to understand deep in your bones what logarithms are and where they come from.

A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Exponential functions, like the amount owed on a n year mortgage at an interest rate of $c\%$ per year, are functions which grow distressingly fast, as anyone who has tried to pay off a mortgage knows.

Thus inverse exponential functions, ie. logarithms, grow refreshingly slowly.

Binary search is an example of an $O(\lg n)$ algorithm. After each comparison, we can throw away half the possible number of keys. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!

If you have an algorithm which runs in $O(\lg n)$ time, take it, because this is blindingly fast even on very large instances.

Properties of Logarithms

Recall the definition, $c^{\log_c x} = x$.

Asymptotically, the base of the log does not matter:

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus, $\log_2 n = (1/\log_{100} 2) \times \log_{100} n$, and note that $1/\log_{100} 2 = 6.643$ is just a constant.

Asymptotically, any polynomial function of n does not matter:

Note that

$$\log(n^{473} + n^2 + n + 96) = O(\log n)$$

since $n^{473} + n^2 + n + 96 = O(n^{473})$, and $\log n^{473} = 473 * \log n$.

Any exponential dominates *every* polynomial. This is why we will seek to avoid exponential time algorithms.

Federal Sentencing Guidelines

2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.

(a) Base offense Level: 6

(b) Specific offense Characteristics

(1) If the loss exceeded \$2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) \$2,000 or less	no increase
(B) More than \$2,000	add 1
(C) More than \$5,000	add 2
(D) More than \$10,000	add 3
(E) More than \$20,000	add 4
(F) More than \$40,000	add 5
(G) More than \$70,000	add 6
(H) More than \$120,000	add 7
(I) More than \$200,000	add 8
(J) More than \$350,000	add 9
(K) More than \$500,000	add 10
(L) More than \$800,000	add 11
(M) More than \$1,500,000	add 12
(N) More than \$2,500,000	add 13
(O) More than \$5,000,000	add 14
(P) More than \$10,000,000	add 15
(Q) More than \$20,000,000	add 16
(R) More than \$40,000,000	add 17
(Q) More than \$80,000,000	add 18

The federal sentencing guidelines are designed to help judges be consistent in assigning punishment. The time-to-serve is a roughly linear function of the total *level.*

However, notice that the increase in level as a function of the amount of money you steal grows *logarithmically* in the amount of money stolen.

This very slow growth means it pays to commit one crime stealing a lot of money, rather than many small crimes adding up to the same amount of money, because the time to serve if you get caught is much less.

The Moral: *"if you are gonna do the crime, make it worth the time!"*

Working with the Asymptotic Notation

Suppose $f(n) = O(n^2)$ and $g(n) = O(n^2)$.

What do we know about $g'(n) = f(n) + g(n)$? Adding the bounding constants shows $g'(n) = O(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? Since the bounding constants don't necessary cancel, $g''(n) = O(n^2)$

We know nothing about the lower bounds on $g' + g''$ because we know nothing about lower bounds on f, g.

Suppose $f(n) = \Omega(n^2)$ and $g(n) = \Omega(n^2)$.

What do we know about $g'(n) = f(n)+g(n)$? Adding the lower bounding constants shows $g'(n) = \Omega(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? We know nothing about the lower bound of this!

The Complexity of Songs

Suppose we want to sing a song which lasts for n units of time. Since n can be large, we want to memorize songs which require only a small amount of brain space, i.e. memory.

Let $S(n)$ be the *space complexity* of a song which lasts for n units of time.

The amount of space we need to store a song can be measured in either the words or characters needed to memorize it. Note that the number of characters is $\Theta(words)$ since every word in a song is at most 34 letters long – Supercalifragilisticexpialidocious!

What bounds can we establish on $S(n)$?

- $S(n) = O(n)$, since in the worst case we must explicitly memorize every word we sing – "The Star-Spangled Banner"

- $S(n) = \Omega(1)$, since we must know something about our song to sing it.

The Refrain

Most popular songs have a refrain, which is a block of text which gets repeated after each stanza in the song:

> Bye, bye Miss American pie
> Drove my chevy to the levy but the levy was dry
> Them good old boys were drinking whiskey and rye
> Singing this will be the day that I die.

Refrains made a song easier to remember, since you memorize it once yet sing it $O(n)$ times. But do they reduce the space complexity?

Not according to the big oh. If

$$n = \text{ repetitions } \times (\text{verse-size} + \text{refrain-size})$$

Then the space complexity is still $O(n)$ since it is only halved (if the verse-size = refrain-size):

$$S(n) = \text{ repetitions } \times \text{verse-size} + \text{refrain-size}$$

The k Days of Christmas

To reduce $S(n)$, we must structure the song differently.
Consider "The k Days of Christmas". All one must memorize is:

> On the kth Day of Christmas, my true love gave to me, $gift_k$
>
> $\vdots$
>
> On the First Day of Christmas, my true love gave to me, a partridge in a pear tree

But the time it takes to sing it is

$$\sum_{i=1}^{k} i = k(k+1)/2 = \Theta(k^2)$$

If $n = O(k^2)$, then $k = O(\sqrt{n})$, so $S(n) = O(\sqrt{n})$.

100 Bottles of Beer

What do kids sing on really long car trips?

n bottles of beer on the wall,
n bottles of beer.
You take one down and pass it around
$n-1$ bottles of beer on the ball.

All you must remember in this song is this template of size $\Theta(1)$, and the current value of n. The storage size for n depends on its value, but $\log_2 n$ bits suffice.

This for this song, $S(n) = O(\lg n)$.

Is there a song which eliminates even the need to count?

That's the way, uh-huh, uh-huh
I like it, uh-huh, huh

Reference: D. Knuth, 'The Complexity of Songs', *Comm. ACM*, April 1984, pp.18-24

Problem 2.1-2: Show that for any real constants a and b, $b > 0$,

$$(n + a)^b = \Theta(n^b)$$

To show $f(n) = \Theta(g(n))$, we must show O and Ω. *Go back to the definition!*

- *Big O* – Must show that $(n+a)^b \leq c_1 \cdot n^b$ for all $n > n_0$. When is this true? If $c_1 = 2^b$, this is true for all $n > |a|$ since $n + a < 2n$, and raise both sides to the b.
- *Big Ω* – Must show that $(n+a)^b \geq c_2 \cdot n^b$ for all $n > n_0$. When is this true? If $c_2 = (1/2)^b$, this is true for all $n > 3|a|/2$ since $n + a > n/2$, and raise both sides to the b.

Note the need for absolute values.

Problem 2.1-4:
(a) Is $2^{n+1} = O(2^n)$?
(b) Is $2^{2n} = O(2^n)$?

(a) Is $2^{n+1} = O(2^n)$?

Is $2^{n+1} \leq c * 2^n$?
Yes, if $c \geq 2$ for all n

(b) Is $2^{2n} = O(2^n)$?

Is $2^{2n} \leq c * 2^n$?
note $2^{2n} = 2^n * 2^n$
Is $2^n * 2^n \leq c * 2^n$?
Is $2^n \leq c$?

No! Certainly for any constant c we can find an n such that this is not true.

Recurrence Relations

Many algorithms, particularly divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

A recurrence relation is an equation which is defined in terms of itself.

Why are recurrences good things?

1. Many natural functions are easily expressed as recurrences:

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n \quad (polynomial)$$

$$a_n = 2a_{n-1}, a_1 = 1 \longrightarrow a_n = 2^{n-1} \quad (exponential)$$

$$a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n! \quad (weird\ function)$$

2. It is often easy to find a recurrence as the solution of a counting problem. *Solving* the recurrence can be done for many special cases as we will see, although it is somewhat of an art.

Recursion *is* Mathematical Induction!

In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces.

The *initial* or boundary condition terminate the recursion.

As we will see, induction provides a useful tool to solve recurrences – guess a solution and prove it by induction.

$$T_n = 2T_{n-1} + 1, T_0 = 0$$

n	0	1	2	3	4	5	6	7
T_n	0	1	3	7	15	31	63	127

Guess what the solution is?

Prove $T_n = 2^n - 1$ by induction:

1. Show that the basis is true: $T_0 = 2^0 - 1 = 0$.
2. Now assume true for T_{n-1}.
3. Using this assumption show:

$$T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

∎

Solving Recurrences

No general procedure for solving recurrence relations is known, which is why it is an art. My approach is:

Realize that linear, finite history, constant coefficient recurrences always can be solved

Check out any combinatorics or differential equations book for a procedure.

Consider $a_n = 2a_{n-1} + 2a_{n-2} + 1$, $a_1 = 1$, $a_2 = 1$

It has history = 2, degree = 1, and coefficients of 2 and 1. Thus it can be solved mechanically! Proceed:

- Find the characteristic equation, eg.

$$\alpha^2 - 2\alpha = 2 = 0$$

- Solve to get roots, which appear in the exponents.
- Take care of repeated roots and inhomogeneous parts.
- Find the constants to finish the job.

$$a_n = -1/3 + (1 - \sqrt{3})^n (1 + \sqrt{3})/3 + (1 + \sqrt{3})^n (-1 + \sqrt{3})/3$$

Systems like Mathematica and Maple have packages for doing this.

Guess a solution and prove by induction

To guess the solution, play around with small values for insight.

Note that you can do inductive proofs with the big-O's notations - just be sure you use it right.

Example: $T(n) \leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n$.

Show that $T(n) \leq c \cdot n \lg n$ for large enough c and n. Assume that it is true for $n/2$, then

$$\begin{aligned}
T(n) &\leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\
&\leq c \cdot n \lg(\lfloor n/2 \rfloor) + n \quad \text{dropping floors makes it bigger} \\
&= c \cdot n(\lg n - (\lg 2 = 1)) + n \quad \text{log of division} \\
&= c \cdot n \lg n - cn + n \\
&\leq c \cdot n \lg n \quad \text{whenever } c > 1
\end{aligned}$$

Starting with basis cases $T(2) = 4$, $T(3) = 5$, lets us complete the proof for $c \geq 2$.

Try backsubstituting until you know what is going on

Also known as the iteration method. Plug the recurrence back into itself until you see a pattern.

Example: $T(n) = 3T(\lfloor n/4 \rfloor) + n$.

Try backsubstituting:

$$\begin{aligned} T(n) &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor) \\ &= n + 3\lfloor n/4 \rfloor + 9(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \end{aligned}$$

The $(3/4)^n$ term should now be obvious.

Although there are only $\log_4 n$ terms before we get to $T(1)$, it doesn't hurt to sum them all since this is a fast growing geometric series:

$$T(n) \leq n \sum_{i=0}^{\infty} (\frac{3}{4})^i + \Theta(n^{\log_4 3} \times T(1))$$

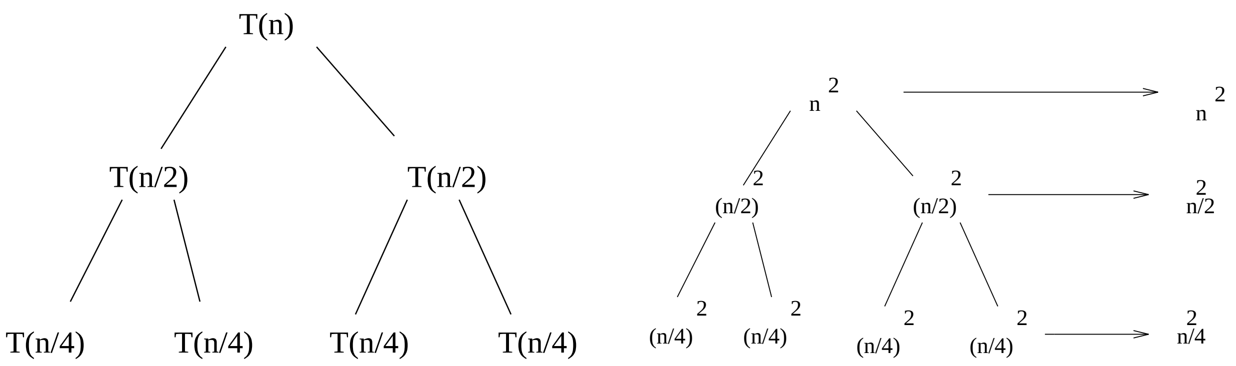
$$T(n) = 4n + o(n) = O(n)$$

Recursion Trees

Drawing a picture of the backsubstitution process gives you a idea of what is going on.

We must keep track of two things – (1) the size of the remaining argument to the recurrence, and (2) the additive stuff to be accumulated during this call.

Example: $T(n) = 2T(n/2) + n^2$



The remaining arguments are on the left, the additive terms on the right.

Although this tree has height $\lg n$, the total sum at each level decreases geometrically, so:

$$T(n) = \sum_{i=0}^{\infty} n^2/2^i = n^2 \sum_{i=0}^{\infty} 1/2^i = \Theta(n^2)$$

The recursion tree framework made this much easier to see than with algebraic backsubstitution.

See if you can use the Master theorem to provide an instant asymptotic solution

The Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Examples of the Master Theorem

Which case of the Master Theorem applies?

- $T(n) = 4T(n/2) + n$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n$.

  Is $n = O(n^{\log_2 4 - \epsilon}) = O(n^{2-\epsilon})$?

  Yes, so case 1 applies and $T(n) = \Theta(n^2)$.

- $T(n) = 4T(n/2) + n^2$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n^2$.

  Is $n^2 = O(n^{\log_2 4 - \epsilon}) = O(n^{2-\epsilon})$?

  No, if $\epsilon > 0$, but it is true if $\epsilon = 0$, so case 2 applies and $T(n) = \Theta(n^2 \log n)$.

- $T(n) = 4T(n/2) + n^3$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n^3$.

  Is $n^3 = \Omega(n^{\log_2 4 + \epsilon}) = \Omega(n^{2+\epsilon})$?

  Yes, for $0 < \epsilon < 1$, so case 3 *might* apply.

  Is $4(n/2)^3 \leq c \cdot n^3$?

  Yes, for $c \geq 1/2$, so there exists a $c < 1$ to satisfy the regularity condition, so case 3 applies and $T(n) = \Theta(n^3)$.

Why should the Master Theorem be true?

Consider $T(n) = aT(n/b) + f(n)$.

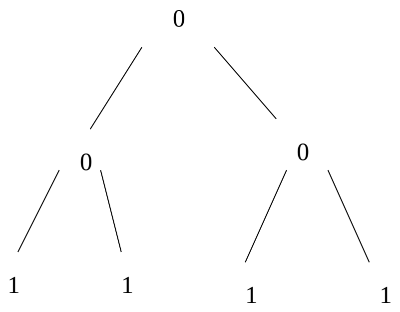
Suppose $f(n)$ is small enough

Say $f(n) = 0$, ie. $T(n) = aT(n/b)$.

Then we have a recursion tree where the only contribution is at the leaves.

There will be $\log_b n$ levels, with a^l leaves at level l.

$$T(n) = a^{\log_b n} = n^{\log_b a} \quad \text{Theorem 2.9 in CLR}$$



so long as $f(n)$ is small enough that it is dwarfed by this, we have case 1 of the Master Theorem!

Suppose f(n) is large enough

If we draw the recursion tree for $T(n) = aT(n/b) + f(n)$.

If $f(n)$ is a big enough function, the one top call can be bigger than the sum of all the little calls.

Example: $f(n) = n^3 > (n/3)^3 + (n/3)^3 + (n/3)^3$. In fact this holds unless $a \geq 27$!

In case 3 of the Master Theorem, the additive term dominates.

In case 2, both parts contribute equally, which is why the log pops up. It is (usually) what we want to have happen in a divide and conquer algorithm.

Famous Algorithms and their Recurrence

Matrix Multiplication

The standard matrix multiplication algorithm for two $n \times n$ matrices is $O(n^3)$.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \qquad \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

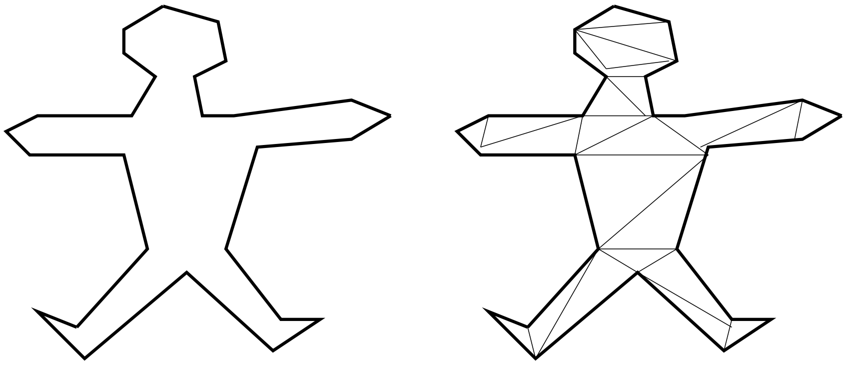
Strassen discovered a divide-and-conquer algorithm which takes $T(n) = 7T(n/2) + O(n^2)$ time.

Since $O(n^{\lg 7})$ dwarfs $O(n^2)$, case 1 of the master theorem applies and $T(n) = O(n^{2.81})$.

This has been "improved" by more and more complicated recurrences until the current best in $O(n^{2.38})$.

Polygon Triangulation

Given a polygon in the plane, add diagonals so that each face is a triangle None of the diagonals are allowed to cross.



Triangulation is an important first step in many geometric algorithms.

The simplest algorithm might be to try each pair of points and check if they see each other. If so, add the diagonal and recur on both halves, for a total of $O(n^3)$.

However, Chazelle gave an algorithm which runs in $T(n) = 2T(n/2) + O(\sqrt{(n)})$ time. Since $n^{1/2} = O(n^{1-\epsilon})$, by case 1 of the Master Theorem, Chazelle's algorithm is linear, ie. $T(n) = O(n)$.

Sorting

The classic divide and conquer recurrence is Mergesort's $T(n) = 2T(n/2) + O(n)$, which divides the data into equal-sized halves and spends linear time merging the halves after they are sorted.

Since $n = O(n^{\log_2 2}) = O(n)$ but not $n = O(n^{1-\epsilon})$, Case 2 of the Master Theorem applies and $T(n) = O(n \log n)$.

In case 2, the divide and merge steps balance out perfectly, as we usually hope for from a divide-and-conquer algorithm.

Mergesort Animations

Approaches to Algorithms Design

Incremental

Job is partly done - do a little more, repeat until done.
A good example of this approach is insertion sort

Divide-and-Conquer

A recursive technique

- Divide problem into sub-problems of the same kind.
- For subproblems that are really small (trivial), solve them directly. Else solve them recursively. (conquer)
- Combine subproblem solutions to solve the whole thing (combine)

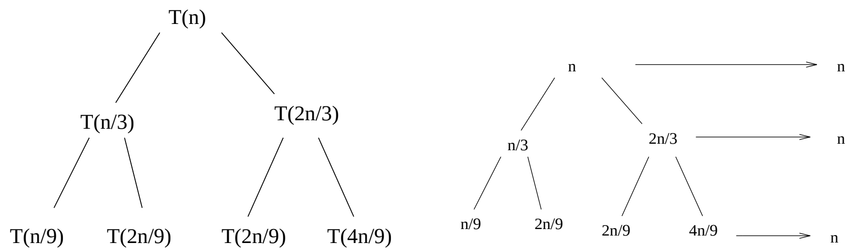
A good example of this approach is Mergesort.

4.2-2 Argue the solution to

$$T(n) = T(n/3) + T(2n/3) + n$$

is $\Omega(n \lg n)$ by appealing to the recursion tree.

Draw the recursion tree.



How many levels does the tree have? This is equal to the longest path from the root to a leaf.

The shortest path to a leaf occurs when we take the heavy branch each time. The height k is given by $n(1/3)^k \leq 1$, meaning $n \leq 3^k$ or $k \geq \lg_3 n$.

The longest path to a leaf occurs when we take the light branch each time. The height k is given by $n(2/3)^k \leq 1$, meaning $n \leq (3/2)^k$ or $k \geq \lg_{3/2} n$.

The problem asks to show that $T(n) = \Omega(n \lg n)$, meaning we are looking for a lower bound

On any ***full*** level, the additive terms sums to n. There are $\log_3 n$ full levels. Thus $T(n) \geq n \log_3 n = \Omega(n \lg n)$

4.2-4 Use iteration to solve $T(n) = T(n-a) + T(a) + n$, where $a \geq 1$ is a constant.

Note iteration is backsubstitution.

$$
\begin{aligned}
T(n) &= T(n-a) + T(a) + n \\
&= (T(n-2a) + T(a) + n - a) + T(a) + n \\
&= (T(n-3a) + T(a) + n - 2a) + 2T(a) + 2n - 3a \\
\vdots & \\
&\approx \sum_{i=0}^{n/a} T(a) + \sum_{i=0}^{n/a} n - ia \\
&\approx (n/a)T(a) + \sum_{i=0}^{n/a} n - a\sum_{i=0}^{n/a} i \\
&\approx (n/a)T(a) + n\sum_{i=0}^{n/a} 1 - a\sum_{i=0}^{n/a} i \\
&\approx (n/a)T(a) + n(n/a) - a(n/a)^2/2
\end{aligned}
$$

Why don't CS profs ever stop talking about sorting?!

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

2. Sorting is the best studied problem in computer science, with a variety of different algorithms known.

3. Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms - we will concentrate on the analysis.

Applications of Sorting

One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.

Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.

Speeding up searching is perhaps the most important application of sorting.

Closest pair

Given n numbers, find the pair which are closest to each other.

Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Element uniqueness

Given a set of n items, are they all unique or are there any duplicates?
Sort them and do a linear scan to check all adjacent pairs.
This is a special case of closest pair above.

Frequency distribution – Mode

Given a set of n items, which element occurs the largest number of times?

Sort them and do a linear scan to measure the length of all adjacent runs.

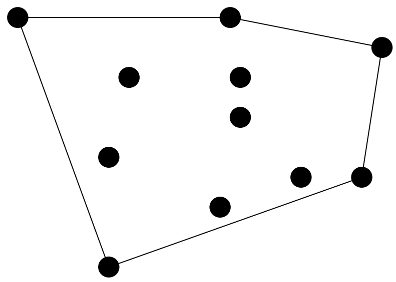
Median and Selection

What is the kth largest item in the set?

Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.

Convex hulls

Given n points in two dimensions, find the smallest area polygon which contains them all.



The convex hull is like a rubber band stretched over the points.

Convex hulls are the most important building block for more sophisticated geometric algorithms.

Once you have the points sorted by x-coordinate, they can be inserted from left to right into the hull, since the rightmost point is always on the boundary.

Without sorting the points, we would have to check whether the point is inside or outside the current hull.

Adding a new rightmost point might cause others to be deleted.

Huffman codes

If you are trying to minimize the amount of space a text file is taking up, it is silly to assign each letter the same length (ie. one byte) code.

Example: e is more common than q, a is more common than z.

If we were storing English text, we would want a and e to have shorter codes than q and z.

To design the best possible code, the first and most important step is to sort the characters in order of frequency of use.

Character	Frequency	Code
f	5	1100
e	9	1101
c	12	100
b	13	101
d	16	111
a	45	0

Selection Sort

A simple $O(n^2)$ sorting algorithm is selection sort.

Sweep through all the elements to find the smallest item, then the smallest remaining item, etc. until the array is sorted.

```
Selection-sort(A)
    for i = 1 to n
        for j = i + 1 to n
            if (A[j] < A[i]) then swap(A[i],A[j])
```

It is clear this algorithm must be correct from an inductive argument, since the ith element is in its correct position.

It is clear that this algorithm takes $O(n^2)$ time.

It is clear that the analysis of this algorithm cannot be improved because there will be $n/2$ iterations which will require at least $n/2$ comparisons each, so at least $n^2/4$ comparisons will be made. More careful analysis doubles this.

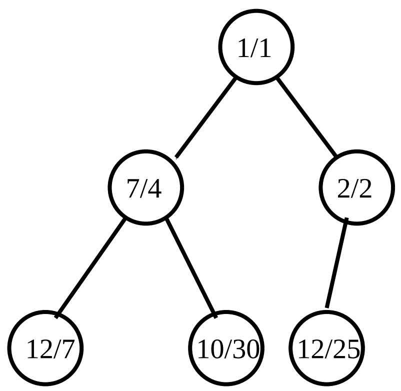
Thus selection sort runs in $\Theta(n^2)$ time.

Binary Heaps

A *binary heap* is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in root is $\geq$ all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.



The ancestor relation in a heap defines a *partial order* on its elements, which means it is reflexive, anti-symmetric, and transitive.

1. *Reflexive:* x is an ancestor of itself.

2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.

3. *Transitive:* if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial orders can be used to model heirarchies with incomplete information or equal-valued elements. One of my favorite games with my parents is fleshing out the partial order of "big" old-time movie stars.

The partial order defined by the heap structure is weaker than that of the total order, which explains

1. Why it is easier to build.

2. Why it is less useful than sorting (but still very important).

Constructing Heaps

Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.

If the new element is greater than its parent, swap their positions and recur.

Since at each step, we replace the root of a subtree by a larger one, we preserve the heap order.

Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.

Doing n such insertions takes $\Theta(n \log n)$, since the last $n/2$ insertions require $O(\log n)$ time each.

Heapify

The bottom up insertion algorithm gives a good way to build a heap, but Robert Floyd found a better way, using a *merge* procedure called *heapify*.

Given two heaps and a fresh element, they can be merged into one by making the new one the root and trickling down.

```
Build-heap(A)
    n = |A|
    For i = ⌊n/2⌋ to 1 do
        Heapify(A,i)


Heapify(A,i)
    left = 2i
    right = 2i + 1
    if (left ≤ n) and (A[left] > A[i]) then
        max = left
        else max = i
    if (right ≤ n) and (A(right] > A[max]) then
        max = right
    if (max ≠ i) then
        swap(A[i],A[max])
        Heapify(A,max)
```

Rough Analysis of Heapify

Heapify on a subtree containing n nodes takes

$$T(n) \leq T(2n/3) + O(1)$$

The $2/3$ comes from merging heaps whose levels differ by one. The last row could be exactly half filled. Besides, the asymptotic answer won't change so long the fraction is less than one.

Solve the recurrence using the Master Theorem.

Let $a = 1$, $b = 3/2$ and $f(n) = 1$.

Note that $\Theta(n^{\log_{3/2} 1}) = \Theta(1)$, since $\log_{3/2} 1 = 0$.

Thus Case 2 of the Master theorem applies.

The Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

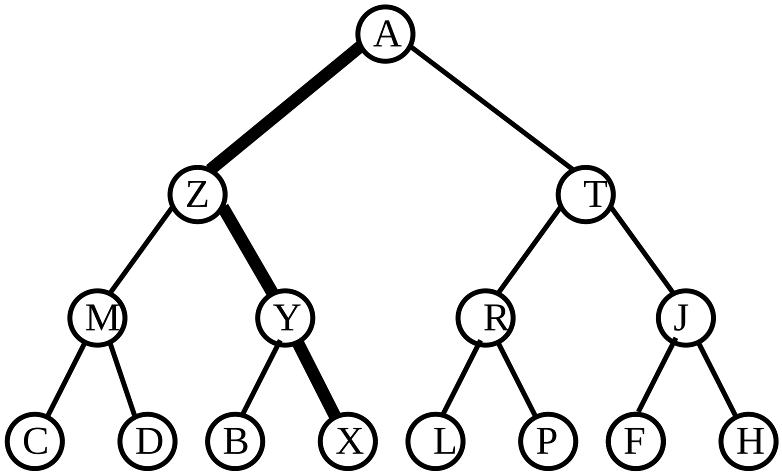
$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Exact Analysis of Heapify

In fact, Heapify performs better than $O(n \log n)$, because most of the heaps we merge are extremely small.



In a full binary tree on n nodes, there are $n/2$ nodes which are leaves (i.e. height 0), $n/4$ nodes which are height 1, $n/8$ nodes which are height 2, ...

In general, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h)$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum. But it should be clear that the series converges.

Proof of Convergence

Series convergence is the "free lunch" of algorithm analysis.
The identify for the sum of a geometric series is

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

If we take the derivative of both sides, ...

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides of the equation by x gives the identity we need:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Substituting $x = 1/2$ gives a sum of 2, so Build-heap uses at most $2n$ comparisons and thus linear time.

The Lessons of Heapsort, I

"Are we doing a careful analysis? Might our algorithm be faster than it seems?"

Typically in our analysis, we will say that since we are doing at most x operations of at most y time each, the total time is $O(xy)$.

However, if we overestimate too much, our bound may not be as tight as it should be!

Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

```
Heapsort(A)
    Build-heap(A)
    for i = n to 1 do
        swap(A[1],A[i])
        n = n - 1
        Heapify(A,1)
```

If we construct our heap from bottom to top using Heapify, we do not have to do anything with the last $n/2$ elements.

With the implicit tree defined by array positions, (i.e. the ith position is the parent of the $2i$th and $(2i+1)$st positions) the leaves start out as heaps.

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm, named *Heapsort*.

Heapsort Animations

The Lessons of Heapsort, II

Always ask yourself, "Can we use a different data structure?"

Selection sort scans throught the entire array, repeatedly finding the smallest remaining element.

For $i = 1$ to n
A: Find the smallest of the first $n - i + 1$ items.
B: Pull it out of the array and put it first.

Using arrays or unsorted linked lists as the data structure, operation A takes $O(n)$ time and operation B takes $O(1)$.

Using heaps, both of these operations can be done within $O(\lg n)$ time, balancing the work and achieving a better tradeoff.

Priority Queues

A *priority queue* is a data structure on sets of keys supporting the following operations:

- *Insert(S, x)* - insert x into set S
- *Maximum(S)* - return the largest key in S
- *ExtractMax(S)* - return and remove the largest key in S

These operations can be easily supported using a heap.

- *Insert* - use the trickle up insertion in $O(\log n)$.
- *Maximum* - read the first element in the array in $O(1)$.
- *Extract-Max* - delete first element, replace it with the last, decrement the element counter, then heapify in $O(\log n)$.

Applications of Priority Queues

Heaps as stacks or queues

- In a stack, *push* inserts a new item and *pop* removes the most recently pushed item.
- In a queue, *enqueue* inserts a new item and *dequeue* removes the least recently enqueued item.

Both stacks and queues can be simulated by using a heap, when we add a new *time* field to each item and order the heap according it this time field.

- To simulate the stack, increment the time with each insertion and put the maximum on top of the heap.
- To simulate the queue, decrement the time with each insertion and put the maximum on top of the heap (or increment times and keep the minimum on top)

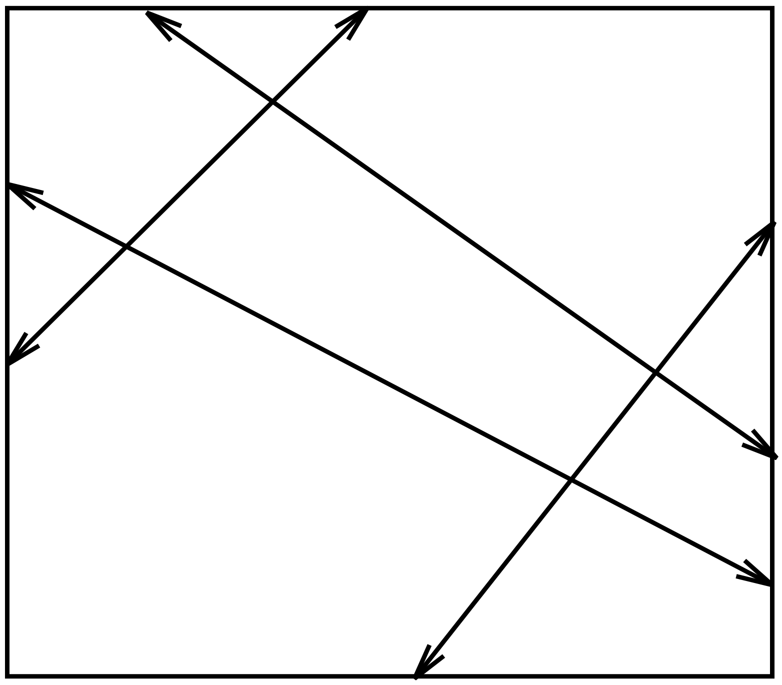
This simulation is not as efficient as a normal stack/queue implementation, but it is a cute demonstration of the flexibility of a priority queue.

