2.1 Lower Bounds for Algorithms

A big question. So far you have learned several sorting algorithms such as Merge Sort, Quick Sort, HeapSort, and Bubble Sort. All of them have running time $\Omega(n \log(n))$. Is this $\Omega(n \log(n))$ bound a true barrier that no sorting algorithm can bypass it? Also you might wonder if there is a faster way to search a sorted array than Binary Search. This leads to a natural question: For a problem $X$, can we show that no algorithm can solve $X$ faster than $\Omega(T(n))$?

Comparison-based algorithms. First note that for the sorting problem, if we are willing to make some certain assumptions on the input then it is possible to bypass the $\Omega(n \log(n))$ barrier. For example, if each element of the array to be sort is a sequence of $d$ digits, then we can use Radix Sort for $O(nd)$ time. To avoid such loopholes, we will consider only comparison-based algorithms. By comparison-based, I mean that the algorithm makes all its decisions based on how the elements of the input array compare to each other, and ignore the actual values. Pictorially, you may view that the algorithm has to process $n$ objects that look identical but have different weights. The only tool your algorithm has is an old-fashioned pan balance: you can put two objects to the weighing dishes and tell which is heavier, but that’s all. Merge Sort, Quick Sort, HeapSort, and Bubble Sort all fall into this category.

The restriction to comparison-based algorithms is not meaningful for many problems, such as finding a shortest path between two nodes in a graph or matrix multiplication. Still, it is useful in several interesting scenarios, such as sorting an array or searching a sorted array.

An illustrative example: Sorting. We now show that any comparison-based sorting algorithm needs $\Omega(n \log(n))$ time on an array of size $n$. When we aim to sort an array $A = [a_1, \ldots, a_n]$, we actually want to output a permutation of the input in which the items are in sorted order. Initially, there are $n!$ possible permutations. A comparison, say “Is $a_1 < a_2$?”, would divide the set of these $n!$ permutations into two groups: (i) the Yes group, consisting of permutations like $(a_1, a_2, \ldots, a_n)$, and (ii) the No group, consisting of permutations like $(a_2, a_1, a_3, \ldots, a_n)$. In other words, your algorithm can be represented as a binary tree (known as a decision tree) where each node is associated with a comparison, its left branch corresponds to a Yes answer, and its right branch corresponds to a No. Each leaf corresponds to a permutation, and running the algorithm on a specific input means going from the root to the corresponding leaf of the sorted output.

The worst-case running time of your algorithm is the height of the decision tree. Since there are at least $n!$ leaves, the height will be at least $\log_2(n!)$. Using the property that $\log_2(ab) = \log_2(a) + \log_2(b)$ for every $a, b > 0$,

\[
\log_2(n!) = \log_2(1 \cdot 2 \cdot 3 \cdot n) = \log_2(1) + \log_2(2) + \cdots + \log_2(n)
\]

In Section 2.3, we’ll learn a powerful technique that helps us to prove that

\[
\log_2(1) + \log_2(2) + \cdots + \log_2(n) = \Omega(n \log(n))
\]

In conclusion, any comparison-based sorting algorithm must use at least $\Omega(n \log(n))$ comparisons, and thus needs $\Omega(n \log(n))$ running time.
Takeaway lesson. To prove lower bounds on comparison-based algorithms for a particular problem, you need to find a way to represent those as decision trees. In our sorting problem, a comparison returns two possible outcomes (Yes/No) so the tree is binary, but a general comparison operator may return more outcomes (for example, “equal”, “less than”, or “greater than”). If each comparison returns $t$ possible outcomes then you’ll have a $t$-ary decision tree. Moreover, you need to turn your problem into a search problem (so sorting becomes searching for a permutation out of $n!$ ones).

2.2 Another Example: Coin Weighing

**Question:** You are given 12 balls, all equal in weight except for one that is either heavier or lighter. You are also given a two-pan balance to use. Your task is to design a strategy to determine which is the odd ball and whether it is heavier or lighter than other balls using as few uses of the balance as possible.

The discovery dialog. Below is a dialog between a teacher and a student for solving this problem.

   *Student:* So here the problem asks us to minimize the number of weighings to find the odd ball. I think it should be easier to drop the requirement of minimum number of weighings. In addition, it might be good to go to the general problem of $N$ balls so that we can see the big picture. For this generalized problem to make sense, we need $N \geq 3$.

2. *Teacher:* Can you construct an input example small enough to solve by hand? What happens when you try to solve it?
   *Student:* I can solve $N = 3$ easily. I’ll put one ball on the left pan, another on the right. If balanced, the odd ball is the remaining one. If not, the remaining ball is an ordinary ball. I can then compare it with, say the ball on the left pan.

