CS 341: Algorithms

Charles Shen

Fall 2016, University of Waterloo

Notes written from Jeffrey Shallit's lectures.

Feel free to email feedback to me at ccshen902@gmail.com.

Contents

1 Introduction

← 2016/09/08

A few things to review first:

- sorting algorithms (quicksort, heapsort, mergesort, counting sort, radix sort)
- lower bound for sorting in comparison-based model
- searching algorithms (binary search, interpolation search)
- data structures (trees, graphs, binary search trees, balanced binary trees, priority queues)
- hashing
- string search (Knuth-Morris-Pratt linear time)
- big-O notation

What is an algorithm?
A series of computational steps intended to solve a problem.
Algorithms should be,

- effective (can actually carry out steps)
- unambiguous — steps are not subject to interpretation
- have a finite description (only finitely many lines of code)
- finite (should eventually halt; some exceptions to this rule, such as servers or operating systems)
- deterministic (although randomized algorithms are also studied these days)

Focuses on,

- Design: what are some techniques to build algorithms to solve problems?
- Analysis: what resources are consumed by the algorithm?
  - Time
  - Space
  - Randomness
- Hardness: what problems cannot be solved with algorithms? Or cannot be solved efficiently? Lower bounds for problems.

2 Algorithm Analysis

Algorithm analysis is hard! A simple algorithm can have a very difficult or incomplete analysis.

We want a rigorous analysis of algorithms, where we count the number of steps an algorithm requires. Steps are usually measured as a function of n, the number of inputs.

The way we count steps depends on our *model of computation*.
There are many models, and most are theoretical models that are far simpler than a modern computer because they are easier to analyze!
More complicated models are much harder to analyze. Think of all the parts of a computer: the RAM, the primary cache, the secondary cache, the disk, etc. Modelling all of these precisely would be a real pain.
So we simply things by assuming that each "basic operations" takes 1 unit of time.
This is the so-called "*unit cost*" model.

The unit-cost model is good if all operations use numbers that fit in a single computer word and if you are interested in analyzing an algorithm's behaviour on n inputs as n gets large.

Another model is the "*log-cost*" model, where all basic operations are charged a cost proportional to the **size** (number of bits) of the number(s) you are dealing with; in other words, the base-2 logarithms of the number(s).
If there is more than one number, you take the logarithms of each and add them up.
When you're all done, you often phrase the result in terms of n, which is now the **number of bits of the input**.
The number of bits required to represent a positive number x is $(\lfloor \lg |x| \rfloor + 1)$ for $x \neq 0$ and 1 if $x = 0$, but there is usually no harm in just using $\lg x$ as an approximation.

Example. Suppose we have $x \times y + z$.
There are two operations: a multiplication and an addition.
The multiplication costs $\lg x + \lg y$.
In the addition, we are adding xy to z, so the cost is $\lg(xy) + \lg z$.
The total log cost is thus $\lg x + \lg y + \lg(xy) + \lg z = 2(\lg x + \lg y) + \lg z$.

Summarizing:
Use the unit-cost model if your algorithm has a variable number of inputs, each of which fits in a machine word.

Use the log-cost model if your algorithm has a single input, or where the individual inputs typically don't fit in a single machine word.

2.1 More on Algorithm Analysis

← 2016/09/13

Sometimes computing the cost of a step strongly depends on how things are implemented.

Example. Consider concatenating (joining) two lists of length n.
The cost in the unit-cost model depends entirely on *how the lists are represented*.

If the lists are represented as arrays, then you'll have to allocate a new array of length $2n$, then copy the elements of the two lists into it. All that is **non-destructive** (doesn't affect the two lists) and it would cost $\Theta(n)$.

If the lists are linked lists, and you maintain pointers to the front and back of the list, then you could concatenate two such lists, in a **destructive** way, by moving pointers.
This would cost only $\Theta(1)$.

Therefore, when estimating costs, we need to think carefully about how the items being manipulated would be represented at the very lowest level of the machine.

2.2 Big-O, Big-Theta, and so forth

Four important kinds of asymptotic notations: O, Ω, Θ, o

Definition. We say $f(n)$ is $O(g(n))$ if there exist constants $c > 0$, n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Definition. We say $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$, n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Definition. We say $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
Equivalently, $f(n)$ is $\Theta(g(n))$ if both $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Definition. We say $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.

If $f = O(g)$, then the function f *grows more slowly than* g or *has slower growth rate* than g.
Slow-growing functions corresponds to quickly-running algorithms.

Stirling's approximation:

$$n! = n^n e^{-n}\sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

2.2.1 Tricks

Trick 1:
If $\lim_{n\to\infty} f(n)/g(n) = c$, then

- if $c = 0$, then $f = o(g)$;
- if $0 < c < \infty$, then $f = \Theta(g)$; and
- if $c = \infty$, then $g = o(f)$

Trick 2:
$n^b = o(a^n)$ for all real numbers a and b such that $a > 1$.
$(\lg n)^b = o(n^a)$ for every fixed positive number a and every fixed real number b.

Trick 3:
L'Hôpital's Rule, which says that if f and g are differentiable functions, and *both* f and g tend to ∞ as n tends to ∞, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Trick 4:
If you're trying to prove $f(n) = O(g(n))$, sometimes it's easier just to prove an explicit inequality, such as $f(n) < cg(n)$ for some appropriate explicit constant c.

Trick 5:
Use fact that if $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$ and $f_1 f_2 = O(g_1 g_2)$.

Trick 6:
If $f(n), g(n) \geq 1$ for all sufficiently large n and $f = o(g)$ then $e^f = o(e^g)$.

2.3 Polynomial Time

Definition. An algorithm runs in *polynomial time* if there exists a $k \geq 0$ such that its worst-case time complexity $T(n)$ is $O(n^k)$.

We generally think of an algorithm as "good" if it runs in polynomial time.

3 Maximum Subrange Sum

Given an array $x[1..n]$ of integers (possibly *negative*) and we want to find the block of contiguous entries with the largest sum.

3.1 Naive and Still Naive

Enumerate all possible subblocks of the array,

```
maxsubrangesum1(x, n):
  max = 0
  for lower = 1 to n do
    for upper = lower to n do
      sum = 0
      for i = lower to upper do
        sum = sum + x[i]
      if sum > max then
        max = sum
```

Run time: $O(n^3)$.

Slightly better:

4 Selection in Deterministic Linear Time

← 2016/10/25

In the selection problem, the goal is to determine the i-th smallest element from an unsorted list of n elements in linear time.
"Deterministic" here means that no random numbers are used.
For simplicity, suppose that all elements are distinct, but this is not crucial as a small modification will make it work any list.

One method to choose a pivot is via randomness.
We want a deterministic method to clever choose a pivot to be used to partition a list.
Thus we arrive at the `medians-of-five` algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan.

Idea:
Split the list into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements, and one additional group of at most 4 elements.
Then sort each group of 5 elements, and find the median of each group of 5. This can be done in constant time, so sorting each group of 5 costs $O(n)$.
Note that that median is the 3rd (center) element in the group of 5. This gives a list of $\lfloor \frac{n}{5} \rfloor$ medians.
Now recursively determine the median of these $\lfloor \frac{n}{5} \rfloor$ medians, which is the element we'll partition around.

We arrive at the algorithm:

function SELECT(A, i)
  $n \leftarrow |A|$
  if $n < 60$ **then**
    SORT(A)
    return i-th smallest element
  else
    $m \leftarrow \lfloor \frac{n}{5} \rfloor$
    divide A up to m groups of 5 elements, with at most one remaining group of ≤ 4 elements
    sort each of the m groups in ascending order
    $M \leftarrow$ array of medians of each group
    $x \leftarrow$ SELECT(M)$\lceil m/2 \rceil$ ▷ median of all the medians
    $k \leftarrow$ X-PARTITION(A, x) ▷ partition array A into elements $\leq$ x and elements $>$ x; returns number of elements on "low side" of the partition
    if $i = k$ **then**
      return x
    else if $i < k$ **then**
      return SELECT($A[1..k-1], i$)
    else
      return SELECT($A[k+1..n], i-k$)
    end if
  end if
end function

Let's obtain a lower bound on the number of elements $> x$, the median of medians.
Here's a picture for $n = 37$:

```
smallest   S    S    S    S    *    *    *    *

           S    S    S    S    *    *    *    *

           S    S    S    m    L    L    L

           *    *    *    L    L    L    L

largest    *    *    *    L    L    L    L

m = median of medians
```

There are $\left\lfloor \frac{n}{5} \right\rfloor$ total columns in which 5 elements appear.
Of these, at least $\frac{1}{2}$ (more precisely, $\lceil \lfloor n/5 \rfloor /2 \rceil$) contain an L. All of these columns, except the one where x appears, contributes 3 to the count of L's; the one where x appears contributes 2.

The conclusion is that $\geq 3 \lceil \lfloor n/5 \rfloor /2 \rceil - 1$ elements are L's, that is, greater than x. Hence, at most $\leq n - (3 \lceil \lfloor n/5 \rfloor /2 \rceil - 1)$ elements are $\leq x$.

Then, $n - (3 \lceil \lfloor n/5 \rfloor /2 \rceil - 1) \leq 7n/10 + 3$ for all $n \geq 1$. To prove this, see that

$$\lfloor n/5 \rfloor \geq n/5 - 1$$

so

$$ceilfloorn/5/2 \geq n/10 - 1/2$$

then

$$3 \lceil \lfloor n/5 \rfloor /2 \rceil - 1 \geq 3n/10 - 5/2$$

hence

$$n - (3 \lceil \lfloor n/5 \rfloor /2 \rceil - 1) \leq 7n/10 + 3$$

We can also claim that at most $(7n/10 + 3)$ elements are $\geq x$ using the same proof above.
So whether we recurse in the smaller elements or larger elements, we can use this bound.

