Contents

1 Introduction

We begin with a computational problem, a problem where, given an input and a set of conditions, the goal is to return an appropriate output. The input set varies for each problem, as well as the output.

A computational problem is solved if we can produce an appropriate output for all possible inputs in some finite time scale.

An algoritm is a sequence of unambiguous instructions that can solve a well-defined computational problem. Essentially, we take the input, run it through the algorithm, and produce an output.

We care about these because most problems that we want to solve in the real world are computational problems, or compositions of computational problems. We want our solution algorithms to be correct, efficient, and understandable. We also might want to compare multiple algorithms for a problem as well, comparing speed or efficiency, memory usage, and space usage. There can also be nice properties that we might want, such as stability.

We want our algorithms to be independent of our programming language, we want to generate a process, rather than instructions in code.

2 Big Notation

The motivation for studying big notation is analyzing an algorithms' time and space complexity, or for comparing which algorithm is 'better'. One metric for this is how performance scales with input size. Big notation is a formal method for comparing runtime and space requirements as the input size grows.

2.1 $\mathcal{O}$ (Upper Bound)

Consider two functions, f and g. We say that $f(x) = O(g(x))$ if

$$(\exists x_0, C > 0)(\forall x \geq x_0)(f(x) \leq C \cdot g(x))$$

Intuitively, this says that there is some constant C and some starting input x_0 such that for all inputs $x \geq x_0$, $f(x)$ is no larger than $C \cdot g(x)$. Alternatively, this says that $f(x)$ is eventually always smaller than some multiple of $g(x)$. For the most vague statement, $f(x)$ is *bounded* by $g(x)$.

Note that it is possible for two functions to be Big-O of each other, for example, if they are the same function, $f = g = x$.

Let us show that $x^2 + \lg(x) + 4 = O(x^2)$ (Note that lg represents the log base 2). We need to show that there exists some $x_0 > 0$ and $C > 0$ such that for all $x \geq x_0$, $f(x) \leq C \cdot x^2$.

We know that for $x > 0$, $\lg(x) < x$. We know that

$$x^2 + \lg(x) + 4 \leq x^2 + x + 4$$

For positive x. We know that when $x \geq 1$, $x \leq x^2$, letting us do another inequality:

$$x^2 + x + 4 \leq x^2 + x^2 + 4$$

for $x \geq 1$. We also know that for $x \geq 2$, $4 \leq x^2$:

$$x^2 + x^2 + 4 \leq x^2 + x^2 + x^2 = 3x^2$$

If we let $C = 3$, we find that we have shown that if $x \geq 2$, then our function is less than or equal to our $g(x)$, x^2. Thus we have that $C = 3$ and $x_0 = 2$, satisfying the definition of our Big O, and proving that our function is bounded by x^2. Essentially, we use a sequence of inequalities to modify the function f, to eventually be some constant scaling our function $g(x)$.

Suppose $f(x) = O(g(x))$, and $g(x) = O(h(x))$. Is it true that $f(x) = O(h(x))$? Intuitively, this asks whether the Big O is transitive. Writing this out semi-formally:

$$x \geq x_0 \rightarrow f(x) \leq Cg(x)$$

$$x \geq x_1 \rightarrow g(x) \leq Dh(x)$$

Via substitution, we have that

$$x \geq \text{MAX}(x_0, x_1) \rightarrow f(x) \leq CDh(x)$$

2.2 Ω (Lower Bound)

Suppose we have two functions, f and g that are both $O(h(x))$. Is it then true that at least one of f or g is O of the other? Via counterexample, we can show that this is not true, as we could have functions like $f(x) = \sin(x)$ and $g(x) = \cos(x)$. Both of these are $O(1)$, and yet due to their oscillatory nature, neither of them are bounded by each other.

Consider two functions $f(x)$ and $g(x)$. We say that $f(x) = \Omega(g(x))$ if

$$(\exists x_0, C > 0)(\forall x \geq x_0)[f(x) \geq C \cdot g(x)]$$

This is essentially a lower bound, where O was an upper bound. Note that the only thing that has changed is that now $f(x)$ will always be greater than or equal to $g(x)$.

We can also think of this as f being eventually always larger than some multiple of g.

Suppose $f(x) = \Omega(g(x))$, is it true that $g(x) = O(f(x))$? We claim that this is true, as the definition of big Omega means that past some x_0, and for some C, $f(x \geq x_0) \geq Cg(x)$. We can then divide by C:

$$\frac{f}{C} \geq g \rightarrow g = O(f(x))$$

Let us do an example of computing Ω. We want to show that $\frac{1}{3}x^3 - \frac{1}{x} = \Omega(x^2)$. We can begin by using the fact that for $x \geq 1$, $x^3 \geq x^2$:

$$\frac{1}{3}x^3 - \frac{1}{x} \geq \frac{1}{3}x^2 - \frac{1}{x} \; (x \geq 1)$$

Now to deal with the $-\frac{1}{x}$, we note that $\frac{1}{x} \leq 1$ when $x \geq 1$, and $-\frac{1}{4}x^2 \geq 1$:

$$\frac{1}{3}x^2 - \frac{1}{x} \geq \frac{1}{3}x^2 - \frac{1}{4}x^2 = \frac{1}{12}x^2 = \Omega(x^2)$$

Where $C = \frac{1}{12}$, and $x_0 = 2$. Note that our choice for $\frac{1}{4}$ was used in order to make sure that $C > 0$.

Suppose that $f(x) \neq O(g(x))$. Must it be true that $f(x) = \Omega(g(x))$? We claim that this is false, because we can once again bring up the oscillatory case, with sines and cosines.

2.3 Θ (Exact Bound)

We have Θ, where $f(x) = \Theta(g(x))$ if

$$(\exists x_0, c, C > 0)(\forall x \geq x_0)(cg(x) \leq f(x) \leq Cg(x))$$

Intuitively, we are saying that f is caught between two multiples of g if you go out far enough. We can also say that $g(x)$ is an exact bound for $f(x)$. Θ is the most descriptive of the 3 definitions of the bound on $f(x)$.

