Section 4: Asymptotics and Array Lists

Raymond Guo

Disclaimer: I'm a little rusty in 61B and actually relearning with y'all :). These are just my personal notes, so there may be errors in this document; if so, let me know! Also, this isn't affilianted with CSM or any Berkeley organization :).

Asymptotics (Q1 & Q2)

In computer science, we often deal with data on the extremely large scale; whenever this is the case, we usually have two scarce resources: time and space. Thus, to determine how "good" an algorithm, we usually try to estimate the amount of time and space an algorithm will take, on the conceptual large scale. This is what we call *analysis of algorithms*. We call the estimated amount of time an algorithm takes to run the **runtime complexity**, and the estimated amount of space the **space complexity**. For this week, we'll focus on time complexity.

There are typically 3 ways we measure the runtime complexity (this also expands to space complexity as well):

- Big Omega: Describes the "slowest" a program can run (lower bound).
- Big O: Describes the "fastest" a program can run (upper bound).
- Big Theta: Describes how a program runs in general (tight bound).

Some General Rules

When we are considering runtime, we get rid of the following:

- Coefficents
- Lower Degree Terms
- Bases of Logarithms

This means that `O(5n^2 + 3n + 2)` actually is the same runtime `O(n^2)`. Also, the order of the most common runtimes, from fastest to slowest, is:

> O(1) [constant] < O(log(n)) [logarithmic] < O(n) [linear] < O(n^k) [polynomial] < O(k^n) [exponential]

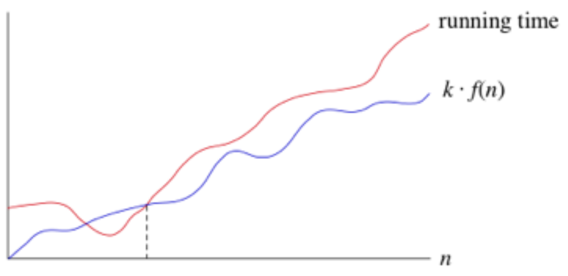
Big Omega (Ω)

We say that Big Omega provides the lower bound of an algorithm's runtime. For example, let's say an algorithm has a runtime of `Ω(n)`. Notice that there aren't any units; that's because we're just comparing the rate that the runtime of the algorithm grows as `n` increases; more specifically for this example, we're saying that as `n` increases, the runtime of the algorithm grows at the same rate or slower than the rate that `k · n` is increasing.

More generally, the mathematical definition of Big Omega is:

> For an algorithm with runtime R(N): R(N) ∈ Ω(f(N)) → ∃ k, k · f(N) ≤ R(N)

Visually, this means that the the runtime of the algorithm is always "above" f(N).



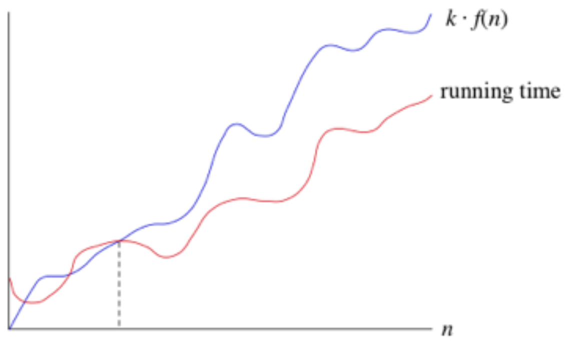
Big O

Big O provides the upper bound of an algorithm's runtime. Let's say an algorithm has a runtime of `O(n)`. This means that as `n` increases, the runtime of the algorithm grows at the same rate or faster than the rate that `k · n` is increasing. We typically care about the Big O of an algorithm much more than the Big Omega, since it provides us with the "worst case scenario" of how much time an algorithm will take to run.

The mathematical definition of Big O is:

> For an algorithm with runtime R(N): R(N) ∊ O(f(N)) → ∃ k, k · f(N) ≥ R(N)

Visually, this means that the the runtime of the algorithm is always "below" f(N).