It follows that the time $T(n)$ to select from a list of n elements satisfies the following inequality:

$$T(n) \leq T(\lfloor n/5 \rfloor) + T(r) + dn$$

where $r \leq 7n/10 + 3$, and the dn term soaks up the time needed to do the sorting of 5-element groups, and the partitioning.

Suppose that this recurrence obeys $T(n) \leq cn$ for some c,

$$T(n) \leq T(\lfloor n/5 \rfloor) + T(r) + dn$$

$$\leq cn/5 + cr + dn$$
$$\leq cn/5 + c(7n/10 + 3) + dn$$
$$\leq 9cn/10 + 3c + dn$$

and we want this to be less than cn.

Now, $9cn/10 + 3c + dn \leq cn \iff (c/10 - d)n \geq 3c$
Now n had better be bigger than 30 here, or else we are in a heap o' trouble. So let's assume $n \geq 60$. Then $(c/10 - d)n \geq (c/10 - d)60$, and if this is to be $\geq 3c$, then we must have $6c - 60d \geq 3c$, and so $c \geq 20d$.
If $n \geq 60$, and $c \geq 20d$, then the induction step indeed works.

To complete the proof, it remains to see that $T(n) \leq cn$ for $n = 1, 2, 3, \ldots, 59$.
To do this, simply choose

$$c = max(20d, T(1), T(2)/2, ..., T(59)/59)$$

That looks like cheating, but it works.
Now the basis (namely $1 \leq n \leq 59$) works out just fine, and so does the induction step.
The presence of these large constants (such as $c = 20d$) suggests that this algorithm is of more theoretical than practical interest.
However, with a bit more work, the number of comparisons can be reduced quite a lot, down to $(3 + o(1))n = 3n + o(n)$.

5 Lower bounds

Can we prove that an algorithm is best possible? Sometimes.

We argued that any comparison-based algorithm for sorting must use $\Omega(n \lg n)$ time. And in reference to the convex hull algorithm, we argued that we couldn't do better than $\Omega(n \lg n)$, because we could use convex hull as a subroutine to create a sorting algorithm; if we could solve convex hull in $o(n \lg n)$ time, then we would end up being able to sort in $o(n \lg n)$ time, a contradiction.

By "lower bounds", we mean a lower bound on the complexity of a *problem*, not an algorithm. Basically we need to prove that *no algorithm*, no matter how complicated or clever, can do better than our bound.
We don't need each algorithm to be "slow" as $f(n)$ on the *same* input, but we do need each algorithm to be "slow" as $f(n)$ on *some* input.
Also, a lower bound doesn't rule out the possibility that some algorithm might be very fast on *some inputs* (but not all).
What do we mean by complexity of a problem?
We say that a function f is a lower bound on the complexity of a problem P if, for *every algorithm* A to solve P, and every positive integer n, there exists some input I of size n such that A uses at least $f(n)$ steps on input I.

The easiest kind of lower bound is the "information-theoretic bound".
Idea: every algorithm that uses *only* yes/no questions to decide among k possible alternatives, *must* ask at least $(\lg k)$ questions in the worst case.

Proof. Use an "adversary argument".
The adversary doesn't decide on one of the alternatives ahead of time, but rather it answers the algorithm's questions by always choosing the alternative that maximizes the size of the solution space.
Since the questions are yes/no, the solution space goes down in size by a factor of at most two with each question.
The algorithm cannot answer correctly with certainty unless the solution space is of size 1.
Thus at least $(\lg k)$ questions are needed. □

Theorem 5.0.1. *Any comparison-based sorting algorithm must use at least* $\lg(n!) = \Omega(nlogn)$ *questions in the worst-case.*

Proof. Use the information-theoretic bound, and observe that the space of possible solutions includes the $n!$ different ways the input could be ordered. □

6 Adversary Strategy

6.1 Adversary Arguments in General

More generally, we can think of a lower bound proof as a game between the algorithm and an "adversary".
The algorithm is "asking questions" (for example, comparing two elements of an array), while the adversary is answering them (providing the results of the comparison).

The adversary should be thought of as a very powerful, clever being that is answering in such a way as to make your algorithm run as slowly as possible.
The adversary cannot "read the algorithm's mind", but it can be prepared for anything the algorithm might do.
Finally, the adversary is not allowed to "cheat"; that is, the adversary cannot answer questions inconsistently. The adversary does not not need to have a particular input in mind at all times when it answers the questions, but when the algorithm is completed, there must be at least one input that matches the answers the adversary gave (otherwise it would have cheated).

The algorithm is trying to run as quickly as possible.
The adversary is trying (through its cleverness) to *force* the algorithm to run slowly.
This interaction proves that the lower-bound obtained by this type of argument applies to *any possible* algorithm from the class under consideration.

Theorem 6.1.1. *Every comparison-based algorithm for determining the minimum of a set of n elements must use at least $(\frac{n}{2})$ comparisons.*

Proof. Every element must participate in at least one comparison; if not, the not compared element can be chosen (by an adversary) to be the minimum.
Each comparison compares 2 elements.
Hence, at least $(\frac{n}{2})$ comparisons must be made. □

Theorem 6.1.2. *Every comparison-based algorithm for determining the minimum of a set of n elements must use at least $(n-1)$ comparisons.*

Proof. To say that a given element, x, is the minimum, implies that (1) every other element has won at least one comparison with another element (not necessarily x). (By "x wins a comparison with y" we mean $x > y$.)
Each comparison produces at most one winner.
Hence at least $(n-1)$ comparisons must be used.

To convert this to an adversary strategy, do the following: for each element, record whether it has won a comparison (W) or not (N).
Initially all elements are labelled with N.
When we compare an N to an N, select one arbitrarily to be the W.
If no values assigned, choose them to make this so.
If values are assigned, decrease the loser (if necessary) to make this so.

When we compare W to N, always make the N the loser and set or adjust values to make sure this is the case. (You might need to decrease the loser's value to ensure this.)
When we compare W to W, use the values already assigned to decide who is the winner.

Now each comparison increases the total number of items labelled W by at most 1.
In order to "know" the minimum, the algorithm must label $(n-1)$ items W, so this proves at least $(n-1)$ comparisons are required.
The values the adversary assigned are the inputs that forced that algorithm to do the $(n-1)$ comparisons. □

6.2 Finding Both the Maximum and Minimum of a List of n Numbers

It is possible to compute both the maximum and minimum of a list of n numbers, using $(\frac{3}{2}n-2)$ comparisons if n is even, and $(\frac{3}{2}n-\frac{3}{2})$ comparisons if n is odd.
It can be proven that *no comparison-based method* can correctly determine both the max and min using fewer comparisons in the worst case.

We do this by constructing an adversary argument.

In order for the algorithm to correctly decide that x is the minimum and y is the maximum, it must know that

1. every element other than x has won at least one comparison (i.e. $x > y$ if x wins a comparison with y),

2. every element other than y has lost at least one comparison

Calling a win W and a loss L, the algorithm must assign $(n-1)$ W's and $(n-1)$ L's. That is, the algorithm must determine $(2n-2)$ "units of information" to always give the correct answer.

We now construct an adversary strategy that will force the algorithm to learn its $(2n-2)$ "units of information" as slowly as possible.
The adversary labels each element of the input as N, W, L, or WL. These labels may change over time.
N signifies that the element has never been compared to any other by the algorithm.
W signifies the element has won at least one comparison.
L signifies the element has lost at least one comparison.
WL signifies the element has won at least one and lost at least one comparison.

← 2016/10/27

So the the adversary uses the following table.

labels when comparing elements (x, y)	the adversary's response	the new label	unit of information given to the alg.
(N, N)	$x > y$	(W, L)	2
(W, N) or (WL, N)	$x > y$	(W, L) or (WL, L)	1
(L, N)	$x < y$	(L, W)	1
(W, W)	$x > y$	(W, WL)	1
(L, L)	$x > y$	(WL, L)	1
(W, L) or (WL, L) or (W, WL)	$x > y$	no change	0
(WL, WL)	consistent with assigned values	no change	0

The adversary also tentatively assigns values to the elements, which may change over time. However, they can only change in a fashion *consistent* with previous answers. That is, an element labelled L may only *decrease* in value (since it lost all previous comparisons, if it is decreased, it will still lose all of them), and an element labelled W may only increase in value. An element labelled WL cannot change.

Example. Consider the following sequence of possible questions asked by the algorithm and the adversary's responses:

Algorithm compares	Adversary's responses and worksheet x1	x2	x3	x4	x5	x6	
(x1, x2)	W-20	L-10	N	N	N	N	
(x4, x5)	W-20	L-10	N	W-30	L-15	N	
(x1, x4)	W-40	L-10	N	WL-30	L-15	N	(*)
(x3, x6)	W-40	L-10	W-11	WL-30	L-15	L-2	
(x2, x5)	W-40	WL-10	W-11	WL-30	L-7	L-2	
(x3, x1)	WL-40	WL-10	W-50	WL-30	L-7	L-2	
(x2, x4)	WL-40	WL-10	W-50	WL-30	L-7	L-2	(**)
(x5, x6)	WL-40	WL-10	W-50	WL-30	WL-7	L-2	

At this point the algorithm knows that x_3 is the maximum, since it is the only element that has never lost a comparison, and x_6 is the minimum, since it is the only element that has never won a comparison.
Note that in step (*), the adversary was forced to reassign the value he had previously assigned to x_1, since x_1 had to win the comparison against $x_4 = 30$.
This is permitted, since x_1 had won every previous comparison and so increasing its value

ensures consistency with previous answers.
Also note that step (**) is superfluous, as the algorithm didn't learn any new information.