Note that in order for $f(x)$ to be $\Theta(g(x))$, it must be both $O(g(x))$ as well as $\Omega(g(x))$.

Let's do an example. Suppose we have the function $x^3 + 2\sin(2x) + 4$. We want to show that it is $\Theta(x^3)$.

Looking at the sine function, we know that it is bounded by ± 1. Thus we cna do this in both directions. Suppose we are looking for an upper bound. We can write that

$$x^3 + 2\sin(2x) + 4 \leq x^3 + 2 + 4 \leq x^3 + x^3 = 2x^3 = O(x^3)(x \geq \sqrt[3]{6})$$

For a lower bound:

$$x^3 + 2\sin(2x) + 4 \geq x^3 - 2 + 4 = x^3 + 2 \geq x^3 = \Omega(x^3)\ (x \geq 0)$$

Thus we have that $f(x) = O(x^3)$ for $C = 2$ and $x_0 = \sqrt[3]{6}$, and $f(x) = \Omega(x^3)$ for $C = 1$ and $x_0 = 0$. Thus putting these together, $f(x) = \Theta(x^3)$ for $c = 1$, $C = 2$, and $x_0 = \sqrt[3]{6}$.

2.4 Limit Method

We care about what happens eventually, so we care about the limit.

Theorem 2.1. *Consider two functions $f(n)$ and $g(n)$. Assume that the limits both exist. Then, if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

Then $f(n) = \mathcal{O}(g(n))$.

If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$$

then $f(n) = \Omega(g(n))$. Using these two, we have that if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0, \infty$$

then $f(n) = \Theta(g(n))$.

Intuitively, these make sense, because if f outpaces g, then we expect an infinite limit, so if its not ∞, then g is an upper bound for f. The same logic holds for the lower bound. Note that the converse is not true, as we can have oscillatory functions.

Note that should the limit not exist, we cannot conclude anything.

Suppose we want to prove that $\ln x = \mathcal{O}(\sqrt{x})$:

$$\lim_{x\to\infty} \frac{\ln x}{\sqrt{x}}$$

We can see that this is an indeterminate form, so we have to use L'hopital's rule, taking the derivative of both numerator and denominator:

$$\lim_{x\to\infty} \frac{\ln x}{\sqrt{x}} = \lim_{x\to\infty} \frac{2}{\sqrt{x}} = 0 \to \ln x = \mathcal{O}(\sqrt{x})$$

Suppose we want to prove that $20n^{10} = \mathcal{O}(4^n)$. Setting up the limit:

$$\lim_{n\to\infty} \frac{20n^{10}}{4^n} = \lim_{n\to\infty} \frac{200n^9}{\ln 44^n}$$

We see that after 1 application of L'hopital's rule, we decreased the order of the numerator, and left the denominator effectively unchanged. To solve this issue, we keep taking derivatives until it works, since this is still indeterminate:

$$\lim_{n\to\infty} \frac{20 \cdot 10!}{(\ln 4)^{10} 4^n} = 0 \neq \infty \to 20n^{10} = \mathcal{O}(4^n)$$

3 Rigorous Time

We often care about the total time an algorithm takes as an overall measure of quality. The general rule is thta anything the compute does takes some amount of time. Rigorous time is a math framework for analyzing the time an algorithm takes independent of the hardware.

In 351, we have our own conventions for what takes time. The first thing that we look at is assignments.

```
x = 0
y = 1
z = 2 + x - y
```

In real life, each one might take some different amount of time, but we don't care, we assume that they all take some time c_a. We only care about scaling behavior, so we can just say that an individual assignment add just $\mathcal{O}(1)$, essentially adding some constant time overhead.

We also have time for loop construction:

```
prod = 1
for i = 1 to n
    prod = prod * i
end
```

In this case, the body takes constant time, since its just an assignment. The total time is given by

$$c_a + n \cdot c_m + n \cdot c_a$$

where c_m is the maintenance time for the loop, and runs as many times as the loop does. We can see that the complexity of this will be $\mathcal{O}(n)$.

For a while loop:

```
prod = 1
i = 1
while (i < n)
    prod = prod * 1
end
```

We have the same amount of maintenance time c_m, and the same body constant time in this case.

For conditionals:

```
if (max < y)
    max = y
end
```

We see that we have two outcomes, either this takes c_c, the time for the conditional, or it takes the comparison time as well as the time in the body. If the body is constant time, then the whole conditional is constant time. However, if there is a linear time function inside, or any other nonconstant time body, we have to account for this using branches.

We can combine these statements:

```
if (x > 5)
    sum = 0
    for i = 1 to n
        sum = sum + x
    end
end
print(sum)
```

The path this program takes changes the overall time complexity. If $x > 5$, we have that $c_c + (n + 1)c_a + \mathcal{O}(1)$, which is $\mathcal{O}(n)$. If $x \leq 5$, then we have constant runtime, as we only have the conditional evaluation and the print statement.

Suppose a conditional branch never happens. In this case, does it affect the time complexity to have anything in that branch? Since we only care about the execution path, we don't care about anything in that conditional.

Suppose we have two algorithms, A and B, and via complexity analysis we see that $A = \mathcal{O}(n)$ and $B = \mathcal{O}(n^2)$, but we see that B still runs faster on every input that we test. How could this happen?

We could have tested small input sizes, we haven't passed n_0 yet. Or it could be that we have better coefficients for the n^2 algorithm than for the n algorithm. We coud also have hardware dependence. It could also be that the worst case input for algorithm B rarely happens, only for a very small set of inputs perhaps. If we know features of the input that make it run faster on B than A, such as B running faster if the numbers inserted are even.

We can see that complexity analysis only really gives us a heuristic, not a perfect comparison.

