EE360C: Algorithms

The Basics

Spring 2018

Department of Electrical and Computer Engineering
University of Texas at Austin

Algorithms

Definition of an Algorithm

Definition 1: Algorithm

An *algorithm* is any well-defined computational procedure that takes some value or set of values as *input* and produces some value or set of values as *output*

An algorithm is:

- a sequence of computational steps that transform the input into an output
- a tool for solving a well-specified computational problem
- said to be *correct* if, for every input instance, it halts with the correct output
- said to *solve* a computational problem if it is correct

Algorithm Efficiency

Fundamental Issues in Algorithms

We'll talk about two fundamental issues in algorithms:

- analysis
- design

But first, . . .

Some Caveats

Rob Pike's Rules of Programming

Rule 1: You can't tell where a program is going to spend its time; bottlenecks occur in surprising places.

Rule 2: Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code *overwhelms* the rest.

Rule 3: Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants.

Rule 4: Fancy algorithms are always buggier than simple ones, and they're much harder to implement.

Rule 5: Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident.

Algorithm Analysis Basics

Analyzing an algorithm refers to predicting the resources (memory, communication bandwidth, computer hardware) that the algorithm requires.

We must have a model of the implementation technology to underlie our analysis.

But What's Our Goal?

Our goal is to develop (correct) *efficient* algorithms as solutions to well-defined problems.

Let's consider some working definitions...

Algorithm Efficiency Definition 1

First try

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

This is a good start, but...

- it's awfully vague
- even bad algorithms can run fast when applied to small test cases
- even good algorithms can run slow when implemented poorly
- what is a "real" input instance? (part of the problem is that we don't know the range on possible inputs *a priori*)
- this definition doesn't consider how well the algorithm's performance *scales* as the problem size grows

Algorithm Efficiency Definition 1 (cont.)

We would like a definition of algorithm efficiency that is:

- platform-independent
- instance-independent
- of predictive value with respect to increasing instance sizes

Example

Consider the stable matching problem. A problem instance has a "size" N, which is the total size of the input preference lists (what must be input to run the algorithm).

- there are n men and women
- each has a preference list of $2n$
- $N = 2n^2$

Algorithm Efficiency Definition 1 (cont.)

Input Size

The definition of **input size** depends on the particular computational problem being studied.

- As a general rule, the running time grows with the size of the input

Running Time

The **running time** of an algorithm on a particular input is the number of primitive operations or “steps” executed.

- running time should be machine independent
- we assume a constant amount of time is required to execute each line of pseudocode

Algorithm Efficiency Definition 2

Worst-Case Running Time

The *worst-case* running time of an algorithm is the worst possible running time the algorithm could have over all inputs of size N.

This tends to be a better measure than *average-case* running time, which averages running times over “random” instances

Second Try

An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

Consider the stable matching algorithm again.

- the brute-force search generates all $n!$ possible pairings
- a running time on the order of n^2 is clearly better

Algorithm Efficiency Definition 3

But this is still a little vague. What is “qualitatively better”?

A better working definition is:

Third (and Final) Try

An algorithm is efficient if it has polynomial running time.

Of course, running time of n^{100} is clearly not great, and a running time of $n^{1+.02(\log n)}$ is not clearly bad. But in practice, polynomial time is generally good.

In addition to being precise, this definition is also *negatable*.

Algorithm Efficiency Summary

Brute Force

For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution

- Typically takes 2^N time or worse for inputs of size N
- Unacceptable in practice

Desirable Scaling Property

When the input size doubles, the algorithm should only slow down by some constant factor, C.

> *There exists constants $c > 0$ and $d > 0$ such that on every input of size N, the running time is bounded by cN^d steps.*

An algorithm is considered poly-time if the above scaling property holds.

Running times for algorithms on inputs of increasing size on a processor executing a million instructions a second.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Asymptotic Notation

Asymptotic Notation

We study the asymptotic efficiency of algorithms; i.e., how the running time scales with increasing input size in the limit

An algorithm with the best asymptotic performance will be the best choice for all but very small inputs (but remember our caveats...).