Big Theta (Θ)

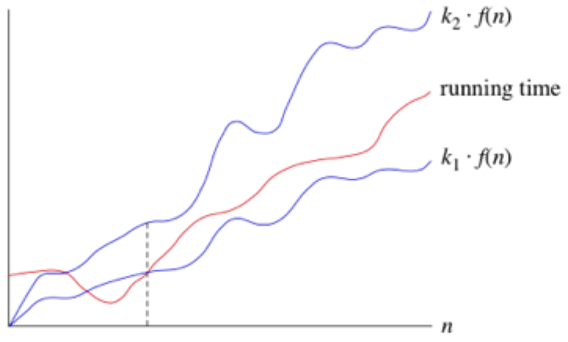
Big Theta provides the tight bound of an algorithm's runtime. What this means is that the algorithm's runtime growth can be modelled with a single function. As you may guess, even though every algorithm has a (or infinite) Big O and a Big Omega runtime (since the bounds can be arbitrarily loose), not every algorithm has a Big Theta runtime.

Let's say an algorithm has a runtime of `θ(n)`. This means that as `n` increases, the runtime of the algorithm grows at the same rate that `k · n` is increasing. If there exists a function `f(n)` where the algorith has both a Big O and Big Omega of `f(n)`, then the algorithm has a Big Theta of `f(n)` as well. Otherwise, if there does not exist a function that can satisfy both the upper bound and lower bound at once, then the algorithm doesn't have a Big Theta runtime.

The mathematical definition of Big Theta is:

> For an algorithm with runtime R(N): R(N) ∊ Θ(f(N)) → ∃ k_1, k_2, k_1 · f(N) ≤ R(N) ≤ k_2 · f(N)

Visually, this means that the the runtime of the algorithm is always "below" f(N).



Calculating Runtime - Iterative Algorithms

In general, calculating the runtime of an iterative algorithm is typically more simple than for recursive algorithms. The general rule of thumb is to trace the execution of the program and count the number of times an operation is being looped over. For example, if you're iterating over list, the runtime is probably `O(n)` . If you're iterating over a 2D matrix (with dimension `m x n`), the runtime is probably `O(mn)` . However, be careful! Just like on the worksheet, sometimes the program terminates early, leading to a much faster runtime than one would expect. So in the end, make sure to not just look at the number of `for` or `while` loops; instead, trace through the execution!

Calculating Runtime - Recursive Algorithms

The key to calculating the runtime of a recursive algorithm is to draw a tree. Each node in the tree represents a new call to the recursive function, with the value of the node being the input size (usually in terms of `n` , the original input size). There are 3 main steps in calculating the runtime:

1. Draw the tree. Get the general shape.
2. Determine the tree's height.
    - If the `parent -> child` relationship is `n -> n-1` , then the height is most likely `n` . One example of this is Fibonacci.
    - If the `parent -> child` relationship is `n -> n/2` , then the height is most likely `log(n)` . One example of this is Binary Search.
    - If the `parent -> child` relationship is `n -> sqrt(n)` , then the height is most likely `log(log(n))` . This is typically more rare than the other two.
    - There are others, but these 3 are the most common ones I personally run into.
3. Determine the amount of work that's being done at each node. This is equal to the amount of work that's being done in each function call, outside of the recursive relations.
4. From step 3, determine the amount of work being done at each level of the tree. Just multiply the amount of work done at each node by the number of nodes in that level.
5. Sum up all the levels from step 4, taking into account the number of levels (from step 2) to determine the runtime. Here are a few useful sums to know:
    - `1 + 2 + 3 + ... + n = O(n^2)`
    - `1 + 4 + 9 + ... + n^2 = O(n^3)`
    - `1 + 2 + 4 + ... + 2^n = O(2^n)`
    - `1 + 2 + 4 + ... + n = O(n)`
    - `1 + 2 + 4 + ... + 2^(log n) = O(n)`