Theorem 6.2.1. *Every* comparison-based *method for determining both the maximum and minimum of a set of n numbers must use at least $(\frac{3}{2}n - 2)$ comparisons (if n even), or $(\frac{3}{2}n - \frac{3}{2})$ comparisons (if n odd), in the worst case.*

Proof. Suppose that n is even.
As shown before, the algorithm must learn $(2n - 2)$ units of information.
The most it can learn in one comparison is 2 units; when both elements have never participated before in a comparison, labelled by (N, N). This can happen at most $(\frac{n}{2})$ times.
To learn the remaining $(n - 2)$ units of info., the algorithm must ask $(n - 2)$ questions.
The total number of questions is therefore at least $\frac{n}{2} + n - 2 = \frac{3}{2}n - 2$.
The same kind of argument works for n odd. □

So we have a lower bound for the problem of finding both the maximum and minimum of a set of n numbers.

So the algorithm is as followed:
If n is even, compare $x[1]$ with $x[2]$ (determining both the maximum and minimum with 1 comparison), $x[3]$ with $x[4]$, etc. We get $(\frac{n}{2})$ maxima and $(\frac{n}{2})$ minima.
To find the maximum, use the usual method on the $(\frac{n}{2})$ maxima, which uses $(\frac{n}{2} - 1)$ comparisons, and the same for the minima.
The total cost is $\frac{n}{2} + \frac{n}{2} - 1 + \frac{n}{2} - 1 = \frac{3}{2}n - 2$. A similar method works when n is odd.

Notice that each comparison adds at most one W, and we need to get $(n - 1)$ W's before we know the remaining element must be the minimum.
So we need at least $(n - 1)$ comparisons.
If the algorithm declares a minimum before getting $(n - 1)$ W's, then there are at least two elements without W, so if the algorithm declares one of the minimum, the adversary can truthfully produce the other as the minimum.

6.3 Remarks on Constructing an Adversary Strategy

Lower bound arguments are some of the most difficult and deepest areas in theoretical computer science.

Tips for producing an adversary strategy:

Recall that the lower bound argument can be viewed as a game between you (playing the role of an algorithm) and an adversary.
You want the algorithm to run quickly (for example, by making a small number of queries to the input).
The adversary wants the algorithm to run slowly.
The adversary "wins" (and the lower bound of $f(n)$ is proved) if he/she can force the algorithm

to execute $f(n)$ steps.

A proof behaves something like: for all sequences of queries into the input made by the algorithm, there exists a sequence of replies by the adversary, such that the algorithm is forced to execute f(n) steps. So your adversary argument must apply to *every* possible "move" the algorithm could make.

In constructing its replies, the adversary can do as much or as little bookkeeping as necessary. There is no requirement that the adversary's computation be efficient or run in any particular time bound.

6.4 Finding the Second Largest

Problem: given a list of n elements, find the 2nd largest.

The naive method (finding the largest, remove it, and then find the largest again) takes $2n - 3$ comparisons.
Instead, we can do a "tournament" method.
Have the elements in pairs, the larger one (winner) advances; then the 2nd largest (runner-up) is among those defeated by the winner! This is because any other element must have played two other elements superior to it.

This algorithm uses $(n - 1)$ comparisons to determine the largest, plus $\lceil \lg n \rceil - 1$ comparisons to determine the runner-up (2nd largest).
The total is $(n + \lceil \lg n \rceil - 2)$ comparisons.

We can prove that the algorithm just given is *optimal*, in the sense that no comparison-based algorithm uses fewer than $(n + \lceil \lg n \rceil - 2)$ comparisons in the worst case.

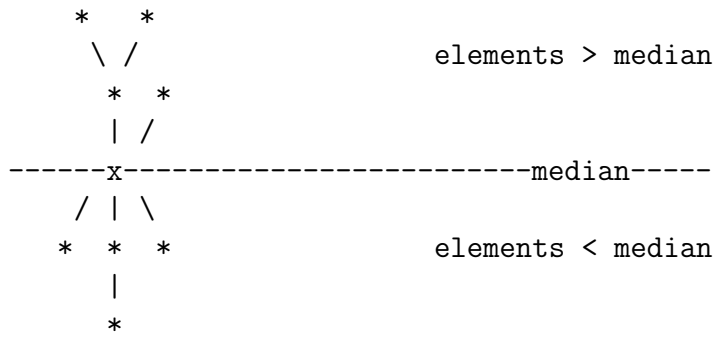
Note: see Lecture 14 for full proof! (Click here.)

6.5 A Lower Bound for Finding the Median

The best algorithm known (in terms of number of comparisons used) uses $3n+o(n)$ comparisons in the worst case (see Section 4).
The best *lower bound* known for the median-finding problem is $2n + o(n)$ comparisons.

We will prove a weaker lower bound: $\frac{3}{2}n + O(1)$, where n odd for convenience.

First, note that any comparison-based algorithm for finding the median must use at least $(n - 1)$ comparisons. That's because the algorithm *must* establish the relationship of each element to the median—either by direct comparison, or by comparison with another element whose relationship to the median is known.
The algorithm *must* eventually gain enough knowledge equivalent to the following tree:



Note that if there were an element that didn't participate in a comparison, we couldn't know the median.
For an adversary could change the value of that element from, say, less than the median to greater than the median, changing the median to some other element.
Similarly, if there is some element whose relationship to the median is not known (because, for example, it is greater than some element that is less than the median), we could simply move it from one side of the line above to the other, without changing comparisons. This would also change the median.

Now there are n elements in the tree above, so there are $(n-1)$ edges, and hence $(n-1)$ comparisons.
We now improve this to $\frac{3}{2}n + O(1)$.
First, we call a comparison involving x crucial, say x against y, if it is the first comparison where

1. $x < y$ and $y \leq$ median OR if
2. $x > y$ and $y \geq$ median

So the algorithm must perform at least $(n-1)$ crucial comparisons in order to determine the median.

← 2016/11/01

We now give an adversary strategy to *force* the algorithm to perform $(n-1)/2$ *non-crucial* comparisons (since we've proved earlier that the algorithm requires $(n-1)$ *crucial* comparisons.

Idea:
First, the adversary picks some value (*not* some element) to be the median.
Then it assigns a label in $\{N, L, S\}$ to each element, and also values, as appropriate.
Initially each element is assigned an N ("Never participated in any comparison"). L means Larger than the median and S means Smaller than the median.
The adversary follows the following strategy:

Algorithm compares	Adversary responds
(N, N)	assign one element to be larger than the median, one smaller; result is (L, S)
(S, N) or (N, S)	assign the N-element to be larger than the median; result is (S, L) or (L, S)
(L, N) or (N, L)	assign the N-element to be smaller than the median; result is (L, S) or (S, L)
(S, L) or (L, S) or (S, S) or (L, L)	consistent with previously assigned values

Note 1: this strategy continues until $((n-1)/2)$ S's (or $((n-1)/2)$ L's) have been assigned. If at some point $((n-1)/2)$ S's are assigned, then the adversary assigns the remaining elements to be greater than the median, except for one, which *is* the median. A similar thing is done if $((n-1)/2)$ L's have been assigned.

Note 2: the last element assigned is *always* the median.

I claim that this strategy will always force the algorithm to perform $((n-1)/2)$ non-crucial comparisons.
For any time an N-element is compared, a non-crucial comparison is done (except at the very end, when a crucial comparison may be done with the median itself).
The least number of comparisons with N-elements that can be done is $((n-1)/2)$.
The *total* number of comparisons is therefore $n - 1 + (n-1)/2 = (3/2)(n-1)$.

Example. The adversary strategy when $n = 7$. The adversary arbitrarily chooses the median to be 43. Then

```
-----------------------------------------------------
Algorithm   Adversary's strategy (label:value)
compares    x1    x2    x3    x4    x5    x6    x7
-----------------------------------------------------
            N     N     N     N     N     N     N
(x1, x3)   L:50   N    S:31   N     N     N     N
(x3, x4)   L:50   N    S:31  L:47   N     N     N
(x6, x7)   L:50   N    S:31  L:47   N    L:60  S:20

-------  at this point there are (7-1)/2 L's -------
-------  assigned, so adversary assigns the --------
-------  remaining elements to L and the median ----

           L:50  43    S:31  L:47  S:15  L:60  S:20
(x2, x5)   a crucial comparison
(x2, x1)   a crucial comparison
(x2, x3)   a crucial comparison
(x2, x4)   a crucial comparison
```

```
(x2, x6)   a crucial comparison
(x2, x7)   a crucial comparison
```

At this point the algorithm concludes that `x2` is the median.
The first three comparisons were non-crucial; only the last six are crucial.

The best lower bound known for the median: Dor and Zwick, Median selection requires $(2+\epsilon)n$ comparisons.

6.6 Lower Bound on Average-Case Sorting

Theorem 6.6.1. *Any comparison-based algorithm must use* $\Omega(n \lg n)$ *comparisons on* average.