The goal is to identify similar classes of algorithms with similar behavior.

We measure running times in the number of primitive “steps” an algorithm must perform.

Asymptotic Bounds

We don't need to be overly precise about running times, since we care about the rates of growth.

Asymptotic Bounds

We want to represent the asymptotic running time of an algorithm (its growth rate relative to the input size) independent of any constant factors.

Asymptotic Upper Bounds: O-Notation

Given some function $T(n)$ that represents an algorithm's running time, we say $T(n)$ is $O(f(n))$ ("$T(n)$ is order $f(n)$") if, for sufficiently large n, $T(n)$ is bounded above by a constant multiple of $f(n)$.

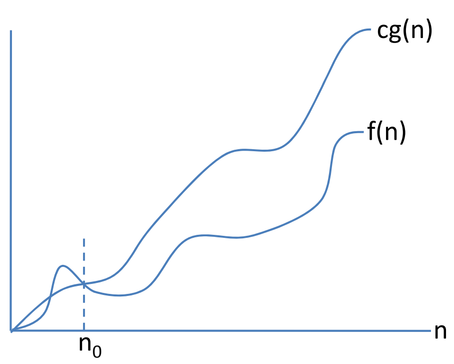
Definition 2

Given $g(n)$, we denote by $O(g(n))$ the set of functions:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

- $O(g(n))$ is a set, but we usually abuse notation and write: $f(n) = O(g(n))$

O-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ is on or below $cg(n)$
- We say that $g(n)$ is an **upper bound** for $f(n)$

O-Notation: An Example

Claim

$T(n) = pn^2 + qn + r$ is in $O(n^2)$

$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$
for all $n \geq 1$.

This is the required definition of $O(\cdot)$:
$T(n) \leq cn^2$, where $c = p + q + r$.

O-Notation: Another Example

Claim

$an + b$ is in $O(n^2)$

- Take $c = a + |b|$ and $n_0 = \max(1, -b/a)$

Some Notes on O-Notation

- Some texts use O to informally describe asymptotically tight bounds (i.e., when we use Θ)
- O-notation can be useful in quickly and easily bounding the running time of an algorithm by inspection

Asymptotic Lower Bounds: Ω-Notation

Lower bounds can be useful for stating that an algorithm's running time is at least some magnitude

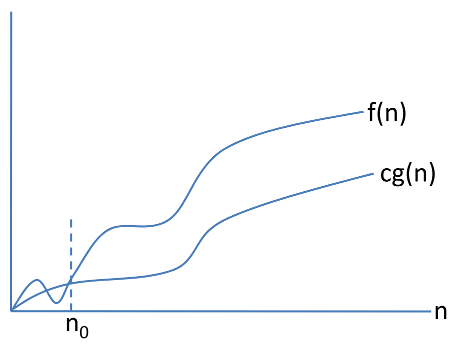
Definition 3

Given $f(n)$, we denote by $\Omega(g(n))$ the set of functions:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Ω-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ is on or above $cg(n)$
- We say that $g(n)$ is a **lower bound** for $f(n)$

Ω-Notation: An Example

Claim

$T(n) = pn^2 + qn + r$ is in $\Omega(n^2)$.

$T(n) = pn^2 + qn + r \geq pn^2$ for all $n \geq 0$.

This is the required definition of $\Omega(\cdot)$:
$T(n) \geq cn^2$, where $c = p$.

Asymptotically Tight Bounds: Θ-Notation

If a running time $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, then we've found a "tight" bound.