Moving back to rigorous time, we cna do simplified analysis, where we condense constants into one constant if they happen in succession and all take the same order of time.

```
a = 1
b = 2
c = 3
for i = 1 to n
    a = a + i
    b = b - (c * i)
end
for j = 1 to n
    b = b + i
end
```

For example, we could condense the 3 starting assignments into some constant c_1. The body of the first for loop can be condensed to c_2, and we see that it runs n times in the loop. The body of the second loop is also constant, call that c_3, and also runs n times. Note that we are lumping the maintenance into the constants. Thus this code will run in time:

$$T = c_1 + c_2 n + c_3 n = n(c_2 + c_3) + c_1 = c_4 n + c_1 = \mathcal{O}(n)$$

4 Types of Analyses

For tracking any quantity in an algorithm, there are 3 main cases. The first is the worst case, where we have the worst possible input. The second is the opposite of this, the best case, given an input that minimizes the amount of the quantity. Finally, we have the average case, which takes in an "average" input, which depends on knowing what every input might look like, as well as their probabilities.

We can look at these in terms of exact sums or $\mathcal{O}$ complexity, or ideally the Θ complexity.

Suppose A and B are algorithms such that A's worst case runtime is $\mathcal{O}(n^3)$ while B's worst case is $\mathcal{O}(n^2 \log(n))$. It may seem obvious that we want to pick B, but there are some times we might want to use A instead. For example, if the worst case for A is unlikely, then A might be a better choice, or if the best/average case for A is much better than the best/average cases for B. Another possibility is that we have better constants for A than for B. Another possibility is that you have limited amounts of space, in which case A might have better space complexity than B.

5 Maximum Contiguous Sum

This is the first computational problem that we will be solving.

Consider a list of integers. A continuous sublist is a sublist that contains any consecutive block of elements from the original list, in the same order. For example, if our original list is $[1, 2, -7, 3, 1, 1, -8]$, the lists $[2, -7, 3]$ and $[-7, 3, 1, 1, -8]$ would be contiguous sublists. Note that not all sublists are contiguous.

We define the sum of a list as the sum of all its elements.

The MCS problem is that we are given a list, and we want to find the largest sum of any of its nonempty contiguous sublists it has. In the example given before, the maximum contiguous sum is 5, given by the contiguous sublist $[3, 1, 1]$.

We will look at 4 different approaches to solving this problem.

5.1 Brute Force Solution

The brute force solution is to test every possible contiguous sum. This seems like a bad idea. If we have a list of length n, how many nonempty continguous sublists do we have?

The number is given by the Gaussian sum, and is $\frac{n(n+1)}{2}$. This is because starting at the first element, we have n contiguous sublists starting from there. For the second element, we have $n-1$. For the third, $n-2$. We see that we have the sum

$$n + (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

```
max = A[0]
for i = 0 to n -1
    for j = 0 to n - 1
        sum = 0
        for k = i to j
            sum = sum + A[k]
        end
        if sum > max
            max = sum
        end
    end
end
```

We see that we have a starting constant time operation c_1. We then have two loops that run n times, and then another constant assignment c_2. The inner loop from i to j will run on average $n/2$ times (0 times best case, n times worst case), and has a constant time body, c_3. We then have a constant time comparison, c_4. Thus we have

$$c_1 + n(n(c_2 + \frac{n}{2}c_3 + c_4)) = c_1 + n^2(c_2 + \frac{c_3 n}{2} + c_4) = c_1 + \frac{c_3}{2}n^3 + n^2(c_2 + c_4) = \Theta(n^3)$$

5.2 Naive Solution

The brute force method kept recomputing the same sums, and then adding on a new term. The new solution computes all the sums starting at index i in order, storing the current sum and just adding the next element as a constant time operation.

Consider a list indexed from 0 to $n-i$. We have $n-i$ contiguous sublists starting from any index i.

```
max = A[0]
for i = 0 to n-1
    sum = 0
    for j = 0 to n - 1
        sum = sum + A[j]
        if sum > max
            max = sum
        end
    end
end
```

We can see that this becomes a $\Theta(n^2)$ algorithm.

5.3 Divide and Conquer

There are three cases for the sublist with the MCS. Either it is entirely contained in the first half of the list, it is contained in the second half of the list, or it straddles both halves of the list.

If our list is length 1, then it is its own MCS. Thus we can recurse until we hit length-1 lists.

Consider a length n list. How many times can we reduce its size by half before it is length 1. The answer to this is $\lg(n)$.

```
function mcs(A, 1, r)
    if 1 == r
        return A[1]
    else
        c = (1 + r) / 2
        lmax = mcs(A,1,c)
        rmax = mcs(A, c+1, r)
        cmax = // omitted O(1)
        return max(lmax, rmax, cmax)
    end
end
```

We have that this algorithm is $n \lg n$.

5.4 Kadane's Algorithm

The best way to solve MCS is dynamic programming, soring and reusing information rather than recomputing it, wherever possible. Kadane's uses the property that if M_i is the MCS that ends at index i. We have two cases, either adding the next element will be larger than the sum, or the stored sum will still be the maximum ending at that position. Kadane's algorithm exploits this theorem to solve the problem.

Theorem 5.1. *Let M_i denote the MCS ending at i. We have two cases. Either $M_i = M_{i-1} + A[i]$ or $M_i = A[i]$.*

```
maxoverall = A[0]
maxendingati = A[0]
for i = 1 to n - 1
    maxendingati = max(maxendingati + A[i], A[i])
    maxoverall = max(maxoverall, maxendingati)
end
```

We see that the time complexity is

$$c_1 + \sum_{i=1}^{n-1} c_2 = c_1 + c_2 \sum_{i=1}^{n-1} 1 = c_1 + c_2(n-1)$$

Thus we have that Kadane's algorithm is $\mathcal{O}(n)$. Note that we can also say this is $\Theta(n)$, because we don't depend on the input.