Lemma 6.6.2. In any binary tree T where each node has 0 or 2 children,

$$\sum_{\ell \text{ a leaf of } T} 2^{-\text{depth}(T,\ell)} = 1$$

where $\text{depth}(T, \ell)$ is the length of the path from ℓ to the root.

Proof. Prove by induction on the height of T.
Easy to see the lemma holds when T consists of a single node.
Assume it holds for all tree of height $< k$; prove it for trees of height k.
Then a tree of height k consists of a root and two subtrees, T' and T''.
Then the lemma applies to each of these subtrees recursively, so we know that

$$\sum_{\ell \text{ a leaf of } T'} 2^{-\text{depth}(T',\ell)} = 1$$

and

$$\sum_{\ell \text{ a leaf of } T''} 2^{-\text{depth}(T'',\ell)} = 1$$

Then we have

$$\sum_{\ell \text{ a leaf of } T'} 2^{-\text{depth}(T,\ell)} = \frac{1}{2}$$

since the depth of a leaf in T goes up by 1 when considered as a leaf of T.
The same result holds for T''.
Adding these together yields the result. □

Lemma 6.6.3. Suppose $a, b \in \mathbb{R}$ and $a, b > 0$.
If $a \neq b$, then we have

$$\lg a - \lg b > 2 \lg((a + b)/2)$$

Proof. If $a \neq b$, then

$$(\sqrt{a} - \sqrt{b})^2 > 0$$
$$a - 2\sqrt{ab} + b > 0$$

$$\begin{aligned}(a+b)/2 &> \sqrt{ab} \\ \lg((a+b)/2) &> \frac{1}{2}\lg(ab) \\ 2\lg((a+b)/2) &> \lg a + \lg b \\ -2\lg((a+b)/2) &< -\lg a - \lg b\end{aligned}$$

as required. □

Lemma 6.6.4. Suppose $x_1, x_2, \ldots, x_k \in \mathbb{R}$ and $x_1, x_2, \ldots, x_k > 0$ such that $\sum_{1 \le i \le k} x_i = 1$. Then

$$\sum_{1 \le i \le k} (-\lg x_i) \ge k \lg k$$

Proof. If all the $x_i = \frac{1}{k}$, then the inequality clearly holds - actually then it is an equality!
Suppose we claim this sum actually achieves its minimum at this point.
For if the minimum occurred at some *other* point, then two of the x_i must be different, say a and b.
Then by Lemma 3.6.3, we could replace $(-\lg a) + (-\lg b)$ with two copies of $-\lg((a+b)/2)$ and get a smaller sum, a contradiction. □

Proof. Consider the comparison tree used by the algorithm.
There are at least $k = n!$ leaves, since there are are $n!$ possible ways to order n numbers.
There are also at most this number of leaves, since otherwise two leaves would be labelled with the same ordering, and then, by tracing up to the lowest common ancestor, we would get to a node where there are two different answers, implying the orderings cannot be the same.
So there are exactly $n!$ leaves.
By Lemma 3.6.2,

$$\sum_{\ell \text{ a leaf of } T} 2^{-\text{depth}(\ell)} = 1$$

Then by Lemma 3.6.4, with $x_i = 2^{-\text{depth}(\ell)}$, we get

$$\sum_{\ell \text{ a leaf of } T} \text{depth}(\ell) \ge n! \lg(n!)$$

It follows that

$$\frac{1}{n!} \sum_{\ell \text{ a leaf of } T} \text{depth}(\ell) \ge \lg(n!)$$

but the expression on the left is just the average depth of each leaf, i.e., the average number of comparisons used (averaged over all possible arrangements of the input).
Hence the average number of comparisons used is $\Omega(n \lg n)$. □

7 Graphs and Graph Algorithms

A graph consists of a bunch of points, called vertices (or nodes), and a bunch of connections between pairs of points, called edges (or arcs).
So $G = (V, E)$, where V is a set of vertices and E is a subset of $V \times V$.
Where $|V|$ is the number of vertices and $|E|$ is the number of edges in G.

Graphs can be directed or undirected; in a directed graph, edges have a direction, and you can only traverse the graph by going in that direction. In an undirected graph, edges can be traversed in any direction.

Sometimes we allow graphs to have edges from a vertex to itself; such an edge is called a "loop" or a "self-loop" and the graph is no longer *simple*.
Most graphs in this course won't have such edges, we will allow self-loops for directed graphs and only sometimes for undirected graphs. We will rarely have two or more different edges between the same two vertices.

Note: in any graph, $|E| = O(|V|^2)$, and if the graph is connected, $|E| = \Omega(|V|)$.

A *path* is a series of vertices where each consecutive vertex is connected by an edge to the previous vertex.
An undirected graph is *connected* if there is a path from every vertex to every other vertex.
The *length* of a path is the number of edges in the path.
A path is a *cycle* if the first vertex is the same as the last vertex. Usually in a path we demand that there are no repeated vertices (with the exception in a cycle).
A *walk* (resp., *closed walk*) is a path (resp., cycle) where repeated vertices and edges are allowed.
A graph is *acyclic* if there are no cycles.

A *tree* is a connected acyclic graph.
An undirected graph is a tree if every pair of vertices are connected by a *unique* simple path.
Trees can be rooted (have a distinguishable vertex called the root) or unrooted.
A *forest* is a vertex-disjoint collection of 1 or more trees.

Two ways to store a graph:

1. **Adjacency Matrix**
   We represent a graph by a $|V| \times |V|$ matrix with a 1 in row i and column j is (i, j) is an edge, and 0 otherwise.

2. **Adjacency List**
   We represent a graph by an array Adj of $|V|$ lists, one list for each vertex.
   For each vertex Adj[u] contains all vertices (or pointers to the vertices) v such that uv is an edge.

Advantages and disadvantages of each representation:
Adjacency Matrix:

- easy to manipulate
- can check if uv is an edge in $O(1)$ time
- but storage is always $\Omega(|V|^2)$—not good for sparse graphs
- time to add an edge:
  - existing vertex and new edge: $O(1)$ time
  - existing vertex and existing edge: $O(1)$ time, can check in $O(1)$ time
  - new vertex and new edge: $O(|V|^2)$ time, create a new, bigger matrix

Adjacency List:

- harder to manipulate
- checking if uv is an edge takes $O(|V|)time$
- but storage is $O(|V| + |E|)$
- time to add an edge:
  - existing vertex and new edge: $O(|V|)$ time
  - existing vertex and existing edge: $O(|V|)$ time
  - new vertex and new edge: $O(|V|)$ time

Sometimes there are weights on the edges.
In this case we can represent a weighted graph by a $|V| \times |V|$ matrix with a number x in row i and column j if the edge from i to j has weight x.
If i is not directly connected to j, then the weight is typically taken to be infinite.
Or we can use the adjacency list representation, with weights given together with the vertices in a list.

The *weight of a path* is the sum of the weights of all the edges traversed along that path.

7.1 Minimum Spanning Tree

Note: here we are working with undirected graphs.
A spanning tree is an (unrooted) tree that connects all the vertices of a graph.
Every connected graph has at least one spanning tree; typically, a graph can have many, many different spanning trees.
If there are weights on the edges, a minimum spanning tree is one that minimizes the sum of the weights of the edges used.
Note that edge weights need not be distances! They can even be 0 or negative.
A graph can have more than one minimum spanning tree.

If the graph is not connected, we sometimes talk about spanning *forests* instead of spanning trees.

Minimum spanning trees come up all the time in network problems.
Here the nodes represent nodes of a network that need to be connected, and the weights represent distances (or perhaps, amounts of wire required) between the vertices.
We want to hook up all the nodes in our network using the smallest total distance (or amount of wire).

← 2016/11/03

Algorithms for *minimum spanning tree*:

Kruskal: many different trees that are eventually joined up to form one tree

Prim: one tree that grows by sequentially adding edges

Neither Kruskal nor Prim work for directed graphs.
That problem is called the "minimum cost arborescence" problem; it can be solved in $min(n^2, m \lg n)$ time.

7.1.1 Kruskal's Algorithm

We consider each edge in turn, starting with the lowest-weight edge and continuing in ascending order by weight. We only add the edge if it doesn't create a cycle.
It's a greedy algorithm where, at each step, we choose the minimum-weight edge that doesn't create a cycle.

To implement Kruskal's algorithm, we need to consider the edges in turn, ordered in ascending order by weight; which we can do by sorting the edges.
Then when we pick an edge uv to consider, we have to decide if they are in different trees, or the same tree. This is not so easy! But it is an example of the "disjoint-set data structure problem."

In the disjoint-set data structure problem, we have three operations:

`MAKE-SET`: create a set containing a single vertex

`FIND`: given a vertex, find the "name" of the set it is in (sets are disjoint so only one name is possible for each vertex)

`UNION`: union together two sets, possibly giving the resulting set a new name (destructive)

Due to the two main operations, this is also called the "union-find" problem.

It is possible to do m `MAKE-SET`, `UNION` and `FIND` operations,n of which are `MAKE-SET`, in $O(m\ \alpha(n))$ time.
$\alpha(n)$ is an extremely slow-growing function called the "inverse Ackerman" function; the function grows so slowly that $\alpha(n) \leq 4$ for all values of n that one would ever encounter in real life.