Discrete Event Simulations

In simulations of airports, parking lots, and jai-alai – priority queues can be used to maintain who goes next.

The stack and queue orders are just special cases of orderings. In real life, certain people cut in line.

Sweepline Algorithms in Computational Geometry



In the priority queue, we will store the points we have not yet encountered, ordered by x coordinate. and push the line forward one stop at a time.

Greedy Algorithms

In greedy algorithms, we always pick the next thing which locally maximizes our score. By placing all the things in a priority queue and pulling them off in order, we can improve performance over linear search or sorting, particularly if the weights change.

Example: Sequential strips in triangulations.

Danny Heep

4-2 Find the missing integer from 0 to n using $O(n)$ "is bit[j] in A[i]" queries.

Note - there are a total of $n \lg n$ bits, so we are not allowed to *read* the entire input!

Also note, the problem is asking us to minimize the number of bits we read. We can spend as much time as we want doing other things provided we don't look at extra bits.

How can we find the last bit of the missing integer?

Ask all the n integers what their last bit is and see whether 0 or 1 is the bit which occurs less often than it is supposed to. That is the last bit of the missing integer!

How can we determine the second-to-last bit?

Ask the $\approx n/2$ numbers which ended with the correct last bit! By analyzing the bit patterns of the numbers from 0 to n which end with this bit.

By recurring on the remaining candidate numbers, we get the answer in $T(n) = T(n/2) + n = O(n)$, by the Master Theorem.

Quicksort

Although mergesort is $O(n \lg n)$, it is quite inconvenient for implementation with arrays, since we need space to merge.

In practice, the fastest sorting algorithm is Quicksort, which uses *partitioning* as its main idea.

Example: Pivot about 10.

17 12 6 19 23 8 5 10 – before

6 8 5 10 23 19 12 17 – after

Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.

Note that the pivot element ends up in the correct place in the total order!

Partitioning the elements

Once we have selected a pivot element, we can partition the array in one linear scan, by maintaining three sections of the array: $<$ pivot, $>$ pivot, and unexplored.

Example: pivot about 10

— 17 12 6 19 23 8 5 — 10

— 5 12 6 19 23 8 — 17

5 — 12 6 19 23 8 — 17

5 — 8 6 19 23 — 12 17

5 8 — 6 19 23 — 12 17

5 8 6 — 19 23 — 12 17

5 8 6 — 23 — 19 12 17

5 8 6 ——23 19 12 17

5 8 6 10 19 12 17 23

As we scan from left to right, we move the left bound to the right when the element is less than the pivot, otherwise we swap it with the *rightmost unexplored* element and move the right bound one step closer to the left.

Since the partitioning step consists of at most n swaps, takes time linear in the number of keys. But what does it buy us?

1. The pivot element ends up in the position it retains in the final sorted order.

2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

Thus we can sort the elements to the left of the pivot and the right of the pivot independently!

This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem.

Quicksort Animations

Pseudocode

```
Sort(A)
        Quicksort(A,1,n)


Quicksort(A, low, high)
        if (low < high)
                pivot-location = Partition(A,low,high)
                Quicksort(A,low, pivot-location - 1)
                Quicksort(A, pivot-location+1, high)


Partition(A,low,high)
        pivot = A[low]
        leftwall = low
        for i = low+1 to high
                if (A[i] < pivot) then
                        leftwall = leftwall+1
                        swap(A[i],A[leftwall])
        swap(A[low],A[leftwall])
```

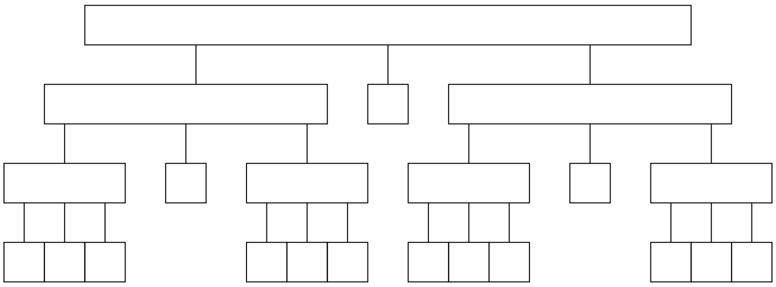
Best Case for Quicksort

Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?

The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.

The partition step on each subproblem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is $O(n)$.

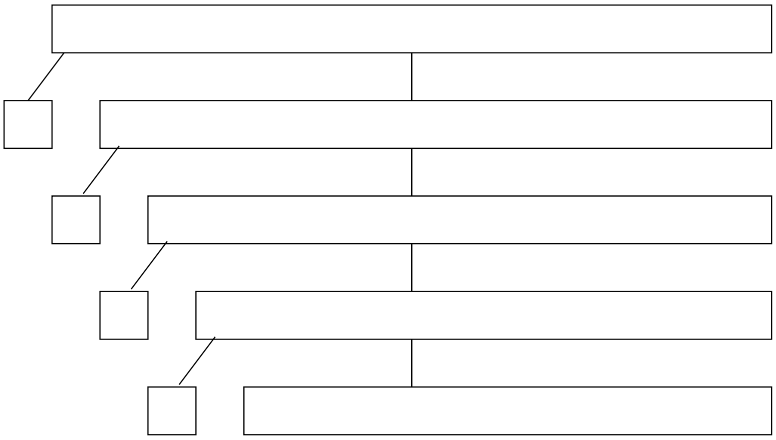
The recursion tree for the best case looks like this:



The total partitioning on each level is $O(n)$, and it take $\lg n$ levels of perfect partitions to get to single element subproblems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is $O(n \lg n)$.

Worst Case for Quicksort

Suppose instead our pivot element splits the array as unequally as possible. Thus instead of $n/2$ elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.

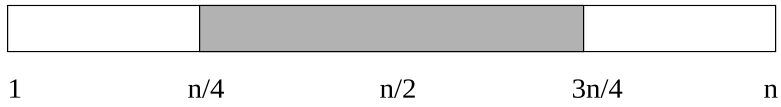
Thus the worst case time for Quicksort is worse than Heapsort or Mergesort.

To justify its name, Quicksort had better be good in the average case. Showing this requires some fairly intricate analysis.

The divide and conquer principle applies to real life. If you will break a job into pieces, it is best to make the pieces of equal size!

Intuition: The Average Case for Quicksort

Suppose we pick the pivot element at random in an array of n keys.



Half the time, the pivot element will be from the center half of the sorted array.

Whenever the pivot element is from positions $n/4$ to $3n/4$, the larger remaining subarray contains at most $3n/4$ elements.

If we assume that the pivot element is always in this range, what is the maximum number of partitions we need to get from n elements down to 1 element?

$$(3/4)^l \cdot n = 1 \longrightarrow n = (4/3)^l$$

$$\lg n = l \cdot \lg(4/3)$$

Therefore $l = \lg(4/3) \cdot \lg(n) < 2 \lg n$ good partitions suffice.

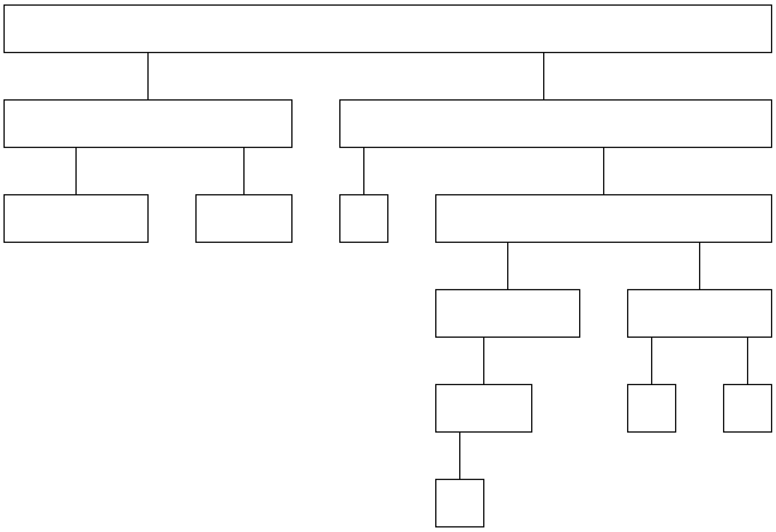
What have we shown?

At most $2\lg n$ levels of *decent partitions* suffices to sort an array of n elements.

But how often when we pick an arbitrary element as pivot will it generate a decent partition?

Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, we get one half the time on average.

If we need $2\lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4\lg n$ levels.



Since $O(n)$ work is done partitioning on each level, the average time is $O(n\lg n)$.

More careful analysis shows that the expected number of comparisons is $\approx 1.38n\lg n$.

Average-Case Analysis of Quicksort

To do a precise average-case analysis of quicksort, we formulate a recurrence given the exact expected time $T(n)$:

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}(T(p-1) + T(n-p)) + n - 1$$

Each possible pivot p is selected with equal probability. The number of comparisons needed to do the partition is $n-1$.

We will need one useful fact about the Harmonic numbers H_n, namely

$$H_n = \sum_{i=1}^{n} 1/i \approx \ln n$$

It is important to understand (1) where the recurrence relation comes from and (2) how the log comes out from the summation. The rest is just messy algebra.

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}(T(p-1) + T(n-p)) + n - 1$$

$$T(n) = \frac{2}{n} \sum_{p=1}^{n} T(p-1) + n - 1$$

$$nT(n) = 2 \sum_{p=1}^{n} T(p-1) + n(n-1) \quad \text{multiply by n}$$

$$(n-1)T(n-1) = 2 \sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2) \quad \text{apply to n-1}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

rearranging the terms give us:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

substituting $a_n = A(n)/(n+1)$ gives

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$

$$a_n \approx 2\sum_{i=1}^{n}\frac{1}{(i+1)} \approx 2\ln n$$

We are really interested in $A(n)$, so

$$A(n) = (n+1)a_n \approx 2(n+1)\ln n \approx 1.38n\lg n$$

What *is* the Worst Case?

The worst case for Quicksort depends upon how we select our partition or pivot element. If we always select either the first or last element of the subarray, the worst-case occurs when the input is already sorted!

A B D F H J K
B D F H J K
D F H J K
F H J K
H J K
J K
K

Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.

To eliminate this problem, pick a better pivot:

1. Use the middle element of the subarray as pivot.
2. Use a *random* element of the array as the pivot.
3. Perhaps best of all, take the median of three elements (first, last, middle) as the pivot. Why should we use median instead of the mean?

Whichever of these three rules we use, the worst case remains $O(n^2)$. However, because the worst case is no longer a natural order it is much more difficult to occur.

Is Quicksort really faster than Heapsort?

Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.

But how can we compare two $\Theta(n \lg n)$ algorithms to see which is faster? Using the RAM model and the big Oh notation, we can't!

When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort. The primary reason is that the operations in the innermost loop are simpler. The best way to see this is to implement both and experiment with different inputs.

Since the difference between the two programs will be limited to a multiplicative constant factor, the details of how you program each algorithm will make a big difference.

If you don't want to believe me when I say Quicksort is faster, I won't argue with you. It is a question whose solution lies outside the tools we are using.

Randomization

Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.

If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.

But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at *random*.

Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:

> "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time."

Where before, all we could say is:

> "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."

Since the time bound how does not depend upon your input distribution, this means that unless we are *extremely* unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.

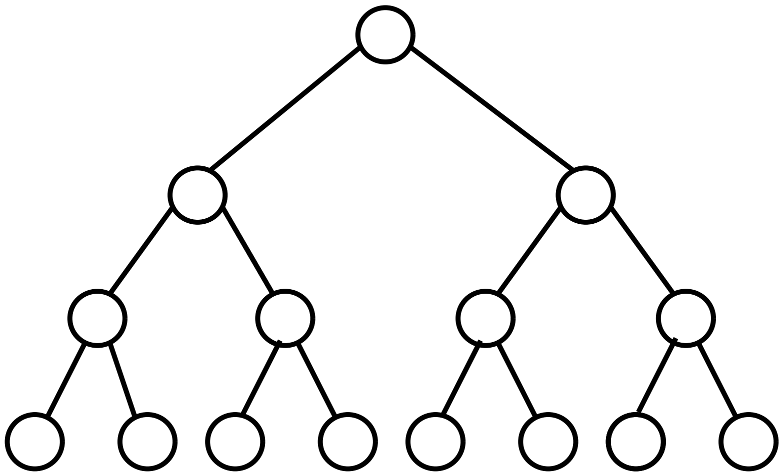
Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

The worst-case is still there, but we almost certainly won't see it.

7.1-2: Show that an n-element heap has height $\lfloor \lg n \rfloor$.

Since it is balanced binary tree, the height of a heap is clearly $O(\lg n)$, but the problem asks for an exact answer.

The height is defined as the number of edges in the longest simple path from the root.



The number of nodes in a complete balanced binary tree of height h is $2^{h+1} - 1$.

Thus the height increases only when $n = 2^{\lg n}$, or in other words when $\lg n$ is an integer.

7.1-5 Is a reverse sorted array a heap?

In a heap, each element is greater than or equal to each of its descendants.

In the array representation of a heap, the descendants of the ith element are the $2i$th and $(2i + 1)$th elements.

If A is sorted in reverse order, then $A[i] \geq A[j]$ implies that $i \leq j$.

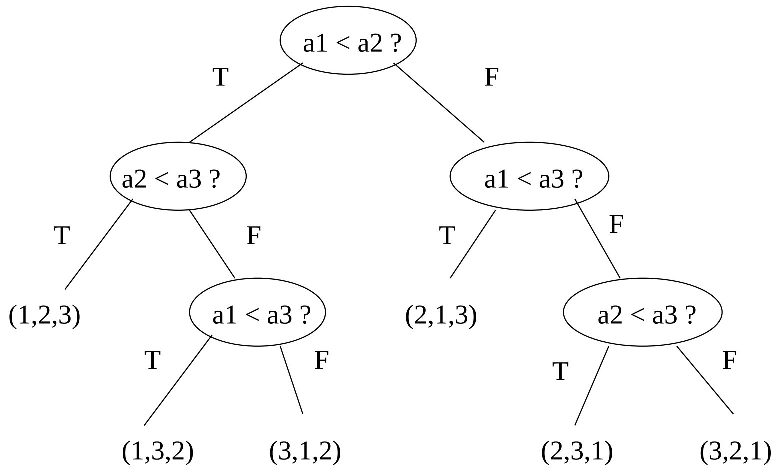
Since $2i > i$ and $2i + 1 > i$ then $A[2i] \leq A[i]$ and $A[2i + 1] \leq A[i]$.

Thus by definition A is a heap!

Can we sort in better than $n \lg n$?

Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.

Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.



Claim: the height of this decision tree is the worst-case complexity of sorting.

Once you believe this, a lower bound on the time complexity of sorting follows easily.

Since any two different permutations of n elements requires a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree, ie. at least $n!$ different leaves in the tree.

Since only binary comparisons (less than or greater than) are used, the decision tree is a binary tree.

Since a binary tree of height h has at most 2^h leaves, we know $n! \leq 2^h$, or $h \geq \lg(n!)$.

By inspection $n! > (n/2)^{n/2}$, since the last $n/2$ terms of the product are each greater than $n/2$. By Sterling's approximation, a better bound is $n! > (n/e)^n$ where $e = 2.718$.

$$h \geq \lg(n/e)^n = n \lg n - n \lg e = \Omega(n \lg n)$$

Non-Comparison-Based Sorting

All the sorting algorithms we have seen assume binary comparisons as the basic primative, questions of the form "is x before y?".

Suppose you were given a deck of playing cards to sort. Most likely you would set up 13 piles and put all cards with the same number in one pile.

A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K

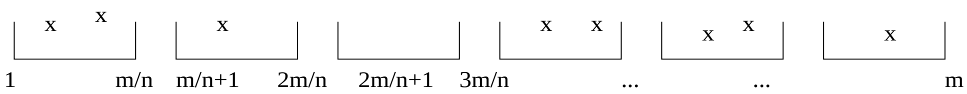
With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.

If we could find the correct pile for each card in constant time, and each pile gets $O(1)$ cards, this algorithm takes $O(n)$ time.

Bucketsort

Suppose we are sorting n numbers from 1 to m, where we know the numbers are approximately uniformly distributed.

We can set up n buckets, each responsible for an interval of m/n numbers from 1 to m



Given an input number x, it belongs in bucket number $\lceil xn/m \rceil$.

If we use an array of buckets, each item gets mapped to the right bucket in $O(1)$ time.

With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes $O(1)$ time!

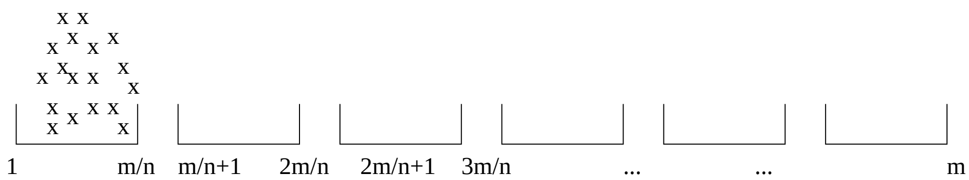
The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is $O(n)$.

What happened to our $\Omega(n \lg n)$ lower bound!

We can use bucketsort effectively whenever we understand the distribution of the data.

However, bad things happen when we assume the wrong distribution.

Suppose in the previous example all the keys happened to be 1. After the bucketing phase, we have:



We spent linear time distributing our items into buckets and learned *nothing*. Perhaps we could split the big bucket recursively, but it is not certain that we will ever win unless we understand the distribution.

Problems like this are why we worry about the worst-case performance of algorithms!

Such distribution techniques can be used on strings instead of just numbers. The buckets will correspond to letter ranges instead of just number ranges.

The worst case "shouldn't" happen if we understand the distribution of our data.

Real World Distributions

Consider the distribution of names in a telephone book.

- Will there be a lot of Skiena's?
- Will there be a lot of Smith's?
- Will there be a lot of Shifflett's?

Either make *sure* you understand your data, or use a good worst-case or randomized algorithm!

The Shifflett's of Charlottesville

For comparison, note that there are seven Shifflett's (of various spellings) in the 1000 page Manhattan telephone directory.

Shifflett Debbie K Ruckersville 985-7957
Shifflett Debra S SR 617 Quinque 985-8813
Shifflett Delma SR609 985-3688
Shifflett Delmas Crozet 823-5901
Shifflett Dempsey & Marilynn
  100 Greenbrier Ter 973-7195
Shifflett Denise Rt 627 Dyke 985-8097
Shifflett Dennis Stanardsville 985-4560
Shifflett Dennis H Stanardsville 985-2924
Shifflett Dewey E Rt667 985-6576
Shifflett Dewey O Dyke 985-7269
Shifflett Diana 508 Bainbridge Av 979-7035
Shifflett Doby & Patricia Rt6 286-4227
Shifflett Don&Ola Rt 621 974-7463

Shifflett James 2219 Williamsburg Rd
Shifflett James B 801 Stonehenge Av .
Shifflett James C Stanardsville .....
Shifflett James E Earlysville ....
Shifflett James E Jr 552 Cleveland Av
Shifflett James F & Lois LongMeadow
Shifflett James F & Vernell Rt671 ...
Shifflett James J 1430 Rugby Av ....
Shifflett James K St George Av .....
Shifflett James L SR33 Stanardsville .
Shifflett James O Earlysville ....
Shifflett James O Stanardsville .....
Shifflett James R Old Lynchburg Rd ..
Shifflett James R Rt753 Esmont ....

Rules for Algorithm Design

The secret to successful algorithm design, and problem solving in general, is to make sure you ask the right questions. Below, I give a possible series of questions for you to ask yourself as you try to solve difficult algorithm design problems:

1. Do I really understand the problem?
   (a) What exactly does the input consist of?
   (b) What exactly are the desired results or output?
   (c) Can I construct some examples small enough to solve by hand? What happens when I solve them?
   (d) Are you trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Might your problem be formulated in more than one way? Which formulation seems easiest?

2. Can I find a simple algorithm for the problem?
   (a) Can I find the solve my problem exactly by searching all subsets or arrangements and picking the best one?
      i. If so, why am I sure that this algorithm always gives the correct answer?
      ii. How do I measure the quality of a solution once I construct it?
      iii. Does this simple, slow solution run in polynomial or exponential time?
      iv. If I can't find a slow, *guaranteed* correct algorithm, am I sure that my problem is well defined enough to permit a solution?
   (b) Can I solve my problem by repeatedly trying some heuristic rule, like picking the biggest item first? The smallest item first? A random item first?
      i. If so, on what types of inputs does this heuristic rule work well? Do these correspond to the types of inputs that might arise in the application?

ii. On what types of inputs does this heuristic rule work badly? If no such examples can be found, can I show that in fact it always works well?

iii. How fast does my heuristic rule come up with an answer?

3. Are there special cases of this problem I know how to solve exactly?

   (a) Can I solve it efficiently when I ignore some of the input parameters?

   (b) What happens when I set some of the input parameters to trivial values, such as 0 or 1?

   (c) Can I simplify the problem to create a problem I can solve efficiently? How simple do I have to make it?

   (d) If I can solve a certain special case, why can't this be generalized to a wider class of inputs?

4. Which of the standard algorithm design paradigms seem most relevant to the problem?

   (a) Is there a set of items which can be sorted by size or some key? Does this sorted order make it easier to find what might be the answer?

   (b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search, or a partition of the elements into big and small, or left and right? If so, does this suggest a divide-and-conquer algorithm?

   (c) Are there certain operations being repeatedly done on the same data, such as searching it for some element, or finding the largest/smallest remaining element? If so, can I use a data structure of speed up these queries, like hash tables or a heap/priority queue?

5. Am I still stumped?

   (a) Why don't I go back to the beginning of the list and work through the questions again? Do any of my answers from the first trip change on the second?

8.2-3 Argue that insertion sort is better than Quicksort for sorting checks

---

In the best case, Quicksort takes $\Theta(n \lg n)$. Although using median-of-three turns the sorted permutation into the best case, we lose if insertion sort is better on the given data.

$$1\ 2\ 3\ 4\ 6\ 7\ 9\ 11 — 5$$

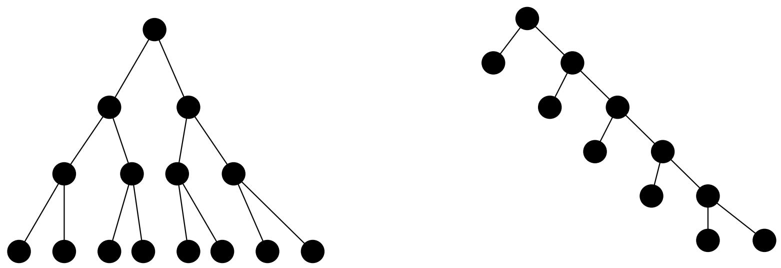
In insertion sort, the cost of each insertion is the number of items which we have to jump over. In the check example, the expected number of moves per items is small, say c. We win if $c << \lg n$.

8.3-1 Why do we analyze the average-case performance of a randomized algorithm, instead of the worst-case?

In a randomized algorithm, the worst case is not a matter of the input but only of luck. Thus we want to know what kind of luck to expect. Every input we see is drawn from the uniform distribution.

8.3-2 How many calls are made to Random in randomized quicksort in the best and worst cases?

Each call to random occurs once in each call to partition.
The number of partitions is $\Theta(n)$ in any run of quicksort!!



There is some potential variation depending upon what you do with intervals of size 1 – do you call partition on intervals of size one? However, there is no asymptotic difference between best and worst case.

The reason – any binary tree with n leaves has $n-1$ internal nodes, each of which corresponds to a call to partition in the quicksort recursion tree.

Elementary Data Structures

"Mankind's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps will be the "of-the-shelf" components we build our algorithm from. There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementation of these operations.

The fact that we can describe the behavior of our data structures in terms of abstract operations explains why we can use them without thinking, while the fact that we have different implementation of the same abstract operations enables us to optimize performance.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

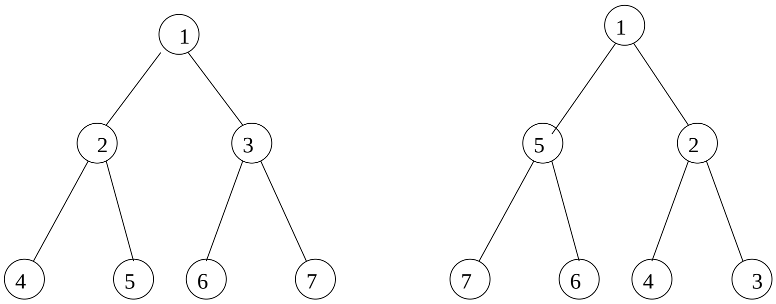
A *stack* supports last-in, first-out operations: push and pop.

A *queue* supports first-in, first-out operations: enqueue and dequeue.

A *deque* is a double ended queue and supports all four operations: push, pop, enqueue, dequeue.

Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

Both can be used to traverse a tree, but the order is completely different.



Queue Stack

Which order is better for WWW crawler robots?

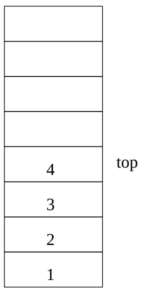
Stack Implementation

Although this implementation uses an array, a linked list would eliminate the need to declare the array size in advance.

STACK-EMPTY(S)
    if top[S] $= 0$
        then return TRUE
        else return FALSE

PUSH(S, x)
    $top[S] \leftarrow top[S] + 1$
    $S[top[S] \leftarrow x$

POP(S)
    if STACK-EMPTY(S)
        then error "underflow"
        else $top[S] \leftarrow top[S] - 1$
            return $S[top[S] + 1]$

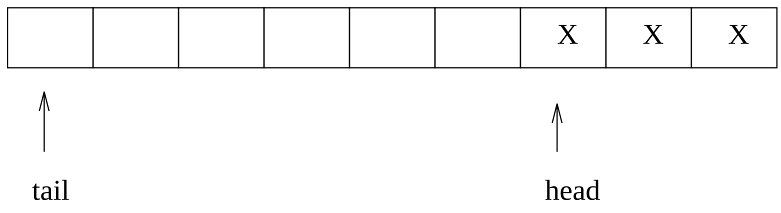


All are $O(1)$ time operations.

Queue Implementation

A circular queue implementation requires pointers to the head and tail elements, and wraps around to reuse array elements.

```
ENQUEUE(Q, x)
        Q[tail[Q]] ← x
        if tail[Q] = length[Q]
                then tail[Q] ← 1
                else tail[Q] ← tail[Q] + 1
```




```
DEQUEUE(Q)
        x = Q[head[Q]]
        if head[Q] = length[Q]
                then head[Q] = 1
                else head[Q] = head[Q] + 1
        return x
```

A list-based implementation would eliminate the possibility of overflow. All are $O(1)$ time operations.

Dynamic Set Operations

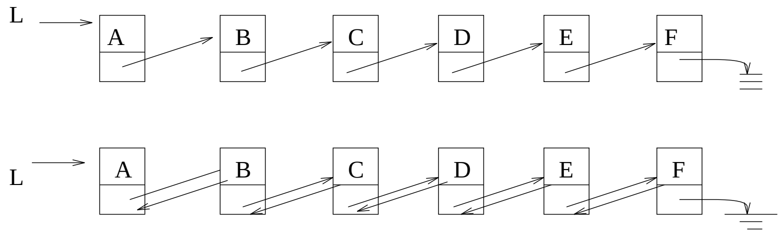
Perhaps the most important class of data structures maintain a set of items, indexed by keys.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

- *Search(S,k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.
- *Insert(S,x)* – A modifying operation that augments the set S with the element x.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.
- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- *Next(S,x), Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

Pointer Based Implementation

We can maintain a dictionary in either a singly or doubly linked list.



We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.

Since the extra big-Oh costs of doubly-linkly lists is zero, we will usually assume they are, although it might not be necessary.

Singly linked to doubly-linked list is as a Conga line is to a Can-Can line.

Array Based Sets

Unsorted Arrays

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot, $O(1)$
- Delete(S,x) - copy nth item to the xth spot, $O(1)$
- Min(S,x), Max(S,x) - sequential search, $O(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $O(n)$

Sorted Arrays

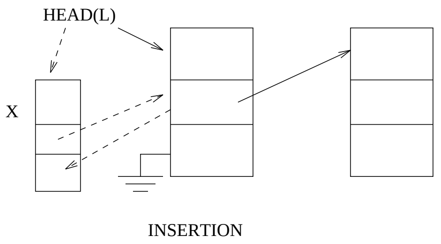
- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S,x), Max(S,x) - first or last element, $O(1)$
- Successor(S,x), Predecessor(S,x) - Add or subtract 1 from pointer, $O(1)$

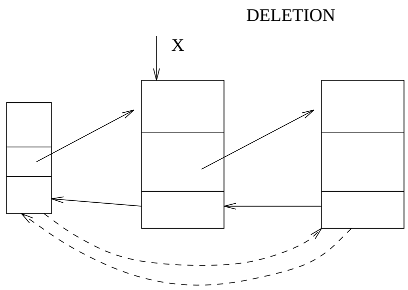
What are the costs for a heap?

Unsorted List Implementation

LIST-SEARCH(L, k)
    x = head[L]
    while $x <> NIL$ and $key[x] <> k$
        do x = next[x]
    return x

Note: the while loop might require two lines in some programming languages.



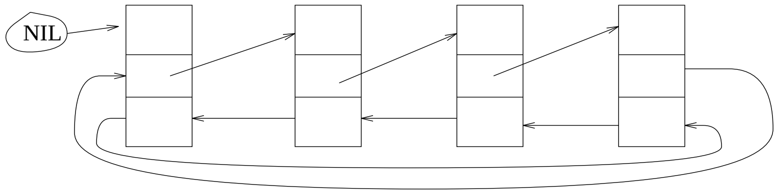


LIST-INSERT(L, x)
    next[x] = head[L]
    if head[L] <> NIL
        then prev[head[L]] = x
    head[L] = x
    prev[x] = NIL

LIST-DELETE(L, x)
    if $prev[x] <> NIL$
        then next[prev[x]] = next[x]
        else head[L] = next[x]
    if $next[x] <> NIL$
        then prev[next[x]] = prev[x]

Sentinels

Boundary conditions can be eliminated using a sentinel element which doesn't go away.



LIST-SEARCH'(L, k)
    x = next[nil[L]]
    while $x <> NIL[L]$ and $key[x] <> k$
        do x = next[x]
    return x

LIST-INSERT'(L, x)
    next[x] = next[nil[L]]
    prev[next[nil[L]]] = x
    next[nil[L]] = x
    prev[x] = NIL[L]

LIST-DELETE'(L, x)
    next[prev[x]] <> next[x]
    next[prev[x]] = prev[x]

Hash Tables

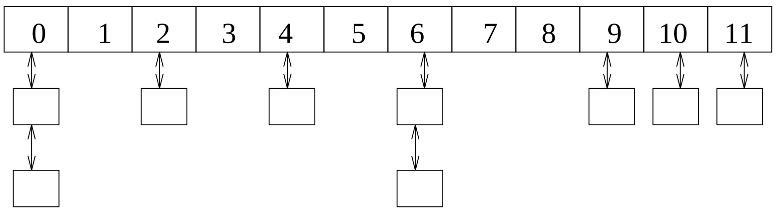
Hash tables are a *very practical* way to maintain a dictionary. As with bucket sort, it assumes we know that the distribution of keys is fairly well-behaved.

The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index. A hash function is a mathematical function which maps keys to integers.

In bucket sort, our hash function mapped the key to a bucket based on the first letters of the key. "Collisions" were the set of keys mapped to the same bucket.

If the keys were uniformly distributed, then each bucket contains very few keys!

The resulting short lists were easily sorted, and could just as easily be searched!



Hash Functions

It is the job of the hash function to map keys to integers. A good hash function:

1. Is cheap to evaluate
2. Tends to use all positions from $0 \ldots M$ with uniform frequency.
3. Tends to put similar keys in different parts of the tables (Remember the Shifletts!!)

The first step is usually to map the key to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

This large number must be reduced to an integer whose size is between 1 and the size of our hash table.

One way is by $h(k) = k \bmod M$, where M is best a large prime not too close to $2^i - 1$, which would just mask off the high bits.