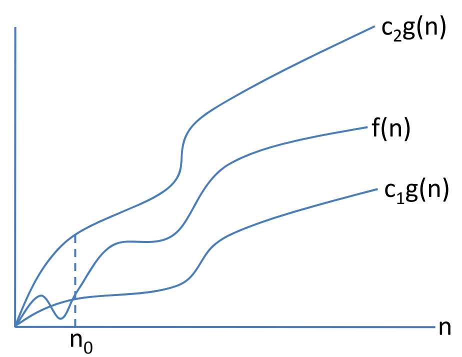
Definition 4

Given $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

- Effectively, f is "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$ for large n

Θ-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$.
- We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$

Θ-Notation: An Example

Claim

Consider $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

- We must show that there exists c_1, c_2, and n_0 such that
$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$
for all $n \geq n_0$
- This is equivalent to showing
$$c1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$
for all $n \geq n_0$
- Take $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ and $n_0 = 7$

There are many other viable choices for c_1, c_2, and n_0. And for other functions in $\Theta(n^2)$ we may need different c_1, c_2, and n_0.

Θ-Notation: Another Example

Claim

$6n^3 \neq \Theta(n^2)$

- Were this the case, then there exist c_1, c_2, and n_0 such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$.
- Equivalently, this means that, for all $n \geq n_0$, $n \leq \frac{c_2}{6}$.
- Since c_2 is a constant, and n is not, this is impossible!

In English...

Upper Bound

$f(n) = O(g(n))$:

- $f(n)$ is bounded above by a constant multiple of $g(n)$.
- $f(n)$ is asymptotically upper bounded by $g(n)$.

Lower Bound

$f(n) = \Omega(g(n))$

- $f(n)$ is at least a constant multiple of $g(n)$.
- $f(n)$ is asymptotically lower bounded by $g(n)$.

o-Notation

- Consider $2n^2 = O(n^2)$ and $2n = O(n^2)$.
- While both are true, the bound on the left is **asymptotically tight** while the bound on the right is not.
- We use o-notation to refer to upper bounds that are not asymptotically tight.

Definition 5

Given $g(n)$, we denote by $o(g(n))$ the set of functions:

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0\text{, there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

- $2n = o(n^2)$ but $2n^2 \neq o(n^2)$

ω-Notation

- We define ω-notation similarly to refer to lower bounds that are not asymptotically tight.

Definition 6

Given $g(n)$, we denote by $\omega(g(n))$ the set of functions:

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

- This is equivalent to $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$
- $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$

The Limit Theorems

Theorem

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *implies* $f(n) = o(g(n))$

Can you derive limit theorems for O, Θ, Ω, and ω?

Theorem

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ *implies* $f(n) = \omega(g(n))$

Theorem

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *or* c *(where* $0 < c < \infty$*) implies* $f(n) = O(g(n))$

Theorem

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ *or* c *(where* $0 < c < \infty$*) implies* $f(n) = \Omega(g(n))$

Theorem

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ *(where* $0 < c < \infty$*) implies* $f(n) = \Theta(g(n))$

The Limit Theorems (other direction)

Limit theorems also go in the other direction if the limit exists. E.g.:

Theorem

$f(n) = O(g(n))$ implies $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ or c (where $0 < c < \infty$), if the limit exists.

Comparison of Functions

Transitivity

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n))$$

Reflexivity

$$f(n) = \Theta(f(n))$$

Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

Transpose Symmetry

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Analogies to Traditional Comparisons

Analogies between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b.

$$
\begin{aligned}
f(n) = O(g(n)) &\approx a \leq b \\
f(n) = \Omega(g(n)) &\approx a \geq b \\
f(n) = \theta(g(n)) &\approx a = b \\
f(n) = o(g(n)) &\approx a < b \\
f(n) = \omega(g(n)) &\approx a > b
\end{aligned}
$$

The analogy does break down in some cases. For two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

Exercise

What is wrong with this statement?

Any comparison based sorting algorithm requires at least $O(n \log n)$ comparisons.

Fix it.

Standard Functions

Standard Notations

Monotonicity

- $f(n)$ is **monotonically increasing** if $m \leq n$ implies $f(m) \leq f(n)$
- $f(n)$ is **monotonically decreasing** if $m \leq n$ implied $f(m) \geq f(n)$
- $f(n)$ is **strictly increasing** if $m < n$ implies $f(m) < f(n)$
- $f(n)$ is **strictly decreasing** if $m < n$ implies $f(m) > f(n)$

Floors and Ceilings

- For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$.
- For any real number x, we denote the least integer greater than or equal to x by $\lceil x \rceil$.