Note that it is not possible to do this any better, because we have to visit every single element at least once. To prove this, suppose we have an array of all -1s, and the last element is $10,000$. If we don't visit the last element, we would have missed the MCS. This means that we have to be at least $\mathcal{O}(n)$, we can't do any better, Kadane's is an optimal solution.

6 Sorting

Mathematically, sorting is based on an *ordering*, which is a function that takes a pair of elements $a_1, a_2 \in A$, and says whether a_1 is "less" than a_2. Formally, it is a function $f : A \times A \to \{\text{True}, \text{False}\}$. Generally, we use $<$ to represent the ordering.

Consider a list L whose elements are all in set A. Assume we have an ordering $<$ on A. The problem of sorting L asks us to return a list L' such that L' contains exactly the same elements as L, with the same frequency, and if $i > j$, then either $L[i] = L[j]$ or $L[i] < L[j]$. Note that this defines a descending sort.

There are 4 main properties of sorting algorithms that we care about. The first is time complexity, how does the time taken scale based on the input size? The second is Axuiliary Space complexity, which represents how the overall memeory used scales as the input scales. The third is stability, which represents if idential entries are kept in the same relative order after the sort. Finally, we have whether it is in-place, which is whether it reuses the original list's memore space or whether it creates and returns a new list.

6.1 Bubble Sort

Consider a list of length n. How many adjacent pairs of elements does it have? We can see that there are $n-1$ pairs of elements. Bubble sort iterates through all $n-1$ pairs of elements, and if they are out of ourder, we switch their positions. We can guarantee this because the largest element will "win" every comparison, and will "bubble" to the right until it hits the end. We can now iterate through the first $n-2$ pairs of elements, skipping the sorted last element. We continue iterating for fewer and fewer elements until we have just a single pair to check, after which we have a sorted list. Note that we iterate a total of $n-1$ times.

```
for i = 0 to n - 1
    for j = 0 to n - i - 2
        if A[j] > A[j + 1]
            swap A[j] and A[j+1]
        end
    end
end
```

Looking at the runtime:

$$T = \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-2} c_1$$

Factoring out the c_1:

$$T = c_1 \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-2} 1 = c_1 \sum_{i=0}^{n-1} (n-i-2+1) = c_1 \sum_{i-0}^{n-1} (n-i-1)$$

We can now split this sum into 3 sums:

$$= -c_1 \sum_{i=0}^{n-1} i + c_1 \sum_{i=0}^{n-1} (n-1)$$

$$= -c_1 \frac{n(n-1)}{2} + c_1 n(n-1) = c_1 n(n-1) - c_1 \frac{n(n-1)}{2}$$

$$= c_1 \left(\frac{n(n-1)}{2} \right) = \Theta(n^2)$$

Consider running Bubble Sort on a list containing all integers between 0 and 99, in reverse order. It takes 99 swaps to swap 99 to its correct place (100 elements). The element 0 requires:

$$\sum_{i=0}^{99} i = \frac{99(100)}{2}$$

We see that this is the worst possible placement for 0, because it requires the maximum amount of swaps. It sorts large elements faster than small elements. This is the main reason Bubble Sort is so slow.

When doing a Bubble sort, we have two cases to watch for. Rabbits are big elements near the start of the list, while small elements at the end of the list are called turtles. The rabbits will be sorted very quickly, while the turtles will be sorted extremely slowly. The more turtles we have, the slower our list will be sorted. There are several ways to minimize this effect. We can alternate the directions of the iterations, known as the Cocktail Shaker Sort. Consider an array of elements from 1 to 99:

$$[1, 3, 4, 5, \ldots, 97, 98, 2, 99]$$

We can see that Bubble Sort would sort this incredibly slowly, while the Cocktail Shaker Sort would do this in two iterations, rather than 99 iterations.

6.2 Selection Sort

The idea of selection sort is to scan the unsorted part of the list for the element that should go in the current index, and then swap it with whatever is there.

Suppose the first k indices of a list are sorted and the rest is not. How do we know which element should be sorted into position $k+1$?

Intuitively, we want to choose the minimum element of the unsorted section of the list, based on the ordering on our set.

```
for i = 0 to n - 2
    minindex = i
    for j = i + 1 to n - 1
        if A[j] < A[minindex]
            minindex = j
        end
    end
    swap A[i] with A[minindex]
end
```

Tracing this on $[5, 2, 7, 4, 1]$:

$$[5, 2, 7, 4, 1]$$
$$[1, 2, 7, 4, 5]$$
$$[1, 2, 7, 4, 5]$$
$$[1, 2, 4, 7, 5]$$
$$[1, 2, 4, 5, 7]$$

How many swaps does Selection Sort make in the worst case, this is $n - 1$, because the last swap makes both of the last two elements fall into place. In the best case, we have the same number of swaps, because we always do a swap, even if the minimum is the same element.

What is the sum for the exact runtime?

$$T = \sum_{i=0}^{n-2} \left[c_1 + \sum_{j=i+1}^{n-1} c_2 \right]$$

$$= \sum_{i=0}^{n-2} [c_1 + c_2(n - 1 - i - 1 + 1)] = \sum_{i=0}^{n-2} [c_1 + c_2(n - i - 1)]$$

$$\sum_{i=0}^{n-2} (c_1 + c_2(n-1)) - c_2 \sum_{i=0}^{n-2} i = (c_1 + c_2 n - c_2)(n-1) - c_2 \frac{(n-2)(n-1)}{2}$$

$$= c_1 n + c_2 n^2 - c_2 n - c_1 - c_2 n + c_2 - \frac{c_2}{2}(n^2 - n - 2n + 2)$$

$$= c_1 n + c_2 n^2 - 2c_2 n - c_1 + c_2 - \frac{c_2}{2}(n^2 - 3n + 2) = \Theta(n^2)$$

Where we have used Θ because all that changes from best to worst case is the value of c_2.