How Kruskal's algorithm may be implemented:

```
function KRUSKAL(G, w)                    ▷ G = (V, E) and w is the weight function
    A ← empty set
    for each v in V do
        MAKE-SET(v)
    end for
    sort E into increasing order of weight
    for each edge uv in E, in increasing order do
        if FIND(u) ≠ FIND(v) then
            A ← A ∪ {uv}
            UNION(u, v)
        end if
    end for
    return A
end function
```

`MAKE-SET(v)` creates a new set whose only element is v.
`FIND(u)` returns the name of the set that u is contained in.
`UNION(u,v)` unions together the trees containing u and v and gives the result a name (which can be found with `FIND`).

Sets are stored as linked lists of vertices.
Each element in the list has a pointer to its next element, and a pointer to the head of the list.
There is also a tail pointer to the end of the list, and the list maintains how many elements it contains.
The "name" is the first vertex in the list.

To do `MAKE-SET`, we create a linked list of a single element, and the `num` field set to 1. This costs $O(1)$.

To do a `FIND`, we follow the pointer to the head of the list and return the element there. This costs $O(1)$.

To do a `UNION(u,v)`, we follow the pointer of u to its head, and the pointer of v to its head.
There we find the number of elements in each list.
We can do this via "weighted-union heuristic."
If (say) u is in the smaller list, we link that list to the end of the bigger list (v's list).
Then traverse u's list, moving each head pointer to point to the head of v's list.
Finally, we update the `num` field of v's list.
This costs $O(t)$, where t is the number of vertices of the smaller list.

Because we always update the head pointers of the elements of the smaller list, each `UNION` at least doubles the size of the smaller list.
So an element's name gets updated at most $O(logn)$ times.
Thus, over the course of the entire algorithm, we see that (all) n `UNION`s take at most $O(n \lg n)$ time.

Theorem 7.1.1. *Kruskal's algorithm takes* $O(|E| \lg |E|) = O(|E| \lg |V|)$ *time.*

Proof. First, note that the algorithm does indeed produce a spanning tree.
The output cannot have a cycle, because if it did, the edge added that forms the cycle would join together two vertices in the same tree.
The output must be connected, for if it weren't, when we considered the edge that connected two components, we would have added it.

Next, we need to argue that the spanning tree T produced by Kruskal's algorithm is a minimum spanning tree.
Assume it isn't. Among all minimum spanning trees for G, the input, let T' be the one with the largest number of edges in T.
If $T' = T$, we're done.
Otherwise, among all the edges in $T - T'$, let e be the one that is added earliest by Kruskal's algorithm when we run it on G.
Because T' is a spanning tree, $T' \cup \{e\}$ has a cycle C.
T cannot contain all the edges of this cycle (otherwise it would not be a tree) so C contains an edge f that is not in T.
Now consider $T'' = (T' \cup \{e\}) - \{f\}$. T" is also a spanning tree; since T' was minimum, $weight(e) \geq weight(f)$.

If $weight(f) < weight(e)$, then the algorithm would have considered the edge f before it would have considered the edge e.
Adding f wouldn't create a cycle (e was the first edge not in T'), so it would have been added.
But it wasn't; e was.
So $weight(f) \geq weight(e)$.
Thus $weight(e) = weight(f)$.

So T'' is also a minimum spanning tree.
But T'' has one more edge in common with T than T' does, contradicting the choice of T'. □

7.1.2 Prim's Algorithm

← 2016/11/08

It is a greedy algorithm where, at each step, we choose a previously unconnected vertex that becomes connected by a lowest-weight edge.

At every step, we need to pick an appropriate vertex that is not in the tree T, say r.
The vertex v that we want will have the minimum-cost edge connecting v to some vertex already in T.
So it makes sense to keep those vertices not in T in a min-priority queue (min-heap).

For each vertex v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in T; $key[v] = \infty$ if there is no such edge. Initially the keys of all vertices equal ∞.

At every step we do an `EXTRACT-MIN` operation on the heap to get the vertex with the minimum edge weight associated with it.

Once we get that vertex, we have to look at all edges adjacent to it, and find the minimum weight edge.
We then update the key associated with each vertex, decreasing the key (via `DECRESE-KEY` if we find a lower-weight edge.

We use an array $\pi[u]$, indexed by the vertices u of V, such that $\pi[u]$ is the parent of u in the minimum spanning tree.
The actual tree itself will consist of the edges $(v, \pi[v])$ for all vertices v in V (except r, which has no parent).

The key idea is that whenever a new vertex v is added to the tree T, we look at all the new edges (v, u) incident on v and update our knowledge about the lowest-cost edge connecting u to T.
That way we examine all edges, but the priority queue only has to hold vertices, not edges.

```
function PRIM(G, w, r)
    for each u ∈ V do
                ▷ key[u] is current estimate on cheapest edge uv joining u to vertices not yet considered
        key[u] ← ∞
                    ▷ π[u] is the current predecessor of u in minimum spanning tree created
        π[u] = nil
    end for
    key[r] = 0
                                ▷ Q is the priority queue, vertices that remain to be considered
    Q = V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        for each v in Adj[u] do
            if v ∈ Q and w(u, v) < key[v] then
                π[v] = u
                key[v] = w(u, v)
            end if
        end for
    end while
end function
```

Note: as each vertex is removed from the priority queue Q, its connection to the tree is established through the parent pointer. Before that time, the parent pointer of a vertex might point to a different vertex.

How do we determine, in `PRIM`, whether $v \in Q$?
One simple way is just to have every vertex have a flag, and when we do an `EXTRACT-MIN` and the vertex is removed, we set this flag.
Alternatively we can assume the vertices are numbered 1 through n and use a bit-vector (an array of size n) to record whether a vertex is in Q or not; this would be updated over time.

What is the running time?
We can build the heap initially in $O(|V|)$ time.
We call `EXTRACT-MIN` $|V|$ times, and each call costs $O(\lg |V|)$. So this part is $O(|V| \lg |V|)$.
We have to traverse all the edges, so that is $O(|E|)$.
For each edge, we potentially call `DECREASE-KEY`, which costs $O(\lg |V|)$. So that costs $O(|E| \lg |V|)$.
The total is $O(|V| \lg |V| + |E| \lg |V|) = O(|E| \lg |V|)$, which is the same time as Kruskal's algorithm.

Its running time can be *improved* to $O(|E| + |V| \lg |V|)$ with a more sophisticated data structure — Fibonacci heaps — that supports `DECREASE-KEY` more efficiently.
If the graph is dense (more than a linear number of edges), this time is asymptotically better than before.

7.2 Shortest Path Problems

The *single-source* shortest path problem is the following: given a source vertex s, and a sink vertex v, we'd like to find the shortest path from s to v.
Here shortest path means a sequence of directed edges from s to v with the smallest total weight.
Now it turns out that for most of the known algorithms, it is just as efficient to find the shortest paths from s to all the other vertices of the graph.

Applications of shortest paths include speech recognition, to try to figure out the meaning of sentences.
Here we have to distinguish between different words that sound alike (homophones), such as "to", "two", and "too".
To do this, construct a graph whose vertices are words and whose edges are words that follow the given words in a sentence.
The edge between two words carries a weight measuring the likelihood of the transition; the lower the weight, the more reasonable the transition.
So "to school" would have low weight, while "two school" would have a high weight.
Then a shortest path gives the best interpretation of an uttered sentence.

Other applications include network routing protocols such as "IS-IS" (Intermediate System to Intermediate System) and "OSPF" (Open Shortest Path First) use shortest paths algorithms.

Another application is graph drawing.
We'd like to "center" a graph on a page.
But what's the center? It could be a vertex that minimizes the maximum distance to any other vertex in the graph.
We could find this by finding the shortest path between all pairs of vertices.

7.2.1 Aspects of Shortest Paths

The definition of shortest paths involves some some subtleties.
Maybe we really want a shortest walk, not shortest path?
So should we allow *negative edges*? Of course, there are no negative distances; nevertheless, there are actually some cases where negative edges make logical sense.
But then there may not *be* a shortest walk, because if there is a *cycle* with negative weight, we could simply go around that cycle as many times as we want and reduce the cost of the path as much as we like.
To avoid this, we might want to detect negative cycles. This can be done in the shortest-path algorithm itself.
Non-negative cycles aren't helpful, either.
Suppose our shortest walk contains a cycle of non-negative weight.
Then by cutting it out we get a walk with the same weight or less, so we might as well cut it out.
So we can assume our walks are actually paths.

7.3 Shortest Path Algorithms

Given a source vertex s, a shortest-path algorithm constructs:

- a predecessor $\pi[v]$ for each vertex v; this is the predecessor of v in the shortest path from s to v
- $d[v]$, the current upper bound on the weight of the shortest path from s to v ("shortest-path estimate")

function INITIALIZE(G, s)
  for each vertex $v \in V$ **do**
    $d[v] = \infty$
    $\pi[v] = nil$
  end for
  $d[s] = 0$
end function

Main idea: "relaxation" of edges.
Here we already have a shortest-path estimate to both u and v.
We consider an edge (u, v) and then update our estimate on v, if $d[u] + w(u, v) < d[v]$.

function RELAX(u, v, w)
  $t = d[u] + w(u, v)$
  if $d[v] > t$ **then**
    $d[v] = t$
    $\pi[v] = u$
  end if
end function

The *Bellman-Ford* algorithm: it computes the length of a shortest path $d[]$ originating from s to all other vertices on the graph $G = (V, E)$, using the weight function w.