3. *Teacher:* Let’s try to apply this idea for the general problem. For the first weighing, let’s say you put $B$ balls on the left pan, and other $B$ balls on the right pan. If balanced, how would you proceed?
   *Student:* It means that the odd ball is within the remaining $N - 2B$ balls. I can recursively run the algorithm on those $N - 2B$ balls.

4. *Teacher:* What if the first weighing is not balanced? For concreteness, go back to $N = 12$ and $B = 3$.
   *Student:* In that case it means that the remaining 6 balls are ordinary balls. I can weigh three of them against the balls on the left pan. In any case, I can locate the odd ball within three balls, and recursively run the algorithm.

5. *Teacher:* Describe your algorithm for general $N$ and $B$ when the first weighing is not balanced.
   *Student:* This means that the remaining $N - 2B$ balls are ordinary balls. I can weigh the $B$ balls on the left against $B$ ordinary balls on the remaining pile. In any case, I can locate the odd ball within $B$ balls, and recursively run the algorithm.

6. *Teacher:* Any requirement on the number $B$?
   *Student:* I need $B \leq N - 2B$ so that I can pick $B$ balls from the remaining pile of $N - 2B$ balls. This means that $B \leq N/3$. 
7. **Teacher**: Any intuition on what’s the best choice of $B$?

**Student**: It looks like the bigger $B$ is, the faster the algorithm is, because after the first weighing, we need to recursively solve the problem of size up to $\max\{B, N - 2B\} = N - 2B$. Given that $B \leq N/3$, the best choice is probably $B/3$.

8. **Teacher**: So now you have an algorithm to find the odd ball with a parameterization on $B$. Let’s pick $B = N/3$ as you suggest. What’s the asymptotic cost?

**Student**: This is like binary search but better because after $O(1)$ operations, you can narrow down the odd ball to pile of size $N/3$. So the cost is of order $\log_3(N)$. Thus the running time is $\Theta(\log(N))$.

9. **Teacher**: Let’s use the algorithm on $N = 12$. How many weighings would you need?

**Student**: It would take me 3 weighings.

10. **Teacher**: Now we need to find an algorithm with optimal cost. We aim to prove that your algorithm is optimal. The question asks about concrete cost, which is difficult. Let’s first aim for asymptotic cost first. Let’s say we want to prove that any algorithm needs $\Omega(\log(N))$ time to find the odd ball. Do you know how to prove a lower bound of running time for a problem? Have you seen anything like that?

**Student**: Yes, I know that comparison-based sorting needs at least $\Omega(\log(N))$ time. An our coin-weighing problem also deals with comparison-based algorithms. But I’m still not sure how to represent a coin-weighing algorithm as a decision tree.

11. **Teacher**: Let’s re-examine the sorting problem. A key insight there is to realize that the output is from the set of all permutations of $[a_1, \ldots, a_n]$. What about our coin-weighing problem? What exactly is the output?

**Student**: A number $i \in \{1, \ldots, N\}$ (meaning that ball $i$ is the odd one), and whether it’s heavier or lighter than other balls.

12. **Teacher**: Could you encode that using some math symbols so that it’s easier to manipulate? Let’s say the odd ball is ball 9, and it’s heavier.

**Student**: I can write this as $9+$.

13. **Teacher**: So what are all possible values of the output?

**Student**: The output would be from $\{1+, 1-, \ldots, N-, N+\}$.

14. **Teacher**: Back to the sorting problem. Another insight there is to realize that each comparison would divide the target set of permutations into two groups: the Yes group, and the No one. For our coin-weighing problem, when you weigh some balls, how many groups would the answer divide?

**Student**: Three groups, depending on the three possible outcomes of the weighing. So we actually have a tertiary decision tree in our case. Each leaf of the tree corresponds to an element of $\{1+, 1-, \ldots, N-, N+\}$.

15. **Teacher**: What is the worst-case running time of an algorithm for the coin-weighing problem?

**Student**: It’s the height of the tree. Since there are $2N$ leaves, the height is at least $\log_3(2N) = \Omega(\log(N))$.

16. **Teacher**: We are done with proving optimal asymptotic cost. What about concrete cost? Let’s get back to our original problem where $N = 12$. How many weighings would you need?