Standard Notations (cont.)

Modular Arithmetic

For any integer a and positive integer n, the value $a \bmod n$ is the **remainder** of the quotient a/n.

$$a \bmod n = a - \lfloor a/n \rfloor n$$

Polynomials

Given a nonnegative integer d, a **polynomial in n of degree d** is a function $p(n)$ of the form:

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the constants $a_0, a_1, \ldots a_d$ are the **coefficients** of the polynomial and $a_d \neq 0$

- For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$.

Exponentials

For all real $a > 0$, m, and n, these identities hold:

- $a^0 = 1$
- $a^1 = a$
- $a^{-1} = 1/a$
- $(a^m)^n = a^{mn}$
- $(a^m)^n = (a^n)^m$
- $a^m a^n = a^{m+n}$

Any exponential function with base strictly greater than 1 grows faster than any polynomial function:

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

Logarithm Notations

$$
\begin{aligned}
\lg n &= \log_2 n \\
\ln n &= \log_e n \\
\lg^k n &= (\lg n)^k \\
\lg \lg n &= \lg(\lg n)
\end{aligned}
$$

For all real $a > 0$, $b > 0$, $c > 0$, and n:

$$
\begin{aligned}
a &= b^{\log_b a} \\
\log_c (ab) &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b (1/a) &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}
$$

Common Functions

Bounds for Common Functions

Polynomials

$a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial Time Algorithm

An algorithm whose running time is $O(n^d)$ for some constant d (where d is independent of the input size).

Logarithms

$O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$. (You can ignore the base in logarithms.)

Logarithms Again

For every $d > 0$, $\log n = O(n^d)$. (Any log grows slower than any polynomial.)

Exponentials

For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$. (Every exponential grows faster than every polynomial.)

Common Running Times

Linear Time: $O(n)$

Linear Time

Running time is at most a constant factor times the size of the input.

What are some things you think you can do in linear time?

Compute the maximum of n numbers $a_1, \ldots a_n$

```
max ← a_1
for i = 2 to n
    do if (a_i > max)
        then max ← a_i
```

Linear Time: $O(n)$ (cont.)

Merge

Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_m$ into a sorted whole

```
i = 1, j = 1
while (both lists are nonempty)
      do if (a_i < b_j) append a_i to output and increment i
           else append b_j to output and increment j
append remainder of nonempty list to output
```

Claim

Merging two sorted lists of total size n takes $O(n)$ time.

Proof

After each comparison, the length of the output list increases by 1.

Linearithmic Time: $O(n \log n)$

Linearithmic Time

Commonly arises in divide and conquer algorithms (we'll see why ad nauseum).

What kinds of problems do you think take $O(n \log n)$ time?
Sorting!

Largest Empty Interval

Given n time stamps $x_1, \ldots, x_n$, on which copies of a file arrive at a server, what is the largest interval of time when no copies of the file arrive?

An $O(n \log n)$ Solution

Sort the time stamps. Scan the sorted list in order, identifying the maximum gap between successive time stamps.

Quadratic Time: $O(n^2)$

What kinds of things do you think take quadratic time?

Enumerate all pairs of elements.

Closest Pair of Points

Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is the closest.

$O(n^2)$ solution?

Try all pairs of points

```
min ← (x_1 − x_2)^2 + (y_1 − y_2)^2
for i = 1 to n
    do for j = i + 1 to n
        do d ← (x_i − x_j)^2 + (y_i − y_j)^2
           if d < min
              then min ← d
```

Do you think $\Omega(n^2)$ is a lower bound?

Cubic Time: $O(n^3)$

What kinds of things do you think take cubic time?

Set Disjointness

Given n sets $S_1, \ldots S_n$, each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ Solution

For each pair of sets, determine if they're disjoint.

```
for each set S_i
    do for each other set S_j
        do for each element p ∈ S_i
            do determine if p ∈ S_j
        if no element of S_i belongs to S_j
            then report S_i and S_j are disjoint
```

Polynomial Time: $O(n^k)$

Independent Set of Size k

Given a graph, are there k nodes such that no two are joined by an edge? (Where k is a constant.)