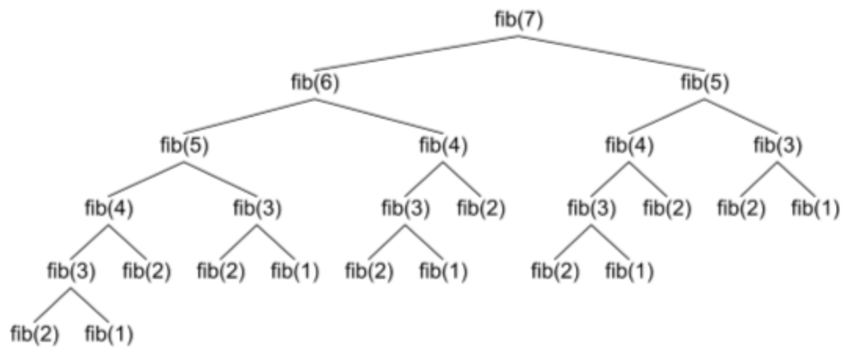
Example: Fibonacci

In section, I went over Fibonacci as an example. Recall the code for Fibonacci:

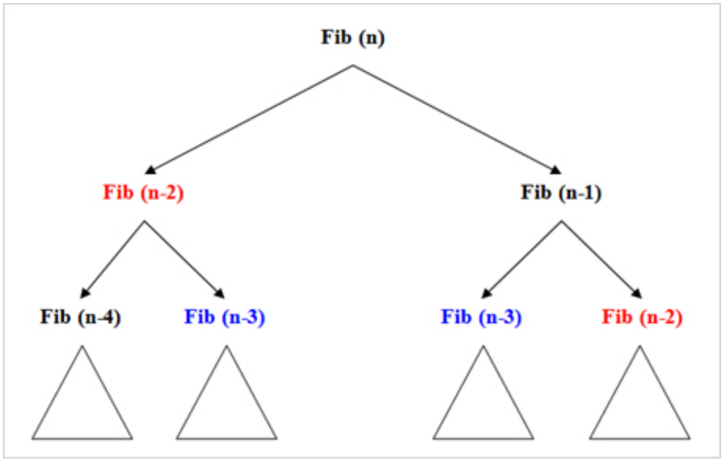
```
static int fib(int n)  {
    if (n <= 1)
```

```
        return n;
    return fib(n-1) + fib(n-2);
}
```

Let's draw out a tree with a sample input `7` :



If we generalize this with an arbitrary input `n` , we get:



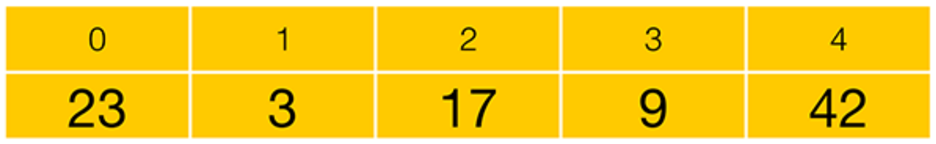
Now, to determine the tree's height, we notice that the `parent -> child` relationship is `n -> n-1` , so the height is `n` .

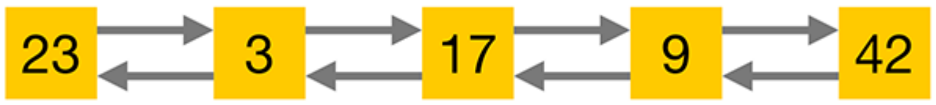
At each node, we're only doing addition, which is a `O(1)` operation. Thus, at level `k` , the runtime of that level is `O(2^k)` , since there are `O(2^k)` nodes and each node is running at `O(1)` time.

Now, we just need to sum all the levels up: `2^0 + 2^1 + 2^2 + ... 2^n = 1 + 2 + 4 + ... + 2^n = O(2^n)` , so the overall runtime of recursive Fibonacci is `O(2^n)` .

Array Lists (Q3)

In this class, we'll mainly be dealing with two types of lists: linked lists and array lists. As we learned before, linked lists are essentially nodes that are connected to each other through a `next` pointer. Array lists, on the other hand, are lists that are in essence arrays "behind-the-scenes" -- behind every array list lies an array.