Further, it returns either true or false, false iff. the graph contains a reachable negative-weight cycle, true otherwise.
It makes no assumptions at all about the weights on the edges; they can be negative. ← 2016/11/10

```
function BELLMAN-FORD(G, w, s)
    INITIALIZE(G, s)
    for i = 1 to |V| − 1 do
        for each edge uv in E do
            RELAX(u, v, w)
        end for
    end for
    for each edge uv in E do
        if d[v] > d[u] + w(u, v) then
            return false
        end if
    end for
    return true
end function
```

Let's prove that this method works.
Intuitively, the idea is that each iteration for the first `for` loop increases the total number of edges in a path from s by one.
So after at most $(|V| - 1)$ iterations, we will have considered all the paths (if there are no negative cycles).
If there's a negative cycle, then we can go on improving our cost vertices within the loop, and this is detected by the second `for` loop.

Lemma 7.3.1. After i repetitions of the `for` loop in `BELLMAN-FORD`, the following properties hold:

- If $d[u] \neq \infty$, it is equal to the weight of some path from s to u
- If there is a path from s to u with at most i edges, then $d[u] \leq$ weight of the shortest path from s to u having at most i edges.

Proof. By induction on i.

The base case is $i = 0$.
Consider what happens before the first `for` loop is executed.
Then $d[s] = 0$ for the source vertex s, which is correct.
For all the other vertices u, $d[u] = \infty$, which is also correct because there is no path from s to u with 0 edges.

For the induction step, consider when a vertex's estimated distance is updated by $d[v] = d[u] + w(u, v)$.
By out inductive hypothesis, $d[u]$ is the weight of some path p from s to u.
Then $(d[u] + w(u, v))$ is the weight of the path that follows p from s to u and then goes to v.

So the first property holds.

For the second property, consider the shortest path from s to u with at most i edges.
Let v be the last vertex before u on this path.
Then the part of the path from s to v is the shortest path from s to v with at most $(i-1)$ edges.
By the inductive hypothesis, $d[v]$ after $(i-1)$ executions of the first `for` loop is at most the weight of this path.
Therefore, $(d[v] + w(u,v))$ is at most the weight of the path from s to u.
In the i-th iteration of the first `for` loop, $d[u]$ gets compared with $(d[v] + w(u,v))$, and is set to equal to it if $(d[v] + w(u,v))$ is smaller.

Therefore, after i iterations of the first `for` loop, $d[u]$ is at most the weight of the shortest path from s to u that uses at most i edges.
So the second property holds.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so the second `for` loop does not detect any possible shorter paths.

Conversely, suppose no shorter path is detected in the second `for` loop.
Then for any cycle with vertices $v_0, \ldots, v_{k-1}$, we have

$$\begin{aligned} d[v_1] &\leq d[v_0] + w(v_0, v_1) \\ d[v_2] &\leq d[v_1] + w(v_1, v_2) \\ &\vdots \\ d[v_0] &\leq d[v_{k-1}] + w(v_{k-1}, v_0) \end{aligned}$$

If we now sum both sides, the $d[v_i]$ terms cancel, giving us

$$w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{k-1}, v_0) \geq 0$$

which does not have negative weight. □

Note: `BELLMAN-FORD` actually creates a tree!

What is the running time of `BELLMAN-FORD`? The `INITIALIZE` step costs $O(|V|)$. The first `for` loop runs $(|V|-1)$ times, and inside the loop we do $|E|$ work. The second `for` loop costs $|E|$. So the total cost is $O(|V||E|)$.

7.4 Dijkstra's Algorithm

Dijkstra's algorithm is another shortest-paths algorithm.
If implemented carefully, it can run faster than `BELLMAN-FORD`, but it has the disadvantage that the edge weights *must be non-negative.*
In `BELLMAN-FORD`, every edge is relaxed $|V|$ times. In `DIJKSTRA`, we avoid this, at the cost of

having (essentially) to do some sorting.

Dijkstra's algorithm is quite similar to Prim's algorithm for the minimum spanning tree.
The idea is to maintain a set of S of vertices for which you already know the shortest path from the source.
At every step you choose a vertex u from $V - S$ with the minimum shortest-path estimate, add u to S, and relax all the edges leaving u.
This can be done by keeping the vertices in a priority queue.

```
function DIJKSTRA(G, w, s)
    INITIALIZE(G, s)
    S ← ∅
    Q ← a min-priority queue on the vertices V
    while Q ≠ ∅ do
        u ← EXTRACT-MIN(Q)
        S ← S ∪ {u}
        for each vertex v in Adj[u] do
            RELAX(u, v, w)
        end for
    end while
end function
```

Informally, the correctness of the algorithm follows because we keep the vertices in three groups: the vertices already considered (those in S) for which we already know the shortest-path; the vertex currently being considered; and the remaining vertices.
At each step we choose the unconsidered vertex for which our distance element is smallest and update our distance estimates.

Let's figure out the running time.
Each vertex v is added S exactly once, so each edge in $Adj[v]$ is examined exactly once.
Since the total number of edges is $|E|$, there are $|E|$ iterations of the for loop, and so at most $|E|$ `DECREASE-KEY` operations (done in `RELAX`).
The total cost is there for $O(|V| + |E| \lg |V|)$.
If all vertices are reachable from the source, then $|E| \geq |V| - 1$, so this is $O(|E| \lg |V|)$.
This can be improved to $O(|V| \lg |V| + |E|)$ with a fancy data structure, the Fibonacci heap.

By the way, how does Google Maps find shortest paths?
Because there are millions and millions of possible vertices in the database, Google doesn't run a shortest path algorithm on this giant database.
Rather, it uses some heuristics.
For example, to go from Kitchener to Winnipeg, it first routes you through Kitchener to some big road going in the right direction (using shortest paths on the much-smaller local database), then follows the big road to the vicinity of Winnipeg, then runs shortest-paths on that local area.

7.5 Floyd-Warshall Algorithm

← 2016/11/15

We're interested in the *all-pairs shortest path* problem; we are given a weighted directed graph and we want to find the shortest-path distance between *every* pair of vertices.
Note that the edges can possibly be negative, but no negative-weight cycles.

It's easy to find shortest paths when there are 0 edges, or 1 edge, in the path.
So maybe the right way to do it is solve all-pairs shortest paths for paths having 0 edges, or 1 edges, etc.
Let $cost[i, j, k]$ be the minimum cost of a path from node i to node j using at most k edges.
We arrive at the following recursion,

$$cost[i, j, k] = \min_{1 \le \ell \le n} w(i, \ell) + cost[\ell, j, k-1]$$

We have to fill in all the entries for $1 \le i, j, k \le n$, and filling each entry costs $O(n)$. This gives an $O(n^4)$ algorithm which is rather slow, we can do better!

Floyd came up with the idea.
Let the vertices be numbered $1, 2, \ldots, n$.
We consider all paths from vertex i to vertex j in which all intermediate vertices are numbered k or lower.
Let $d_{ij}^{(k)}$ be the weight of the shortest path from i to j in which all intermediate vertices are numbered k or lower.

Then we get the following recurrence:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \ge 1 \end{cases}$$

Note that if $k = 0$, then there can be no intermediate vertices at all, so the only possibility is a one-step path from i to j, which has weight w_{ij}.

Now $d_{ij}^{(k)}$ is supposed to reflect paths where all intermediate vertices are numbered $\le k$.
So either k appears as an intermediate vertex in a shortest path, or it doesn't.
In the latter case, $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.
In the former case, k appears only once so we can break the path into the part before k appears and the part after. In both of these parts k does not appear again, so $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

```
function FLOYD(W, n)
    for i = 1 to n do
        for j = 1 to n do
            cost[i, j, 0] = W[i, j]
        end for
    end for
    for k = 1 to n do
```

```
        for i = 1 to n do
            for j = 1 to n do
                cost[i, j, k] = min(cost[i, j, k − 1], cost[i, k, k − 1] + cost[k, j, k − 1])
            end for
        end for
    end for
    for i = 1 to n do
        for j = 1 to n do
            M[i, j] = cost[i, j, n]
        end for
    end for
    return M
end function
```

The algorithm runs in $O(n^3)$ in the unit-cost model, so with this we can match the running time of Dijkstra's algorithm run from every vertex, even if we use the complicated Fibonacci heaps in Dijkstra's algorithm.

This uses $\Theta(n^3)$ space, but it is easy to rewrite it so that only $\Theta(n^2)$ space is needed.
To do this, we can just store the $k - 1$ value since we only look at the previous one.

8 P, NP, and all that

There are some problems that are solvable, but for which it is possible to prove that **no faster algorithm exists**.
There are also problems for which no fast algorithm is *currently known* to exist, and for which there is some evidence that no fast algorithm will ever be found.

8.1 P

What does "fast" mean?
The answer that has become traditional by now is that the program's running time should be **bounded by a polynomial in the input size**, in the log-cost model; in other words, the worst-case running time should be $O(n^k)$ for some k.
This is called **polynomial time**, and the complexity class P is the class of all problems solvable in polynomial time. Strictly speaking, P is actually the class of *languages* denoting encoding of polynomial-time-solvable decision problems, but we don't need to be so rigorous for this course.

P contains pretty much every problem we have looked at so far in this course: maximum subrange sum, greatest common divisor, matrix multiplication, all-pairs shortest paths, etc.

One problem we discussed that is *not* in P is the problem of listing all the permutations.
This cannot be in P because we need to output $n!$ results on an input of size n.
But this isn't so interesting because the complexity of the problem arises from the large output size. So sometimes it makes more sense to measure complexity of a problem in terms of the size of the input *and* the size of the output.

Why is polynomial-time a synonym for "fast"?