Suppose we are running Selection Sort on a list, but we terminate the algorithm just avfter it completes iteration $\frac{n}{2}$. We can guarantee that the first half will be the $\frac{n}{2}$ smallest elements, and will be sorted. The second half will contain the $\frac{n}{2}$ largest elements, but not sorted. After iteration k, the first k elements are sorted and placed properly.

Suppose we change Selection Sort so that we use a conditional to see whether we need to swap. The worst-case input would then be something that requires all of the swaps. The worst case possible number of swaps are $n - 1$. The worst case input is when no swap places the larger element into its proper position. The best case input would be a sorted list.

6.3 Insertion Sort

The idea is to skim the list from left to right, look at each element and move it as far left as it needs to go, moving any elements it passed one space to the right to realign the list. This is similar to Selection Sort, except instead of swapping, we move the element to the correct spot, and then shove everything to the right by one space.

Suppose that the first k indices of the list are sorted and the rest is not. How do we know where the element in position $k + 1$ should be placed?

We can descend through the list, and compare with each element, until we see a smaller one, or reach the end of the list.

```
for i = 0 to n-1
    key = A[i]
    j = i - 1
    while j >= 0 and key < A[i]
        A[j+1] = A[j]
        j = j-1
    end
    A[j+1] = key
end
```

Suppose we have the input array $[5, 2, 7, 4, 1]$. We can trace out Insertion Sort:

$$[5, 2, 7, 4, 1]$$

$$[2, 5, 7, 4, 1]$$

$$[2, 5, 7, 4, 1]$$

$$[2, 4, 5, 7, 1]$$

$$[1, 2, 4, 5, 7]$$

If we track the number of right shifts for each iteration:

1. 0
2. 1
3. 1
4. 3
5. 7

Even when we have an iteration that does not change the list, we have a change in info, we can extend the sorted part of the list.

Note that we have a while loop to deal with, so we have an input dependent performance time. We have to independently analyze the best, worst, and average cases.

The best case input for Insertion Sort is that it is already sorted, we have 0 shifts at all. The worst case is if the list is reverse sorted, because we have to do the maximum number of shifts.

For the best case, we can do the runtime analysis:

$$T_b = \sum_{0}^{n-1} c_1 + c_2$$

Where c_1 is the two assignments at the start of each loop and the assignment at the end, and c_2 is the while loop, since it never runs and the conditional just takes some constant time:

$$T_b = (c_1 + c_2)n = \Theta(n)$$

Note that this function is $\Theta(n)$. Insertion Sort overall is $\Omega(n)$, but the function for the best case runtime is $\Theta(n)$.

For the worst case, we have a reverse sorted list. We have the same linear time for loop, and the same constant time assignments. For the while loop, it passes the conditional and executes the code inside i times, and then fail the conditional once. Writing this out formally:

$$\sum_{i=0}^{n-1}(c_1 + i(c_2 + c_3) + c_2)$$

Where c_2 is the time for the conditional, and c_3 is the time for the conditional body. Solving this:

$$= (c_1 + c_2)\sum_{i=0}^{n-1} 1 + (c_2 + c_3)\sum_{i=0}^{n-1} i$$

$$= n(c_1 + c_2) + (c_2 + c_3)\frac{n(n-1)}{2} = \Theta(n^2)$$

We see that the worst-case is $\mathcal{O}(n^2)$.

For the average case, we can define the concept of an inversion, a metric for how "scrambled" a list is.

An inversion in a list is a pair of distrinct elements in a list where the smaller element is located at a later index than the larger element. Formally it is when $i > j$ and $A[j] > A[i]$. If we have the list

$$[5, 2, 7, 4, 1]$$

We have 7 inversions. Note that this is the same number of shifts we did in the insertion sort trace.

If we have a list of length n, and it is completely sorted, we have 0 inversions. If it is reverse order (maximally unsorted), we have the Gaussian sum, $\frac{n(n-1)}{2}$ inversions. We see that the number of inversions is the number of times that the while loop runs in the best and worst cases respectively.

The fundamental principles of insertion sort is that the number of shifts needed is equal to the number of inversions in the list. Every shift in the list corrects exactly 1 inversion.

For the average case, we want to compute the runtime in terms of the input length n and the number of inversions I:

$$\sum_{i=0}^{n-1} c_1 + c_2 + I(c_2 + c_3) = n(c_1 + c_2) + I(c_2 + c_3)$$

For an arbitrary list, what is the expected number of inversions?

We have $\binom{n}{2}$ different pairs of elements, and a 50% chance that each corresponds to an inversion. Thus we have $\frac{\binom{n}{2}}{2}$ inversions, which is $\frac{n(n-1)}{4}$. This is $\Theta(n^2)$. This doesn't tell us much about Insertion Sort overall, we already have the best and worst case analysis, but this tells us about the real world usage for most lists.

Suppose we are running insertion sort on a list, but we terminate the algorithm after iteration $\frac{n}{2}$. We can guarantee that the first half will be sorted order with respect to the first half of the list. The second half has no guarantees.

Is Insertion Sort stable? The answer is yes, because we only right shift if the element is strictly smaller. Suppose we have the list

$$[2, 1, 2, 1, 2]$$

Adding subscripts for differentiation purposes:

$$[2_a, 1_a, 2_b, 1_b, 2_c]$$

Now running Insertion sort:

$$[1_a, 2_a, 2_b, 1_b, 2_c]$$

$$[1_a, 2_a, 2_b, 1_b, 2_c]$$

$$[1_a, 1_b, 2_a, 2_b, 2_c]$$

We see that the sort is stable (I didn't do the later iterations because they don't do anything).

In Insertion sort, every shift corrects one inversion. Does every swap in Bubble Sort do the same thing? Bubble Sort cannot create new inversions, and each swap fixes one inversion (there is no gap in between the elements that we are moving, so we fix that singular inversion). What about Selection Sort? When we swap, any inversions that we create were already there, and we could have fixed more than 1 inversion. The following array has 7 inversions. When we run the first iteration of Selection sort:

$$[7, 2, 3, 4, 1]$$

$$[1, 2, 3, 4, 7]$$

We see that this has 0 inversions. We have eliminated all 7 inversions with a single swap.