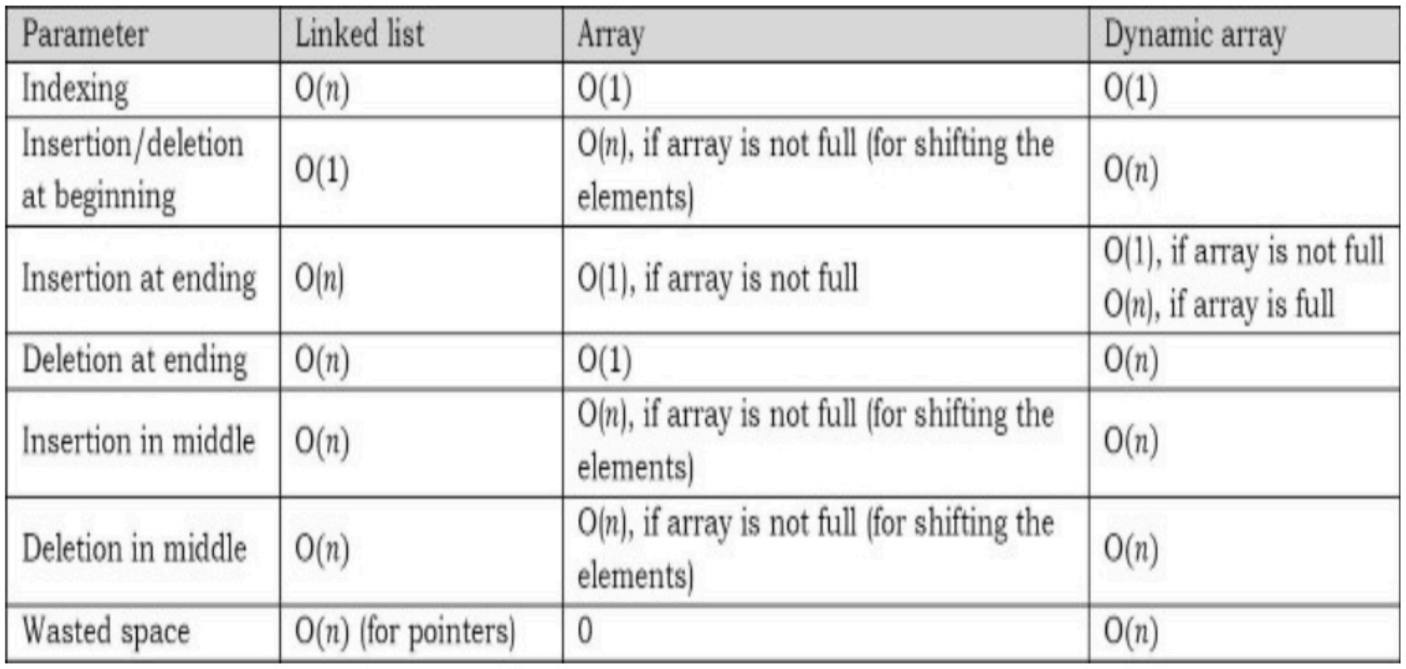


The biggest functionality that an array list has that an array doesn't is the ability to add elements. Arrays aren't able to do this because they aren't resizeable. Thus, in order to overcome this, whenever the underlying array of an array list becomes full and a new element is inserted, the array list goes through a resizing procedure:

1. Make a new array with double the size.
2. Copy over the elements from the old array to the new array.
3. Delete the old array.

The advantage of linked lists is that it makes appending to the beginning / end of the list extreme fast (`O(1)` time). Removing is also a lot faster for linked lists. However, array lists are very fast at retrieving elements at a specified index (`O(1)` time), whereas linked lists have to iterate through every node from the beginning to reach the right index before returning the value (`O(n) time`).

Here's the runtimes for linked list, array, and array list (called dynamic array below):



Parameter	Linked list	Array	Dynamic array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$ (for pointers)	0	$O(n)$