- Nearly every algorithm that people actually consider to be fast in practice runs in polynomial time in theory. (A possible exception is the simplex algorithm for linear programming, which runs relatively quickly on most natural examples, but has an exponential worst-case run time. However, there are algorithms for linear programming which are genuinely polynomial time, like the ellipsoid method.)

- A program that runs in polynomial time in one model runs in polynomial time in almost all other models. This is not completely trivial and there is definitely something to prove here. (The relatively recent invention of the quantum computing model means this is probably no longer true for all pairs of models.)

- The class P has nice closure properties. It is closed under composition. P is also closed under using a polynomial-time subroutine, *provided* that the input is not expanded each time the subroutine is called. (If the input is expanded in size, then you could get something that runs in exponential, not polynomial time.)

Objections to the definition of P:

- It doesn't consider the exponent of the polynomial. An algorithm running in n^{10000} time is considered "fast" and one running in $2^{n/10000}$ is not, although for practical problem sizes the second will be more useful that the first.
  Response: true, but "most" natural problems in P has algorithms with small polynomial exponents.

- It doesn't consider the size of the constant out in front. An algorithm running in $1000000n^2$ will be less useful than an algorithm running in $2^{n/10000}$.
  Response: true again, but "most" natural problems in P have algorithms with relatively small constants out in front.

- It considers only worst-case behaviour. Some problems may be efficiently solvable for most inputs, and only require lots of time on a small set of inputs.
  Response: true, but such problems may be relatively rare.

- It doesn't consider other models of computation, such as quantum computing.
  Response: true, but we don't yet have a practical quantum computer

Recall that P is the class of problems solvable in polynomial time.
There is one subtlety here, which we have already encountered, and that is how the size of the input is measured.
The convention is that numbers are usually represented in base-2 (or any base ≥ 2), *not* in unary.
Forcing the input to be in unary can transform an algorithm that doesn't run in polynomial time to one that does.

Example. Consider prime factorization, which is generally considered to be a "hard" problem. Nobody currently knows an algorithm for the prime factorization of n in time polynomial in $\lg n$, but there are easy algorithms that run in time polynomial in n.

Similarly, if we choose a crazy encoding for a problem (the way the problem is actually represented as input) then we could get a crazy result, like hard problems suddenly become easy.
Normally there is an obvious way to encode things, and usually two different reasonable encoding are polynomial-related; there is a polynomial-time algorithm converting one to the other.

8.2 Problems not in P

There are some problems, like the halting problem, that are not solvable in *any* time bound.

Consider the following problem:
ERE problem: Given two extended regular expressions E and F over an alphabet Σ, decide if E and F specify the same set.

An extended regular expression (ERE) is just like an ordinary regular expression, except that exponentiation is allowed.

Example. $(a^3 + b^2)^2 = (aaa + bb)^2 = (aaa + bb)(aaa + bb) = aaaaaa + aaabb + bbaaa + bbbb$

We may assume that the exponents are decimal numbers. No other exponents are allowed.

The ERE problem is *solvable*: to see this, we can take any ERE and expand out all the exponents to get a regular RE.
Do this for E and F, getting RE's E' and F'.
Now convert these to equivalent deterministic finite automata, minimize the automata, and check if the resulting minimized automata are the same.

However, the ERE problem is not *efficiently* solvable.

Theorem 8.2.1. *There exists a constant c such that every algorithm that solves the ERE problem must use at least $2^{cn/\lg n}$ time on infinitely many inputs. (Here n, as usual, measures the size of the input.)*

Therefore the ERE problem cannot be solved in polynomial time.
Unfortunately, the number of such natural examples that are known is fairly small.

← 2016/11/17

Up to now we have been talking about problems in general.
Now we restrict our attention to one particular kind of problem: the *decision problem*.
A decision problem is a problem with some inputs (parameters) and a *single output*, which is either "yes" or "no".

An "instance" of a problem is a particular set of inputs to the problem.
We say this set of inputs if a "yes" instance if the answer to the problem on these inputs is "yes", and it is a "no" instance if the answer is "no".

8.3 NP

What is NP?
NP is the class of decision problems having the property that their "yes" instances can be *verified* in polynomial time.

Example. Let us say an undirected graph has a *Hamiltonian cycle* if there is a simple cycle visiting all the vertices in the graph, never repeating an edge or vertex.
The decision problem is: `HAM-CYCLE`: given a graph G, does it have a Hamiltonian cycle?

The associated "yes" instances are those graphs *having* a Hamiltonian cycle.
It is not obvious how to solve `HAM-CYCLE` in polynomial time; actually, no polynomial-time algorithm is currently known.
However, if I claim that a particular graph *has* a Hamiltonian cycle, and you demand proof, then there is a certificate I can produce that you can *easily* check: namely, the list of vertices forming the cycle!
All you have to do is check that that it really is a simple cycle, and that it actually visits each vertex. You can verify this quickly, *if* I provide you with the cycle.
Of course, finding the cycle appears to be hard — but *verifying* it is easy.
That's the essence of NP.

Example. Another example is the `SMALL DIVISORS` problem.
Here the decision problem is as follows: given a positive integer n, and a bound B, is n divisible by an integer c with $1 < c < B$?

Suppose I claim that, say, 187 has a divisor < 12.
If you are skeptical, I can convince you by producing the factorization $187 = 11 \times 17$. You check to make sure that the factors indeed multiply to the given number.
Of course, it may take me a long time to find the factorization, but once found, it can be quickly verified.

That's the crux of the differences between P and NP.
Problems in P can be quickly solved; problems in NP are not necessarily solvable quickly, but the answer can be *verified* quickly provided you have the certificate.

8.4 Formal Definition of NP

A *decision problem* is a parameterized problem with a yes/no answer.

A *verifier* for a decision problem is an algorithm V that takes two inputs: an instance I of the decision problem and a certificate C.
V returns "true" if C is a valid certificate and "false" otherwise.
Furthermore, for each "yes" instance I, there must be *at least one* valid certificate, and for each "no" instance there must be *no* valid certificates at all.
If, further, V runs in time bounded by a polynomial in the size of I, then it is called a *polynomial-time* verifier.

A decision problem is in NP if it has a polynomial-time verifier.

Example. Given a graph G, does G have a Hamiltonian cycle?
A polynomial-time verifier takes $I = G$ and $C =$ a claimed list of the vertices forming the cycle as input.
It then checks to ensure that all vertices are represented and the edges actually exist in G.

Example. Is n a composite number?
A polynomial-time verifier takes $I = n$ and $C = (a, b)$ as input.
It then verifies that both a and b are ≥ 2 and that $n = ab$.
If this is the case, it answer "yes"; otherwise it answers "false".

Example. The travelling salesman problem. Here you are given a directed graph G with weights on the edges, and a number B, and you want to find a directed cycle passing through all the vertices, with total weight $\leq B$.

Example. We say a subset of the vertices of an undirected graph is an "independent set" if no two of the vertices are connected by an edge.
The problem is, given a graph G, and an integer b, is there an independent set of size $\geq b$?

Although not every problem is a decision problem, the following principle applies:
Principle: Usually, a problem p has an associated decision problem p' such that

(a) Given an algorithm for p', we can solve p with a small amount of additional work, and

(b) Given an algorithm for p, we can solve p' with a small amount of additional work.

Theorem 8.4.1. *P is a subset of NP.*

Proof. We just need to see that if a problem is in P, then it has a polynomial-time verifier.
If the problem is in P, then there is already a polynomial-time algorithm A to solve it. So the verifier just runs A and checks that the answer is "yes". The certificate plays no role, so it can just be the empty string. □

The big question is: is P actually *equal* to NP?
This is one of the most important question in all of mathematics and computer science.

8.5 Why is it called NP?

"NP" stands for "non-deterministic polynomial", not "not polynomial."
We don't yet know that the hard problems in NP can't be solved in polynomial time.

Why is "NP" called "non-deterministic polynomial time"?
It comes from another way of thinking about NP, the way people originally thought about it.
Instead of using a verifier to define this class, we could instead consider a different machine model: the non-deterministic RAM.
Such a machine M is just like the ordinary RAM, but also has an instruction `NCHOICE(l1, l2)`. When this instruction is executed, the machine has a choice to branch to *either* line `l1` or line `l2` of the program.
The machine is said to accept the input if *some sequence of choices* results in the machine halting with "1" on the output tape. (It doesn't matter if some sequences of choices fail, just as long as at least one succeeds.)
The machine M fails to accept if *no sequence of choices* results in this action.
uch a machine is said to run in polynomial time if, for every accepted input, there is some accepting sequence of choices that causes M to run in time bounded by a polynomial in the size of the input.
It is easy to see that this is equivalent to the definition with the verifier.

To see this, note that we can create a certificate by listing the series of non-deterministic choices that causes M to halt with "1" on the output tape.
Similarly, given a verifier V we can make a non-deterministic RAM that simply non-deterministically runs V on all possible certificates of length bounded by a polynomial.

8.5.1 co-NP

There is a certain asymmetry in our definition of NP.
We only need certificates for the instances with "yes" answers.

For "no" answers we need not provide any certificate at all.
For many problems in NP, it is not obvious how we could produce a certificate for the "no" answers.

Due to this symmetry, it is reasonable to consider those decision problems for which there is a verifier for the "no" instances (instead of the "yes" instances).
This class is called co-NP ("co-" stands for "complement" or "complementary").

Nobody currently knows the relationship between NP and co-NP.
It could be that NP = co-NP, and it could be that P = NP $\cap$ co-NP.
Most experts don't believe that either of these two equalities holds.