This works on the same principle as a roulette wheel!

Good and Bad Hash functions

The first three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The last three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The Birthday Paradox

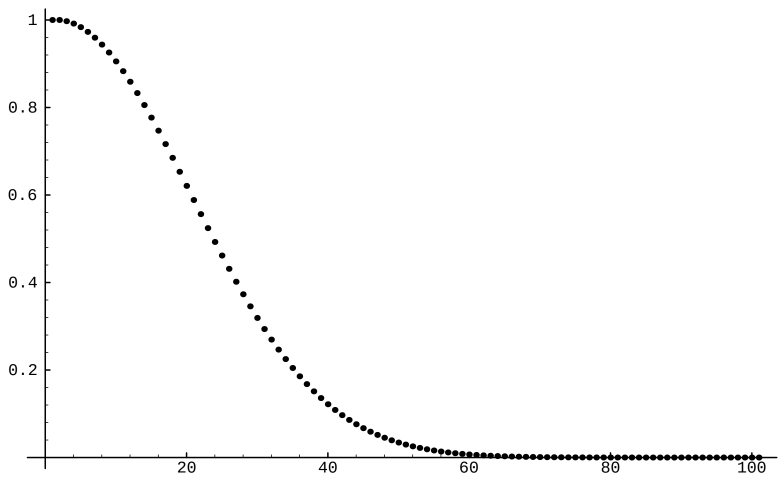
No matter how good our hash function is, we had better be prepared for collisions, because of the birthday paradox.

J	F	M	A	M	J	J1	A	S	O	N	D

The probability of there being *no* collisions after n insertions into an m-element table is

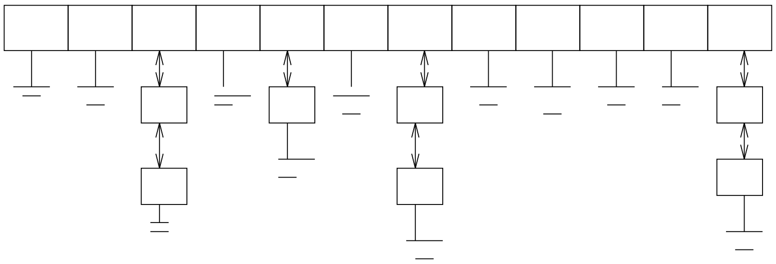
$$(m/m) \times ((m-1)/m) \times \ldots \times ((m-n+1)/m) = \Pi_{i=0}^{n-1}(m-i)/m$$

When $m = 366$, this probability sinks below 1/2 when $N = 23$ and to almost 0 when $N \geq 50$.



Collision Resolution by Chaining

The easiest approach is to let each element in the hash table be a pointer to a list of keys.



Insertion, deletion, and query reduce to the problem in linked lists. If the n keys are distributed uniformly in a table of size m/n, each operation takes $O(m/n)$ time.

Chaining is easy, but devotes a considerable amount of memory to pointers, which could be used to make the table larger. Still, it is my preferred method.

Open Addressing

We can dispense with all these pointers by using an implicit reference derived from a simple function:

1	2	3	4	5	6	7	8	9	10	11	
		X		X	X		X	X			

If the space we want to use is filled, we can examine the remaining locations:

1. Sequentially $h, h+1, h+2, \ldots$
2. Quadratically $h, h+1^2, h+2^2, h+3^2 \ldots$
3. Linearly $h, h+k, h+2k, h+3k, \ldots$

The reason for using a more complicated science is to avoid long runs from similarly hashed keys.

Deletion in an open addressing scheme is ugly, since removing one element can break a chain of insertions, making some elements inaccessible.

Performance on Set Operations

With either chaining or open addressing:

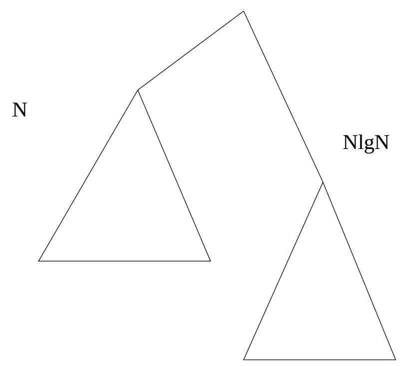
- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary. However, we will not use it much in proving the efficiency of our algorithms, since the worst-case time is unpredictable.

The best worst-case bounds come from balanced binary trees, such as red-black trees.

9.1-3 Show that there is no sorting algorithm which sorts at least $(1/2^n) \times n!$ *instances in* $O(n)$ *time.*

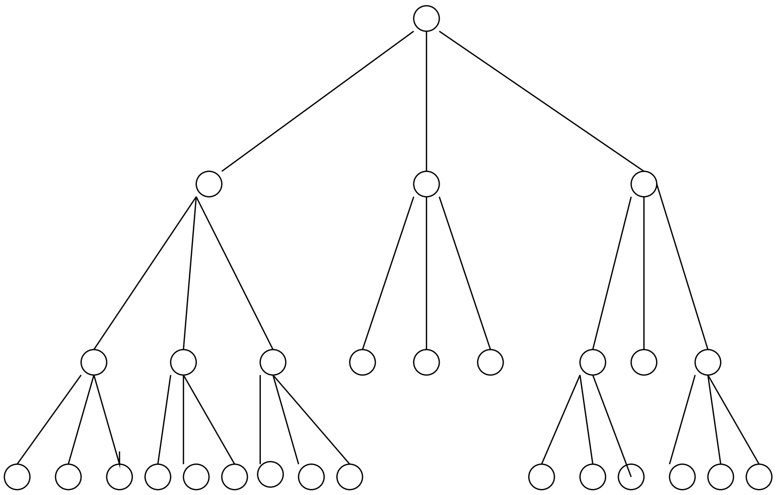
Think of the decision tree which can do this. What is the shortest tree with $(1/2^n) \times n!$ leaves?



$$\begin{aligned} h > \lg(n!/2^n) &= \lg(n!) - \lg(2^n) \\ &= \Theta(n \lg n) - n \\ &= \Theta(n \lg n) \end{aligned}$$

Moral: there cannot be too many good cases for any sorting algorithm!

9.1-4 Show that the $\Omega(n \lg n)$ lower bound for sorting still holds with ternary comparisons.



The maximum number of leaves in a tree of height h is 3^h,

$$\lg_3(n!) = \Theta(n \lg n)$$

So it goes for any constant base.

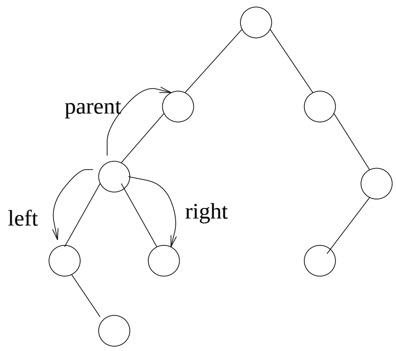
Binary Search Trees

"I think that I shall never see
a poem as lovely as a tree Poem's
are wrote by fools like me but only
G-d can make a tree "
– Joyce Kilmer

Binary search trees provide a data structure which efficiently supports all six dictionary operations.

A binary tree is a rooted tree where each node contains at most two children.

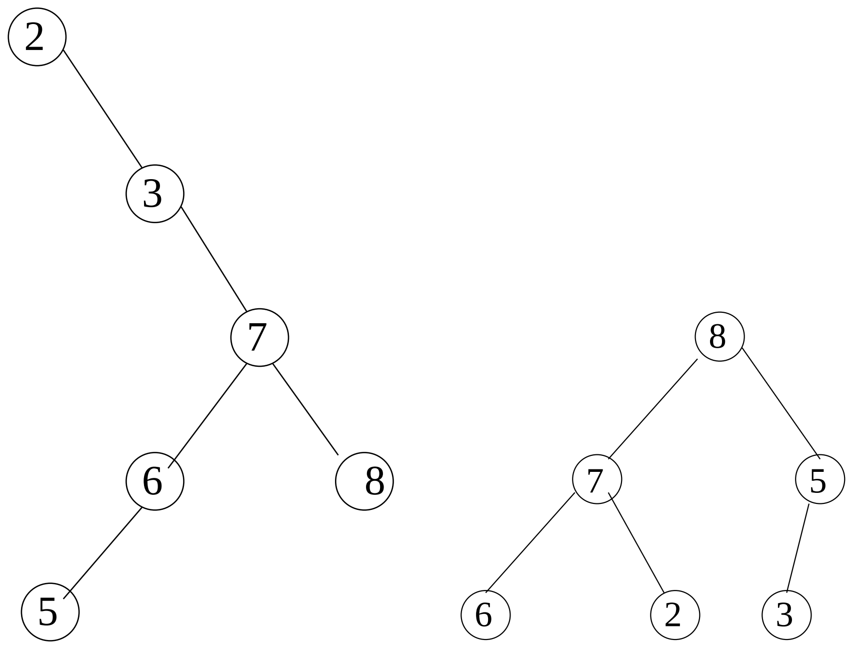
Each child can be identified as either a left or right child.



A binary tree can be implemented where each node has *left* and *right* pointer fields, an (optional) *parent* pointer, and a data field.

Binary Search Trees

A binary *search* tree labels each node in a binary tree with a single key such that for any node x, and nodes in the left subtree of x have keys $\leq x$ and all nodes in the right subtree of x have key's $\geq x$.



Left: A binary search tree. Right: A heap but not a binary search tree.

The search tree labeling enables us to find where any key is. Start at the root - if that is not the one we want, search either left or right depending upon whether what we want is $\leq$ or $\geq$ then the root.

Searching in a Binary Tree

Dictionary search operations are easy in binary trees ...

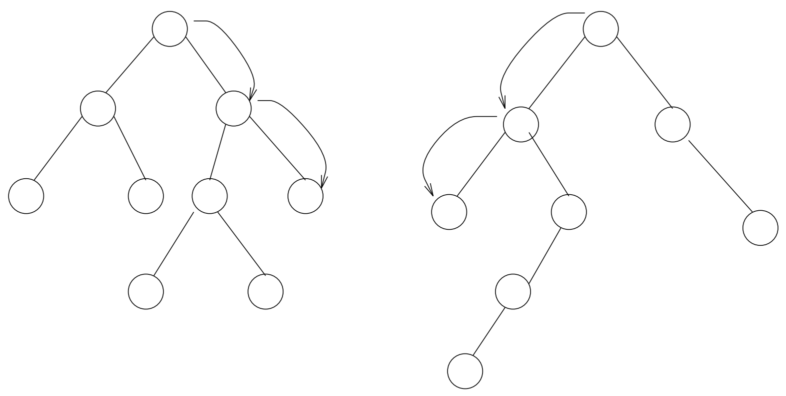
TREE-SEARCH(x, k)
    if ($x = NIL$) and ($k = key[x]$)
        then return x
    if ($k < key[x]$)
        then return TREE-SEARCH(left[x],k)
        else return TREE-SEARCH(right[x],k)

The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees – recursive structure, recursive algorithm.

This takes time proportional to the height of the tree, $O(h)$.

Maximum and Minimum

Where are the maximum and minimum elements in a binary tree?



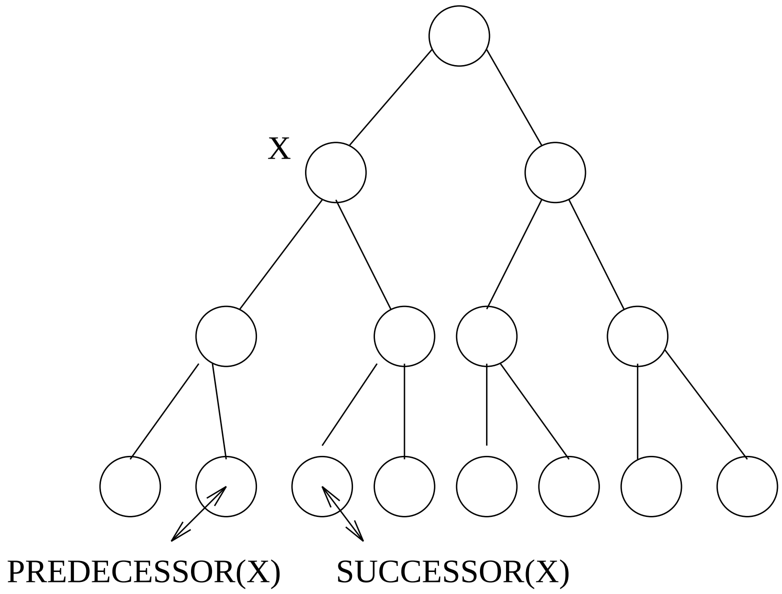
TREE-MAXIMUM(X)
  while $right[x] \neq NIL$
    do x = right[x]
  return x

TREE-MINIMUM(x)
  while $left[x] \neq NIL$
    do x = left[x]
  return x

Both take time proportional to the height of the tree, $O(h)$.

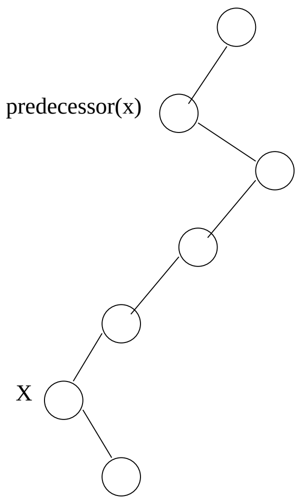
Where is the predecessor?

Where is the predecessor of a node in a tree, assuming all keys are distinct?



If X has two children, its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

What if a node doesn't have children?



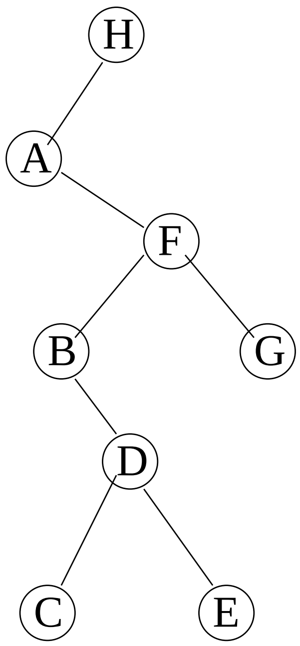
If it does not have a left child, a node's predecessor is its first left ancestor.

The proof of correctness comes from looking at the in-order traversal of the tree.

Tree-Successor(x)
    if $right[x] \neq NIL$
        then return Tree-Minimum($right[x]$)
        $y \leftarrow p[x]$
    while ($y \neq NIL$) and ($x = right[y]$)
        do $x \leftarrow y$
        $y \leftarrow p[y]$
    return y

Tree predecessor/successor both run in time proportional to the height of the tree.

In-Order Traversal

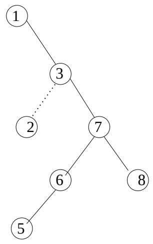



```
Inorder-Tree-walk(x)
    if (x <> NIL)
    then Inorder-Tree-Walk(left[x])
        print key[x]
        Inorder-Tree-walk(right[x])
```

A-B-C-D-E-F-G-H

Tree Insertion

Do a binary search to find where it should be, then replace the termination NIL pointer with the new item.



Tree-insert(T, z)
$y =$ NIL
$x = root[T]$
while $x \neq NIL$
do $y = x$
if $key[z] < key[x]$
then $x = left[x]$
else $x = right[x]$
$p[z] \leftarrow y$
if $y =$ NIL
then $root[T] \leftarrow z$
else if $key[z] < key[y]$
then $left[y] \leftarrow z$
else $right[y] \leftarrow z$

y is maintained as the parent of x, since x eventually becomes NIL.

The final test establishes whether the NIL was a left or right turn from y.

Insertion takes time proportional to the height of the tree, $O(h)$.

Tree Deletion

Deletion is somewhat more tricky than insertion, because the node to die may not be a leaf, and thus effect other nodes.

Case (a), where the node is a leaf, is simple - just NIL out the parents child pointer.

Case (b), where a node has one chld, the doomed node can just be cut out.

Case (c), relabel the node as its successor (which has at most one child when z has two children!) and delete the successor!

This implementation of deletion assumes parent pointers to make the code nicer, but if you had to save space they could be dispensed with by keeping the pointers on the search path stored in a stack.

```
Tree-Delete(T, z)
    if (left[z] = NIL) or (right[z] = NIL)
        then y ← z
        else y ← Tree-Successor(z)
    if left[y] ≠ NIL
        then x ← left[y]
        else x ← right[y]
    if x ≠ NIL
        then p[x] ← p[y]
    if p[y] = NIL
        then root[T] ← x
        else if (y = left[p[y]])
            then left[p[y]] ← x
            else right[p[y]] ← x
    if (y <> z)
        then key[z] ← key[y]
            /* If y has other fields, copy them, too. */
    return y
```

Lines 1-3 determine which node y is physically removed.

Lines 4-6 identify x as the non-nil decendant, if any.

Lines 7-8 give x a new parent.

Lines 9-10 modify the root node, if necessary

Lines 11-13 reattach the subtree, if necessary.

Lines 14-16 if the removed node is deleted, copy.

Conclusion: deletion takes time proportional to the height of the tree.

Balanced Search Trees

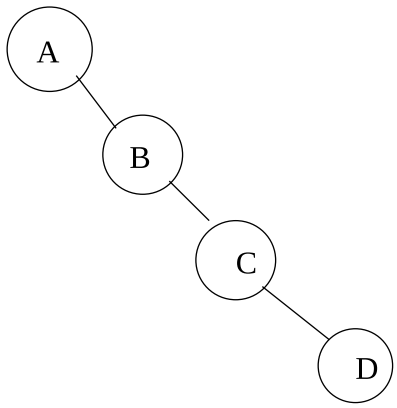
All six of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.

The best height we could hope to get is $\lg n$, if the tree was perfectly balanced, since

$\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$

But if we get unlucky with our order of insertion or deletion, we could get linear height!

insert(a)
insert(b)
insert(c)
insert(d)
$\vdots$

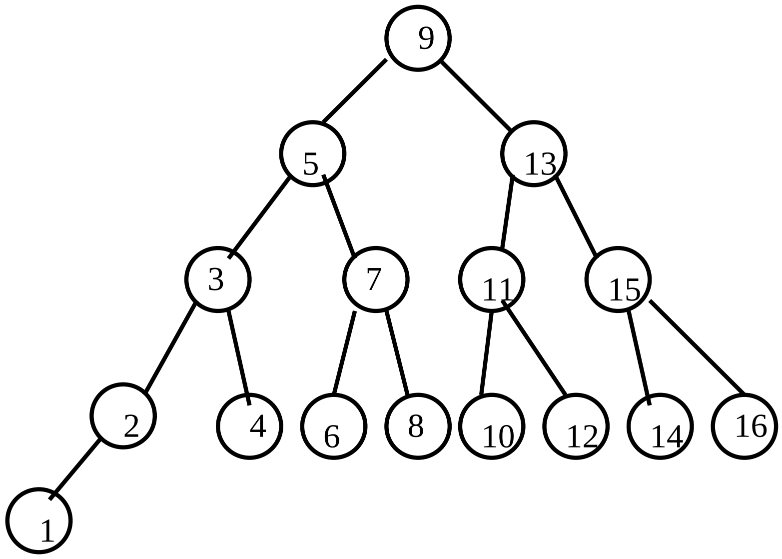


In fact, random search trees on average have $\Theta(\lg N)$ height, but we are worried about worst case height.

We can't easily use randomization - Why?

Perfectly Balanced Trees

Perfectly balanced trees require a lot of work to maintain:



If we insert the key 1, we must move every single node in the tree to rebalance it, taking $\Theta(n)$ time.

Therefore, when we talk about "balanced" trees, we mean trees whose height is $O(\lg n)$, so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take $O(\lg n)$ time.

Red-Black trees are binary search trees where each node is assigned a color, where the coloring scheme helps us maintain the height as $\Theta(\lg n)$.

Red-Black Tree Definition

Red-black trees have the following properties:

1. Every node is colored either red or black.
2. Every leaf (NIL pointer) is black.
3. If a node is red then both its children are black.
4. Every single path from a node to a decendant leaf contains the same number of black nodes.

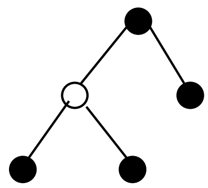
What does this mean?

If the root of a red-black tree is black can we just color it red?

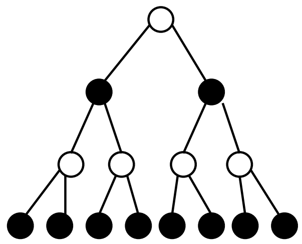
No! For one of its children might be red.

If an arbitrary node is red can we color it black?

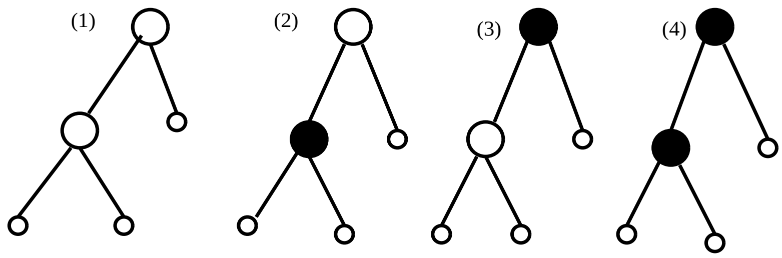
No! Because now all nodes may not have the same black height.



What tree maximizes the number of nodes in a tree of black height h?



What does a red-black tree with two real nodes look like?



Not (1) - consecutive reds Not (2), (4) - Non-Uniform black height

Red-Black Tree Height

Lemma: A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Proof: Our strategy; first we bound the number of nodes in any subtree, then we bound the height of any subtree.

We claim that any subtree rooted at x has at least $2^{bh(x)}$ - 1 *internal* nodes, where $bh(x)$ is the black height of node x.

Proof, by induction:

$$bh(x) = 0 \to x \text{ is a leaf}, \to 2^0 - 1 = 0$$

Now assume it is true for all tree with black height $< bh(x)$.

If x is black, both subtrees have black height $bh(x) - 1$. If x is red, the subtrees have black height $bh(x)$.

Therefore, the number of internal nodes in any subtree is

$$n \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 \geq 2^{bh(x)} - 1$$

Now, let h be the height of our red-black tree. At least half the nodes on any single path from root to leaf must be black if we ignore the root.

Thus $bh(x) \geq h/2$ and $n \geq 2^{h/2} - 1$, so $n + 1 \geq 2^{h/2}$.

This implies that $\lg(n+1) \geq h/2$,so $h \leq 2\lg(n+1)$. ∎

Therefore red-black trees have height at most twice optimal. We have a balanced search tree if we can maintain the red-black tree structure under insertion and deletion.

11-1 For each of the four types of linked lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

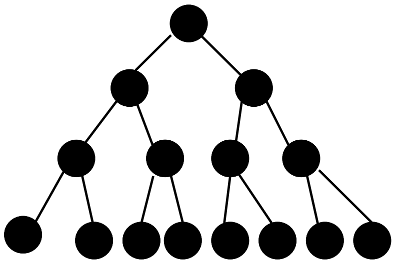
---

	singly unsorted	singly sorted	doubly unsorted	doubly sorted
Search(L, k)	O(N)	O(N)	O(N)	O(N)-
Insert(L, x)	O(1)	O(N)	O(1)	O(N)-
Delete(L, x)	O(N)*	O(N)*	O(1)	O(1)
Successor(L, x)	O(N)	O(1)	O(N)	O(1)
Predecessor(L, x)	O(N)	O(N)	O(N)	O(1)
Minimum(L)	O(N)	O(1)	O(N)	O(1)
Maximum(L)	O(N)	O(1)+	O(N)	O(1)+

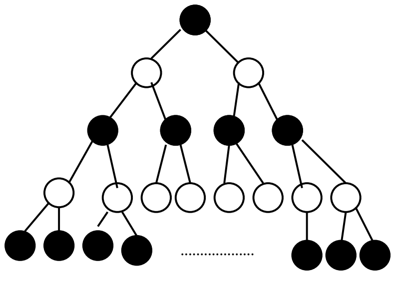
- I need a pointer to the predecessor! (*)
- I need a pointer to the tail! (+)
- Only bottlenecks in otherwise perfect dictionary! (-)

14.1-5 Describe a Red-Black tree with the largest and smallest ratio of red nodes.

To minimize the ratio of red-black nodes, make all black (possible for $n = 2^k - 1$)



To maximize the ratio of red nodes, interleave with red nodes as *real* leaves



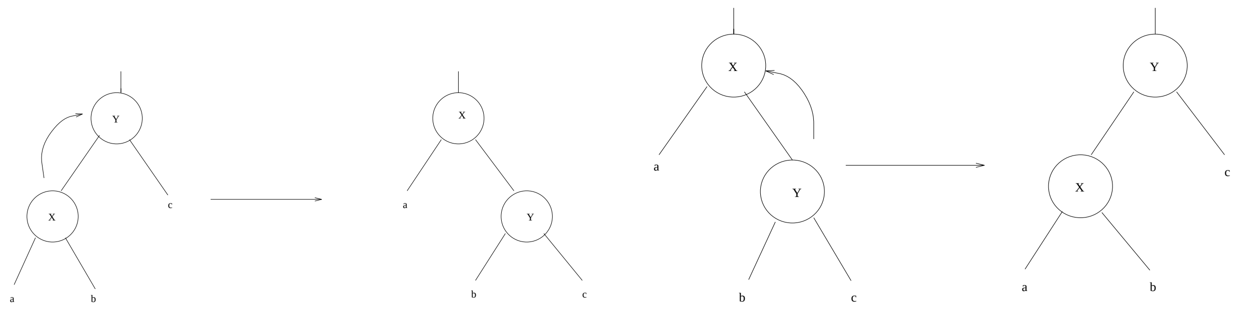
$$\#red/n \leq n/2 + n/8 + n/32 = .656n$$

$$\#black/n \leq n/4 + n/16 + n/64 = .328n$$

$$n - n \sum_{i=1}^{max} (1/4)^i = 2n/3$$

Rotations

The basic restructuring step for binary search trees are left and right rotation:



1. Rotation is a local operation changing $O(1)$ pointers.
2. An in-order search tree before a rotation ***stays*** an in-order search tree.
3. In a rotation, one subtree gets one level closer to the root and one subtree one level further from the root.

LEFT-ROTATE(T,x)

$y \leftarrow right[x]$ (* Set y*)

$right[x] \leftarrow left[y]$ (* Turn y's left into x's right*)

if $left[y] = \mathrm{NIL}$

  then $p[left[y]] \leftarrow x$

$p[y] \leftarrow p[x]$ (* Link x's parent to y *)

if $p[x] = \mathrm{NIL}$

  then $root[T] \leftarrow y$

  else if $x = left[p[x]]$

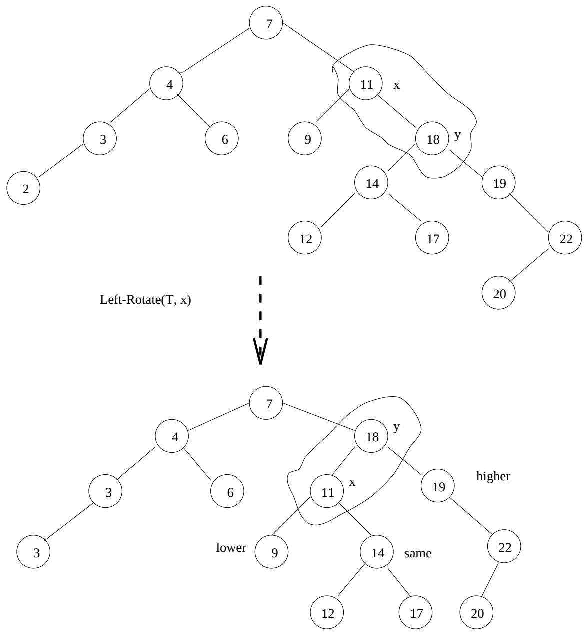
    then $left[p[x]] \leftarrow y$

    else $right[p[x]] \leftarrow y$

$left[y] \leftarrow x$

$p[x] \leftarrow y$

Note the in-order property is preserved.



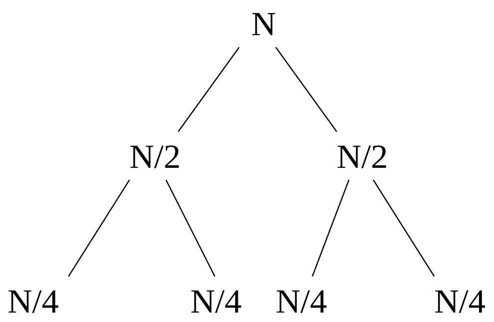
14.2-5 Show that any n-node tree can be transformed to any other using $O(n)$ rotations (hint: convert to a right going chain).

I will start by showing weaker bounds - that $O(n^2)$ and $O(n \log n)$ rotations suffice - because that is how I proceeded when I first saw the problem.

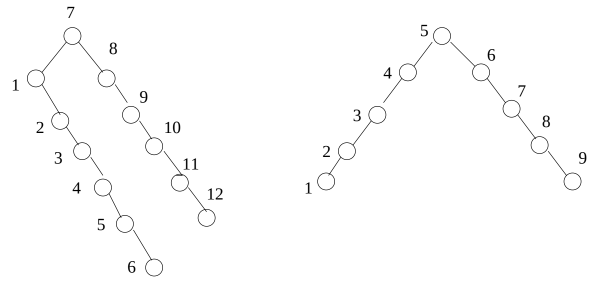
First, observe that creating a right-going, for t_2 path from t_1 ¡ and reversing the same construction gives a path from t_1 to t_2.

Note that it will take at most n rotations to make the lowest valued key the root. Once it is root, all keys are to the right of it, so no more rotations need go through it to create a right-going chain. Repeating with the second lowest key, third, etc. gives that $O(n^2)$ rotations suffice.

Now that if we try to create a completely balanced tree instead. To get the $n/2$ key to the root takes at most n rotations. Now each subtree has half the nodes and we can recur...

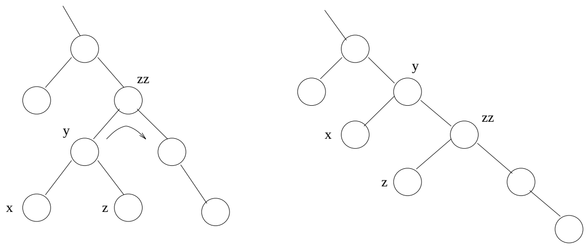


To get a linear algorithm, we must beware of trees like:



The correct answer is that $n-1$ rotations suffice to get to a rightmost chain.

By picking the lowest node on the rightmost chain which has a left ancestor, we can add one node *per* rotation to the right most chain!



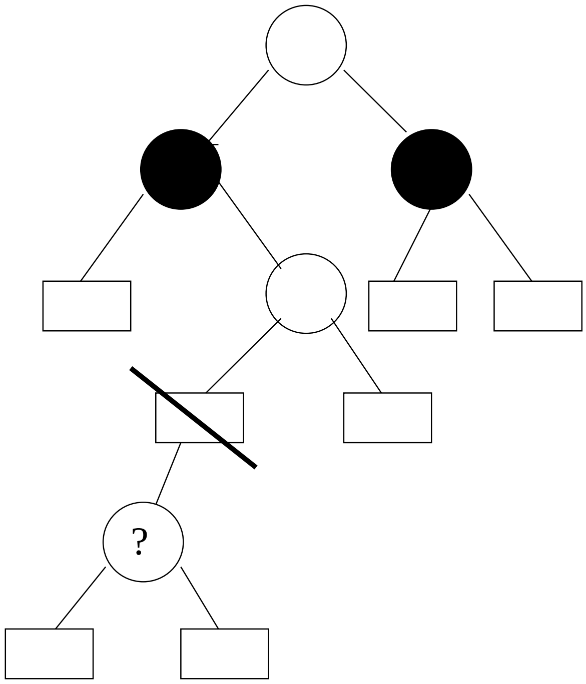
Initially, the rightmost chain contained at least 1 node, so after *1* rotations it contains all n. Slick!

Red-Black Insertion

Since red-black trees have $\Theta(\lg n)$ height, if we can preserve all properties of such trees under insertion/deletion, we have a balanced tree!