7 Binary Search

We first must define searching as a computational problem. Consider a list L that contains elements of a set A, and a target element $x \in A$. The problem of search L for x asks us to return an index i such that $L[i] = x$, or fail if no such index exists. If x occurs multiple times, we can return any of them.

Consider a list of length n. What is the time complexity of searching the list? Intuitively, this is $\mathcal{O}(n)$.

Binary search is based on the useful knowledge that the list is sorted in advance. We can then use this information. If we see some element y such that $x < y$, then we know that x must occur before the index of y. We can then eliminate an entire section of the list.

The value of k that maximizes the amount of the list that we can get rid of will be $\frac{n}{2}$.

```
function binarysearch(A, TARGET)
    L = 0
    R = n-1
    while L <= R
        C = floor((L + R)/2)
        if A[C] == TARGET
            return C
        else if TARGET < A[C]
            R = C - 1
        else
            L = C + 1
        end
    end
```

```
    return FAIL
end
```

We can trace this on the array $[1, 2, 4, 6, 8, 9, 12, 15]$, searching for 12:

$$[1, 2, 4, 6, 8, 9, 12, 15]$$

Comparing 6 against 12, we see that 12 is larger, so we can get rid of everything before the 8. Our search space is now just indices 4 and above.

$$[8, 9, 12, 15]$$

Now comparing 9 against 12, we see that 12 is larger, so we discard 8 and 9:

$$[12, 15]$$

comparing 12 against 12, we see that this is the target, so we return the index, which is 6.

The best case is that we have that the first item that we check, right in the middle, is the correct target. The worst case is that the element is not in the array.

For the best case runtime, we have 2 constant time assignments c_1, and then a constant time while loop body, c_2, and then a constant time return statement c_3. We claim that the loop body is c_2 even if we terminate the program early by returning in the first check. In the best case, we have total runtime $c_1 + c_2 = \Theta(1)$. This tells us that the overall algorithm is $\Omega(1)$.

For the best case, we have the same constant time c_1 assignments, and the same loop body c_2, and the same fail return time c_3. Every iteration, we divide our search space in half. We do this over and over again until we finally realize that the target is not in the array. The original search space has size n, first iteration has $n/2$, and so on. The kth iteration will have search space $\frac{n}{2^k}$ elements. We want this to be 1:

$$\frac{n}{2^k} = 1 \to n = 2^k \to k = \lg n$$

Thus we see that the loop runs $\lg n$ times. In total:

$$T = c_1 + \lg n c_2 + c_3 = \Theta(\lg n)$$

This tells us that overall, the algorithm is $\mathcal{O}(\lg n)$. Note that due to change of base formulas, we can change this to $\mathcal{O}(\log n)$.

Now for the average case analysis. We define the average case as a target located uniformly randomly somewhere in the list. The goal is to compute the expected value of the runtime.

Suppose $n = 2^N - 1$ for some $N \in \mathbb{Z}$. How many indices require k steps of binary search to be picked as the center?

For the first iteration, we have 1 index, the exact center. For the second step, we have 2 indices, the centers of the first half of the list, and the cetner of the back half of the list. For the third step, we have 4 elements. We see that the pattern is that 2^k indices require k iterations.

Suppose we have an arbitrary element. What is the approximate probability that it requires exactly $\frac{1}{2}\lg n$ steps to be chosen as the center? We have $2^N - 1$ elements, and 2^{k-1} elements that require

k steps to be the center, and the deepest elements require $\lg n$ steps. We can see that the exact probability is

$$\frac{2^{\frac{1}{2}\lg n - 1}}{n}$$

We can see that the numerator can be represented as $\frac{1}{2}n^{\frac{1}{2}}$. Thus the exact probability simplifies down to

$$\frac{1}{2\sqrt{n}}$$

We wanto to do the average case analysis now. The average case is that the target is uniformly randomly somewhere in the list. We assume that the input size is $n = 2^{N-1}$, for some $N \in \mathbb{Z}$. To return an element, we must choose it as the center C on some iteration. For $k \in \{1, 2, \ldots, n-1\}$, the list has 2^k indices that need k steps to be chosen as C. Therefore, the probability that the target needs k steps to be chosen is $\frac{2^{k-1}}{n}$.

We can now use two useful formulae:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

For the average case, we can set up the sum for the runtime:

$$\frac{1}{n}\sum_{i=0}^{N-1} 2^i(c_1 + (i+1)c_2)$$

where the bounds represent all possible steps the algorithm can do. The $\frac{1}{n}$ is the probability of picking any particular value as our target. The 2^i is the number of values at depth $i+1$ in the tree. The inner $i+1$ is the time taken for the while loop.

Now solving this sum:

$$= \frac{1}{n}\left[\left(c_1\sum_{i=0}^{N-1} 2^i\right) + \left(c_2\sum_{i=0}^{N-1}(i+1)2^i\right)\right]$$

Now rewriting:

$$= \frac{1}{n}\left[\left(c_1\sum_{i=0}^{N-1} 2^i\right) + \left(c_2\sum_{j=1}^{N} j2^{j-1}\right)\right]$$

Where we have made the substitution $j = i + 1$. Now pulling out a $\frac{1}{2}$ from the exponential, we can pull this out of the sum and make it match our formula:

$$= \frac{1}{n}\left[c_1(2^N - 1) + c_2\left(\frac{1}{2}(N-1)2^{N+1} + 2\right)\right]$$

Cleaning this up, using the fact that $n = 2^N - 1$:

$$= \frac{1}{n}\left[c_1 n + \frac{1}{2}c_2(\lg(n+1)2(n+1) - 2(n+1) + 2)\right]$$

And doing some arithmetic:

$$\frac{1}{n}[c_1 n + c_2 \lg(n+1)n + c_2 \lg(n+1) - c_2 n]$$

$$= c_1 + c_2 \lg(n+1) + \frac{c_2 \lg(n+1)}{n} - c_2$$

We see that the highest order term is the $\lg(n+1)$, telling us that the average case is $\Theta(\lg n)$.

