

1 Motivation: Problems vs Algorithms

Algorithms solve problems.

Problem	Algorithm
Closest pair of points	Divide-and-Conquer BruteForce Modified BruteForce ⋮
sorting	merge sort insertion sort bubble sort ⋮

So far, we have developed an important technique for analyzing algorithms: Time complexity analysis. Given an algorithm to a problem, how long does it take to execute on an input of size n ? Time complexity analysis is a cornerstone of *optimization*: Find the most efficient algorithm to a given problem.

What do we do when we have trouble designing an algorithm faster than what we currently have? It might very well be the case that we are just not thinking hard enough, or maybe it is *impossible* to design a faster algorithm. If it is indeed the latter, we are making a claim about the problem instance.

Take-home message:

**Every problem has an intrinsic “hardness,” “difficulty,” “complexity.”
Lower bounds allow us to *quantify* this attribute.**

There are 2 ways of attacking a problem: Optimization and Tightening of Lower Bounds.

2 Lower Bounds: Definition and Caveats

Characterizes the complexity of a problem. Informally, a lower bound is “the best you can do.” If we prove a lower bound L to a problem P , then *any* algorithm that solves this problem P *has* to have time complexity at least L . Well, this definition isn’t exactly perfect — we need to refine it a little bit. The pedant in us alerts us of a few important caveats:

Important Caveats:

- Lower bounds only apply to correct algorithms
 - Any problem can be solved incorrectly in constant time.
- Any lower bounds analysis assumes a Model of Computation
 - Collection of resources to help us solve the problem.

- Think of these resources as “black-boxes” or “oracles”.

If we prove a lower bound L to P , then any algorithm that solves P *correctly* and *assumes the same model of computation* as our analysis has to have time complexity at least L .

3 Specific Problem: Sorting

Problem: Given an array A of ints, return A' with the same elements as A , sorted in increasing order.

Model of Computation: Our model of computation is the comparison-based model:

- compare $A[i]$ and $A[j]$ for any index i and j .

How about swap, insert and delete? These do not help us solve the problem in a direct sense (they are needed only if we have to build our solution explicitly), so we do not include them within our model.

3.1 Decision Trees: Definition and Interpretation

Represents all possible computations with respect to a model of computation. $T = (V, E)$.

- Each node in V represents one query to an oracle. Think of this as “1 unit of computation”.
- Each edges in E represents the oracle’s answer. This is the information you gained at the price of the unit of computation you just paid.

3.2 Key Observations about decision trees

- Each leaf represents a possible answer to the problem.
- Each path from root to leaf represents a complete computation.
- Every input defines a path from the root to a leaf. In particular, it is the path corresponding to the computation the model executes on the input.
- A worst-case input is one that has a longest possible path length from root to leaf.

Height of decision tree

- Define the height of a tree to be the length of the longest path in it.
- Notice the length of such a path is equal to the number of internal nodes in it.
- Therefore,

$$\boxed{\text{worst case time complexity} \geq \text{height of decision tree}}$$

- Hence, in order to determine the worst case time complexity, we need to find the height of our decision tree

Leaves of decision tree

Claim 1: There must be at least one leaf for each possible permutation of the input array.

Proof. We will proceed by contradiction. Suppose not. Let P be a permutation for which there is no corresponding leaf. If we order the input by P , the algorithm will return a wrong answer. This contradicts the correctness of our algorithm (c.f. Important Caveats). \square

Claim 2: The maximum number of leaves in a binary tree of height h is 2^h .

Proof. Easy proof induction. □

By Claim 1, we have that

$$\text{number of leaves has to be } \geq n!$$

By Claim 2, we have that

$$\text{number of leaves } \leq 2^h$$

Therefore, it is necessarily the case that

$$2^h \geq n!$$

Solving for h gives us

$$h \geq \log_2 n!$$

Recall that by Stirling's approximation, we have that $n! > \left(\frac{n}{e}\right)^n$. Hence, it follows that

$$\begin{aligned} h &\geq \log_2 n! \\ &> \log_2 \left(\frac{n}{e}\right)^n \text{ since log is a monotonically increasing function} \\ &= n \log_2 n - n \log_2 e \end{aligned}$$

Therefore, asymptotically,

$$h = \Omega(n \log n)$$

We have established a lower bound on comparison based sorting: the worst case time complexity is at least $n \log n$. Equivalently, any correct, comparison based sorting algorithm is $\Omega(n \log n)$.

4 Generalizing the analysis

In the previous section, we proved that $h \geq \log_2 n!$. Where do the 2 and $n!$ come from?

Definition. The branching factor of a tree T is the number of children each node in T has.

For example, the branching factor of a binary tree is 2. In our interpretation, the branching factor of a decision tree is the number of different answers the oracle can give to each query.

If b is the branching factor of our decision tree, and l is the number of leaves, then we have that $h \geq \log_b l$. Equivalently,

$$h = \Omega(\log_b l)$$

5 Examples: Marble weighing puzzle

Problem: Given 9 marbles. 8 of them have the same weight, 1 of them is lighter than the rest. Find it in the least number of weighings.

Model of Computation: Balance

Algorithm:

```

if w(1,2,3) = w(4,5,6) \\ we know that it is 7, 8 or 9
  if w(7) = w(8) return 9
  else if w(7) < w(8) return 7
  else return 8
else if w(1,2,3) < w(4,5,6) \\ we know that it is 1, 2 or 3
  if w(1) = w(2) return 3
  else if w(1) < w(2) return 1
  else return 2
else if w(4,5,6) < w(1,2,3) \\ we know that it is 4, 5 or 6
  (symmetric to the above)
...

```

We have an algorithm that finds the odd marble in 2 weighings. Is this optimal?

Yes: Branching factor = 3, number of leaves = 9, so $h \geq \log_3 9 = 2$.

Problem: Given 9 marbles. 8 of them have the same weight, 1 of them is either lighter or heavier than the rest. Find it in the least number of weighings.

Model of Computation: Balance

Algorithm:

```

if w(1,2,3) = w(4,5,6) \\ we know that it is 7, 8 or 9
  if w(7) = w(8) return 9
  else if w(7) < w(8)
    if w(7) = w(9) return 8
    else if w(7) < w(9) return 7
  else if w(1,2,3) < w(4,5,6) \\ we know that it is NOT 7, 8 or 9
    if w(1,2,3) = w(7,8,9) \\ we know it is 4, 5 or 6, and it is heavier
      ...
    else if w(1,2,3) < w(7,8,9) \\ we know it is 1, 2 or 3, and it is lighter
      ...
    else if w(1,2,3) > w(7,8,9) \\ we know it is 1, 2 or 3, and it is heavier
      ...
  else if w(1,2,3) > w(4,5,6) \\ symmetric to the above

```

Is this algorithm optimal?

Yes: Branching factor = 3, number of leaves = 18, so $h \geq \lceil \log_3 18 \rceil = 3$.

6 Limitations of the Decision Tree Method

Problem: Given an array A of ints, return the maximal element in A .

Model of Computation: Comparison-based model (same as sorting)

Using the decision tree method, we yield a lower bound of $\log_2 n$. But is this bound tight?