Suppose we just did a regular insertion. Under what conditions does it stay a red-black tree?

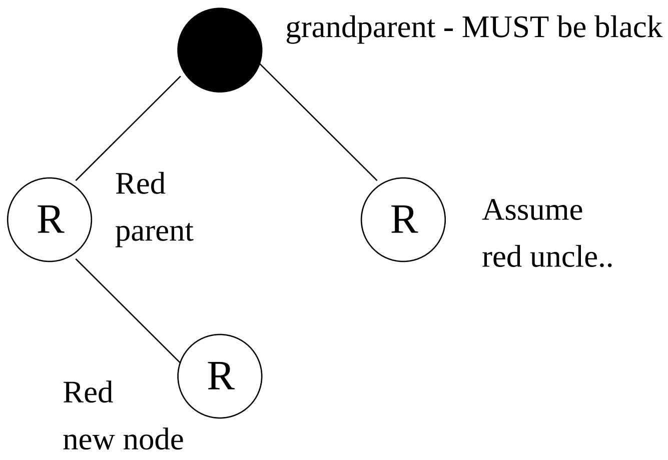
Since every insertion take places at a leaf, we will change a black NIL pointer to a node with two black NIL pointers.



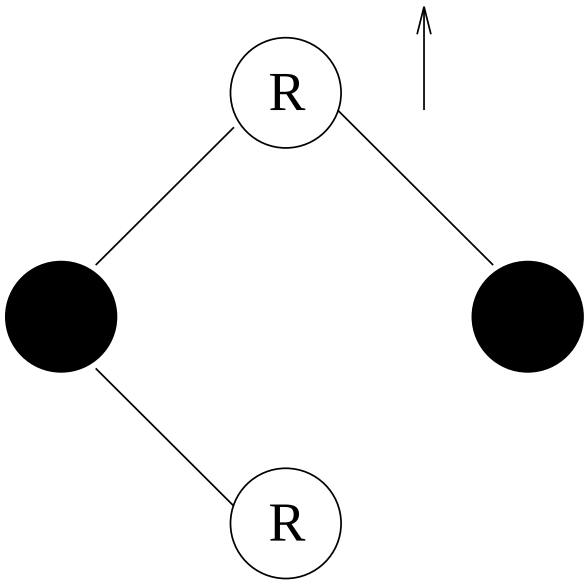
To preserve the black height of the tree, the new node must be *red*. If its *new* parent is black, we can stop, otherwise we must restructure!

How can we fix two reds in a row?

It depends upon our uncle's color:



If our uncle is red, reversing our relatives' color either solves the problem or pushes it higher!



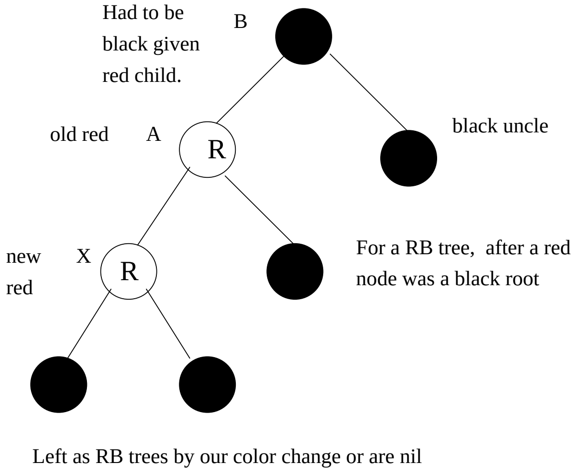
Note that after the recoloring:

1. The black height is unchanged.
2. The shape of the tree is unchanged.
3. We are done if our great-grandparent is black.

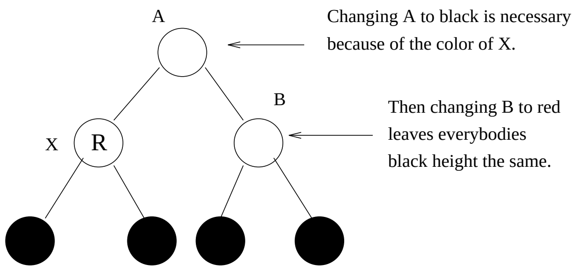
If we get all the way to the root, recall we can always color a red-black tree's root black. We always will, so initially it *was* black, and so this process terminates.

The Case of the Black Uncle

If our uncle was black, observe that all the nodes around us have to be black:



Solution - rotate right about B:



Since the root of the subtree is now black with the same black-height as before, we have restored the colors and can stop!

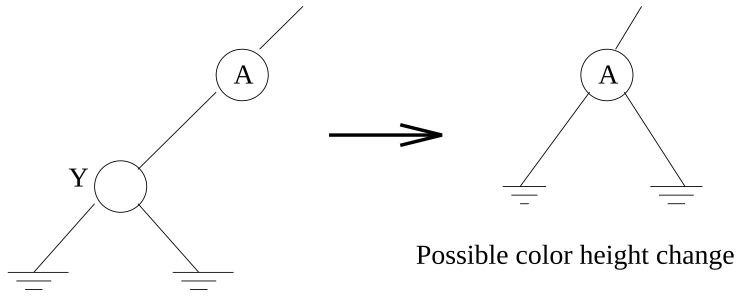
A double rotation can be required to set things up depending upon the left-right turn sequence, but the principle is the same.

DOUBLE ROTATION ILLUSTRATION

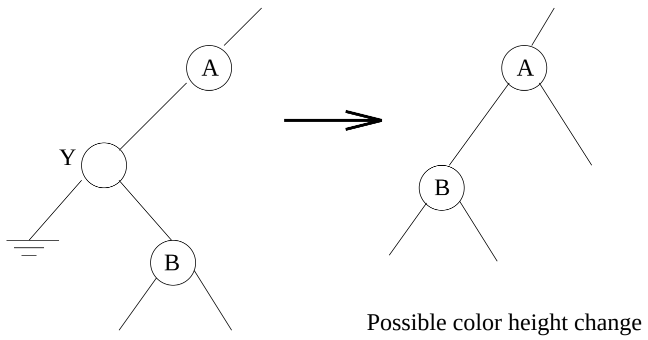
Pseudocode and Figures

Deletion from Red-Black Trees

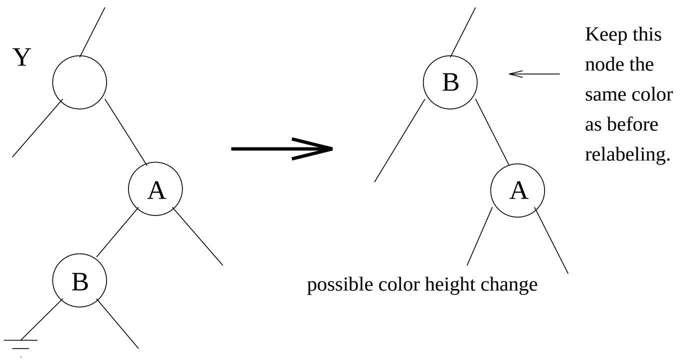
Recall the three cases for deletion from a binary tree:
Case (a) The node to be deleted was a leaf;



Case (b) The node to be deleted had one child;



Case (c) relabel to node as its successor and delete the successor.



Deletion Color Cases

Suppose the node we remove was *red*, do we still have a red-black tree?

Yes! No two reds will be together, and the black height for each leaf stays the same.

However, if the dead node y was black, we must give each of its decendants another black ancestor. If an appropriate node is red, we can simply color it black otherwise we must restructure.

Case (a) black NIL becomes "double black";

Case (b) red β becomes black and black β becomes "double black";

Case (c) red β becomes black and black β becomes "double black".

Our goal will be to recolor and restructure the tree so as to get rid of the "double black" node.

In setting up any case analysis, we must be sure that:

1. All possible cases are covered.

2. No case is covered twice.

In the case analysis for red-black trees, the breakdown is:

Case 1: The double black node x has a red brother.

Case 2: x has a black brother and two black nephews.

Case 3: x has a black brother, and its left nephew is red and its right nephew is black.

Case 4: x has a black brother, and its right nephew is red (left nephew can be any color).

Conclusion

Red-Black trees let us implement all dictionary operations in $O(\log n)$. Further, in no case are more than 3 rotations done to rebalance. Certain very advanced data structures have data stored at nodes which requires a lot of work to adjust after a rotation — red-black trees ensure it won't happen often.

Example: Each node represents the endpoint of a line, and is augmented with a list of segments in its subtree which it intersects.

We will not study such complicated structures, however.

Parallel Bubblesort

In order for me to give back your midterms, please form a line and sort yourselves in alphabetical order, from A to Z.

There is traditionally a strong correlation between the midterm grades and the number of daily problems attempted:

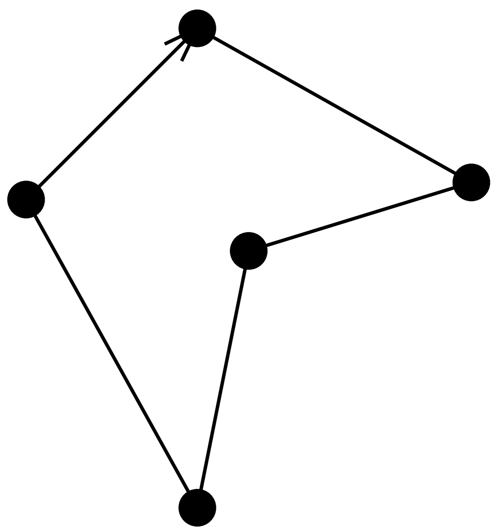
daily: 0, sum: 134, count: 3, avg: 44.67
daily: 1, sum: 0, count: 2, avg: 0.00
daily: 2, sum: 63, count: 1, avg: 63.00
daily: 3, sum: 194, count: 3, avg: 64.67
daily: 4, sum: 335, count: 5, avg: 67.00
daily: 5, sum: 489, count: 8, avg: 61.12
daily: 6, sum: 381, count: 6, avg: 63.50
daily: 7, sum: 432, count: 6, avg: 72.00
daily: 8, sum: 217, count: 3, avg: 72.33
daily: 9, sum: 293, count: 4, avg: 73.25

Combinatorial Search

We have seen how clever algorithms can reduce sorting from $O(n^2)$ to $O(n \log n)$. However, the stakes are even higher for combinatorially explosive problems:

The Traveling Salesman Problem

Given a weighted graph, find the shortest cycle which visits each vertex once.



Applications include minimizing plotter movement, printed-circuit board wiring, transportation problems, etc.

There is no known polynomial time algorithm (ie. $O(n^k)$ for some fixed k) for this problem, so search-based algorithms are the only way to go if you need an optional solution.

But I want to use a Supercomputer

Moving to a faster computer can only buy you a relatively small improvement:

- Hardware clock rates on the fastest computers only improved by a factor of 6 from 1976 to 1989, from 12ns to 2ns.
- Moving to a machine with 100 processors can only give you a factor of 100 speedup, even if your job can be perfectly parallelized (but of course it can't).
- The fast Fourier algorithm (FFT) reduced computation from $O(n^2)$ to $O(n \lg n)$. This is a speedup of 340 times on $n = 4096$ and revolutionized the field of image processing.
- The fast multipole method for n-particle interaction reduced the computation from $O(n^2)$ to $O(n)$. This is a speedup of 4000 times on $n = 4096$.

Can Eight Pieces Cover a Chess Board?

Consider the 8 main pieces in chess (king, queen, two rooks, two bishops, two knights). Can they be positioned on a chessboard so every square is threatened?

			N	B			R
			N				
R							
	B						
		Q			K		

Only 63 square are threatened in this configuration. Since 1849, no one had been able to find an arrangement with bishops on different colors to cover all squares.

Of course, this is not an important problem, but we will use it as an example of how to attack a combinatorial search problem.

How many positions to test?

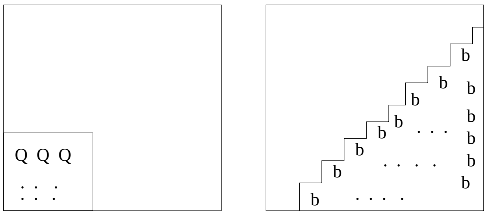
Picking a square for each piece gives us the bound:

$$64!/(64-8)! = 178,462,987,637,760 \approx 10^{15}$$

Anything much larger than 10^8 is unreasonable to search on a modest computer in a modest amount of time.

However, we can exploit symmetry to save work. With reflections along horizontal, vertical, and diagonal axis, the queen can go in only 10 non-equivallent positions.

Even better, we can restrict the white bishop to 16 spots and the queen to 16, while being certain that we get all distinct configurations.



$$16 \times 16 \times 32 \times 64 \times 2080 \times 2080 = 2,268,279,603,200 \approx 10^{12}$$

Backtracking

Backtracking is a systematic way to go through all the possible configurations of a search space.

In the general case, we assume our solution is a vector $v = (a_1, a_2, ..., a_n)$ where each element a_i is selected from a finite ordered set S_i,

We build from a partial solution of length k $v = (a_1, a_2, ..., a_k)$ and try to extend it by adding another element. After extending it, we will test whether what we have so far is still possible as a partial solution.

If it is still a candidate solution, great. If not, we delete a_k and try the next element from S_k:

Compute S_1, the set of candidate first elements of v.
$k = 1$
While $k > 0$ do
  While $S_k \neq \emptyset$ do (*advance*)
    a_k = an element in S_k
    $S_k \leftarrow S_k - a_k$
    if $(a_1, a_2, ..., a_k)$ is solution, print!
    $k = k + 1$
    compute S_k, the candidate kth elements given v.
  $k = k - 1$ (*backtrack*)

Recursive Backtracking

Recursion can be used for elegant and easy implementation of backtracking.

Backtrack(a, k)
if a is a solution, print(a)
else {
    $k = k + 1$
    compute S_k
    while $S_k \neq \emptyset$ do
        a_k = an element in S_k
        $S_k = S_k - a_k$
        Backtrack(a, k)
}

Backtracking can easily be used to iterate through all subsets or permutations of a set.

Backtracking ensures correctness by enumerating all possibilities.

For backtracking to be efficient, we must prune the search space.

Constructing all Subsets

How many subsets are there of an n-element set?

To construct all 2^n subsets, set up an array/vector of n cells, where the value of a_i is either true or false, signifying whether the ith item is or is not in the subset.

To use the notation of the general backtrack algorithm, $S_k = (true, false)$, and v is a solution whenever $k \geq n$.

What order will this generate the subsets of $\{1, 2, 3\}$?

$$(1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow$$

$$(1,2,-)* \rightarrow (1,-) \rightarrow (1,-,3)* \rightarrow$$

$$(1,-,-)* \rightarrow (1,-) \rightarrow (1) \rightarrow$$

$$(-) \rightarrow (-,2) \rightarrow (-,2,3)* \rightarrow$$

$$(-,2,-)* \rightarrow (-,-) \rightarrow (-,-,3)* \rightarrow$$

$$(-,-,-)* \rightarrow (-,-) \rightarrow (-) \rightarrow ()$$

Constructing all Permutations

How many permutations are there of an n-element set?

To construct all $n!$ permutations, set up an array/vector of n cells, where the value of a_i is an integer from 1 to n which has not appeared thus far in the vector, corresponding to the ith element of the permutation.

To use the notation of the general backtrack algorithm, $S_k = (1, \ldots, n) - v$, and v is a solution whenever $k \geq n$.

$$
\begin{array}{rcl}
(1) & \rightarrow & (1,2) \rightarrow (1,2,3)* \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow \\
(1,3,2)* & \rightarrow & (1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow \\
(2,1,3)* & \rightarrow & (2,1) \rightarrow (2) \rightarrow (2,3) \rightarrow (2,3,1)* \rightarrow (2,3) \rightarrow () \\
(2) & \rightarrow & () \rightarrow (3) \rightarrow (3,1)(3,1,2)* \rightarrow (3,1) \rightarrow (3) \rightarrow (3,2) \\
(3,2,1)* & \rightarrow & (3,2) \rightarrow (3) \rightarrow ()
\end{array}
$$

The n-Queens Problem

The first use of pruning to deal with the combinatorial explosion was by the king who rewarded the fellow who discovered chess!

In the eight Queens, we prune whenever one queen threatens another.

Covering the Chess Board

In covering the chess board, we prune whenever we find there is a square which we *cannot* cover given the initial configuration!

Specifically, each piece can threaten a certain maximum number of squares (queen 27, king 8, rook 14, etc.) Whenever the number of unthreated squares exceeds the sum of the maximum number of coverage remaining in unplaced squares, we can *prune*.

As implemented by a graduate student project, this backtrack search eliminates 95% of the search space, when the pieces are ordered by decreasing mobility.

With precomputing the list of possible moves, this program could search 1,000 positions per second. But this is too slow!

$$10^{12}/10^{3} = 10^{9} \text{ } seconds \; > 1000 \text{ d}ays$$

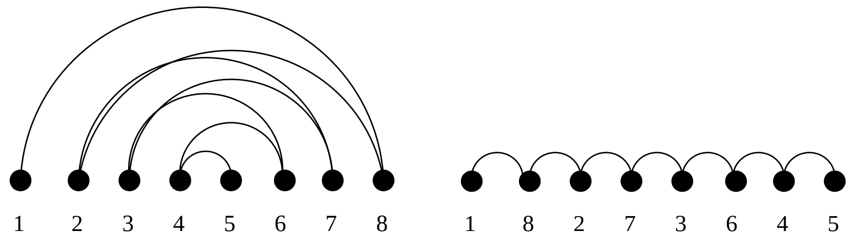
Although we might further speed the program by an order of magnitude, we need to prune more nodes!

By using a more clever algorithm, we eventually were able to prove no solution existed, in less than one day's worth of computing.

You too can fight the combinatorial explosion!

The Backtracking Contest: Bandwidth

The *bandwidth problem* takes as input a graph G, with n vertices and m edges (ie. pairs of vertices). The goal is to find a permutation of the vertices on the line which minimizes the maximum length of any edge.



The bandwidth problem has a variety of applications, including circuit layout, linear algebra, and optimizing memory usage in hypertext documents.

The problem is NP-complete, meaning that it is *exceedingly* unlikely that you will be able to find an algorithm with polynomial worst-case running time. It remains NP-complete even for restricted classes of trees.

Since the goal of the problem is to find a permutation, a backtracking program which iterates through all the $n!$ possible permutations and computes the length of the longest edge for each gives an easy $O(n! \cdot m)$ algorithm. But the goal of this assignment is to find as practically good an algorithm as possible.

Rules of the Game

1. Everyone must do this assignment separately. Just this once, you are not allowed to work with your partner. The idea is to think about the problem from scratch.

2. If you do not completely understand what the bandwidth of a graph is, you don't have the *slightest* chance of producing a working program. *Don't be afraid to ask for a clarification or explanation!!!!!*

3. There will be a variety of different data files of different sizes. Test on the smaller files first. Do not be afraid to create your own test files to help debug your program.

4. The data files are available via the course WWW page.

5. You will be graded on how fast and clever your program is, not on style. No credit will be given for incorrect programs.

6. The programs are to run on the whatever computer you have access to, although it must be vanilla enough that I can run the program on something I have access to.

7. You are to turn in a listing of your program, along with a brief description of your algorithm and any interesting optimizations, sample runs, and the time it takes on sample data files. Report the largest test file your program could handle in one minute or less of wall clock time.

8. The top five self-reported times / largest sizes will be collected and tested by me to determine the winner.

Producing Efficient Programs

1. **Don't optimize prematurely:** Worrying about recursion vs. iteration is counter-productive until you have worked out the best way to prune the tree. That is where the money is.

2. **Choose your data structures for a reason:** What operations will you be doing? Is case of insertion/deletion more crucial than fast retrieval?

   When in doubt, keep it simple, stupid (KISS).

3. **Let the profiler determine where to do final tuning:** Your program is probably spending time where you don't expect.

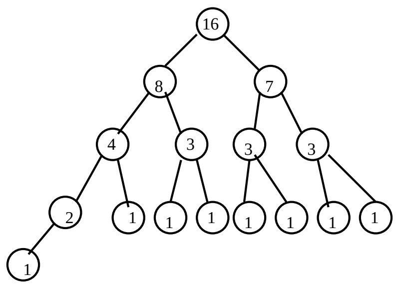
15.1-5 Given an element x in an n-node order-statistic binary tree and a natural number i, how can the ith successor of x be determined in $O(\lg n)$ time.

This problem can be solved if our data structure supports two operations:

- Rank(x) – what is the position of x in the total order of keys?
- Get(i) – what is the key in the ith position of the total order of keys?

What we are interested in is $Get(Rank(x) + i)$.

In an order statistic tree, each node x is labeled with the number of nodes contained in the subtree rooted in x.



Implementing both operations involves keeping track of how many nodes lie to the left of our path.

Optimization Problems

In the algorithms we have studied so far, correctness tended to be easier than efficiency. In optimization problems, we are interested in finding a *thing* which maximizes or minimizes some function.

In designing algorithms for optimization problem - we must prove that the algorithm in fact gives the best possible solution.

Greedy algorithms, which makes the best local decision at each step, occasionally produce a global optimum - but you need a proof!

Dynamic Programming

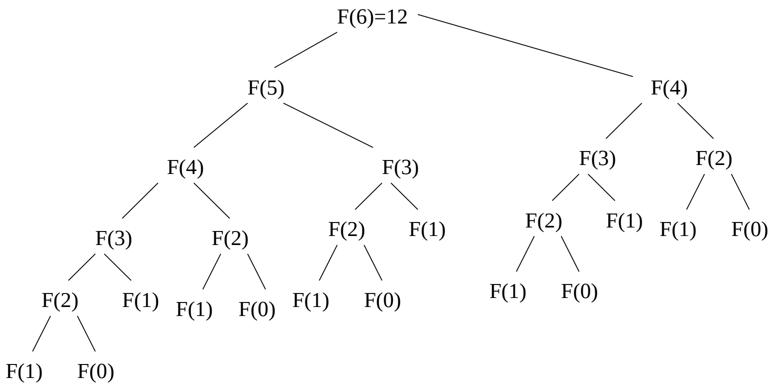
Dynamic Programming is a technique for computing recurrence relations efficiently by sorting partial results.

Computing Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0, F_1 = 1$$

Implementing it as a recursive procedure is easy but slow!
We keep calculating the same value over and over!



How slow is slow?

$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$

Thus $F_n \approx 1.6^n$, and since our recursion tree has 0 and 1 as leaves, means we have $\approx 1.6^n$ calls!

What about Dynamic Programming?

We can calculate F_n in linear time by storing small values:

$F_0 = 0$
$F_1 = 1$
For $i = 1$ to n
    $F_i = F_{i-1} + F_{i-2}$

Moral: we traded space for time.

Dynamic programming is a technique for efficiently computing recurrences by storing partial results.

Once you understand dynamic programming, it is usually easier to reinvent certain algorithms than try to look them up!

Dynamic programming is best understood by looking at a bunch of different examples.

I have found dynamic programming to be one of the most useful algorithmic techniques in practice:

- Morphing in Computer Graphics
- Data Compression for High Density Bar Codes
- Utilizing Grammatical Constraints for Telephone Keypads

Multiplying a Sequence of Matrices

Suppose we want to multiply a long sequence of matrices $A \times B \times C \times D \ldots$.

Multiplying an $X \times Y$ matrix by a $Y \times Z$ matrix (using the common algorithm) takes $X \times Y \times Z$ multiplications.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \qquad \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

We would like to avoid big intermediate matrices, and since matrix multiplication is *associative*, we can parenthesise however we want.

Matrix multiplication is *not communitive*, so we cannot permute the order of the matrices without changing the result.

Example

Consider $A \times B \times C \times D$, where A is 30×1, B is 1×40, C is 40×10, and D is 10×25.

There are three possible parenthesizations:

$$((AB)C)D = 30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20,700$$

$$(AB)(CD) = 30 \times 1 \times 40 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41,200$$

$$A((BC)D) = 1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1400$$

The order makes a big difference in real computation. How do we find the best order?

Let $M(i,j)$ be the *minimum* number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$.

The key observations are

- The outermost parentheses partition the chain of matricies (i,j) at some k.
- The optimal parenthesization order has optimal ordering on either side of k.

A recurrence for this is:

$$\begin{aligned} M(i,j) &= Min_{i \le k \le j-1}[M(i,k) + M(k+1,j) + d_{i-1}d_k d_j] \\ M(i,i) &= 0 \end{aligned}$$

If there are n matrices, there are $n+1$ dimensions.

A direct recursive implementation of this will be exponential, since there is a lot of duplicated work as in the Fibonacci recurrence.

Divide-and-conquer is seems efficient because there is no overlap, but ...

There are only $\binom{n}{2}$ substrings between 1 and n. Thus it requires only $\Theta(n^2)$ space to store the optimal cost for each of them.

We can represent all the possibilities in a triangle matrix. We can also store the value of k in another triangle matrix to reconstruct to order of the optimal parenthesisation.

The diagonal moves up to the right as the computation progresses. On each element of the kth diagonal $|j-i| = k$.

For the previous example:

Procedure MatrixOrder
for $i = 1$ to n do $M[i,j] = 0$
for $diagonal = 1$ to $n-1$
    for $i = 1$ to $n - diagonal$ do
        $j = i + diagonal$
        $M[i,j] = \min_{i=k}^{j-1}[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$
        faster$(i,j) = k$
return $[m(1,n)]$

Procedure ShowOrder(i,j)
if $(i = j)$ write (A_i)
else
    $k =$factor(i,j)
    write "("
    ShowOrder(i,k)
    write "*"
    ShowOrder $(k+1,j)$
    write ")"

A dynamic programming solution has three components:

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence so you always have what you need.

Approximate String Matching

A common task in text editing is string matching - finding all occurrences of a word in a text.

Unfortunately, many words are mispelled. How can we search for the string closest to the pattern?

Let p be a pattern string and T a text string over the same alphabet.

A k-approximate match between P and T is a substring of T with at most k differences.

Differences may be:

1. the corresponding characters may differ: KAT$\rightarrow$ CAT
2. P is missing a character from T: CAAT$\rightarrow$ CAT
3. T is missing a character from P: CT$\rightarrow$ CAT

Approximate Matching is important in genetics as well as spell checking.

A 3-Approximate Match

A match with one of each of three edit operations is:

$$P = \text{unescessaraly}$$
$$T = \text{unnecessarily}$$

Finding such a matching seems like a hard problem because we must figure out where you add *blanks*, but we can solve it with dynamic programming.

$D[i,j]$ = the minimum number of differences between $P_1, P_2, \ldots, P_i$ and the segment of T ending at j.

$D[i,j]$ is the *minimum* of the three possible ways to extend smaller strings:

1. If $P_i = t_i$ then $D[i-1,j-1]$ else $D[i-1,j-1]+1$ (corresponding characters do or do not match)

2. $D[i-1,j]+1$ (extra character in text – we do not advance the pattern pointer).

3. $D[i,j-1]+1$ (character in pattern which is not in text).

Once you accept the recurrence it is easy.

To fill each cell, we need only consider three other cells, not $O(n)$ as in other examples. This means we need only store two rows of the table. The total time is $O(mn)$.

Boundary conditions for string matching

What should the value of $D[0,i]$ be, corresponding to the cost of matching the first i characters of the text with none of the pattern?

It depends. Are we doing string matching in the text or substring matching?

- If you want to match all of the pattern against all of the text, this meant that would have to delete the first i characters of the pattern, so $D[0,i]=i$ to pay the cost of the deletions.

- if we want to find the place in the text where the pattern occurs? We do not want to pay more of a cost if the pattern occurs far into the text than near the front, so it is important that starting cost be equal for all positions. In this case, $D[0,i]=0$, since we pay no cost for deleting the first i characters of the text.

In both cases, $D[i,0]=i$, since we cannot excuse deleting the first i characters of the pattern without cost.

What do we return?

If we want the *cost* of comparing all of the pattern against all of the text, such as comparing the spelling of two words, all we are interested in is $D[n, m]$.

But what if we want the cheapest match between the pattern anywhere in the text? Assuming the initialization for substring matching, we seek the cheapest matching of the full pattern ending anywhere in the text. This means the cost equals $\min_{1 \leq i \leq m} D[n, i]$.

This only gives the cost of the optimal matching. The actual alignment – what got matched, substituted, and deleted – can be reconstructed from the pattern/text and table without an auxiliary storage, once we have identified the cell with the lowest cost.

How much space do we need?

Do we need to keep all $O(mn)$ cells, since if we evaluate the recurrence filling in the columns of the matrix from left to right, we will never need more than two columns of cells to do what we need. Thus $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity at all.

Unfortunately, because we won't have the full matrix we cannot reconstruct the alignment, as above.

Saving space in dynamic programming is very important. Since memory on any computer is limited, $O(nm)$ space is more of a bottleneck than $O(nm)$ time.

Fortunately, there is a clever divide-and-conquer algorithm which computes the actual alignment in $O(nm)$ time and $O(m)$ space.

16.3-5 Give an $O(n^2)$ algorithm to find the longest montonically increasing sequence in a sequence of n numbers.

Build an example first: $(5, 2, 8, 7, 1, 6, 4)$

Ask yourself what would you like to know about the first $n-1$ elements to tell you the answer for the entire sequence?

1. The length of the longest sequence in $s_1, s_2, \ldots, s_{n-1}$. (seems obvious)
2. The length of the longest sequence s_n will extend! (not as obvious - this is the idea!)

Let s_i be the length of the longest sequence ending with the ith character:

sequence	5	2	8	7	3	1	6	4
s_i	1	1	2	2	2	1	3	3

How do we compute si?

$s_i = \max_{0<j<i, seq[j]<seq[i]} s_j + 1$

$s_0 = 0$

To find the longest sequence - we know it ends somewhere, so Length $= \max_{i=1}^{n} s_i$

The Principle of Optimality

To use dynamic programming, the problem must observe the *principle of optimality*, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision.

Combinatorial problems may have this property but may use too much memory/time to be efficient.

Example: The Traveling Salesman Problem

Let $T(i; j_1, j_2, \ldots, j_k)$ be the cost of the optimal tour for i to 1 that goes thru each of the other cities once

$$T(i; i_1, j_2, \ldots, j_i) = Min_{1 \leq m \leq k} C[i, j_m] + T(j_m; j_1, j_2, \ldots, j_k)$$

$$T(i, j) = C(i, j) + C(j, 1)$$

Here there can be any subset of $j_1, j_2, \ldots, j_k$ instead of any subinterval - hence exponential.

Still, with other ideas (some type of pruning or best-first search) it can be effective for combinatorial search.

When can you use Dynamic Programming?

Dynamic programming computes recurrences efficiently by storing partial results. Thus dynamic programming can only be efficient when there are not too many partial results to compute!

There are $n!$ permutations of an n-element set – we cannot use dynamic programming to store the best solution for each subpermutation. There are 2^n subsets of an n-element set – we cannot use dynamic programming to store the best solution for each.

However, there are only $n(n-1)/2$ continguous substrings of a string, each described by a starting and ending point, so we can use it for string problems.

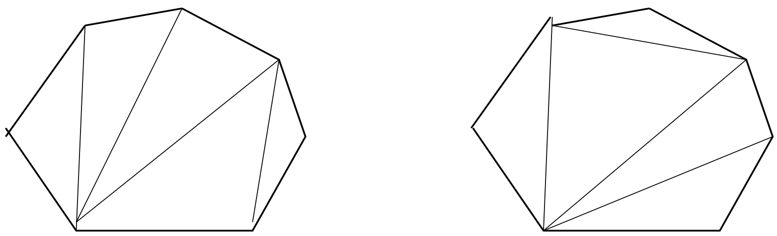
There are only $n(n-1)/2$ possible subtrees of a binary search tree, each described by a maximum and minimum key, so we can use it for optimizing binary search trees.

Dynamic programming works best on objects which are linearly ordered and cannot be rearranged – characters in a string, matrices in a chain, points around the boundary of a polygon, the left-to-right order of leaves in a search tree.

Whenever your objects are ordered in a left-to-right way, you should smell dynamic programming!

