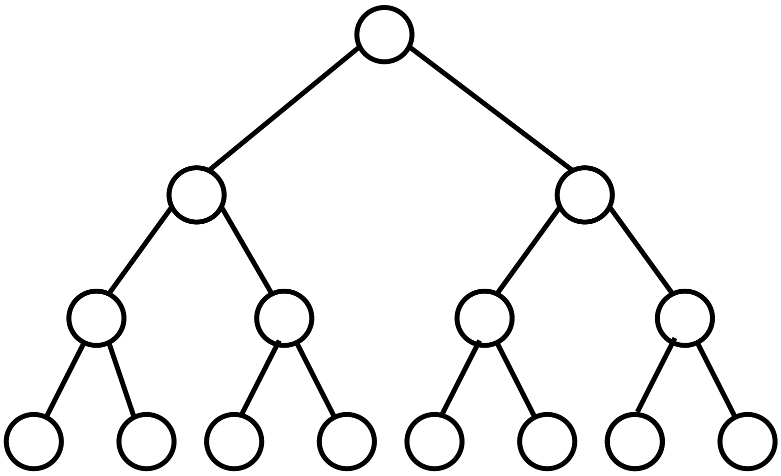
7.1-2: Show that an n-element heap has height $\lfloor \lg n \rfloor$.

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 $i \leq j$.

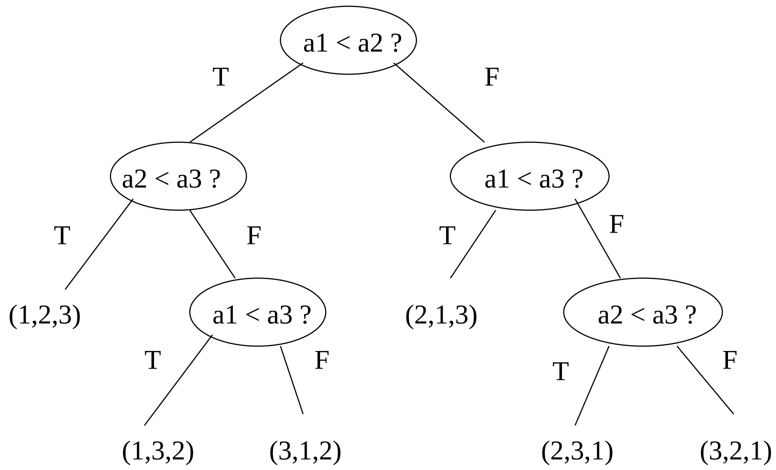
Since $2i > i$ and $2i + 1 > i$ then $A[2i] \leq A[i]$ and $A[2i + 1] \leq A[i]$.

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

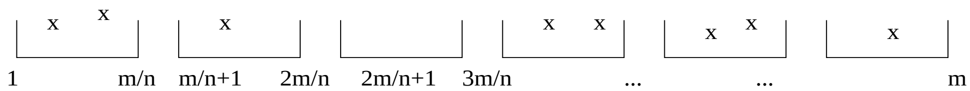
With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.

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!

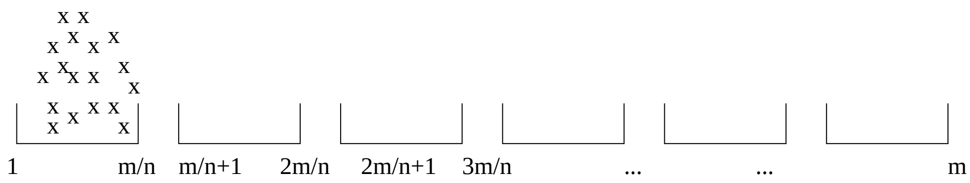
The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is $O(n)$.

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 Shifflitt'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 Rt733 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?