$O(n^k)$ Solution

Enumerate all subsets of k nodes.

```
for each subset S of k nodes
    do check whether S is an independent set
       if S is an independent set
          then report S is an independent set
```

- Checking if S is an independent set is $O(k^2)$ (enumerate pairs)
- The number of subsets of size k ("n choose k"):

$$\frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$$

- $O(k^2 n^k / k!) = O(n^k)$

Exponential Time

Some things are just plain expensive. And they're not always obviously expensive. (More later.)

Independent Set

Given a graph, what is the size of the largest independent set?

$O(n^2 2^n)$ **Solution**

Enumerate all subsets.

$S^* \leftarrow \emptyset$
for each subset S of nodes
    do check whether S is an independent set
        if S is largest independent set seen so far
            then update $S^* \leftarrow S$

Sublinear Time

Some things, when phrased properly, are just plain ~~easy~~ fast.

Can you think of anything you can do faster than $O(n)$?

Binary Search

Given a sorted array A of size n, determine whether p is in the array.
Start with BINARYSEARCH($A, 1, n, p$).

BINARYSEARCH(A, i, j, p)

```
m ← ⌊(j − i)/2⌋
if A[m] = p
    then return true
else if p < A[m]
    then return BINARYSEARCH(A, i, m − 1, p)
else return BINARYSEARCH(A, m + 1, j, p)
```

Binary search's running time is $O(\log n)$.

Lower Bounds for Sorting

Comparison Sorting

The algorithms you've likely encountered for sorting are comparison sorting algorithms that use only direct comparisons between the elements to sort them.

Consider only comparisons of $\leq$ (the following arguments generalize).

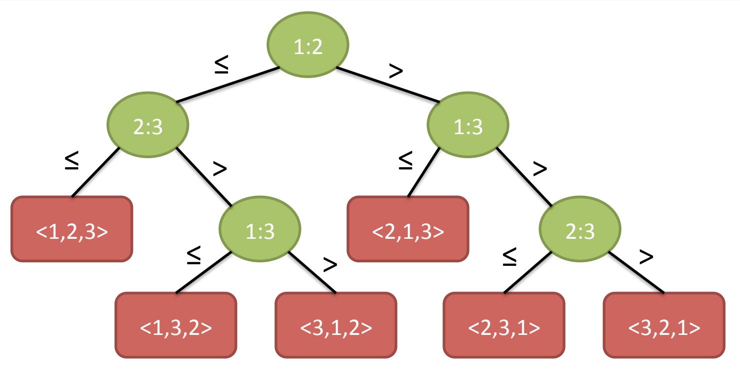
Decision Trees for Comparison Sorting

Decision Trees

We abstract comparison sorting in terms of *decision trees*.

- a full tree
- represents the comparisons between elements perfomed by a sorting algorithm
- each internal node is annotated by $i : j$ for some i and j in $1 \leq i, j \leq n$, for n elements in the input sequence
- each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$
- executing the sorting algorithm equates to tracing a path through the decision tree

Decision Tree Example



Any correct sorting algorithm must be able to generate each of the n! peruations on n elements; each permutation must appear as a leaf in the decision tree.

Lower Bound for the Worst-Case

Worst Case

The length of the longest path from the root of a decision tree to any of its reachable leaves is the worst-case number of comparisons that the corresponding sorting algorithm performs.

Said another way... the worst case number of comparisons for a comparison sort algorithm is the height of its decision tree.

Lower Bound

A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

Lower Bound for the Worst-Case (cont.)

Theorem

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof

We have to determine the height of a decision tree with $n!$ leaves. A binary tree of height h has no more than 2^h leaves. Therefore $n! \leq 2^h$, so $h \geq \lg(n!) = \Omega(n \lg n)$.

Corollary

Heapsort and Mergesort are asymptotically optimal comparison sorts.

Questions?