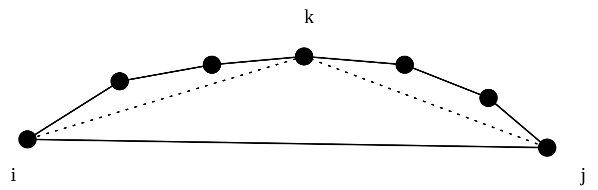
Minimum Length Triangulation

A triangulation of a polygon is a set of non-intersecting diagonals which partiions the polygon into diagonals.



The length of a triangulation is the sum of the diagonal lengths.

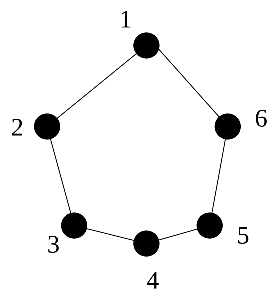
We seek to find the minimum length triangulation. For a convex polygon, or part thereof:



Once we identify the correct connecting vertex, the polygon is partitioned into two smaller pieces, both of which must be triangulated optimally!

$$\begin{aligned} t[i, i+1] &= 0 \\ t[i, j] &= \min_{k=i}^{j} t[i, k] + t[k, j] + |ik| + |kj| \end{aligned}$$

Evaluation proceeds as in the matrix multiplication example - $\binom{n}{2}$ values of t, each of which takes $O(j - i)$ time if we evaluate the sections in order of increasing size.



J-i = 2

13, 24, 35, 46, 51, 62

J-i = 3

14, 25, 36, 41, 52, 63

J-i = 4

15, 26, 31, 42, 53, 64

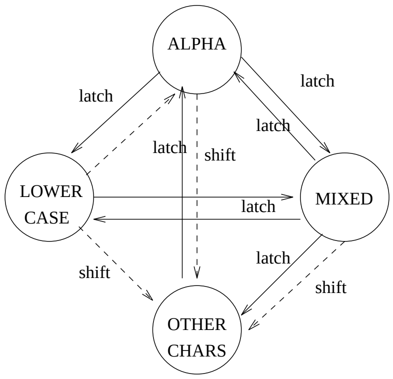
Finish with 16

What if there are points in the interior of the polygon?

Dynamic Programming and High Density Bar Codes

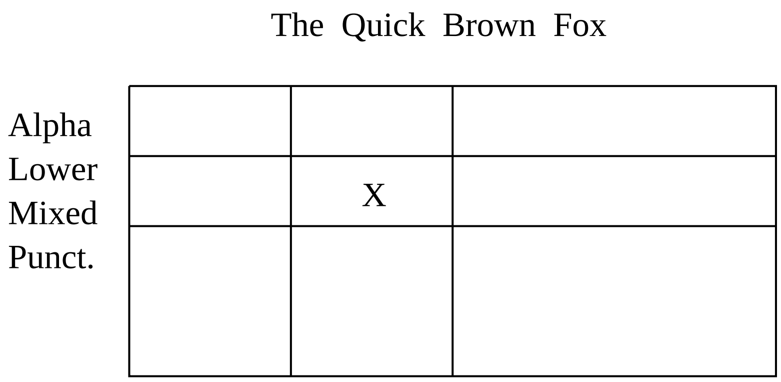
Symbol Technology has developed a new design for bar codes, PDF-417 that has a capacity of several hundred bytes. What is the best way to encode text for this design?

They developed a complicated mode-switching data compression scheme.



Latch commands permanently put you in a different mode. Shift commands temporarily put you in a different mode.

Originally, Symbol used a greedy algorithm to encode a string, making local decisions only. We realized that for any prefix, you want an optimal encoding which might leave you in every possible mode.



$M[i, j] = \min(M[i-1, k] +$ the cost of encoding the ith character and ending up in node $j)$.

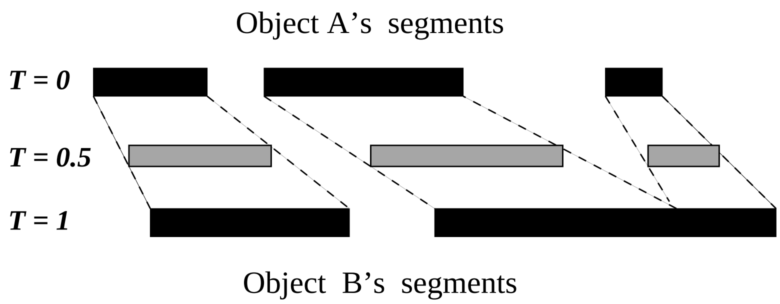
Our simple dynamic programming algorithm improved to capacity of PDF-417 by an average of 8%!

Dynamic Programming and Morphing

Morphing is the problem of creating a smooth series of intermediate images given a starting and ending image.

The key problem is establishing a correspondence between features in the two images. You want to morph an eye to an eye, not an ear to an ear.

We can do this matching on a line-by-line basis:



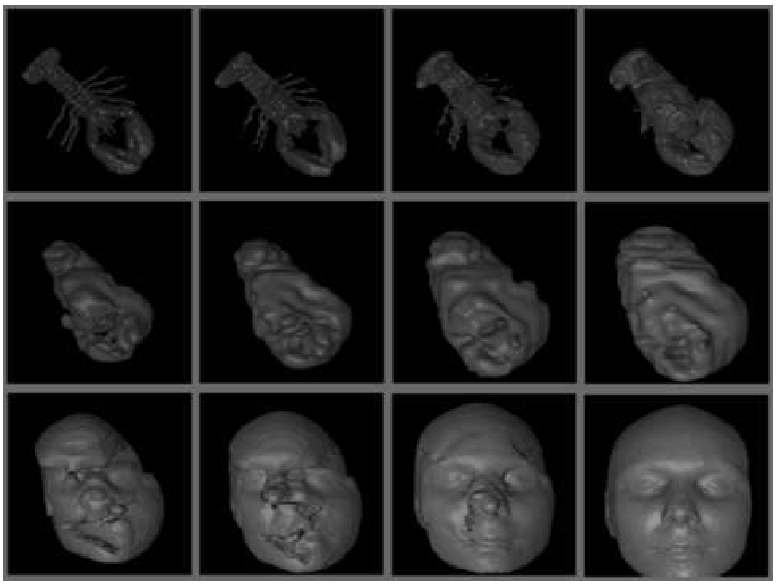
This should sound like string matching, but with a different set of operations:

- *Full run match:* We may match run i on top to run j on bottom for a cost which is a function of the difference in the lengths of the two runs and their positions.

- *Merging runs:* We may match a string of consecutive runs on top to a run on bottom. The cost will be a function of the number of runs, their relative positions, and lengths.

- *Splitting runs:* We may match a big run on top to a string of consecutive runs on the bottom. This is just the converse of the merge. Again, the cost will be a function of the number of runs, their relative positions, and lengths.

This algorithm was incorporated into a morphing system, with the following results:



Problem Solving Techniques

Most important: make sure you understand exactly what the question is asking – if not, you have no hope of answer it!!

Never be afraid to ask for another explanation of a problem until it is clear.

Play around with the problem by constructing examples to get insight into it.

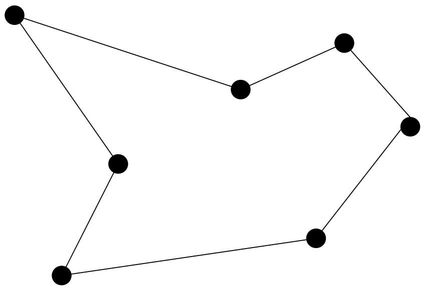
Ask yourself questions. Does the first idea which comes into my head work? If not, why not?

Am I using all information that I am given about the problem?

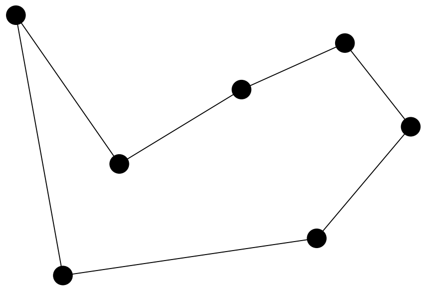
Read Polya's book *How to Solve it.*

16-1: The Euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane.

Bentley suggested simplifying the problem by restricting attention to bitonic tours*, that is tours which start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right back to the starting point.*



non-bitonic

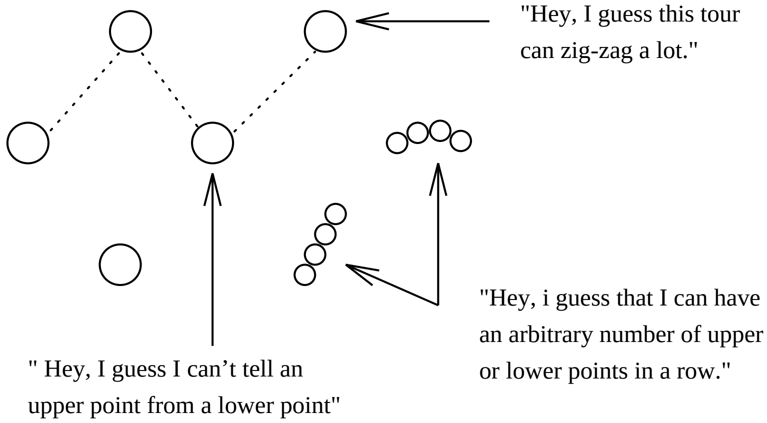


bitonic

Describe an $O(n^2)$ algorithm for finding the optimal bitonic tour. You may assume that no two points have the same x-coordinate. (Hint: scan left to right, maintaining optimal possibilities for the two parts of the tour.)

Make sure you understand what a bitonic tour is, or else it is hopeless.

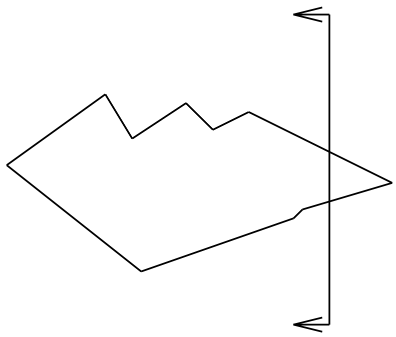
First of all, play with the problem. Why isn't it trivial?



Am I using all the information?

Why will they let us assume that no two x-coordinates are the same? What does the hint mean? What happens if I scan from left to right?

If we scan from left to right, we get an open tour which uses all points to the left of our scan line.



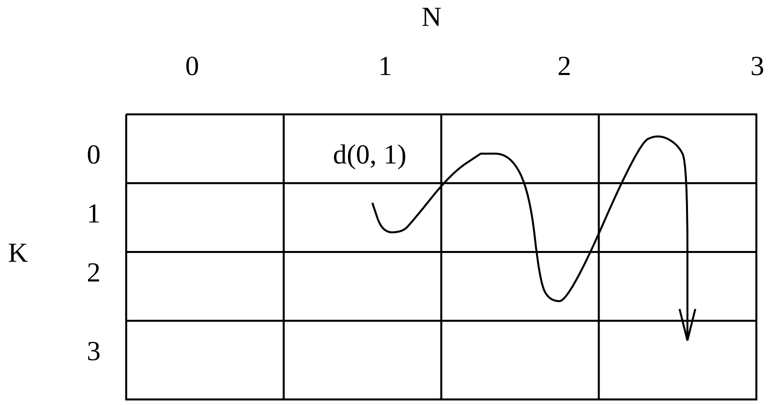
In the optimal tour, the kth point is connected to exactly one point to the left of k. $(k \neq n)$ Once I decide which point that is, say x. I need the optimal partial tour where the two endpoints are x and $k-1$, because if it isn't optimal I could come up with a better one. Hey, I have got a recurrence! And look, the two parameters which describe my optimal tour are the two endpoints.

Let $c[k,n]$ be the optimal cost partial tour where the two endpoints are $k < n$.

$c[k,n] \leq c[k,n-1] + d[n,n-1]$ (when $k < n-1$)

$c[n-1,n] \leq c[k,n-1] + d[k,n]$

$c[0,1] = d[0,1]$



Filling the entities in from N=1 to N', k=1 to N, means we always have what we need waiting for us.

$c[n-1, n]$ takes $O(n)$ to update, $c[k,n]$ $k < n-1$ takes $O(1)$ to update. Total time is $O(n^2)$.

But this doesn't quite give the tour, but just an open tour. We simply must figure where the last edge to n must go.

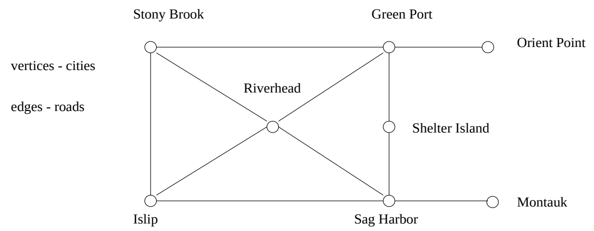
$$Tourcost = \min_{k=1}^{n} C[k,n] + d_{kn}$$

Graphs

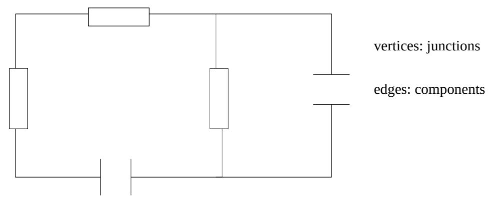
A graph G consists of a set of *vertices* V together with a set E of vertex pairs or *edges*.

Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially *any* relationship.

Example: A network of roads, with cities as vertices and roads between cities as edges.



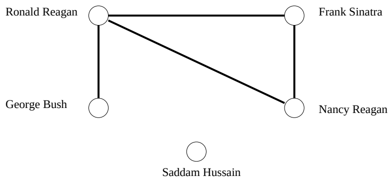
Example: An electronic circuit, with junctions as vertices as components as edges.



To understand many problems, we must think of them in terms of graphs!

The Friendship Graph

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



This graph is well-defined on any set of people: SUNY SB, New York, or the world.

What questions might we ask about the friendship graph?

- **If I am your friend, does that mean you are my friend?**

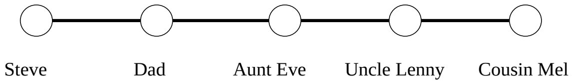
  A graph is *undirected* if (x, y) implies (y, x). Otherwise the graph is directed. The “heard-of” graph is directed since countless famous people have never heard of me! The “had-sex-with” graph is presumably undirected, since it requires a partner.

- **Am I my own friend?**

  An edge of the form (x, x) is said to be a *loop*. If x is y’s friend several times over, that could be modeled using *multiedges*, multiple edges between the same pair of vertices. A graph is said to be *simple* if it contains no loops and multiple edges.

- **Am I linked by some chain of friends to the President?**

  A *path* is a sequence of edges connecting two vertices. Since *Mel Brooks* is my father’s-sister’s-husband’s cousin, there is a path between me and him!



- **How close is my link to the President?**

  If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny. Thus we are often interested in the *shortest path* between two nodes.

- **Is there a path of friends between any two people?**

  A graph is *connected* if there is a path between any two vertices. A directed graph is *strongly connected* if there is a directed path between any two vertices.

- **Who has the most friends?**

  The *degree* of a vertex is the number of edges adjacent to it.

- **What is the largest clique?**

  A social clique is a group of mutual friends who all hang around together. A graph theoretic *clique* is a complete subgraph, where each vertex pair has an edge between them. Cliques are the densest possible subgraphs. Within the friendship graph, we would expect that large cliques correspond to workplaces, neighborhoods, religious organizations, schools, and the like.

- **How long will it take for my gossip to get back to me?**

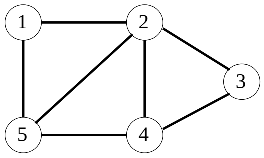
  A *cycle* is a path where the last vertex is adjacent to the first. A cycle in which no vertex repeats (such as 1-2-3-1 verus 1-2-3-2-1) is said to be *simple*. The shortest cycle in the graph defines its *girth*, while a simple cycle which passes through each vertex is said to be a *Hamiltonian cycle.*

Data Structures for Graphs

There are two main data structures used to represent graphs.

Adjacency Matrices

An *adjacency matrix* is an $n \times n$ matrix, where $M[i, j] = 0$ iff there is no edge from vertex i to vertex j



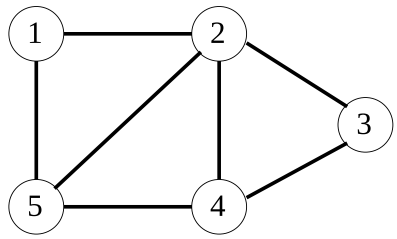
0	1	0	0	1
1	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	1	0	1	0

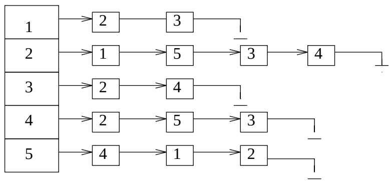
It takes $\Theta(1)$ time to test if (i, j) is in a graph represented by an adjacency matrix.

Can we save space if (1) the graph is undirected? (2) if the graph is sparse?

Adjacency Lists

An *adjacency list* consists of a $N \times 1$ array of pointers, where the ith element points to a linked list of the edges incident on vertex i.





To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex.

Note that d_i can be much less than n when the graph is sparse. If necessary, the two *copies* of each edge can be linked by a pointer to facilitate deletions.

Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *efficiency*, we must make sure we visit each edge at most twice.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Marking Vertices

The idea in graph traversal is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

For each vertex, we can maintain two flags:

- *discovered* - have we ever encountered this vertex before?
- *completely-explored* - have we finished exploring this vertex yet?

We must also maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is considered to be discovered.

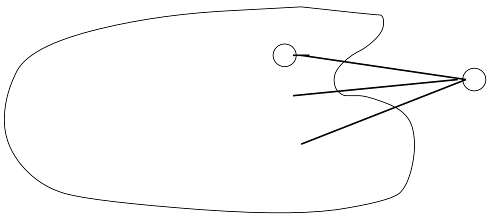
To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do.

Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, ie. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it:



Traversal Orders

The order we explore the vertices depends upon what kind of data structure is used:

- *Queue* – by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a so-called *breadth-first search.*
- *Stack* - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only if we are surrounded by previously discovered vertices. Thus our explorations quickly wander away from our starting point, defining a so-called *depth-first search.*

The three possible colors of each node reflect if it is unvisited (white), visited but unexplored (grey) or completely explored (black).

Breadth-First Search

```
BFS(G,s)
for each vertex u ∈ V[G] − {s} do
        color[u] = white
        d[u] = ∞, ie. the distance from s
        p[u] = NIL, ie. the parent in the BFS tree
color[u] = grey
d[s] = 0
p[s] = NIL
Q = {s}
while Q ≠ ∅ do
        u = head[Q]
        for each v ∈ Adj[u] do
                if color[v] = white then
                        color[v] = gray
                        d[v] = d[u] + 1
                        p[v] = u
                        enqueue[Q,v]
                dequeue[Q]
                color[u] = black
```

Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack.

Discovery and final times are sometimes a convenience to maintain.

DFS(G)
for each vertex $u \in V[G]$ do
    $color[u] = white$
    $parent[u] = nil$
$time = 0$
for each vertex $u \in V[G]$ do
    if $color[u] = white$ then DFS-VISIT[u]

Initialize each vertex in the main routine, then do a search from each connected component. BFS must also start from a vertex in each component to completely visit the graph.

DFS-VISIT[u]
$color[u] = grey$ (*u had been white/undiscovered*)
$discover[u] = time$
$time = time + 1$
for each $v \in Adj[u]$ do
    if $color[v] = white$ then
        $parent[v] = u$
        DFS-VISIT(v)
$color[u] = black$ (*now finished with u*)
$finish[u] = time$
$time = time + 1$

23.1-5 – The square *of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ iff for some $v \in V$, both $(u, v) \in E$ and $(v, w) \in E$; ie. there is a path of exactly two edges.*

Give efficient algorithms for both adjacency lists and matricies.

Given an adjacency matrix, we can check in constant time whether a given edge exists. To discover whether there is an edge $(u, w) \in G^2$, for each possible intermediate vertex v we can check whether (u, v) and (v, w) exist in $O(1)$.

Since there are at most n intermediate vertices to check, and n^2 pairs of vertices to ask about, this takes $O(n^3)$ time.

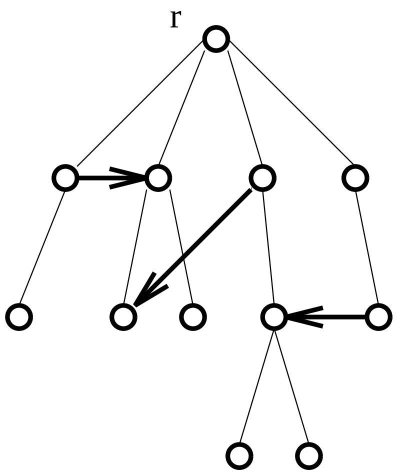
With adjacency lists, we have a list of all the edges in the graph. For a given edge (u, v), we can run through all the edges from v in $O(n)$ time, and fill the results into an adjacency matrix of G^2, which is initially empty.

It takes $O(mn)$ to construct the edges, and $O(n^2)$ to initialize and read the adjacency matrix, a total of $O((n+m)n)$. Since $n \leq m$ unless the graph is disconnected, this is usually simplified to $O(mn)$, and is faster than the previous algorithm on sparse graphs.

Why is it called the square of a graph? Because the square of the adjacency matrix is the adjacency matrix of the square! This provides a theoretically faster algorithm.

BFS Trees

If BFS is performed on a connected, undirected graph, a tree is defined by the edges involved with the discovery of new nodes:



This tree defines a shortest path from the root to every other node in the tree.

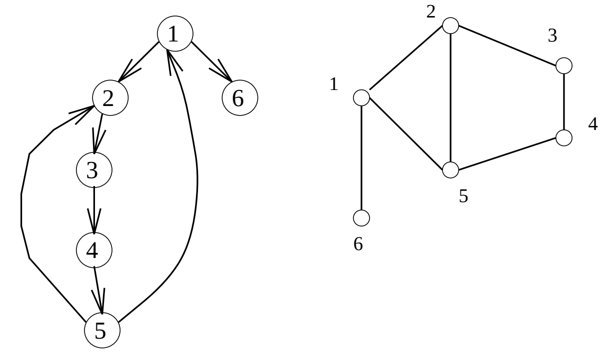
The proof is by induction on the length of the shortest path from the root:

- *Length = 1* First step of BFS explores all neighbors of the root. In an unweighted graph one edge must be the shortest path to any node.
- *Length = s* Assume the BFS tree has the shortest paths up to length $s-1$. Any node at a distance of s will first be discovered by expanding a distance $s-1$ node.

The *key* idea about DFS

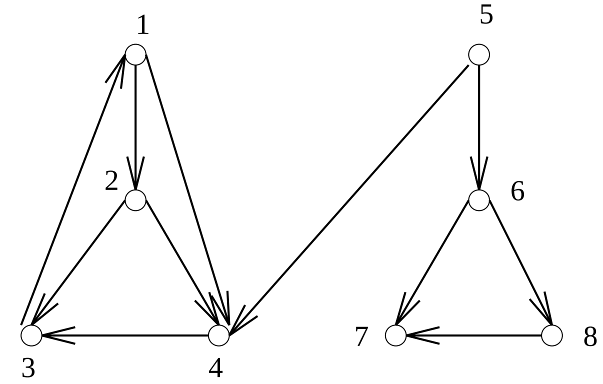
A depth-first search of a graph organizes the edges of the graph in a precise way.

In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:



In a DFS of a directed graph, every edge is either a tree edge or a black edge.

In a DFS of a directed graph, no cross edge goes to a higher numbered or rightward vertex. Thus, no edge from 4 to 5 is possible:

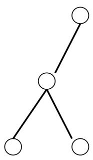


Edge Classification for DFS

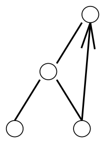
What about the other edges in the graph? Where can they go on a search?

Every edge is either:

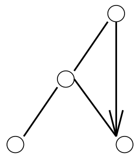
1. A Tree Edge



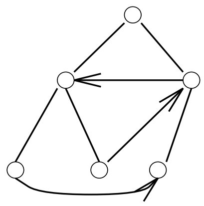
2. A Back Edge
   to an ancestor



3. A Forward Edge
   to a decendant



4. A Cross Edge
   to a different node



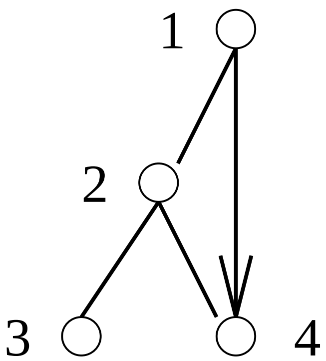
On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

DFS Trees

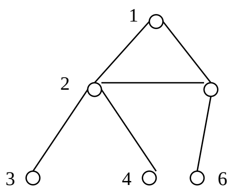
The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

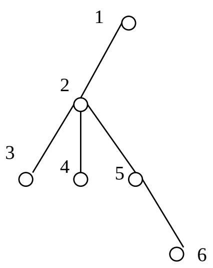
Why? Suppose we have a forward edge. We would have encountered $(4, 1)$ when expanding 4, so this is a back edge.



Suppose we have a cross-edge

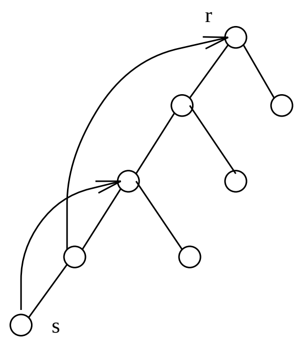


When expanding 2, we would discover 5, so the tree would look like:



Paths in search trees

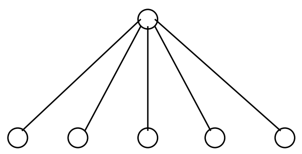
Where is the shortest path in a DFS?



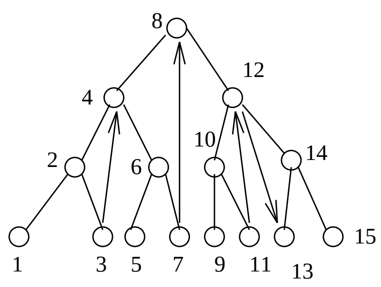
It could use multiple back and tree edges, where BFS only used tree edges.

It could use multiple back and tree edges, where BFS only uses tree edges.

DFS gives a better approximation of the longest path than BFS.



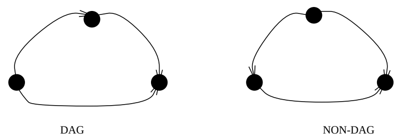
The BFS tree can have height 1, independant of the length of the longest path.



The DFS must always have height >= log P, where P is the length of the longest path.

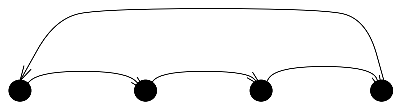
Topological Sorting

A directed, acyclic graph is a directed graph with no directed cycles.



A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.

Only a DAG can have a topological sort.



Any DAG has (at least one) topological sort.

Applications of Topological Sorting

Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given constraints, such as a set of left-right constraints on the positions of objects.

Example: Dressing schedule from CLR.

Example: Identifying errors in DNA fragment assembly.

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

```
A B R A C          A B R A C A D A B R A
                   ---------------------
A C A D A          A B R A C
                       R A C A D
A D A B R                A C A D A
                             A D A B R
D A B R A                      D A B R A

R A C A D
```

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistant ordering. If there are cycles, there must be errors.

A DFS can test if a graph is a DAG (it is iff there are no back edges - forward edges are allowed for DFS on directed graph).

Algorithm

Theorem: Arranging vertices in decreasing order of DFS finishing time gives a topological sort of a DAG.

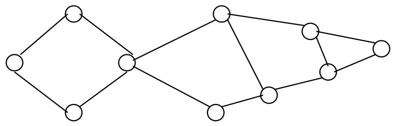
Proof: Consider any directed edge u, v, when we encounter it during the exploration of vertex u:

- If v is white - we then start a DFS of v before we continue with u.
- If v is grey - then u, v is a back edge, which cannot happen in a DAG.
- If v is black - we have already finished with v, so $f[v] < f[u]$.

Thus we can do topological sorting in $O(n + m)$ time.

Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



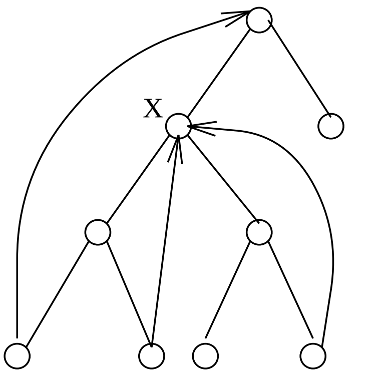
An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.

Clearly connectivity is an important concern in the design of any network.

Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster $O(n+m)$ DFS Algorithm

Theorem: In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.



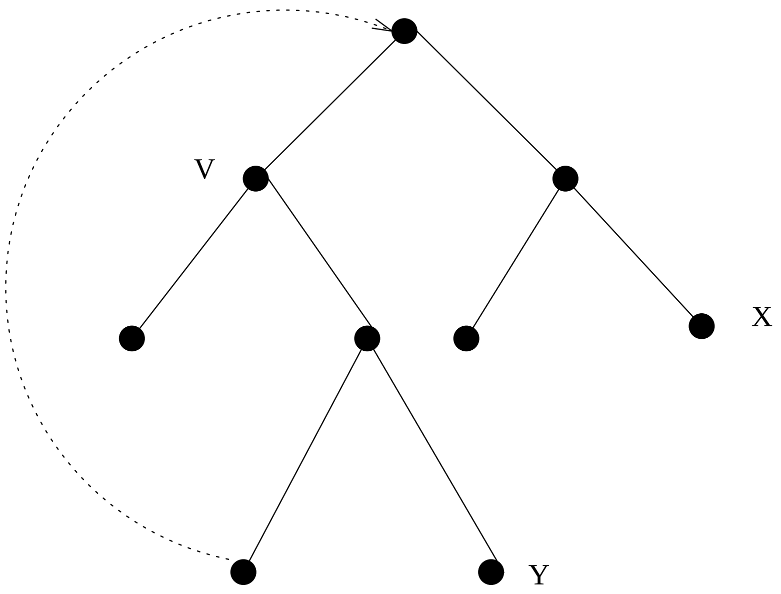
The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

Leaves cannot be articulation vertices

Proof: (1) v is an articulation vertex $\rightarrow$ v cannot be a leaf.

Why? Deleting v must seperate a pair of vertices x and y. Because of the other tree edges, this cannot happen unless y is a decendant of v.



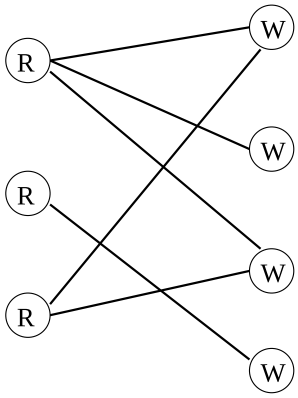
v separating x, y implies there is no back edge in the subtree of y to a proper ancestor of v.

(2) Conditions $\rightarrow v$ is a non-root articulation vertex. v separates any ancestor of v from any decendant in the appropriate subtree.

Actually implementing this test in $O(n + m)$ is tricky – but believable once you accept this theorem.

23.2-6 Give an efficient algorithm to test if a graph is bipartite.

Bipartite means the vertices can be colored red or black such that no edge links vertices of the same color.



Suppose we color a vertex red - what color must its neighbors be? *black!*

We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black!

Bipartite graphs arise in many situations, and special algorithms are often available for them. What is the interpretation of a bipartite "had-sex-with" graph?

How would you break people into two groups such that no group contains a pair of people who hate each other?

23.4-3 Give an $O(n)$ algorithm to test whether an undirected graph contains a cycle.

If you do a DFS, you have a cycle iff you have a back edge. This gives an $O(n+m)$ algorithm. But where does the m go? If the graph contains more than $n-1$ edges, it must contain a cycle! Thus we never need look at more than n edges if we are given an adjacency list representation!

23.4-5 Show that you can topologically sort in $O(n+m)$ by repeatedly deleting vertices of degree 0.