**Student**: I will need $\lceil \log_3(24) \rceil = 3$ weighings at the very least. Since my algorithm uses exactly 3 weighings, it is optimal.
Reflection. When you have a hard problem, it’s always a good strategy to consider a related, easier problem. This might be a problem that you solved before, or you can relax the current problem by dropping some requirements, or make some additional assumptions. A goal of this course is to teach you how to pick the “right” related problems via many examples. In this situation, we have a series of related problems.

• First, we relax the problem by dropping the requirement of using the minimum number of weighings, making the problem easier. Then, instead of dealing with 12 balls, we consider a general problem of N balls. Generalization will help you escape the mass of numerical details, so that you can see the forest from the trees. In this case we realize that after the first weighing, we can narrow down the odd ball to a smaller pile, and can use a recursion to solve it.

• Next, to tackle the question of finding an optimal use of the balance, instead of dealing with concrete numbers, we only consider asymptotic complexity, making the problem easier. This reminds us of the related problem of proving lower bound for comparison-based sorting.

2.3 Tight Estimation of A Sum

Suppose that we have a sum \( T(n) = f(1) + \cdots + f(n) \), and we want to find the Big-Theta of \( T(n) \). In this section, we’ll see a powerful method for this task.

2.3.1 Lower Bound

A warmup problem. Let’s begin by giving a lower bound of
\[
T(n) = 1^2 + 2^2 + \cdots + n^2.
\]

Let’s plot the curve \( f(x) = x^2 \) as shown in Figure 2.1. Note that the area of the first rectangle is exactly \( 1^2 \) because its width is 1 and its height is \( f(1) = 1 \). Likewise, the area of the second rectangle is \( 2^2 \), and so on. Thus \( T(n) \) is the total area of those \( n \) rectangles. Moreover, those rectangles cover the area under the curve \( f(x) \) from 0 to \( n \). Hence the sum of the area of those rectangles is at least the area under \( f(x) \), from 0 to \( n \). But if you recall from your calculus class, the latter is simply
\[
\int_0^n f(x)dx = \frac{n^3}{3}.
\]

Hence
\[
T(n) \geq \frac{n^3}{3}
\]
for every \( n \geq 1 \). To see that the bound is good, note that
\[
T(n) = \frac{n(n+1)(n+1/2)}{3},
\]
and thus the estimation is quite close to the actual value.

A harder problem. Let’s now give a lower bound for
\[
T(n) = \ln(1) + \ln(2) + \cdots + \ln(n).
\]
Our original problem uses $\log_2$, but remember that the base of the log doesn’t matter in Big-Omega, as the difference is just a multiplicative factor. Here if you deal with integration, it’s much easier to use the natural base. That’s why we consider $\ln$ instead.

The approach above will continue to work for this problem. Again, let’s plot the curve $f(x) = \ln(n)$, as shown in Figure 2.2,

The area of the first rectangle is again $\ln(2)$, and the area of the second one is $\log_2(3)$, and so on. Since $\ln(1) = 0$, the value $T(n)$ is simply the total area of those rectangles, which is greater than the area under the curve, from 1 to $n$. However, the latter is simply

$$\int_1^n f(x)dx = n\ln(n) - (n - 1) \in \Omega(n \log(n)) .$$

### 2.3.2 Upper Bounds

Suppose that we want to estimate an upper bound of

$$T(n) = 1^4 + 2^4 + \cdots + n^4 .$$
A simple approach is to observe that each term $i^4$ is smaller than $n^4$. Since there are $n$ terms, $T(n) \leq n^5$. In many cases, this naive solution is enough. However, sometimes we want a tighter estimation. Again, the integration method is useful here. Let’s plot the curve $f(x) = x^4$ as shown in Figure 2.3.

![Figure 2.3: The curve $f(x) = x^4$ and its rectangles.](image)

Note that the area of the first rectangle is exactly $1^4$ because its width is 1 and its height is $f(1) = 1^4$. Likewise, the area of the second rectangle is $2^4$, and so on. Thus $T(n)$ is the total area of those $n$ rectangles. Moreover, those rectangles is completely covered by the area under the curve $f(x)$ from 1 to $n + 1$. Hence the sum of the area of those rectangles is at least the area under $f(x)$, from 1 to $n + 1$. But if you recall from your calculus class, the latter is simply

$$
\int_{1}^{n+1} f(x)dx = \frac{(n + 1)^5 - 1}{5}.
$$

Hence

$$
T(n) \leq \frac{(n + 1)^5 - 1}{5}
$$

for every $n \geq 1$. To see that the bound is good, note that

$$
T(n) = \frac{n(n + 1)(n + 1/2)(n^2 + n - 1/3)}{5}
$$

for every $n$. So the estimation is very close to the actual value.