We have that the best case is $\Theta(1)$, the worst case is $\Theta(\lg n)$, and the average case is $\Theta(\lg n)$.

Suppose we have a list of real numbers you want to search. Which has better time complexity, sorting the list and then binary searching or linear searching? We know that the linear search is $\mathcal{O}(n)$. The best sorting we have is $\mathcal{O}(n \lg n)$, on top of the binary searches $\mathcal{O}(\lg n)$, which gives us $\mathcal{O}(n \lg n)$. This tells us that the linear search is better than sorting and then running binary search.

8 Recurrence Trees

Oftentimes, our runtime is computed recursively as a function of smaller inputs. We define a recurrence relation as an equation expressing each term of a sequence as a function of both its own index and preceding terms' values.

We have two components. The first is an initial condition, or base case. We then have a recursive rule, which is how we generate future terms from past ones.

The prototypical recurrence relation is the Fibonnaci numbers, and another common one is the factorial function.

Recursion is in general not very nice. Sometimes we get a nice closed form expression, like for the Fibonacci sequence, but other times we don't because of weird growth rates.

Sometimes we have really chaotic functions:

$$x_0 = \frac{1}{5}, x_{n+1} = 2x_n - 4x_n^2$$

Consider the following recurrence relation:

$$T(1) = 2$$

$$T(n) = 3T\left(\lfloor \frac{n}{4} \rfloor\right) + 3n + 1$$

We can compute $T(4)$:

$$T(4) = 6 + 12 + 1 = 19$$

And then try to compute $T(16)$:

$$T(16) = 106$$

What if we try to compute $T(2)$? We can see that it relies on $T(0)$, which is undefined, so we can't get a value for $T(2)$. This is not a complete definition. However, from an analysis standpoint, we don't really care about this.

We will also often omit the floor in the argument, and just assume that there is a floor function, to deal with fractional arguments.

We can view recursion as having a tree structure. To get the total time, we just add up every node at every layer of the tree that we draw.

Suppose we assume that $n = 4^k$ for some $k \in \mathbb{Z}$. We begin at the root node, $T(n)$. The time done at this level of the tree is $3n + 1$. This generates 3 child nodes, each with input size $\frac{n}{4}$. Each child node will have $n = 4^{k-1}$, and each child will once again have 3 children itself. We stop recursing when something calls $T(1)$, because that is a constant time computation.

How many iterations will we have until we hit $T(1)$? We will have $\log_4(n)$ iterations, and since we've defined that $n = 4^k$, the height will be k.

The bottom leaves will just be 2, and we can look at the tree layer by layer, and sum up the work done at each layer. At layer 0, we have $3n + 1$ done per node, and 1 node, giving us $3n + 1$ work. Layer 1 has 3 nodes, each with work $\frac{3n}{4} + 1$, giving us $\frac{9}{4}n + 3$ work total for the layer. We can do this all the way down to the leave layer, which has 3^k nodes, and 2 work per node, which will be 2×3^k. We can set up a sum for the layers up to the leaves, and add the leaves:

$$2(3^k) + \sum_{i=0}^{k-1} 3^i \left(\frac{3n}{4^i} + 1 \right)$$

Simplifying these, we have

$$T(n) = 2(3^k) + 3n \sum_{i=0}^{k-1} \left(\frac{3}{4} \right)^i + \sum_{i=0}^{k-1} 3^i$$

We can now use the geometric series sum:

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}}{r-1}$$

For $r \neq 1$.

We can now just use the formula for both sums:

$$T(n) = 2(3^k) + 3n \frac{\left(\frac{3}{4}\right)^k - 1}{\frac{3}{4} - 1} + \frac{3^k - 1}{3 - 1}$$

Simplifying this:

$$2(3^k) + 12n \left(1 - \left(\frac{3}{4} \right)^k \right) + \frac{1}{2}(3^k - 1)$$

Now using the fact that $k = \log_4 n$, we can substitute this in, and simplify. We find that we will end up with

$$12n - \frac{19}{2}(3^{\log_4 n}) - \frac{1}{2}$$

We see that this is actually $\mathcal{O}(n)$. We can actually also show that this is $\Theta(n)$.

8.1 The Master Theorem

Consider an arbitrary recurrence relation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where a is the number of nodes, and $f(n)$ is the work done per node. From this, we can form some intuition. Let $f(n) = 0$, and consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right)$$

What we care about is the number of subproblems over the size of the subproblem, which is $\frac{a}{b}$. If they are the same, we have created $a$new problems of size $\frac{n}{a}$. If $T(1) = t$, and $T(n) = aT\left(\frac{n}{a}\right)$. Can we find a general formula for $T(a^k)$?

In the case where $n = a$, the total work done in the tree will be just the leaves, at. If instead, we have $n = a^2$, we will have a^2t time (a nodes, each of which resolves to at work). Thus we have a generic formula that $T(a^k)$ will take a^kt. This is $\Theta(n)$, since our input size is a^k.

What if $a < b$? When we split, we have a new problems of size $\frac{n}{b}$, which is less than $\frac{n}{a}$. We have created a problems that are easier to solve than linear. Using the same reasoning, if $a > b$, each new subproblem is worse than linear.

In general, we can conclude that

$$T(n) = aT\left(\frac{n}{b}\right) \rightarrow T(n) = \Theta(n^{\log_b a})$$

However, this does not account for any $f(n)$ work done at each layer. We define a critical exponent,

$$c_{crit} = \frac{\log a}{\log b} = \log_b a$$

We care about how c_{crit} compares to $f(n)$.

Before we do this, we care about a property known as regularity. When $f(n)$ is meaningfully large, we want to make sure that the combined work done by splitting is less than the work done at the top later.