8.6 Reductions

We want some way of saying that problem B is as hard as, or harder than, than problem A.
To do so, we introduce a great idea: *reductions*.

We say a **computational problem A reduces to a problem B if you can solve A given access to an algorithm to solve B as a subroutine**.

If problem A reduces to problem B, we say that "problem B is at least as hard as problem A".
This is sensible, because knowing an algorithm for B allows us to solve A, but not necessarily the other way around. So somehow the capability of solving B is more powerful than (or as powerful as) the capability of solving A.
We sometimes write $A \leq B$. This notation is very helpful, because the inequality goes from the "easier" problem to the "harder" problem.

Example. A = "primality testing" reduces to B = "compute prime factorization".
To see this, we need to see how to test n for primality, given an algorithm to compute the prime factorization of n. That's easy: we just compute the prime factorization of n and see if it consists of a single prime.
Note that we can test primality in time polynomial in $\lg n$, but there is still no algorithm for prime factorization known that runs in time polynomial in $\lg n$.
So at least conjecturally, B is truly harder than A.

Example. A = "compute the median of n numbers" reduces to B = "given k and n numbers, compute the k-th smallest".
Less obviously, B also reduces to A.
$A \leq B$: call algorithm to solve B with $k = \lceil n/2 \rceil$.
$B \leq A$: pad/remove elements such that $\lceil n/2 \rceil = k$.

Example. A = "compute the median of a list" reduces to B = "sort a list".
Again, this is easy: first, sort the list of n numbers; then look for the element in position $\lceil n/2 \rceil$.
Note that we know how to find the median in $O(n)$ time, but any comparison-based algorithm for sorting uses $\Omega(nlogn)$ time.
So B is truly harder than A.

8.7 Reductions with Restrictions

Our notion of reduction $A \leq B$ is too liberal for many uses, because it permits our algorithm for solving A to use an algorithm for B, plus *anything* else at all.
For many purposes, it makes more sense to put some restrictions (on time and/or space) on the "additional work" that A does, in addition to calling B.

As an example, we could restrict our reductions to those that use only a constant amount of additional space, not counting the input to A and its output, and what B itself uses.
We could call this a "constant-space" reduction.
In our example of $A =$ "compute the median of n numbers" reduces to $B =$ "given k and n numbers, compute the k-th smallest". then the reduction $A \leq B$ is constant-space, because given an algorithm for B, to solve A all we have to do is compute $k = \lceil n/2 \rceil$ and then call B on the input list and k.

8.8 Polynomial Time Reductions

← 2016/11/22

A decision problem A reduces to a decision problem B in polynomial time (denoted $A \leq_p B$) if there exists a *polynomial-time-computable function* f that maps instances of A to instances of B, such that the *answers to the instances are preserved* under f.
In other words, if x is an instance of decision problem A, then $f(x)$ is an instance of decision problem B, and the answer for x must be the same as that for $f(x)$.
f can be viewed as a *translation*.

Theorem 8.8.1. *If A and B are decision problems, and B is in P, and $A \leq_p B$, then A is in P; i.e. if B is easy and A reduces to B, then A is easy.*

Proof. Since $A \leq_p B$, there is a polynomial-time computable function f mapping instances of A to instances of B, such that the answers are the same in both cases.
Let us say we are given an instance of problem A — call it x.
To decide x, we cam x to $f(x)$ and feed it into our algorithm to solve B. Whatever B answers, we answer the same thing for A. The total running time is the cost of computing $f(x)$ and the cost of running B on $f(x)$.
Since both are polynomial-time, the resulting algorithm is polynomial-time. □

Theorem 8.8.2 (transitive property of reductions)**.** *If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*

Proof. We have a polynomial-time computable function f mapping instances of A to instances of B, and similarly g mapping instances of B to instances of C.
So the composition $x \to g(f(x))$ maps instances of A to instances of C.
If f and g are polynomial-time computable, so if $g(f(x))$. □

Theorem 8.8.3. *If $A \leq_p B$ and $B \in NP$, then $A \in NP$.*

Proof. If $B \in$ NP, then B has a polynomial-time verifier V.
For each "yes" instances of B, say y, there is a short certificate C_y and $V(y, C_y) =$ true. If $A \leq_p B$, there is a function f mapping instances x of A to instances of B that preserves the

answers.
We can create a polynomial-time verifier V' for A: let $V'(x, C) = V(f(x), C)$. Furthermore, a certificate C for $f(x)$ is a certificate for x.
Since V' is a composition of two polynomial-time computable function, it is computable in polynomial time. □

NP-hard and NP-complete:
A decision problem is *NP-hard* if $B \leq_p A$ for every $B \in NP$. Which means A is "as hard as or harder" than every single problem in NP.

A decision problem A is *NP-complete* if A is NP-hard and $A \in$ NP.

Each NP-complete problem is the "hardest" problem in NP.
Furthermore, if a single NP-complete problem is in P, then, by Theorem 8.8.1, NP = P; **every** problem in NP would be polynomial-time solvable.

Theorem 8.8.4. *If there is a polynomial-time algorithm for any one NP-complete problem, then there is a polynomial time algorithm for every problem in NP.*

Proof. Let A be an NP-complete problem, and suppose we find a polynomial-time algorithm for it, call it `solveA`. Let B be any problem in NP.
Since A is NP-complete, we know that $B \leq_p A$; this means there is a polynomial-time computable function f mapping instances of B to instances of A and preserving the answers.
To solve B in polynomial time, map an instance using f and use `solveA` on the result.
The composition runs in polynomial-time. □

Theorem 8.8.5. *If no polynomial-time algorithm exists for any single problem in NP, then there is no polynomial-time algorithm for any NP-complete problem.*

Proof. Suppose no polynomial-time algorithm exists for some problem A in NP.
Let B be any NP-complete problem. If B has a polynomial-time solution, since A reduces to B, A would have a polynomial-time algorithm, a contradiction. □

Table 1: Examples of easy and hard problems

Easy (P)	**Hard (NP-complete)**
2SAT	3SAT
Minimum spanning tree	Travelling salesman
Shortest path	Longest path
Linear programming	Integer linear programming
Eulerian cycle	Hamiltonian cycle

8.8.1 Circuit Satisfiability Problem

We're going to prove this problem is NP-complete.

A *Boolean circuit* consists of a bunch of logical gates joined together by wires. It has inputs that can take 0/1 (true/false) values, and is made up of `AND`, `OR`, and `NOT` gates.

It acyclic. The *size* of the circuit is defined to be the total number of gates and wires.
Generally, a Boolean circuit can have multiple outputs.
Let's restrict to a single output, 0 or 1. Given such a circuit, we want to decide if it is "satisfiable", that is, does there exist an assignment of truth values (0 or 1, F or T) to the wires that results in the circuit outputting a "1"?

Formally, `CIRCUIT-SAT` = the decision problem, given a Boolean circuit C, is it satisfiable?

Theorem 8.8.6. *`CIRCUIT-SAT` is NP-complete.*

Proof. First, we claim that `CIRCUIT-SAT` is in NP. The "certificate" we would want would be a satisfying assignment. Given this, we can easily verify in polynomial time that the circuit outputs a 1, by substituting our values for the gate inputs and computing the value of each gate output.

To prove that `CIRCUIT-SAT` is NP-complete, we will give a polynomial-time transformation from **every** problem L in NP to `CIRCUIT-SAT`.
If $L \in$ NP, then there is a verifier $A(x, y)$ that runs in time $T(|x|) = O(|x|^k)$ for some k.
We now construct a single Boolean circuit M that maps one "configuration" of a machine that carries out the computation of the program $A(x, y)$ to the next "configuration", after executing one step of the program. Here M has many inputs and many outputs.
We now hook together $T(|x|)$ of these circuits together, making the inputs to the circuit at the top the value of y, and the output the single bit that reflects the value of $A(x, y)$.
This big circuit $C(x)$ is satisfiable (by a value of y) iff. x was a "yes" instance of L. The size of this circuit is polynomial in x, and the transformation can be done in polynomial time.
Therefore, `CIRCUIT-SAT` is NP-complete. □

Lemma 8.8.7. Suppose a problem A is known to be NP-complete.
If $A \leq_p B$, then B is NP-hard.
If in addition, B is in NP, then B is NP-complete.

Proof. Let C be any problem in NP. By definition, $C \leq_p A$ and by hypothesis $A \leq_p B$.
By transitivity of reductions, $C \leq_p B$. It follows that B is NP-hard.
If B is also in NP, then B is NP-complete by definition. □

The lemma says that if we want to prove a problem B NP-complete, we can base the proof on previous results, not on the original definition.
The *four-step method* for proving NP-completeness of a decision problem B.

1. Prove B is in NP by describing the polynomial-time verifier V that verifies "yes" instances of B. What is the certificate? How is it verified?

2. Select a problem A that you already know to be NP-hard or NP-complete. Usually this is a problem that is related to B in some way.

3. Design a function f that maps all instances x of A to instances $y = f(x)$ of B, such that the answer on x is always the same as the answer on y, and justify your construction.

4. Explain why f can be computed in polynomial time.

Additional notes:

- We do *not* require that every single possible instance of B gets mapped to by the function f
- Nor do we require that different x's must necessarily map to different y's
- However, *every possible instance — whether a "yes" or "no" instance — of a problem A* must have an image under f, and the answer for instance x of A must always be the same as for the instance $f(x)$ of B
- The function f should not depend on the whether x is a "yes" instance or "no" instance of problem A.