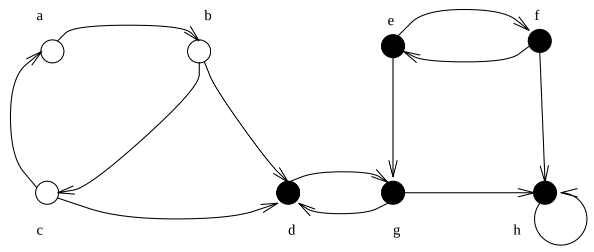
The correctness of this algorithm follows since in a DAG there must always be a vertex of indegree 0, and such a vertex can be first in topological sort. Suppose each vertex is initialized with its indegree (do DFS on G to get this). Deleting a vertex takes O(degree v). Reduce the indegree of each efficient vertex - and keep a list of degree-0 vertices to delete next.

Time: $\sum_{i=1}^{n} O(deg(v_i)) = O(n+m)$

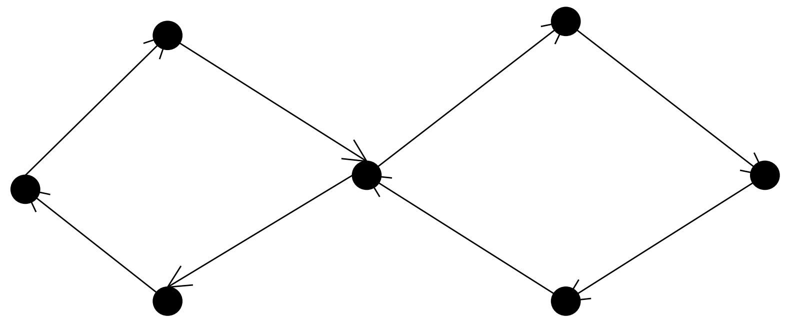
Strongly Connected Components

A directed graph is strongly connected iff there is a directed path between any two vertices.

The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.



Observe that no vertex can be in two maximal components, so it is a partition.

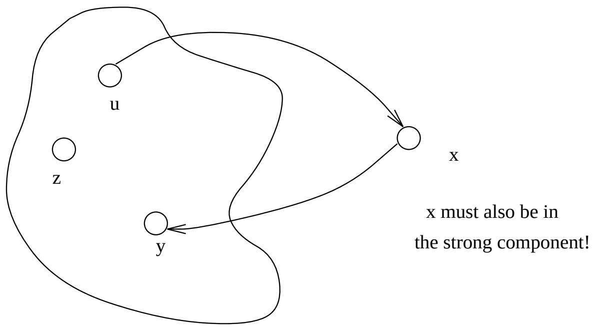


There is an amazingly elegant, linear time algorithm to find the strongly connected components of a directed graph, using DFS.

- Call DFS(σ) to compute finishing times for each vertex.
- Compute the transpose graph G^T (reverse all edges in G)
- Call DFS(G^T), but order the vertices in decreasing order of finish time.
- The vertices of each DFS tree in the forest of DFS(G^T) is a strongly connected component.

This algorithm takes $O(n+m)$, but why does it compute strongly connected components?

Lemma: If two vertices are in the same strong component, no path between them ever leaves the component.

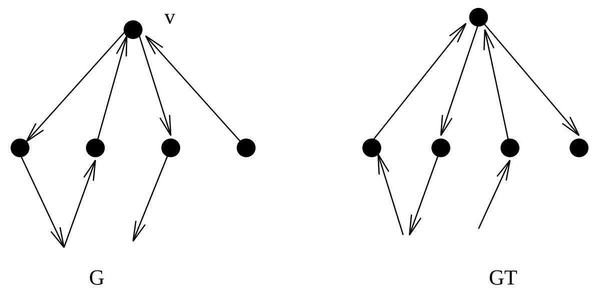


Lemma: In any DFS forest, all vertices in the same strongly connected component are in the same tree.

Proof: Consider the first vertex v in the component to be discovered. Everything in the component is reachable from it, so we will traverse it before finishing with v.

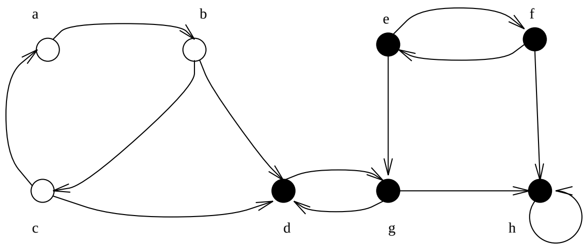
What does DFS(G^T, v) Do?

It tells you what vertices have directed paths to v, while DFS(σ,v) tells what vertices have directed paths from v. But why must any vertex in the search tree of DFS(G^T, v) also have a path from u?

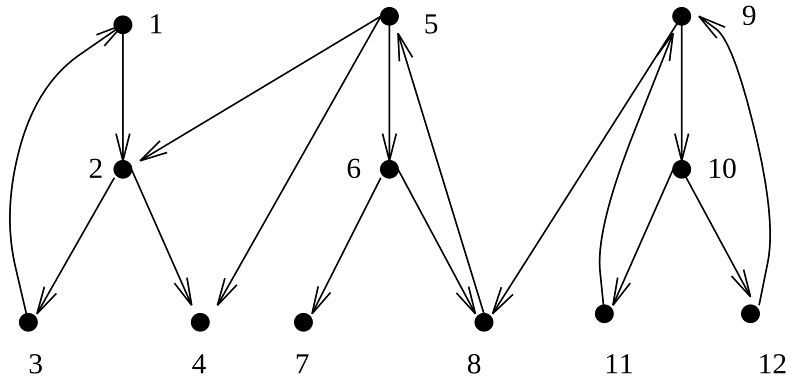


Because there is no edge from any previous DFS tree into the last tree!! Because we ordered the vertices by decreasing order of finish time, we can peel off the strongly connected components from right to left just be doing a DFS(G^T).

Example of Strong Components Algorithm



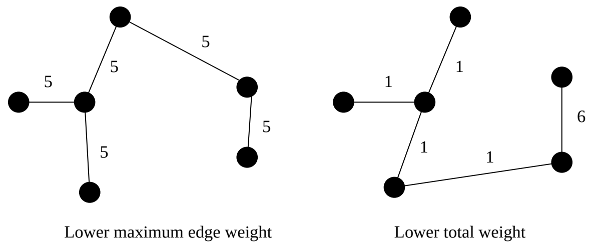
9, 10, 11, 12 can reach 9, oldest remaining finished is 5.
5, 6, 8 can reach 5, oldest remaining is 7.
7 can reach 7, oldest remaining is 1.
1, 2, 3 can reach 1, oldest remaining is 4.
4 can reach 4.



DFG(G) 9 is the last vertex to finish

24.2-6 Describe an efficent algorithm that, given an undirected graph G, determines a spanning tree G whose largest edge weight is minimum over all spanning trees of G.

First, make sure you understand the question



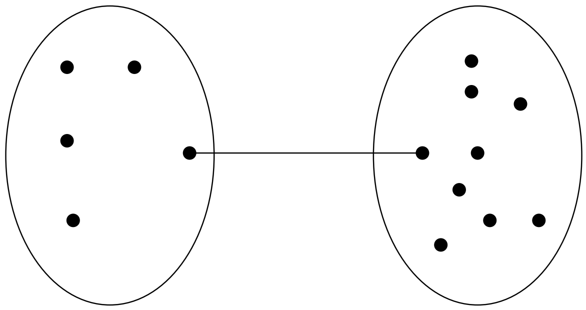
"Hey, doesn't Kruskal's algorithm do something like this."

Certainly! Since Krushal's algorithm considers the edges in order of increasing weight, and stops the moment these edges form a connected graph, the tree it gives must minimize the edge weight.

"Hey, but then why doesn't Prim's algorithm also work?"

It gives the same thing as Kruskal's algorithm, so it must be true that any minimum spanning tree minimizes the maximum edge weight!

Proof: Give me a MST and consider the largest edge weight,



Deleting it disconnects the MST. If there was a lower edge connects the two subtrees, I didn't have a MST!

Minimum Spanning Trees

A tree is a connected graph with no cycles. A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.

A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

There can be more than one minimum spanning tree in a graph $\rightarrow$ consider a graph with identical weight edges.

The minimum spanning tree problem has a long history – the first algorithm dates back at least to 1926!.

Minimum spanning tree is always taught in algorithm courses since (1) it arises in many applications, (2) it is an important example where *greedy* algorithms always give the optimal answer, and (3) Clever data structures are necessary to make it work.

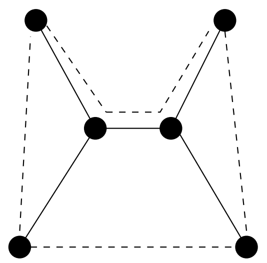
In greedy algorithms, we make the decision of what next to do by selecting the best local option from all available choices – without regard to the global structure.

Applications of Minimum Spanning Trees

Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire. Much of the work on minimum spanning (and related Steiner) trees has been conducted by the phone company.

Minimum spanning trees provide a reasonable way for *clustering* points in space into natural groups.

When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.



The Option Traveling System tour is at most twice the length of the minimum spanning tree.

Note: There can be more than one minimum spanning tree considered as a group with identical weight edges.

Prim's Algorithm

If G is connected, every vertex will appear in the minimum spanning tree. If not, we can talk about a minimum spanning forest.

Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.

As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.

During execution we will label each vertex as either in the tree, *fringe* - meaning there exists an edge from a tree vertex, or *unseen* - meaning the vertex is more than one edge away.

```
Select an arbitrary vertex to start.
While (there are fringe vertices)
    select minimum weight edge between tree and fringe
    add the selected edge and vertex to the tree
```

Clearly this creates a spanning tree, since no cycle can be introduced via edges between tree and fringe vertices, but is it minimum?

Why is Prim's algorithm correct?

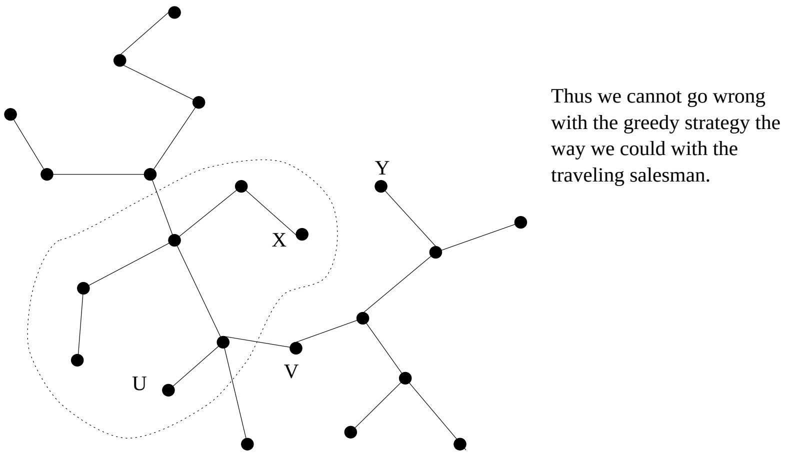
Don't be scared by the proof – the reason is really quite basic:

Theorem: Let G be a connected, weighted graph and let $E' \subset E$ be a subset of the edges in a MST $T = (V, E_T)$. Let V' be the vertices incident with edges in E'. If (x, y) is an edge of minimum weight such that $x \in V'$ and y is not in V', then $E' \cup \{x, y\}$ is a subset of a minimum spanning tree.

Proof: If the edge is in T, this is trivial.

Suppose (x, y) is not in T Then there must be a path in T from x to y since T is connected. If (v, w) is the first edge on this path with one edge in V', if we delete it and replace it with (x, y) we get a spanning tree.

This tree must have smaller weight than T, since $W(v, w) > W(x, y)$. Thus T could not have been the MST.



PRIM's Algorithm is correct!

Prim's Algorithm is correct!

Thus we cannot go wrong with the greedy strategy the way we could with the traveling salesman problem.

But how fast is Prim's?

That depends on what data structures are used. In the simplest implementation, we can simply mark each vertex as tree and non-tree and search always from scratch:

```
Select an arbitrary vertex to start.
While (there are non-tree vertices)
    select minimum weight edge between tree and fringe
    add the selected edge and vertex to the tree
```

This can be done in $O(nm)$ time, by doing a DFS or BFS to loop through all edges, with a constant time test per edge, and a total of n iterations.

Can we do faster? If so, we need to be able to identify fringe vertices and the minimum cost edge associated with it, fast. We will augment an adjacency list with fields maintaining fringe information.

Vertex:

fringelink pointer to next vertex in fringe list.

fringe weight cheapest edge linking v to l.

parent other vertex with v having fringeweight.

status intree, fringe, unseen.

adjacency list the list of edges.

Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list.

After adding a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices (there may be a cheaper way through the new vertex) can be done in $O(n)$ time.

Total time is $O(n^2)$.

Kruskal's Algorithm

Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim's algorithm is optimal in the worst case. Is that all she wrote?

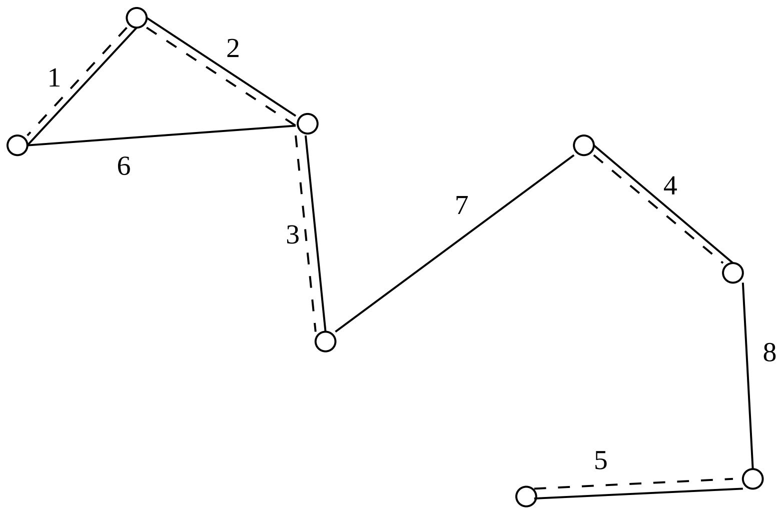
The complexity of Prim's algorithm is independent of the number of edges. Can we do better with sparse graphs? Yes!

Kruskal's algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle. Obviously, this gives a spanning tree, but is it minimal?

Why is Kruskal's algorithm correct?

Theorem: Let G be a weighted graph and let $E' \subset E$. If E' is contained in a MST T and e is the smallest edge in $E - E'$ which does not create a cycle, $E' \cup e \subseteq T$.

Proof: As before, suppose e is not in T. Adding e to T makes a cycle. Deleting another edge from this cycle leaves a connected graph, and if it is one from $E - E'$ the cost of this tree goes down. Since such an edge exists, T could not be a MST.



How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle the forest we have thus far built – if so discard, else add to forest. With a BFS/DFS, this can be done in $O(n)$ time (since the tree has at most n edges).

The total time is $O(mn)$, but can we do better?

Kruskal's algorithm builds up connected components. Any edge where both vertices are in the same connected component create a cycle. Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!

```
Put the edges in a heap
count = 0
while (count < n − 1) do
        get next edge (v, w)
        if (component (v) ≠ component(w))
                add to T
                component (v)=component(w)
```

If we can test components in $O(\log n)$, we can find the MST in $O(m \log m)$!

Question: Is $O(m \log n)$ better than $O(m \log m)$?

Union-Find Programs

Our analysis that Kruskal's MST algorithm is $O(m \log m)$ requires a fast way to test whether an edge links two vertices in the same connected component.

Thus we need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by *UNION* and *FIND* operations:

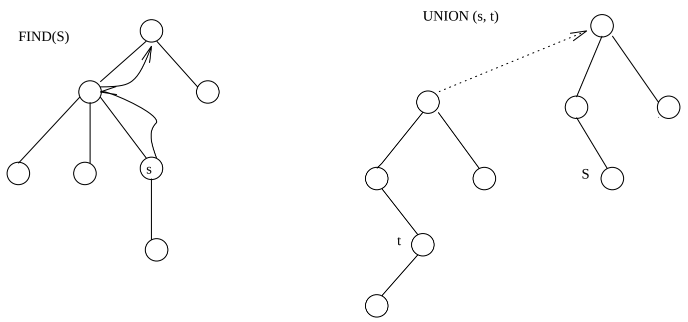
Is $s_i \equiv s_j$
  $t = \text{Find}(s_i)$
  $u = \text{Find}(s_j)$
  Return (Is $t = u$?)

Make $s_i \equiv s_j$
  $t = d(s_i)$
  $u = d(s_j)$
  $\text{Union}(t, u)$

Find returns the name of the set and *Union* sets the members of t to have the same name as u.

We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.

A simple implementation is to represent each set as a tree, with pointers from a node to its parent. Each element is contained in a node, and the *name* of the set is the key at the root:



In the worst case, these structures can be very unbalanced:

For $i = 1$ to $n/2$ do
    UNION(i,i+1)
For $i = 1$ to $n/2$ do
    FIND(1)

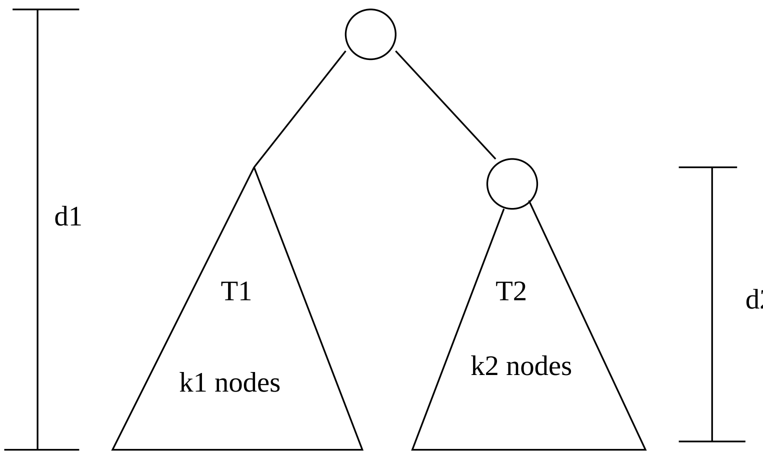
We want the limit the height of our trees which are effected by *UNIONs*. When we union, we can make the tree with fewer nodes the child.

Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal height!

Lemma: If $Union(t, v)$ attaches the root of v as a subtree of t iff the number of nodes in t is greater than or equal to the number in v, after any sequence of unions, any tree with $h/4$ nodes has height at most $\lfloor \lg h \rfloor$.

Proof: By induction on the number of nodes k, $k = 1$ has height 0.

Assume true to $k - 1$ nodes. Let d_i be the height of the tree t_i



k = k1+ k2 nodes

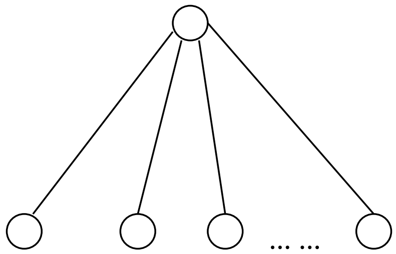
d is the height

If $(d_1 > d_2)$ then $d = d_1 \leq \lfloor \log k_1 \rfloor \leq \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \log k \rfloor$
If $(d_1 \leq d_2)$, then $k_1 \geq k_2$.
$d = d_2 + 1 \leq \lfloor \log k_2 \rfloor + 1 = \lfloor \log 2k_2 \rfloor \leq \lfloor \log(k_1 + k_2) \rfloor = \log k$

Can we do better?

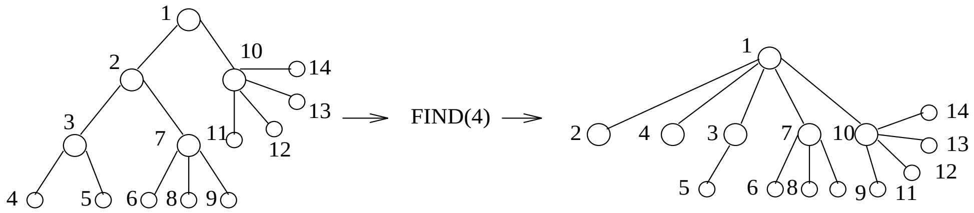
We can do *unions* and *finds* in $O(\log n)$, good enough for Kruskal's algorithm. But can we do better?

The ideal *Union-Find* tree has depth 1:



N-1 leaves

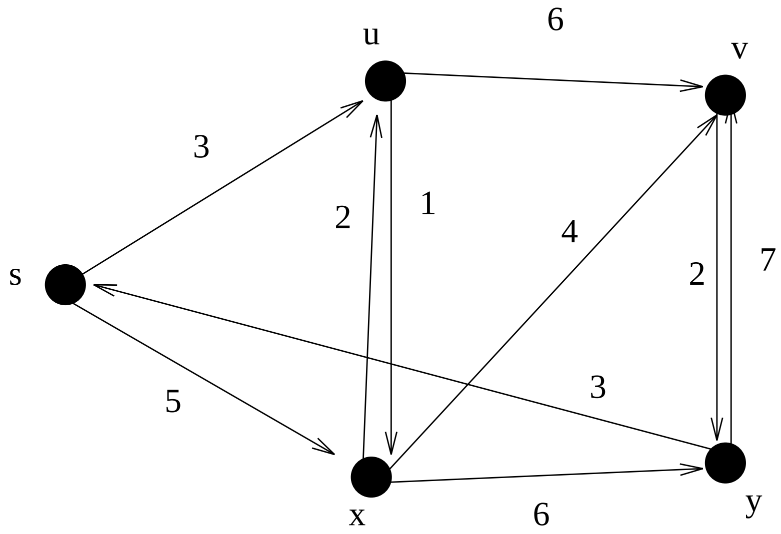
On a find, if we are going down a path anyway, why not change the pointers to point to the root?



This path compression will let us do better than $O(n \log n)$ for n union-finds.

$O(n)$? Not quite ... Difficult analysis shows that it takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and α(number of atoms in the universe)$= 5$.

25.1-1 Give two more shortest path trees for the following graph:



Run through Dijkstra's algorithm, and see where there are ties which can be arbitrarily selected.

There are two choices for how to get to the third vertex x, both of which cost 5.

There are two choices for how to get to vertex v, both of which cost 9.

Lessons from the Backtracking contest

- As predicted, the speed difference between the fastest programs and average program dwarfed the difference between a supercomputer and a microcomputer. Algorithms have a bigger impact on performance than hardware!
- Different algorithms perform differently on different data. Thus even hard problems may be tractable on the kind of data you might be interested in.
- None of the programs could efficiently handle all instances for $n \approx 30$. We will find out why after the midterm, when we discuss NP-completeness.
- Many of the fastest programs were very short and simple (KISS). My bet is that many of the enhancements students built into them actually showed them down! This is where profiling can come in handy.
- The fast programs were often recursive.

Winning Optimizations

- Finding a good initial solution via randomization or heuristic improvement helped by establishing a good upper bound, to constrict search.

- Using half the largest vertex degree as a lower bound similarly constricted search.
- Pruning a partial permutation the instant an edge was $\geq$ the target made the difference in going from (say) 8 to 18.
- Positioning the partial permutation vertices separated by b instead of 1 meant significantly earlier cutoffs, since any edge does the job.
- Mirror symmetry can only save a factor of 2, but perhaps more could follow from partitioning the vertices into equivalence classes by the same neighborhood.

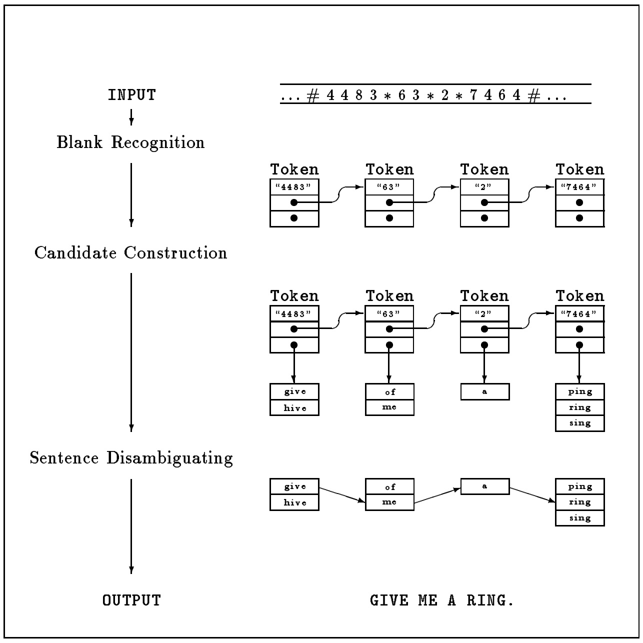
Shortest Paths

Finding the shortest path between two nodes in a graph arises in many different applications:

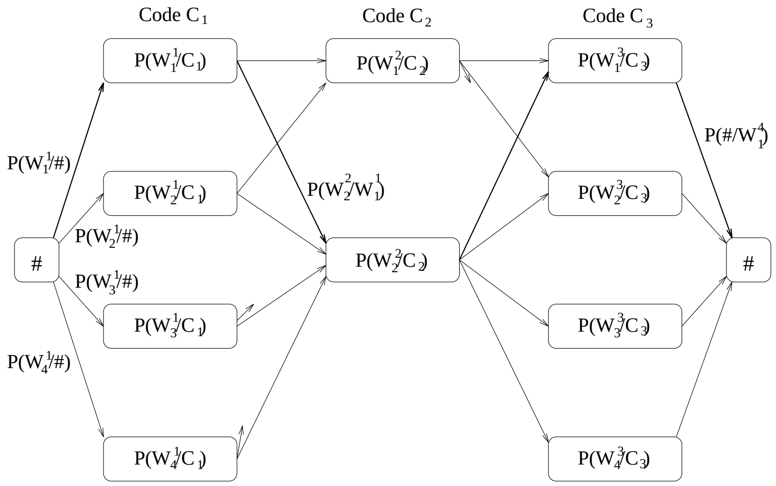
- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how look will it take for a message to get between two places? Which two locations are furthest apart, ie. what is the *diameter* of the network.

Shortest Paths and Sentence Disambiguation

In our work on reconstructing text typed on an (overloaded) telephone keypad, we had to select which of many possible interpretations was most likely.



We constructed a graph where the vertices were the possible words/positions in the sentence, with an edge between possible neighboring words.



The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. ‘hive me’ would be less than ‘give me’, for example.

The final system worked extremely well – identifying over 99% of characters correctly based on grammatical and statistical constraints.

Dynamic programming (the Viterbi algorithm) can be used on the sentences to obtain the same results, by finding the shortest paths in the underlying DAG.

Finding Shortest Paths

In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which can be found in $O(n+m)$ time via breadth-first search.

In a weighted graph, the weight of a path between two vertices is the sum of the weights of the edges on a path.

BFS will not work on weighted graphs because sometimes visiting more edges can lead to shorter distance, ie. $1+1+1+1+1+1+1 < 10$.

Note that there can be an exponential number of shortest paths between two nodes – so we cannot report all shortest paths efficiently.

Note that negative cost cycles render the problem of finding the shortest path meaningless, since you can always loop around the negative cost cycle more to reduce the cost of the path.

Thus in our discussions, we will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges.

Minimum spanning trees are uneffected by negative cost edges.

Dijkstra's Algorithm

We can use *Dijkstra's algorithm* to find the shortest path between any two vertices s and t in G.

The principle behind Dijkstra's algorithm is that if $s, \ldots, x, \ldots, t$ is the shortest path from s to t, then $s, \ldots, x$ had better be the shortest path from s to x.

This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to find the shortest path to more distant nodes.

The shortest path from s to s, $d(s, s) = 0$. If all edge weights are positive, the *smallest* edge incident to s, say (s, x), defines $d(s, x)$.

We can use an array to store the length of the shortest path to each node. Initialize each to ∞ to start.

Soon as we establish the shortest path from s to a new node x, we go through each of its incident edges to see if there is a better way from s to other nodes thru x.

$known = \{s\}$
for $i = 1$ to n, $dist[i] = \infty$
for each edge (s, v), $dist[v] = d(s, v)$
last$=s$
while $(last \neq t)$
    select v such that $dist(v) = \min_{unknown} dist(i)$
    for each (v, x), $dist[x] = \min(dist[x], dist[v] + w(v, x))$
    last$=v$
    $known = known \cup \{v\}$

Complexity $\rightarrow O(n^2)$ if we use adjacency lists and a Boolean array to mark what is known.

This is essentially the same as Prim's algorithm.

An $O(m \lg n)$ implementation of Dijkstra's algorithm would be faster for sparse graphs, and comes from using a heap of the vertices (ordered by distance), and updating the distance to each vertex (if necessary) in $O(\lg n)$ time for each edge out from freshly known vertices.

Even better, $O(n \lg n + m)$ follows from using Fibonacci heaps, since they permit one to do a decrease-key operation in $O(1)$ amortized time.

All-Pairs Shortest Path

Notice that finding the shortest path between a pair of vertices (s,t) in worst case requires first finding the shortest path from s to all other vertices in the graph.

Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices.

We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n^3)$. Can we do better?

Improving the complexity is an open question but there is a ***super-slick*** dynamic programming algorithm which also runs in $O(n^3)$.

Dynamic Programming and Shortest Paths

The four-step approach to dynamic programming is:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute this recurrence in a bottom-up fashion.
4. Extract the optimal solution from computed information.

From the adjacency matrix, we can construct the following matrix:

$$\begin{array}{ll} D[i,j] = \infty, & \text{if } i \neq j \text{ and } (v_i, v_j) \text{ is not in } E \\ D[i,j] = w(i,j), & \text{if } (v_i, v_j) \in E \\ D[i,j] = 0, & \text{if } i = j \end{array}$$

This tells us the shortest path going through no intermediate nodes.

There are several ways to characterize the shortest path between two nodes in a graph. Note that the shortest path from i to j, $i \neq j$, using at most M edges consists of the shortest path from i to k using at most $M-1$ edges$+W(k,j)$ for some k.

This suggests that we can compute all-pair shortest path with an induction based on the number of edges in the optimal path.

Let $d[i,j]^m$ be the length of the shortest path from i to j using at most m edges.

What is $d[i,j]^0$?

$$\begin{aligned} d[i,j]^0 &= 0 \text{ if } i = j \\ &= \infty \text{ if } i \neq j \end{aligned}$$

What if we know $d[i,j]^{m-1}$ for all i, j?

$$\begin{aligned} d[i,j]^m &= \min(d[i,j]^{m-1}, \min(d[i,k]^{m-1} + w[k,j])) \\ &= \min(d[i,k]^{m-1} + w[k,j]), 1 \leq k \leq i \end{aligned}$$

since $w[k,k] = 0$

This gives us a recurrence, which we can evaluate in a bottom up fashion:

for $i = 1$ to n
    for $j = 1$ to n
    $d[i, j]^m = \infty$
    for $k = 1$ to n
        $d[i, j]^0$=Min($d[i, k]^m$, $d[i, k]^{m-1} + d[k, j]$)

This is an $O(n^3)$ algorithm just like matrix multiplication, but it only goes from m to $m + 1$ edges.

Since the shortest path between any two nodes must use at most n edges (unless we have negative cost cycles), we must repeat that procedure n times ($m = 1$ to n) for an $O(n^4)$ algorithm.

We can improve this to $O(n^3 \log n)$ with the observation that any path using at most $2m$ edges is the function of paths using at most m edges each. This is just like computing $a^n = a^{n/2} \times a^{n/2}$. So a logarithmic number of multiplications suffice for exponentiation.

Although this is slick, observe that even $O(n^3 \log n)$ is slower than running Dijkstra's algorithm starting from each vertex!

The Floyd-Warshall Algorithm

An alternate recurrence yields a more efficient dynamic programming formulation. Number the vertices from 1 to n.

Let $d[i,j]^k$ be the shortest path from i to j using only vertices from $1, 2, ..., k$ as possible intermediate vertices.

What is $d[j,j]^0$? With no intermediate vertices, any path consists of at most one edge, so $d[i,j]^0 = w[i,j]$.

In general, adding a new vertex $k+1$ helps iff a path goes through it, so

$$\begin{aligned} d[i,j]^k &= w[i,j] \text{ if } k = 0 \\ &= \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1}) \text{ if } k \geq 1 \end{aligned}$$

Although this looks similar to the previous recurrence, it isn't. The following algorithm implements it:

$d^o = w$
for $k = 1$ to n
    for $i = 1$ to n
        for $j = 1$ to n
        $d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$

This obviously runs in $\Theta(n^3)$ time, which asymptotically is no better than a calls to Dijkstra's algorithm. However, the loops are so tight and it is so short and simple that it runs better in practice by a constant factor.

