The complexity of problems

Problem complexity

So far we've only looked at the time complexity of algorithms.

For a given problem, we can ask:

> What is the worst-case time complexity of the fastest algorithm that solves the problem?

The answer is called the (worst-case time) complexity of the problem.

The answer will be in the form $\Theta(f)$ for some f.

Big Ω notation

Notation: For real-to-real functions f and g. $f \in \Omega(g)$ iff there exist positive c and M such that

$$0 \leq c \times g(N) \leq f(N), \text{ for all } N > M$$

$\Omega(g)$ is the set of functions that grow roughly as fast or faster than g.

Note $\Theta(g) = O(g) \cap \Omega(g)$

Lower bounds and Upper bounds

We can provide partial answers to the question of the complexity of a problem in two ways

- We can show that there is an algorithm for the problem that has a worst-case time complexity in $O(f)$ for some f. This establishes an **upper bound** of $O(f)$ on the complexity of the problem.

- We can show that every algorithm for the problem has a time complexity in $\Omega(g)$. This establishes a **lower bound** on the complexity of the the problem.

Lower bounds

Given a problem: how fast is the fastest algorithm that solves the problem?

Of course a quantitative answer will likely change as faster hardware becomes available.

We may be able to show that every algorithm that solves the problem is in $\Omega(f)$

In this case we know that $\Omega(f)$ is a lower-bound on the complexity of the problem.

Example Shortly we will show, any algorithm for sorting (based on comparisons) requires at least $\lceil \log_2 N! \rceil$ comparison operations and that this in $\Theta(N \log N)$ and so the problem has a lower bound of $\Omega(N \log N)$.

Upper bounds

Conversely if we know that there is an algorithm that is $\Theta(g(N))$ that solves the problem, then $O(g(N))$ is an upper-bound on the complexity of the problem.

Example We know MergeSort is $\Theta(N \log N)$, so sorting has an upper bound of $O(N \log N)$.

Tight bounds

If we know that the same complexity level is both an upper and a lower bound, then we have a “tight bound” or “exact bound”. I.e. if $O(f)$ is an upper bound and $\Omega(f)$ is a lower bound, then $\Theta(f)$ is a tight bound.

A tight bound means we know exactly how hard the problem is.

Until we know a tight bound, then either we have failed to find the asymptotically fastest algorithm or we have failed to prove the best possible lower bound, or both.

Example Since sorting (based on comparisons) has a lower bound of $\Omega(N \log N)$ and an upper bound of $O(N \log N)$, we know exactly how difficult sorting is.

We can make quantitative improvements on MergeSort, but there can be no algorithm with a better time complexity.

A lower bound for sorting

A restricted model of computation

In this model of computation two items may be compared or moved, but we will not otherwise access the value of an item.

An algorithm in this model can be thought of as a set of trees, one for each tree for size of input.

Each tree node either represents an action or a comparison. Action nodes have zero or one child, comparison nodes have 1 or 2 children. (A comparison whose conclusion is forgone has 1 child.)

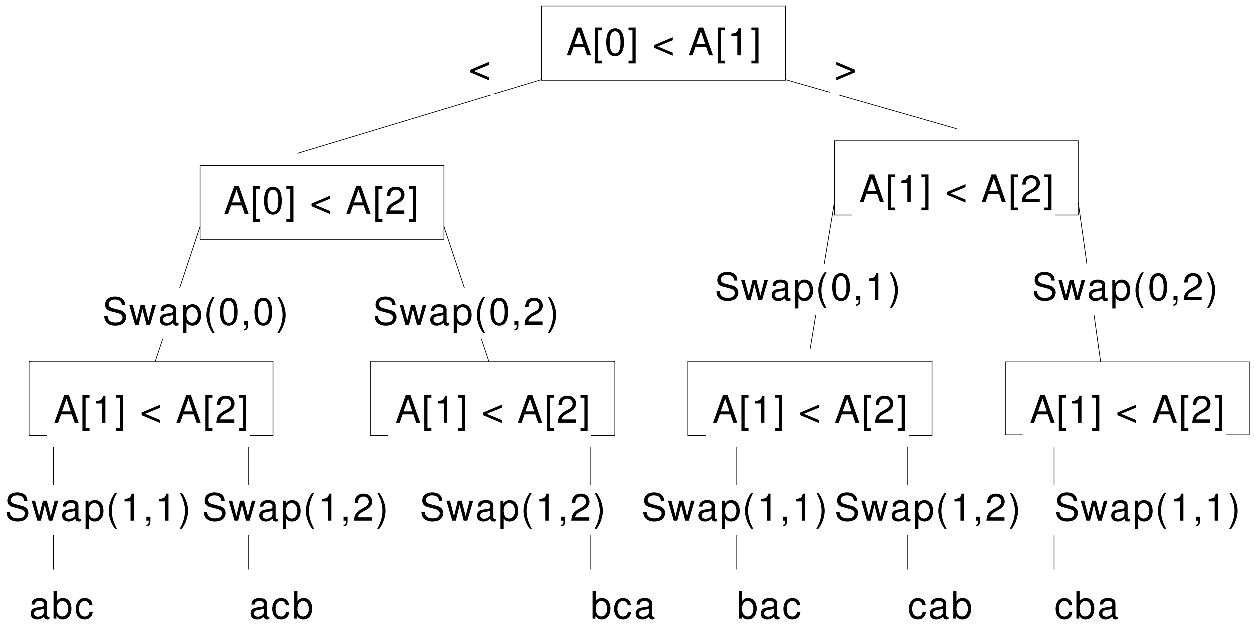
For simplicity, I'll assume that no two items of the input are the same.