Consider a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Then $f(n)$ satisfies the regularity condition if

$$(\exists c < 1, n_0 > 0)(\forall n \geq n_0)\left[af\left(\frac{n}{b}\right) \leq cf(n)\right]$$

Intuitively, this says that the total work done at layer $k + 1$ must eventually always be strictly smaller than the work done at layer k. Splitting must be a good idea. This is a property that is almost always true.

Theorem 8.1. *Suppose $T(n)$ satisfies the recurrence relation*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

for $a \geq 1$, $b > 1$, and where $\frac{n}{b}$ is either the floor or ceiling of $\frac{n}{b}$.

Then if $f(n) = \mathcal{O}(n^c)$, where $c < c_{crit}$, then $T(n) = \Theta(n^{c_{crit}})$. This is the leaf-heavy case.

If $f(n) = \Theta(n^{c_{crit}})$, then $T(n) = \Theta(n^{c_{crit}} \log n)$. This is the balanced case.

If $f(n) = \Omega(n^c)$, where $c > c_{crit}$, and f satisfies regularity for T, then $T = \Theta(f(n))$. This is the root-heavy case.

Recall the recurrence $T(1) = 2$, $T(n) = 3T\left(\lfloor \frac{n}{4} \rfloor\right) + 3n + 1$. Which Master Theorem case applies here?

We have that $f(n) = 3n + 1$, which is $\Theta(n)$. We have that $c_{crit} = \log_b a = \log_4 3 < \log_4 4 = 1$. We see that $n^1 > n^{\log_4 3}$. The time complexity of the function dominates the time complexity of the splitting. This is the root-heavy case. We must now check whether f satisfies regularity for our recurrence.

We need to find constants c and n_0 that satisfy the definition of regularity. We want to find

$$3f\left(\frac{n}{4}\right) \leq cf(n)$$

$$\frac{9}{4}n + 3 \leq 3cn + c$$

We can now pick some c that will make this true, such as $c = \frac{7}{8}$. Now we can find some n_0 that makes this true:

$$\frac{9}{4}n + 3 \leq \frac{21}{8}n + \frac{7}{8}$$

Solving this for n, we can pick $n_0 = \frac{17}{3}$. We have now satisfied regularity. By the Master Theorem, we know that the recurrence relation is $\Theta(f(n)) = \Theta(3n + 1) = \Theta(n)$.

Suppose we have the recurrence relation

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

We see that $c_{crit} = 3$, and $f(n) = \Theta(n^2)$. We are in the leaf-heavy case, and by the Master Theorem, $T(n) = \Theta(n^3)$.

Are there any recurrence relationsh where none of the Master Theorem cases apply?

One case is multiple different recursive calls, since we have different types of splitting. Another case is where our recurrence relation is root-heavy but we have a regularity violation. Another case is where we have a nonconstant a. Another is when we have increased recursive work, $T(n) = T(2n) + f(n)$. Finally, what if we have a negative work function.

We can generalize the balanced case of the Master Theorem, as it is a special case of the following.

For $k \geq 0$, if $f(n) = \Theta(n^{C_{crit}} \log^k n)$, then $T(n) = \Theta(n^{c_{crit}} \log^{k+1} n)$.

8.2 Merge Sort

The idea of Merge Sort is to sort each half of the list, then merge them together like a zipper. This lends itself well to recursion. To find the base case, we need to know when we can stop recursing. The answer is when we are passed in a 1 element list, because, this is trivially sorted.

Let us first look at the merge subroutine. This takes in two sorted lists, and iterates through and compares the elements against each other, and picks the smallest available element, and places this into the merged list. Once one list runs out, we just insert the rest of the remaining list. Note that we only need to do comparisons while both lists still have elements in them.

Suppose L and R have a total of n elements. We have a total of n iterations across all of the while loops, and each while loop does constant work. The time complexity of the subroutine will be linear, $\Theta(n)$.

Now we need to find the time complexity of the overall sorting algorithm. We have the recurrence relation:

$$T(1) = c_1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

We have a critical exponent of 1, and the work function is $\Theta(n)$. We see that $n^1 = n^1$, so we are in the balanced case. This tells us that our relation is $\Theta(n \log n)$.

8.3 Heap Sort

First we have to talk about complete binary trees. A binary tree is complete if all levels before the last are completely filled, and all nodes on the last level are as far left as possible. We indcex nodes left to right, row by row of the tree. The children of a node i are indexed at $2i$ and $2i + 1$, for the left and right children respectively. Given a node, the parent is given by $\lfloor \frac{i}{2} \rfloor$. We define the root to be layer 1. The leftmost node of layer k is given by 2^{k-1}, and we can find what layer a node is on, using $1 + \lfloor \lg i \rfloor$. Using this, we could find the number of layers on the tree using the index of the last index, $1 + \lfloor \lg n \rfloor$. We also note that only the nodes up to $\lfloor \frac{n}{2} \rfloor$ have children.

Let us now talk about max heaps. A complete binary tree stasfies the heap property if each node's value is gretaer than or equal to the value of its children. A max heap is a complete binary tree that satisfies the heap property.

We can place an element properly in the heap by heapifying it. If w ehave a value that violates the heap property, we swap it with its largest child, then repeat until it no longer violates the heap property.

Note that since we swapped it with the max child, the new parent node will maintain the heap property with the other child.

We can convert a binary tree to a max heap by heapifying all the nodes from the bottom up. We claim that we only need to heapify nodes $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$, because the rest of the nodes do not have children, and thus heapifying them will do nothing.

Using these two strategies, we can convert a binary tree into a max heap. To do heap sort, we exchance the root of our max heap with the last index, cut the last index off the heap, and then maxheapify the root. Repeat until the heap is empty. Note that we isolate the last index because

otherwise the maxheapification would put it back to the root. We are essentially taking out the max elements left in the tree.

Note that this is unstable, which can be found via looking at a tree with all the same elements.