The Theory of NP-Completeness

Several times this semester we have encountered problems for which we couldn't find efficient algorithms, such as the traveling salesman problem. We also couldn't prove an exponential time lower bound for the problem.

By the early 1970s, literally hundreds of problems were stuck in this limbo. The theory of NP-Compleness, developed by Stephen Cook and Richard Karp, provided the tools to show that all of these problems were really the same problem.

Polynomial vs. Exponential Time

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01 μs	0.1 μs	1 μs	3.63 ms
20	0.02 μs	0.4 μs	1 ms	77.1 years
30	0.03 μs	0.9 μs	1 sec	8.4×10^{15} years
40	0.04 μs	1.6 μs	18.3 min	
50	0.05 μs	2.5 μs	13 days	
100	0.1 μs	10 μs	4×10^{13} years	
1,000	1.00 μs	1 ms		

The Main Idea

Suppose I gave you the following algorithm to solve the *bandersnatch* problem:

Bandersnatch(G)
    Convert G to an instance of the Bo-billy problem Y.
    Call the subroutine Bo-billy on Y to solve this instance.
    Return the answer of Bo-billy(Y) as the answer to G.

Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a *reduction*.

Now suppose my reduction translates G to Y in $O(P(n))$:

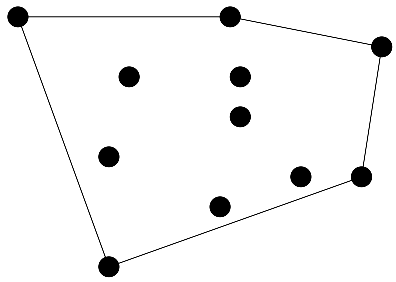
1. If my Bo-billy subroutine ran in $O(P'(n))$ I can solve the Bandersnatch problem in $O(P(n) + P'(n))$

2. If I know that $\Omega(P'(n))$ is a lower-bound to compute Bandersnatch, then $\Omega(P'(n) - P(n))$ must be a lower-bound to compute Bo-billy.

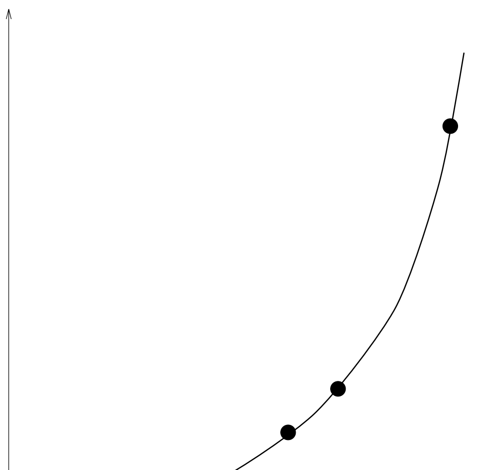
The second argument is the idea we use to prove problems hard!

Convex Hull and Sorting

A nice example of a reduction goes from sorting numbers to the convex hull problem:



We must translate each number to a point. We can map x to (x, x^2).



Why? That means each integer is mapped to a point on the parabola $y = x^2$.

Since this parabola is convex, every point is on the convex hull. Further since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by x-coordinate, ie. the original numbers.

Sort(S)
    For each $i \in S$, create point (i, i^2).
    Call subroutine convex-hull on this point set.
    From the leftmost point in the hull,
        read off the points from left to right.

Creating and reading off the points takes $O(n)$ time.

What does this mean? Recall the sorting lower bound of $\Omega(n \lg n)$. If we could do convex hull in better than $n \lg n$, we could sort faster than $\Omega(n \lg n)$ – which violates our lower bound.

Thus convex hull must take $\Omega(n \lg n)$ *as well!!!*

Observe that any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

What is a problem?

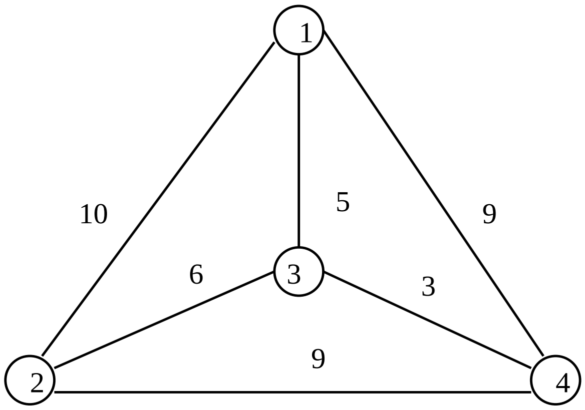
A *problem* is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.

An instance is a problem with the input parameters specified.

Example: The Traveling Salesman

Problem: Given a weighted graph G, what tour $\{v_1, v_2, ..., v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.

Instance: $d[v_1, d_2] = 10$, $d[v_1, d_3] = 5$, $d[v_1, d_4] = 9$, $d[v_2, d_3] = 6$, $d[v_2, d_4] = 9$, $d[v_3, d_4] = 3$



Solution: $\{v_1, v_2, v_3, v_4\}$ cost$= 27$

A problem with answers restricted to *yes* and *no* is called a *decision problem*. Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation.

Example: The Traveling Salesman Decision Problem.

Given a weighted graph G and integer k, does there exist a traveling salesman tour with cost $\leq k$?

Using binary search and the decision version of the problem we can find the optimal TSP solution.

For convenience, from now on we will talk *only* about decision problems.

Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc. All reasonable encodings will be within polynomial size of each other.

The fact that we can ignore minor differences in encoding is important. We are concerned with the difference between algorithms which are polynomial and exponential in the size of the input.

Satisfiability

Consider the following logic problem:

Instance: A set V of variables and a set of clauses C over V.

Question: Does there exist a satisfying truth assignment for C?

Example 1: $V = v_1, v_2$ and $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$

A clause is satisfied when at least one literal in it is TRUE. C is satisfied when $v_1 = v_2 =$TRUE.

Example 2: $V = v_1, v_2$,

$$C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$$

Although you try, and you try, and you try and you try, you can get no satisfaction.

There is no satisfying assigment since v_1 must be FALSE (third clause), so v_2 must be FALSE (second clause), but then the first clause is unsatisfiable!

For various reasons, it is known that satisfiability is a hard problem. Every top-notch algorithm expert in the world (and countless other, lesser lights) have tried to come up with a fast algorithm to test whether a given set of clauses is satisfiable, but all have failed. Further, many strange and impossible-to-believe things have been shown to be true if someone in fact did find a fast satisfiability algorithm.

Clearly, Satisfiability is in NP, since we can guess an assignment of TRUE, FALSE to the literals and check it in polynomial time.

P versus NP

The precise distinction between whether a problem is in P or NP is somewhat technical, requiring formal language theory and Turing machines to state correctly.

However, intuitively a problem is in P, (ie. polynomial) if it can be solved in time polynomial in the size of the input.

A problem is in *NP* if, given the answer, it is possible to verify that the answer is correct within time polynomial in the size of the input.

Example P – Is there a path from s to t in G of length less than k.

Example NP – Is there a TSP tour in G of length less than k. Given the tour, it is easy to add up the costs and convince me it is correct.

Example *not NP* – How many TSP tours are there in G of length less than k. Since there can be an exponential number of them, we cannot count them all in polynomial time.

Don't let this issue confuse you – the important idea here is of reductions as a way of proving hardness.

3-Satisfiability

Instance: A collection of clause C where each clause contains exactly 3 literals, boolean variable v.

Question: Is there a truth assignment to v so that each clause is satisfied?

Note that this is a more restricted problem than SAT. If 3-SAT is NP-complete, it implies SAT is NP-complete but not visa-versa, perhaps long clauses are what makes SAT difficult?!

After all, 1-Sat is trivial!

Theorem: 3-SAT is NP-Complete

Proof: 3-SAT is NP – given an assignment, just check that each clause is covered. To prove it is complete, a reduction from $Sat \propto 3 - Sat$ must be provided. We will transform each clause independantly based on its *length.*

Suppose the clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses:

  $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$.

  Note that the only way all four of these can be satisfied is if z is TRUE.

- If $k = 2$, meaning $\{z_1, z_2\}$, create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$

- If $k = 3$, meaning $\{z_1, z_2, z_3\}$, copy into the 3-SAT instance as it is.

- If $k > 3$, meaning $\{z_1, z_2, ..., z_n\}$, create $n - 3$ new variables and $n - 2$ new clauses in a chain: $\{v_i, z_i, \overline{v_i}\}$, ...

If none of the original variables in a clause are TRUE, there is no way to satisfy all of them using the additional variable:

$$(F, F, T), (F, F, T), \ldots, (F, F, F)$$

But if any literal is TRUE, we have $n-3$ free variables and $n-3$ remaining 3-clauses, so we can satisfy each of them. $(F, F, T), (F, F, T), \ldots, (\mathbf{F}, \mathbf{T}, \mathbf{F}), \ldots, (T, F, F), (T, F, F)$

Since any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets variables giving a SAT solution – the problems are equivallent. If there were n clauses and m total literals in the SAT instance, this transform takes $O(m)$ time, so SAT and 3-SAT.

Note that a slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete. However, it breaks down when we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses. It turns out that resolution gives a polynomial time algorithm for 2-SAT.

Having at least 3-literals per clause is what makes the problem difficult. Now that we have shown 3-SAT is NP-complete, we may use it for further reductions. Since the set of 3-SAT instances is smaller and more regular than the SAT instances, it will be easier to use 3-SAT for future reductions. Remember the direction to reduction!

$$Sat \propto 3 - Sat \propto X$$

36.4-5 Give a polynomial-time algorithm to satisfy Boolean formulas in disjunctive normal form.

Satisfying one clause in DFS satisfied the whole formula. One clause can always be satisfied iff it does not contain both a variable and its complement.

Why not use this reduction to give a polynomial-time algorithm for 3-SAT? The DNF formula can become exponentially large and hence the reduction cannot be done in polynomial time.

A Perpetual Point of Confusion

Note carefully the direction of the reduction.

We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in. If we do the reduction the other way, all we get is a slow way to solve x, by using a subroutine which probably will take exponential time.

This always is confusing at first - it seems bass-ackwards. Make sure you understand the direction of reduction now - and think back to this when you get confused.

Integer Programming

Instance: A set v of integer variables, a set of inequalities over these variables, a function $f(v)$ to maximize, and integer B.

Question: Does there exist an assignment of integers to v such that all inequalities are true and $f(v) \geq B$?

Example:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 3$$

A solution to this is $v_1 = 1$, $v_2 = 2$.

Example:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 5$$

Since the maximum value of $f(v)$ given the constraints is $2 \times 2 = 4$, there is no solution.

Theorem: Integer Programming is NP-Hard

Proof: By reduction from Satisfiability

Any set instance has boolean variables and clauses. Our Integer programming problem will have twice as many variables as the SAT instance, one for each variable and its compliment, as well as the following inequalities:

For each variable v_i in the set problem, we will add the following constraints:

- $1 \leq V_i \leq 0$ and $1 \leq \overline{V}_i \leq 0$

  Both IP variables are restricted to values of 0 or 1, which makes them equivalent to boolean variables restricted to true/false.

- $1 \leq V_i + \overline{V}_i \leq 1$

  Exactly one of the IP variables associated with a given sat variable is 1. This means that exactly one of V_i and $\overline{V}_i$ are true!

- for each clause $C_i = \{v_1, \overline{v}_2, \overline{v}_3 \ldots v_n\}$ in the sat instance, construct a constraint:

$$v_1 + \overline{v}_2 + \overline{v}_3 + \ldots v_n \geq 1$$

  Thus at least one IP variable must be one in each clause! Thus satisfying the constraint is equivalent to satisfying the clause!

Our maximization function and bound are relatively unimportant: $f(v) = V_1$ $B = 0$.

Clearly this reduction can be done in polynomial time.

We must show:

1. Any SAT solution gives a solution to the IP problem.

   In any SAT solution, a TRUE literal corresponds to a 1 in the IP, since if the expression is SATISFIED, at least one literal per clause in TRUE, so the sum in the inequality is ≥ 1.

2. Any IP solution gives a SAT solution.

   Given a solution to this IP instance, all variables will be 0 or 1. Set the literals correspondly to 1 variable TRUE and the 0 to FALSE. No boolean variable and its complement will both be true, so it is a legal assignment with also must satisfy the clauses.

Neat, sweet, and NP-complete!

Things to Notice

1. The reduction preserved the structure of the problem. Note that the reduction did not *solve* the problem – it just put it in a different format.

2. The possible IP instances which result are a small subset of the possible IP instances, but since some of them are hard, the problem in general must be hard.

3. The transformation captures the essence of why IP is hard - it has nothing to do with big coefficients or big ranges on variables; for restricting to 0/1 is enough. A careful study of what properties we do need for our reduction tells us a lot about the problem.

4. It is not obvious that IP $\leq$ NP, since the numbers assigned to the variables may be too large to write in polynomial time - don't be too hasty!

36.5-2 – Given an integer $m \times n$ matrix A, and in integer m-vector b, the 0-1 integer programming problem asks whether there is an integer n-vector x with elements in the set $(0, 1)$ such that $A\times \leq b$. Prove that 0-1 integer programming is NP-hard (hint: reduce from 3-SAT).

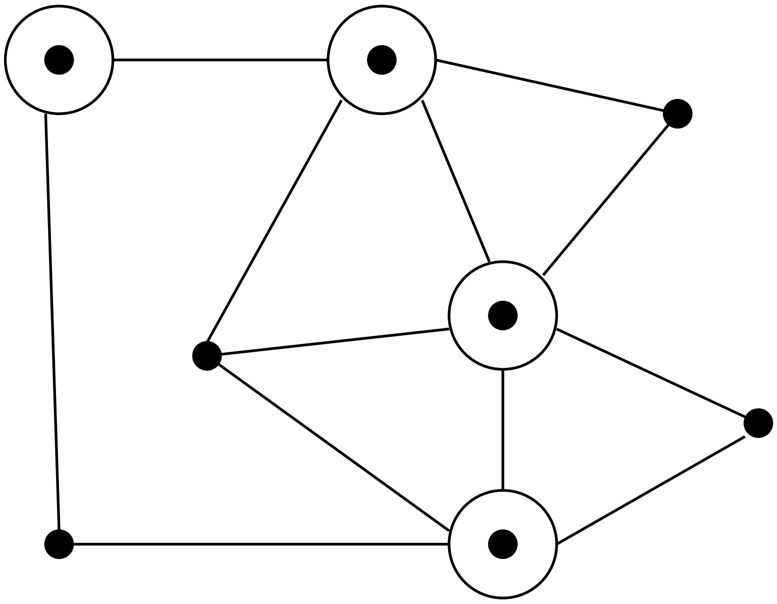
This is really the exact same problem as the previous integer programming problem, slightly concealed by:

- The linear algebra notation – each row is one constraint.
- All inequalities are $\leq$ – multiply both sides by -1 to reverse the constraint from $\geq$ to $\leq$ if necessary.

Vertex Cover

Instance: A graph $G = (V, E)$, and integer $k \leq V$

Question: Is there a subset of at most k vertices such that every $e \in E$ has at least one vertex in the subset?



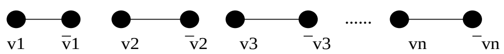
Here, four of the eight vertices are enough to cover. It is trivial to find *a* vertex cover of a graph – just take all the vertices. The tricky part is to cover with as small a set as possible.

Theorem: Vertex cover is NP-complete.

Proof: VC in in NP – guess a subset of vertices, count them, and show that each edge is covered.

To prove completeness, we show 3-SAT and VC. From a 3-SAT instance with n variables and C clauses, we construct a graph with $2N+3C$ vertices.

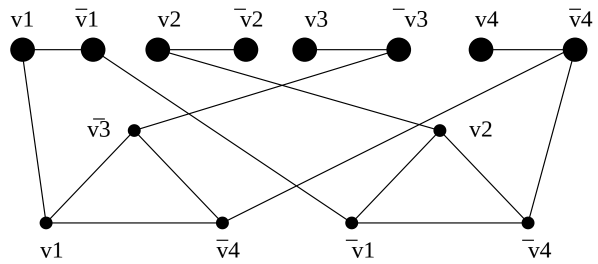
For each variable, we create two vertices connected by an edge:



To cover each of these edges, at least n vertices must be in the cover, one for each pair. For each clause, we create three new vertices, one for each literal in each clause. Connect these in a triangle.

At least two vertices per triangle must be in the cover to take care of edges in the triangle, for a total of at least $2C$ vertices.

Finally, we will connect each literal in the flat structure to the corresponding vertices in the triangles which share the same literal.



Claim: This graph will have a vertex cover of size $N + 2C$ if and only if the expression is satisfiable.

By the earlier analysis, any cover must have at least $N + 2C$ vertices. To show that our reduction is correct, we must show that:

1. *Every satisfying truth assignment gives a cover.*

   Select the N vertices cooresponding to the TRUE literals to be in the cover. Since it is a satisfying truth assignment, at least one of the three cross edges associated with each clause must already be covered - pick the other two vertices to complete the cover.

2. *Every vertex cover gives a satisfying truth assignment.*

   Every vertex cover must contain n first stage vertices and $2C$ second stage vertices. Let the first stage vertices define the truth assignment.

   To give the cover, at least one cross-edge must be covered, so the truth assignment satisfies.

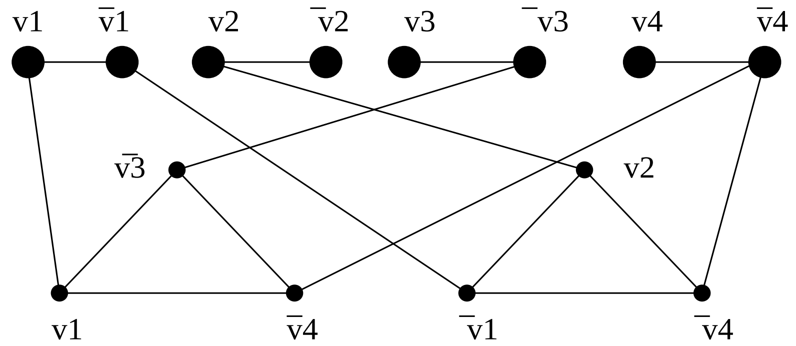
For a cover to have $N + 2C$ vertices, all the cross edges must be incident on a selected vertex.

Let the N selected vertices from the first stage coorespond to TRUE literals. If there is a satisfying truth assignment, that means at least one of the three cross edges from each triangle is incident on a TRUE vertex.

By adding the other two vertices to the cover, we cover all edges associated with the clause.

Every SAT defines a cover and Every Cover Truth values for the SAT!

Example: $V_1 = V_2 = True$, $V_3 = V_4 = False$.



Starting from the Right Problem

As you can see, the reductions can be very clever and very complicated. While theoretically any NP-complete problem can be reduced to any other one, choosing the correct one makes finding a reduction much easier.

$$3 - Sat \propto VC$$

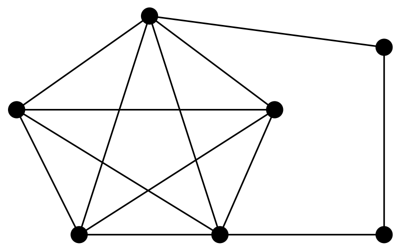
As you can see, the reductions can be very clever and complicated. While theoretically any NP-complete problem will do, choosing the correct one can make it much easier.

Maximum Clique

Instance: A graph $G = (V, E)$ and integer $j \leq v$.

Question: Does the graph contain a clique of j vertices, ie. is there a subset of v of size j such that every pair of vertices in the subset defines an edge of G?

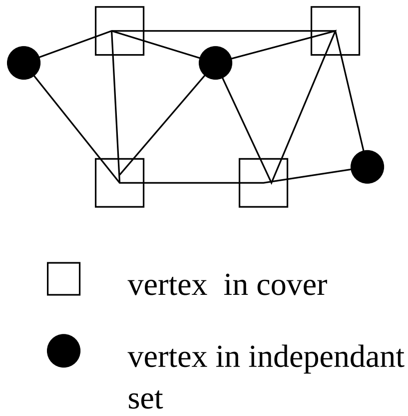
Example: this graph contains a clique of size 5.



When talking about graph problems, it is most natural to work from a graph problem - the only NP-complete one we have is vertex cover!

Theorem: Clique is NP-complete

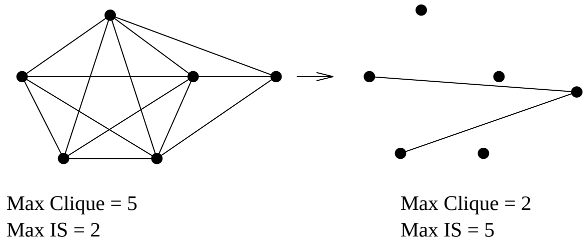
Proof: If you take a graph and find its vertex cover, the remaining vertices form an independent set, meaning there are no edges between any two vertices in the independent set, for if there were such an edge the rest of the vertices could not be a vertex cover.



Clearly the smallest vertex cover gives the biggest independent set, and so the problems are equivalent – Delete the subset of vertices in one from the total set of vertices to get the order!

Thus finding the maximum independent set must be NP-complete!

In an independent set, there are no edges between two vertices. In a clique, there are always between two vertices. Thus if we complement a graph (have an edge iff there was no edge in the original graph), a clique becomes an independent set and an independent set becomes a Clique!



Thus finding the largest clique is NP-complete:

If VC is a vertex cover in G, then $V - VC$ is a clique in G'. If C is a clique in G, $V - C$ is a vertex cover in G'.

36.5-1 Prove that subgraph isomorphism is NP-complete.

---

1. Guessing a subgraph of G and proving it is isomorphism to h takes $O(n^2)$ time, so it is in NP.

2. Clique and subgraph isomorphism. We must transform all instances of clique into some instances of subgraph isomorphism. Clique is a special case of subgraph isomorphism!

   Thus the following reduction suffices. Let $G = G'$ and $H = K_k$, the complete subgraph on k nodes.

Integer Partition (Subset Sum)

Instance: A set of integers S and a target integer t.

Problem: Is there a subset of S which adds up exactly to t?

Example: $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ and $T = 3754$

Answer: $1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = T$

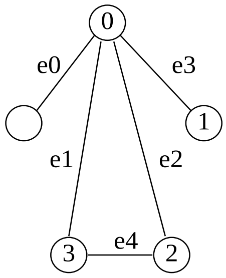
Observe that integer partition is a number problem, as opposed to the graph and logic problems we have seen to date.

Theorem: Integer Partition is *NP*-complete.

Proof: First, we note that integer partition is in *NP*. Guess a subset of the input number and simply add them up.

To prove completeness, we show that vertex cover $\propto$ integer partition. We use a data structure called an incidence matrix to represent the graph G.

	e4	e3	e2	e1	e0
v0	0	1	1	1	1
v1	0	1	0	0	0
v2	1	0	1	0	0
v3	1	0	0	1	0
v4	0	0	0	0	1



How many $1'$s are there in each column? Exactly two.

How many $1'$s in a particular row? Depends on the vertex degree.

The reduction from vertex cover will create $n + m$ numbers from G.

The numbers from the vertices will be a base-4 realization of rows from the incidence matrix, plus a high order digit:

$x_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b[i, j] \times 4^j$

ie. $V_2 = 10100$ becomes $4^5 + (4^4 + 4^2)$.

The numbers from the edges will be $y_i = 4^j$.

The target integer will be

$$t = k \times 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \times 4^j$$

Why? Each column (digit) represents an edge. We want a subset of vertices which covers each edge. We can only use k x vertex/numbers, because of the high order digit of the target.

$x_0 = 100101 = 1041$ $x_2 = 111000 = 1344$ $y_1 = 000010 = 4$

We might get only one instance of each edge in a cover - but we are free to take extra edge/numbers to grab an extra 1 per column.

VC in $G \rightarrow$ Integer Partition in S

Given k vertices covering G, pick the k cooresponding vertex/numbers. Each edge in G is incident on one or two cover vertices. If it is one, includes the cooresponding edge/number to give two per column.

Integer Partition in $S \rightarrow VC$ in G

Any solution to S must contain *exactly* k vertex/numbers. Why? It cannot be more because the target in that digit is k and it cannot be less because, with at most 3 $1'$s per edge/digit-column, no sum of these can carry over into the next column. (This is why base-4 number were chosen).

This subset of k vertex/numbers must contain at least one edge-list per column, since if not there is no way to account for the two in each column of the target integer, given that we can pick up at most one edge-list using the edge number. (Again, the prevention of carrys across digits prevents any other possibilites).

Neat, sweet, and NP-complete!

Notice that this reduction could not be performed in polynomial time if the number were written in unary $5 = 11111$. Big numbers is what makes integer partition hard!

Hamiltonian Cycle

Instance: A graph G

Question: Does the graph contains a HC, i.e. an ordered of the vertices $\{v_1, v_2, ..., v_n\}$?

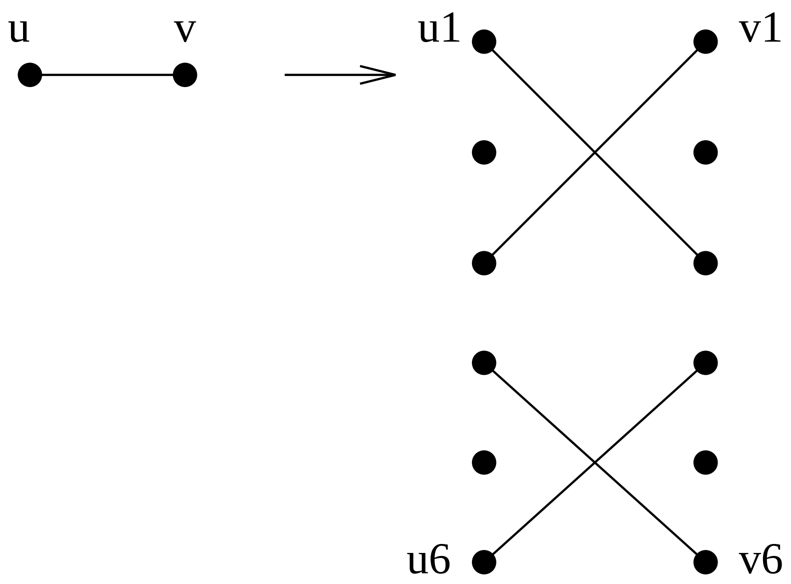
This problem is intimately relates to the Traveling Salesman.

Question: Is there an ordering of the vertices of a weighted graph such that $w(v_1, v_n) + \sum w(v_i, v_{i+1}) \leq k$?

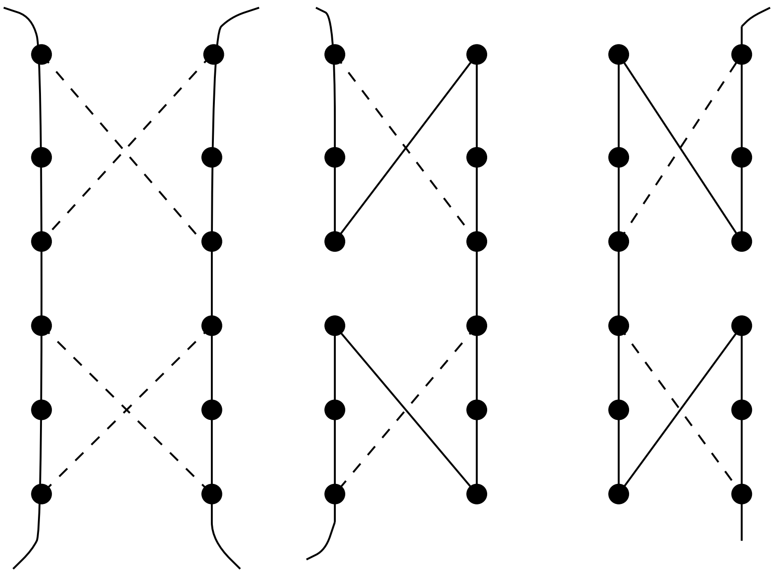
Clearly, $HC \propto TSP$. Assign each edge in G weight 1, any edge not in G weight 2. This new graph has a Traveling Salesman tour of cost n iff the graph is Hamiltonian. Thus TSP is NP-complete if we can show HC is NP-complete.

Theorem: Hamiltonian Circuit is NP-complete

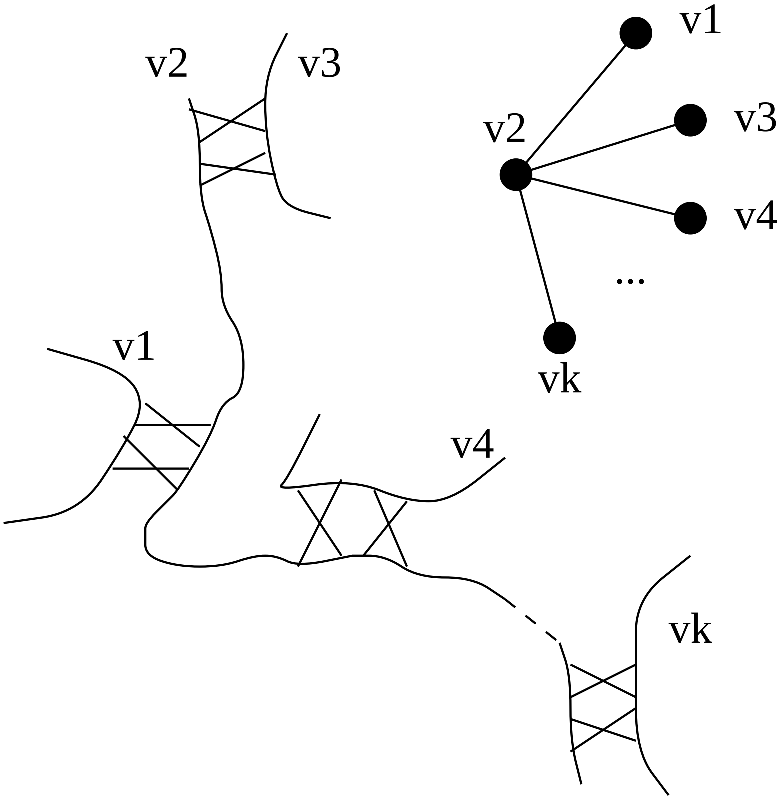
Proof: Clearly HC is in NP-guess a permutation and check it out. To show it is complete, we use vertex cover. A vertex cover instance consists of a graph and a constant k, the minimum size of an acceptable cover. We must construct another graph. Each edge in the initial graph will be represented by the following component:



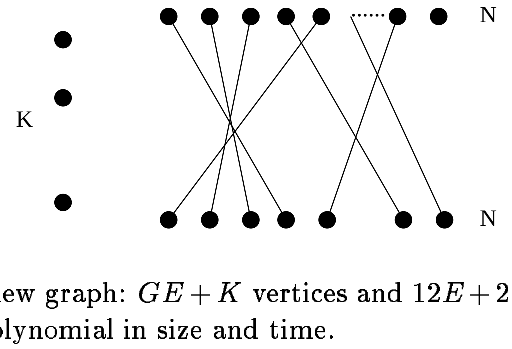
All further connections to this gadget will be through vertices v_1, v_6, u_1 and u_6. The key observation about this gadget is that there are only three ways to traverse all the vertices:



Note that in each case, we exit out the same side we entered. Each side of each edge gadget is associated with a vertex. Assuming some arbitrary order to the edges incident on a particular vertex, we can link successive gadgets by edges forming a chain of gadgets. Doing this for all vertices in the original graph creates n intertwined chains with n entry points and n exits.



Thus we have encoded the information about the initial graph. What about k? We set up k additional vertices and connect each of these to the n start points and n end points of each chain.



Total size of new graph: $GE + K$ vertices and $12E + 2kN + 2E$ edges $\rightarrow$ construction is polynomial in size and time.

We claim this graph has a HC iff G has a VC of size k.

1. Suppose $\{v_1, v_2, ..., v_n\}$ is a HC.

Assume it starts at one of the k selector vertices. It must then go through one of the chains of gadgets until it reaches a different selector vertex.