[Considering only a restricted set of inputs is kosher, because any lower bound we find for the restricted problem must also hold for the unrestricted problem.]

Selection sort

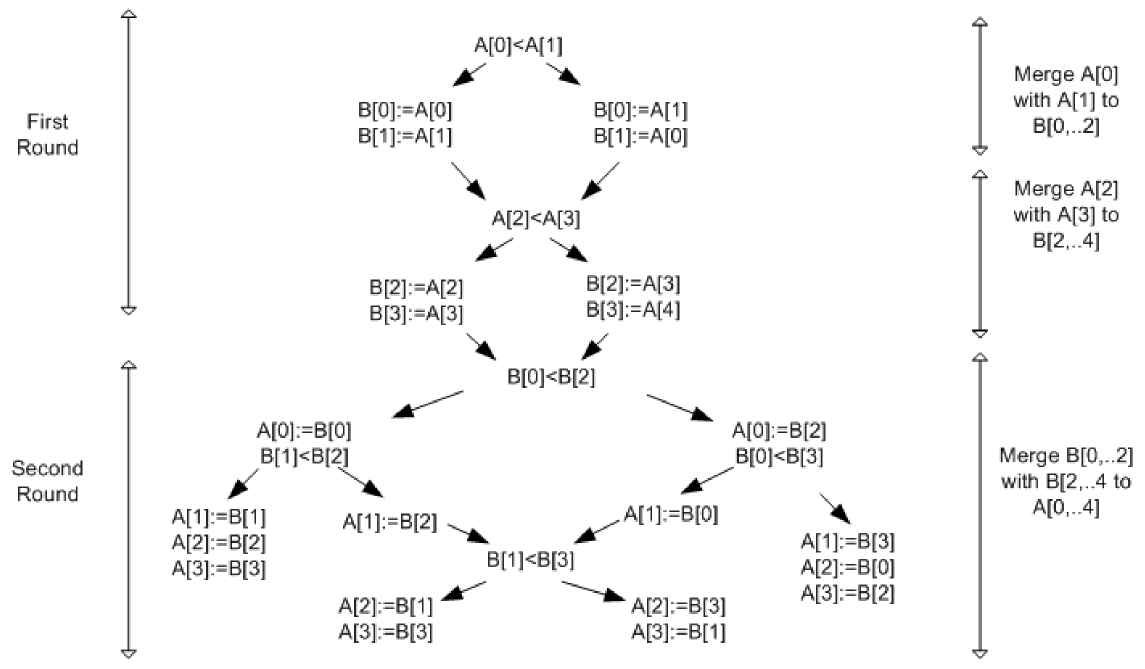
```
  Selection sort
for $i \leftarrow [0, ..n - 1]$ do
    var $j := i$
    for $k \leftarrow [i + 1, ..n]$ do if( $a[k] < a[j]$ then $j := k$ end
  if end for
    $\mathrm{swap}(i, j)$
end for
```

Here is a tree for selection sort with $n = 3$. Assume that $a < b < c$ the leaves show the input that corresponds to each path.



Merge sort

Next is a directed acyclic graph that can be expanded to a tree with 24 leaves representing merge sort for $n = 4$



For the tree model of computation

For a given n

- the best case time is the length of the shortest path from root to leaf
- the worst case time is the length of the longest path from root to leaf.

To simplify we'll only count comparisons.

[Since we are investigating lower bounds, it is kosher to ignore whole classes of operations. If a sorting algorithm requires at least $f(n)$ comparisons, then it must require at least $f(n)$ operations.]

Thus the worst (and best, and average) case time for selection sort is $\frac{n^2-n}{2}$ comparisons.

But selection sort is not the best algorithm. We want a result that even the best algorithm can not beat.

Merge sort, for $n = 2^k$, as we will show, requires $\Theta(n \log n)$ comparisons.

Considering all trees

The best algorithm has the shortest trees (as n approaches ∞)

Key key question is:

- *How short can a tree for an input of size n be?*

First we ask an easier question

- *How many leaves will a tree for an input of size n have?*

The inputs $[2, 1, 3]$ and $[4, 1, 6]$ look the same in this model as the only way to access the data is by comparing.

There are $n!$ distinct inputs, as there are $n!$ permutations of n distinct values.

Any two permutations of the input require the algorithm to do something different. Each possible input requires a different path through the tree and hence a different leaf.

There are $n!$ leaves in each tree for inputs of size n *regardless of the algorithm*.

So our key question becomes

- *If a tree has $n!$ leaves, what is the shortest its longest path can be?*

We need to consider the squattest trees.

Lemma 0 a binary tree of height x has at most 2^x leaves.

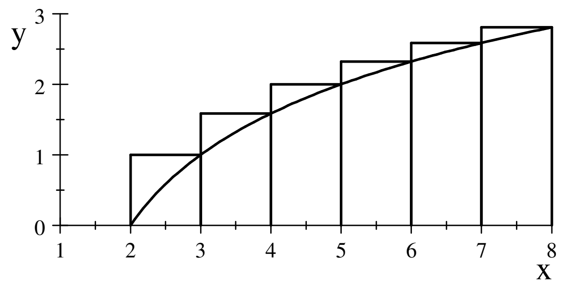
Corollary 0 a binary tree with y leaves has height at least $\lceil \log_2 y \rceil$.

Lemma 1 $\log_2(n!) \in \Omega(n \log n)$

Proof of lemma 1:

$$\begin{array}{ll} & \log_2(n!) \\ = & \\ & \log_2(1 \times 2 \times ...n) \\ = & \\ & \log_2 2 + \log_2 3 + ... + \log_2 n \\ > & \\ & \int_2^{n+1} \log_2(x-1)\ dx \end{array}$$

For the final step above, we provide a "proof by picture"



$\log_2 2 + \log_2 3 ... + \log_2 n > \int_2^{n+1} \log_2(x-1)\ dx$

$$\int_2^{n+1} \log_2(x-1)\ \mathrm{d}x$$

$=$

$$\frac{1}{\ln 2}\int_1^n \ln x\ \mathrm{d}x$$

$=$ “$\int \ln x\,\mathrm{d}x = x\ln x - x$”

$$\frac{1}{\ln 2}\left((n\ln n - n) - (1\ln 1 - 1)\right)$$

$=$

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

$\in$ “The dominant term is $\frac{1}{\ln 2}n\ln n$”

$$\Omega(n\log n)$$

Proof of the main result

The height of a binary tree with $n!$ leaves

$\geq$ “Corollary 0”

$\lceil \log_2 n! \rceil$

$\geq$ “Lemma 1”

$\Omega(n\log n)$

Therefore, for every algorithm, there is at least one input that requires at least $\Omega(n\log n)$ comparisons to sort.

So $\Omega(n\log n)$ is a lower bound on the complexity of sorting.

- There is no point trying to find an algorithm that is significantly better than merge sort or heap sort that accesses the data only by comparison.

Since $O(n \log n)$ is an upper bound and $\Omega(n \log n)$ is a lower bound, $\Theta(n \log n)$ is an exact bound.

We now know the complexity of sorting.

Unfortunately

Lower bounds are seldom so easy to prove.

How good is merge sort quantitatively?

By the way:

$$\lceil \log_2(1024!) \rceil = 8770$$

so merge sort with no more than

$$\begin{aligned} m(1) &= 0 \\ m(2^k) &= 2m(2^{k-1}) + 2^k - 1 \end{aligned}$$

$$m(1024) = 9217$$

comparisons is quite close to optimal.

How good is quicksort quantitatively?

Quicksort, on the other hand, takes (on average) $14,193$ comparison, for $n = 1,024$. Quicksort is quick because it does, on average, fewer moves than merge sort. Merge sort does roughly one move per comparison, whereas quicksort does about one move per two comparisons.)

But maybe there are quicker methods ...

... that don't use just comparison.

Indeed there are.

Consider sorting a large array of 9 digit social insurance numbers:

- Partition the sequence into 10 parts based on the first digit.
- Now re-partition each of the ten parts based on the second digit.
- Now re-partition each of the 100 parts based on the third digit.
- ...
- Finally re-partition each of the 100,000,000 partitions based on the last digit.

Each step can be seen to be $\Theta(N)$ so the 9 steps together are $\Theta(N)$.

But:

- We limited the number of different values that can be sorted to 10^9. Thus we subtly simplified the problem. Comparison based sorting is not thusly limited.