Since the tour is a HC, all gadgets are traversed. The k chains correspond to the vertices in the cover.

Note that if both vertices associated with an edge are in the cover, the gadget will be traversal in two pieces - otherwise one chain suffices.

To avoid visiting a vertex more than once, each chain is associated with a selector vertex.

2. Now suppose we have a vertex cover of size $\leq k$.

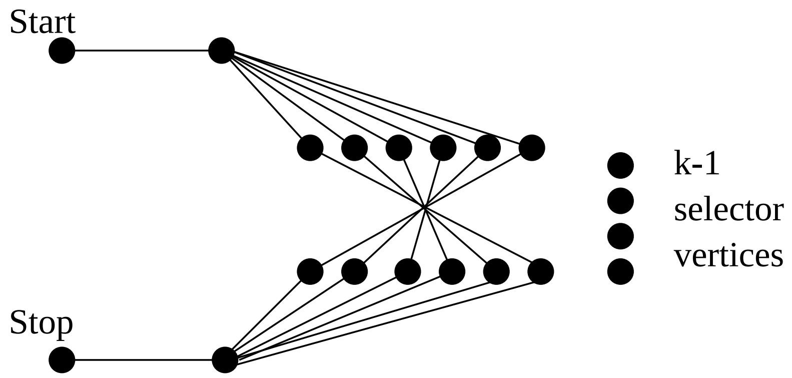
   We can always add more vertices to the cover to bring it up to size k.

   For each vertex in the cover, start traversing the chain. At each entry point to a gadget, check if the other vertex is in the cover and traverse the gadget accordingly.

   Select the selector edges to complete the circuit.

Neat, sweet, and NP-complete.

To show that Longest Path or Hamiltonian Path is *NP*-complete, add start and stop vertices and distinguish the first and last selector vertices.



This has a Hamiltonian path from start to stop iff the original graph has a vertex cover of size k.

Other NP-complete Problems

- Partition - can you partition n integers into two subsets so that the sums of the subset are equal?
- Bin Packing - how many bins of a given size do you need to hold n items of variable size?
- Chromatic Number - how many colors do you need to color a graph?
- $N \times N$ checkers - does black have a forced win from a given position?
- Scheduling, Code Optimization, Permanent Evaluation, Quadratic Programming, etc.

Open: Graph Isomorphism, Composite Number, Minimum Length Triangulation.

Polynomial or Exponential?

Just changing a problem a little can make the difference between it being in P or NP-complete:

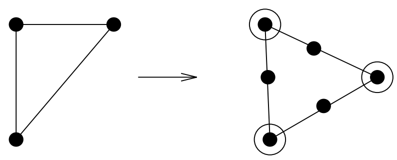
	P	NP-complete
	Shortest Path	Longest Path
	Eulerian Circuit	Hamiltonian Circuit
	Edge Cover	Vertex Cover

The first thing you should do when you suspect a problem might be NP-complete is look in Garey and Johnson, *Computers and Intractability.* It contains a list of several hundred problems known to be NP-complete. Either what you are looking for will be there or you might find a closely related problem to use in a reduction.

Techniques for Proving NP-completeness

1. *Restriction* - Show that a special case of the problem you are interested in is NP-complete. For example, the problem of finding a path of length k is really Hamiltonian Path.

2. *Local Replacement* - Make local changes to the structure. An example is the reduction $SAT \propto 3 - SAT$. Another example is showing isomorphism is no easier for bipartite graphs:

   For any graph, replacing an edge with makes it bipartite.

3. *Component Design* - These are the ugly, elaborate constructions

The Art of Proving Hardness

Proving that problems are hard is an skill. Once you get the hang of it, it is surprisingly straightforward and pleasurable to do. Indeed, the dirty little secret of NP-completeness proofs is that they are usually easier to recreate than explain, in the same way that it is usually easier to rewrite old code than the try to understand it.

I offer the following advice to those needing to prove the hardness of a given problem:

- *Make your source problem as simple (i.e. restricted) as possible.*

  Never use the general traveling salesman problem (TSP) as a target problem. Instead, use TSP on instances restricted to the triangle inequality. Better, use Hamiltonian cycle, i.e. where all the weights are 1 or ∞. Even better, use Hamiltonian path instead of cycle. Best of all, use Hamiltonian path on directed, planar graphs where each vertex has total degree 3. All of these problems are equally hard, and the more you can restrict the problem you are reducing, the less work your reduction has to do.

- *Make your target problem as hard as possible.*

  Don't be afraid to add extra constraints or freedoms in order to make your problem more general (at least temporarily).

- *Select the right source problem for the right reason.*

  Selecting the right source problem makes a big difference is how difficult it is to prove a problem hard. This is the first and easiest place to go wrong.

  I usually consider four and only four problems as candidates for my hard source problem. Limiting them to four means that I know a lot about these problems – which variants of these problems are hard and which are soft. My favorites are:

  - 3-Sat – that old reliable... When none of the three problems below seem appropriate, I go back to the source.
  - Integer partition – the one and only choice for problems whose hardness seems to require using large numbers.

- Vertex cover – for any graph problems whose hardness depends upon *selection*. Chromatic number, clique, and independent set all involve trying to select the correct subset of vertices or edges.

- Hamiltonian path – for any graph problems whose hardness depends upon *ordering*. If you are trying to route or schedule something, this is likely your lever.

- *Amplify the penalties for making the undesired transition.*

  You are trying to translate one problem into another, while making them stay the same as much as possible. The easiest way to do this is to be bold with your penalties, to punish anyone trying to deviate from your proposed solution. "If you pick this, then you have to pick up this huge set which dooms you to lose." The sharper the consequences for doing what is undesired, the easier it is to prove if and only if.

- *Think strategically at a high level, then build gadgets to enforce tactics.*

  You should be asking these kinds of questions. "How can I force that either A or B but not both are chosen?" "How can I force that A is taken before B?" "How can I clean up the things I did not select?"

- *Alternate between looking for an algorithm or a reduction if you get stuck.*

  Sometimes the reason you cannot prove hardness is that there is an efficient algorithm to solve your problem! When you can't prove hardness, it likely pays to change your thinking at least for a little while to keep you honest.

Now watch me try it!

To demonstrate how one goes about proving a problem hard, I accept the challenge of showing how a proof can be built on the fly.

I need a volunteer to pick a random problem from the 400+ hard problems in the back of Garey and Johnson.

Dealing with NP-complete Problems

Option 1: Algorithm fast in the Average case

Examples are Branch-and-bound for the Traveling Salesman Problem, backtracking algorithms, etc.

Option 2: Heuristics

Heuristics are rules of thumb; fast methods to find a solution with no requirement that it be the best one.

Note that the theory of NP-completeness does not stipulate that it is hard to get close to the answer, only that it is hard to get the optimal answer.

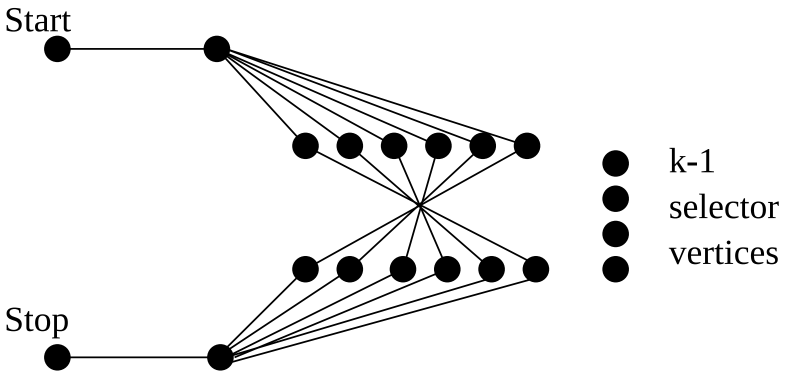
Often, we can prove performance bounds on heuristics, that the resulting answer is within C times that of the optimal one.

36.5-5 Prove that Hamiltonian Path is NP-complete.

This is not a special case of Hamiltonian cycle! (G may have a HP but not cycle)

The easiest argument says that G contains a HP but no HC iff (x, y) in G such that adding edge (x, y) to G causes to have a HC, so $O(n^2)$ calls to a HC function solves HP.

The cleanest proof modifies the VC and HC reduction from the book:



This has a Hamiltown path from start to stop
iff the orignal graph had a vertex cover of size k.

Approximating Vertex Cover

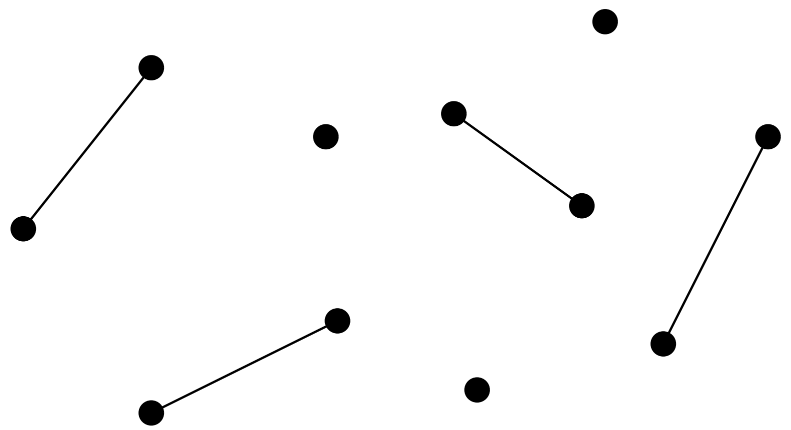
As we have seen, finding the minimum vertex cover is NP-complete. However, a very simple strategy (heuristic) can get us a cover at most twice that of the optimal.

While the graph has edges
    pick an arbitrary edge v, u
    add both u and v to the cover
    delete all edges incident on either u and v

If the graph is represented by an adjacency list this can be implemented in $O(m + n)$ time.

This heuristic must always produce cover, since an edge is only deleted when it is adjacent to a cover vertex.

Further, any cover uses at least half as many vertices as the greedy cover.



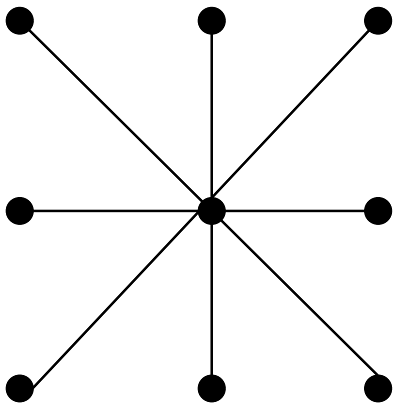
Why? Delete all edges from the graph except the edges we selected.

No two of these edges share a vertex. Therefore, any cover of just these edges must include one vertex per edge, or half the greedy cover!

Things to Notice

- Although the heuristic is simple, it is not stupid. Many other seemingly smarter ones can give a far worse performance in the worst case.

  Example: Pick one of the two vertices instead of both (after all, the middle edge is already covered) The optimal cover is one vertex, the greedy heuristic is two vertices, while the new/bad heuristic can be as bad as $n - 1$.



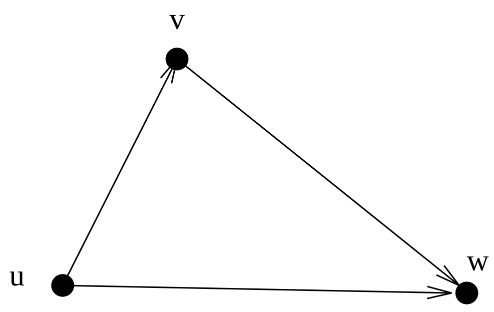
- Proving a lower bound on the optimal solution is the key to getting an approximation result.
- Making a heuristic more complicated does not necessarily make it better. It just makes it more difficult to analyze.
- A post-processing clean-up step (delete any unecessessary vertex) can only improve things in practice, but might not help the bound.

The Euclidean Traveling Salesman

In the traditional version of TSP - a salesman wants to plan a drive to visit all his customers exactly once and get back home.

Euclidean geometry satisfies the triangle inequality, $d(u,w) \leq d(u,v) + d(v,w)$.

TSP remains hard even when the distances are Euclidean distances in the plane.

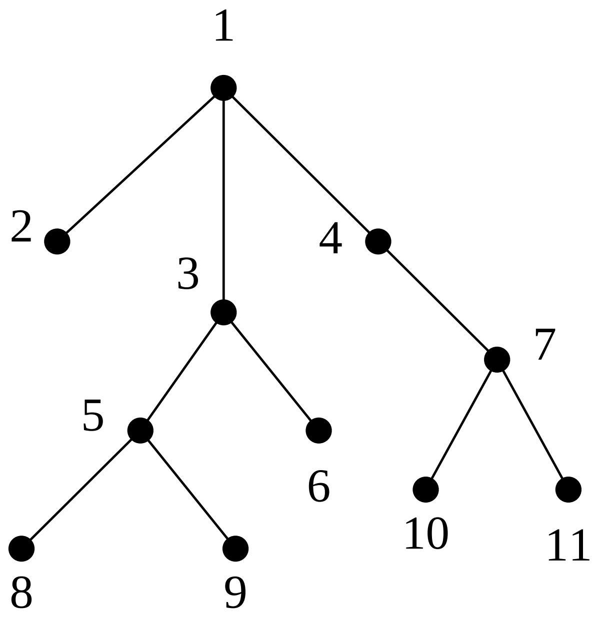


Note that the cost of airfares is an example of a distance function which violates the triangle inequality.

However, we can approximate the optimal Euclidean TSP tour using minimum spanning trees.

Claim: the cost of a MST is a lower bound on the cost of a TSP tour.

Why? Deleting any edge from a TSP tour leaves a path, which is a tree of weight at least that of the MST! If we were allowed to visit cities more than once, doing a depth-first traversal of a MST, and then walking out the tour specified is at most twice the cost of MST. Why? We will be using each edge exactly twice.



Every edge is used exactly twice in the DFS tour: *1*.

However, how can we avoid revisiting cities?

We can take a shortest path to the next unvisited vertex. The improved tour is $1-2-3-5-8-9-6-4-7-10-11-1$. Because we replaced a chain of edges by the edge, the triangle inequality ensures the tour only gets shorter. Thus this is still within twice optimal!

37.1-3 Give an efficient greedy algorithm that finds an optimal vertex cover of a tree in linear time.

In a vertex cover we need to have at least one vertex for each edge.

Every tree has at least two leaves, meaning that there is always an edge which is adjacent to a leaf. Which vertex can we never go wrong picking? The non-leaf, since it is the only one which can also cover other edges!

After trimming off the covered edges, we have a smaller tree. We can repeat the process until the tree as 0 or 1 edges. When the tree consists only of an isolated edge, pick either vertex.

All leaves can be identified and trimmed in $O(n)$ time during a DFS.

Formal Languages and the Theory of NP-completeness

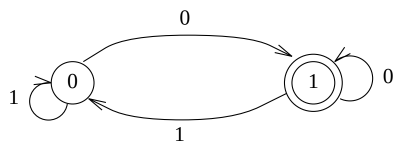
The theory of NP-completeness is based on formal languages and Turing machines, and so we will must work on a more abstract level than usual.

For a given alphabet of symbols Σ =0, 1, &, we can form an infinite set of ***strings*** or words by arranging them in any order: '&10', '111111','&&&', and '&'.

A subset of the set of strings over some alphabet is a ***formal language***.

Formal language theory concerns the study of how powerful a machine you need to recognize whether a string is from a particular language.

Example: Is the string a binary representation of a even number? A simple finite machine can check if the last symbol is zero:

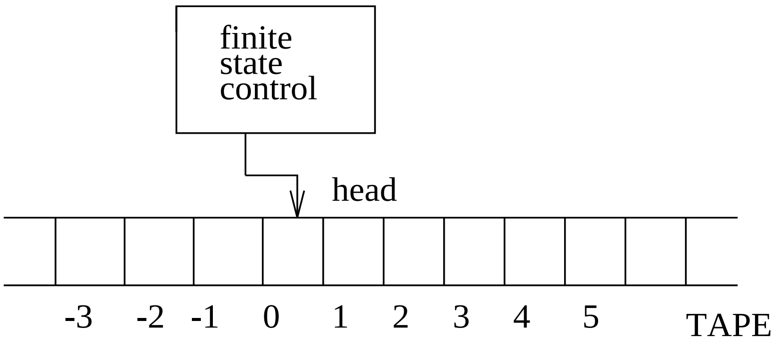


No memory is required, except for the current state.

Observe that solving decision problems can be thought of as formal language recognition. The problem instances are encoded as strings and strings in the language if and only if the answer to the decision problem is YES!

What kind of machine is necessary to recognize this language? A Turing Machine!

A Turing machine has a finite-state-control (its program), a two way infinite tape (its memory) and a read-write head (its program counter)



So, where are we?

Each instance of an optimization or decision problem can be encoded as string on some alphabet. The set of all instances which return True for some problem define a language.

Hence, any problem which solves this problem is equivalent to a machine which recognizes whether an instance is in the language!

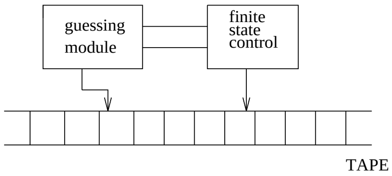
The goal of all this is going to be a formal way to talk about the set of problems which can be solved in polynomial time, and the set that cannot be.

Non-deterministic Turing Machines

Suppose we buy a *guessing module* peripherial for our Turing machine, which looks at a Turing machine program and problem instance and in polynomial time writes something it says is an answer. To convince ourselves it really is an answer, we can run another program to check it.

Ex: The Traveling Salesman Problem

The guessing module can easily write a permutation of the vertices in polynomial time. We can check if it is correct by summing up the weights of the special edges in the permutation and see that it is less than k.



The class of languages which we can recognize in time polynomial in the size of the string or a deterministic Turing Machine (without guessing module) is called P.

The class of languages we can recognize in time polynomial in the length of the string or a non-deterministic Turing Machine is called NP.

Clearly, $P \in NP$, since for any DTM program we can run it on a non-deterministic machine, ignore what the guessing module is doing, and it will just as fast.

P ?= NP

Observe that any NDTM program which takes time $P(n)$ can simulated in $P(N)2^{P(n)}$ time on a deterministic machine, by running the checking program $2^{P(n)}$ times, once on each possible guessed string.

The \$10,000 question is whether a polynomial time simulation exists, or in other words whether $P = NP$?. Do there exist languages which can be verified in polynomial time and still take exponential time on deterministic machines?

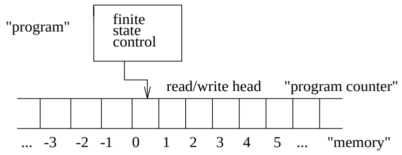
This is the most important question in computer science. Since proving an exponential time lower bound for a problem in NP would make us famous, we assume that we cannot do it.

What we can do is prove that it is at least as hard as any problem in NP. A problem in NP for which a polynomial time algorithm would imply all languages in NP are in P is called NP-complete.

Turing Machines and Cook's Theorem

Cook's Theorem proves that satisfiability is NP-complete by reducing all non-deterministic Turing machines to SAT.

Each Turing machine has access to a two-way infinite tape (read/write) and a finite state control, which serves as the program.



A program for a non-deterministic TM is:

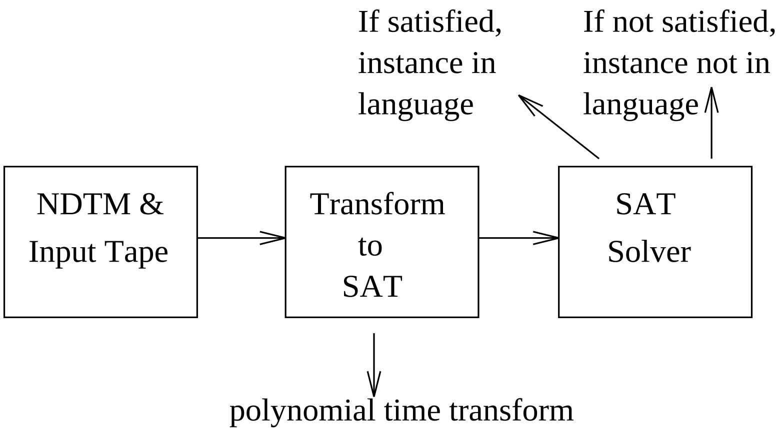
1. Space on the tape for guessing a solution and certificate to permit verification.
2. A finite set of tape symbols
3. A finite set of states Θ for the machine, including the start state q_0 and final states Z_{yes}, Z_{no}
4. A transition function, which takes the current machine state, and current tape symbol and returns the new state, symbol, and head position.

We know a problem is in NP if we have a NDTM program to solve it in worst-case time $p[n]$, where p is a polynomial and n is the size of the input.

Cook's Theorem - Satisfiability is NP-complete!

Proof: We must show that any problem in NP is at least as hard as SAT. Any problem in NP has a non-deterministic TM program which solves it in polynomial time, specifically $P(n)$.

We will take this program and create from it an instance of satisfiability such that it is satisfiable if and only if the input string was in the language.



If a polynomial time transform exists, then SAT must be NP-complete, since a polynomial solution to SAT gives a polynomial time algorithm to anything in NP.

Our transformation will use boolean variables to maintain the state of the TM:

	Variable	Range	Intended meaning
	$Q[i,j]$	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	At time i, M is in state q_k
	$H[i,j]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	At time i, the read-write head is scanning tape square j
	$S[i,j,k]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	At time i, the contents of tape square j is symbol S_k
		$0 \leq k \leq v$	

Note that there are $rp(n)+2p^2(n)+2p^2(n)v$ literals, a polynomial number if $p(n)$ is polynomial.

We will now have to add clauses to ensure that these variables takes or the values as in the TM computation.

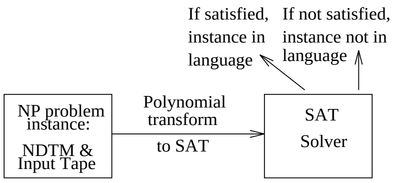
The group 6 clauses enforce the transition function of the machine. If the read-write head is not on tape square j at time i, it doesn't change ....

There are $O(p(^2(n))$ literals and $O(p^2(n))$ clauses in all, so the transformation is done in polynomial time!

Polynomial Time Reductions

A decision problem is NP-hard if the time complexity on a deterministic machine is within a polynomial factor of the complexity of any problem in NP.

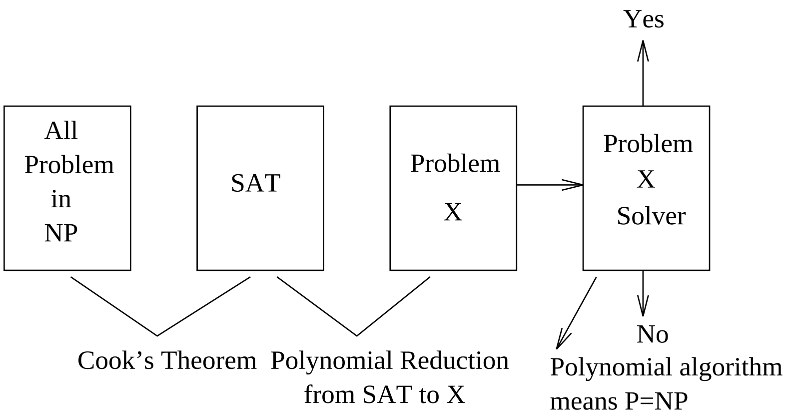
A problem is NP-complete if it is NP-hard and in NP. Cook's theorem proved SATISFIABILITY was NP-hard by using a polynomial time reduction translating each problem in NP into an instance of SAT:



Since a polynomial time algorithm for SAT would imply a polynomial time algorithm for everything in NP, SAT is NP-hard. Since we can guess a solution to SAT, it is in NP and thus NP-complete.

The proof of Cook's Theorem, while quite clever, was certainly difficult and complicated. We had to show that all problems in NP could be reduced to SAT to make sure we didn't miss a hard one.

But now that we have a known NP-complete problem in SAT. For any other problem, we can prove it NP-hard by polynomially transforming SAT to it!



Since the composition of two polynomial time reductions can be done in polynomial time, all we need show is that SAT, ie. any instance of SAT can be translated to an instance of x in polynomial time.

Finding the Optimal Spouse

1. There are up to n possible candidates we will see over our lifetime, one at a time.
2. We seek to maximize our probability of getting the single best possible spouse.
3. Our assessment of each candidate is relative to what we have seen before.
4. We must decided either to marry or reject each candidate as we see them. There is no going back once we reject someone.
5. Each candidate is ranked from 1 to n, and all permutations are equally likely.

For example, if the input permutation is

$$(4, 2, 3, 5, 6, 1)$$

we see $(3, 1, 2)$ after three candidates.

Picking the first or last candidate gives us a probability of $1/n$ of getting the best.

Since we seek maximize our chances of getting the best, it never pays to pick someone who is not the best we have seen.

The optimal strategy is clearly to sample some fraction of the candidates, then pick the first one who is better than the best we have seen.

But what is the fraction?

For a given fraction $1/f$, what is the probability of finding the best?

Suppose $i+1$ is the highest ranked person in the first n/f candidates. We win whenever the best candidate occurs before any number from 2 to i in the last $n(1-1/f)/f$ candidates.

There is a $1/i$ probability of that, so,

$$P = \sum_{i=1}^{\infty} \frac{(\frac{1}{f})(1-\frac{1}{f})^i}{i}$$

In fact, the optimal is obtained by sampling the first n/e candidates.

Index

Ω, 15
Θ, 15
3-satisfiability, 255, 280

abstract data types, 98
adjacency list, 193
adjacency matrix, 192
algorithm, what is a, 2
all-pairs shortest path, 242
approximate string matching, 170
approximation algorithms, 286
articulation vertex, 210
asymptotic notation, 23
average-case analysis, 78
average-case complexity, 9

back edge, 205
backsubstitution, 34, 46
backtracking, 153
backtracking – recursive, 154
backtracking contest, 159, 234
balanced search tree, 125
bandersnatch problem, 249
bandwidth – optimizations, 235
bandwidth problem, 159
bar codes – high density, 180
base of logarithm, 20
best-case complexity, 9, 13
big Oh notation, 14, 28
bin packing, 277
binary heap, 53, 84
binary search, 19, 48
binary search tree, 116
bipartite graphs, 213
birthday paradox, 110
bitonic tours, 185
bo-billy problem, 249
boundary condition, 31, 106
boundary conditions – string matching, 172
breadth-first search, 198, 199
breadth-first search – shortest paths, 239
breadth-first search – tree, 202
Brooks, Mel, 189
bubblesort – parallel, 148
bucketsort, 89

chaining, 111
checkers, 277
chessboard covering, 151
chromatic number, 277
circuit board assembly, 3
clique, 191
clique – hardness, 267
closest pair heuristic, 5
closest pair problem, 48
clustering, 223
coloring graphs, 213
combinatorial search, 149

communications problems, 236
comparison-based sorting, 86
component design, 279
computer graphics, 182
connected components, 200
connected graph, 190
connectivity, 210
convex hull, 50, 250
Cook's theorem, 296
correctness, 2
crime, 21
cross edge, 205
cycle, 191, 214

DAG, 207
data structures, 98
decision problems, 252, 292
decision tree, 86, 114
degree, 190
delete – binary search tree, 124
delete – red-black tree, 144
depth-first search, 198, 200
depth-first search – edge organization, 203
deque, 99
diameter, 236
dictionary data structure, 102, 116
Dijkstra's algorithm, 233, 240
directed acyclic graph, 207
directed graph, 189
discrete event simulation, 66
disjunctive normal form, 257
divide-and-conquer, 44, 69
dominance relation, 18
duplicate elimination, 49
dynamic programming, 163, 186
dynamic programming – components, 169
dynamic programming – space, 174
dynamic programming – when applicable, 177

edges, 188
edit distance, 170
element uniqueness, 49
exhaustive search algorithm, 6
exponential time, 248

Federal sentencing guidelines, 21
Fibonacci numbers, 164
flat earth model, 8
Floyd-Warshall algorithm, 246
formal language theory, 291
forward edge, 205
Fourier transform, 150
fragment assembly, 208
frequency distribution, 49
friendship graph, 189

gadget, 273
Garey and Johnson, 278
geometric algorithms, 50, 66
geometric series, 59
girth, 191
gossip, 191
graph traversal, 195
graphs, 188
greedy heuristic, 67, 163
guessing module, 294

Hamiltonian cycle, 191
Hamiltonian cycle – hardness, 273
Hamiltonian path, 281, 285

harmonic numbers, 79
hash function, 108
hash table, 107
heap data structure, 55
heapify, 56
heapsort, 61, 82
Heep, Danny, 68
heuristic, 287
Huffman codes, 51

incremental algorithms, 44
independent set, 267
induction, 31, 33
inorder traversal, 122
insert – binary search tree, 123
insert – red-black tree, 138
insertion sort, 10, 95
instance, 252
integer partition, 271
integer programming – hardness, 259, 263
inverse exponential function, 19

Kruskal's algorithm, 220, 227

linked list, 103
local replacement, 279
logarithm, 19
logic problem, 253
longest increasing subsequence, 175
longest path, 276
lower bound, 14
lower bound – sorting, 87

mainframe computers, 47
Maple, 32
master theorem, 36, 57, 69
Mathematica, 32
mathematical induction, 31
matrix multiplication, 40, 166
matrix multiplication – applications, 201
maze, 195
median, 49
mergesort, 42
minimum spanning tree, 220, 222
mode, 49
monotonic sequence, 175
morphing, 182
motion planning, 236
movie stars, 54
multigraph, 189

nearest neighbor heuristic, 4
non-comparison based sorting, 88
non-deterministic Turning machine, 294
NP, 254
NP-completeness, 247
NP-completeness – dealing with it, 284

open addressing, 112
optimization problem, 163
optimizing programs for speed, 161
order-statistic tree, 162

P vs. NP, 254, 295
parenthesization, 167
partial order, 54
partitioning step, 70
path, 189
path compression, 232
permutations – generation, 156

pivot, 70
pointers, 103
polygon triangulation, 41
polynomial time, 248
predecessor, 120
Prim's algorithm, 224
principle of optimality, 176
priority queue, 64, 119
problem, 2, 252
problem solving techniques, 93, 184
pseudocode, 7

queens problem, 157
queue, 65, 99, 198
quicksort, 70, 95
quicksort – analysis, 75

RAM model of computation, 8
random-access machine, 8
randomized algorithm, 83, 96
recurrence relations, 30
recursion tree, 35, 38, 45
red-black tree, 133
red-black trees, 127
reduction, 249, 299
reduction – direction of, 258
refrain, 25
refrigerator, 99
restriction, 279
robot arm, 3
Rolling Stones, 253
rotation, 134
rules for algorithm design, 93

satisfiability, 253
satisfiability – Cook's theorem, 297
selection, 49
selection sort, 52, 63
sentence disambiguation, 237
sentinel, 106
series convergence, 59
set data structures, 102, 132
Shifflett, 92
shortest path, 190, 236
simple graph, 189
simulations, 66
social security number, 109
solving recurrences, 32
songs, complexity of, 24
sorting, 47
sorting – applications, 48
sorting – lower bound, 114, 250
space complexity, 24
spanning tree, 222
sparse graphs, 227
spouse, finding the optimal, 301
square of graph, 201
stack, 65, 99, 198
Strassen's algorithm, 40
string matching, 170
strongly connected components, 190, 216
subgraph isomorphism, 270
subset generation, 155
subset sum, 271
successor, 120
supercomputer, 7, 150
sweepline algorithms, 66
symmetry, exploiting, 152

telephone book, 91
telephone keypad, 237
telephone network, 210, 223

terrorist, 210
text compression, 51
topological sorting, 207, 215
transportation problems, 236
traveling salesman problem, 6, 149, 176, 185, 252
traveling salesman problem – approximation, 223, 288
triangulation, 41
triangulation – minimum length, 178
Turing machine, 292

undirected graph, 189
union-find data structure, 230
upper bound, 14

vertex cover, 264, 281
vertex cover – approximation, 286
vertex cover – hardness, 264
vertex cover – tree, 290
vertices, 188
Viterbi algorithm, 238

worst-case complexity, 9