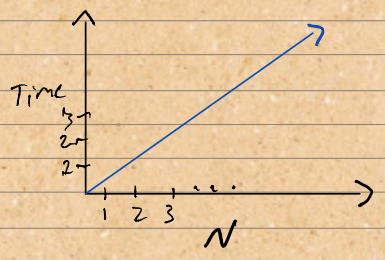
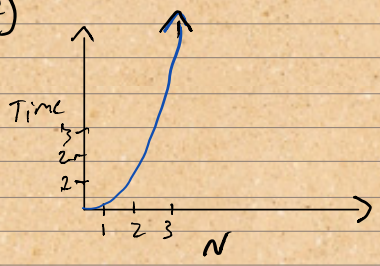
A) What is it?
- The language & metric used to describe the efficiency of algorithms
- **Incredibly Important**

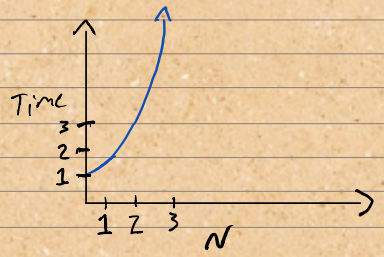
B) Time Complexity (Big-O)
- What the concept of asymptotic runtime means
- Common Runtimes & their graphs
  - $O(N)$ -> time complexity increases with respect to N



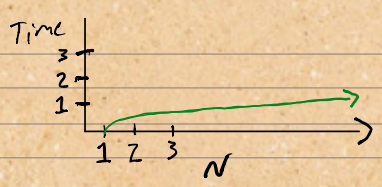
- $O(N^2)$



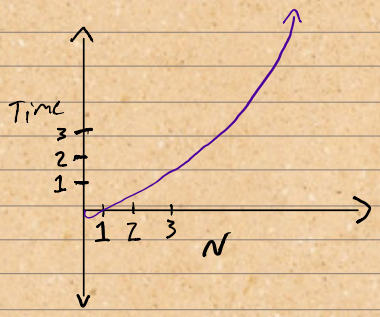
- $O(2^N)$ -> an exponential function



- $O(\log N)$ Which algorithms are $O(\log N)$?



- $O(N \cdot \log N)$



C) Best Case, Worst Case, and Expected Case

- Using Quick sort as an example
  - Quick sort picks a random element as a 'pivot' & then swaps values in the array s.t. the elements less than the pivot appear before elements greater than pivot. It then recursively sorts the left & right sides using a similar process

- Best Case ⇒ not very useful concept
  - If all elements are equal then big-O will be $O(N)$ where N is the size of the array

- Worst Case
  - If the pivot is repeatedly the biggest element in the array then the runtime degenerates to $O(N^2)$

- Expected Case
  - $O(N \cdot \log N)$

Note:
Expected & worst case are usually the same

- Big-O describes the upper bound on the time

- Big-Omega describes the lower

1 N

bound on the time

- Big-Theta is both big-O & big-Omega (tight bounds)

D) Space Complexity
- Amount of memory (space) required by an algorithm
- Parallel concept to time complexity. That is, if we create an array of size n this will require O(n) space. If we create a 2-dimensional array of size nxn will require $O(n^2)$ space

Note: Review Recursion at some point

E) Drop the Constants & Non-Dominant Terms
- O(N) code could run faster than O(2) code for specific inputs
  - This is why we drop the constants in runtime. That is O(2·N) becomes O(N)
- O(N) is not always better than $O(N^2)$ - How?
- $O(N^2 + N) = O(N^2)$
  (N: non-dominant)

F) Multi-Part Algorithms: Add vs. Multiply
- Suppose you have an algorithm that has 2 steps. When do you multiply the runtimes and when do you add them?

i) Add
- When algo is in form "do this, and when you're done, do that"
- Ex: O(A+B)

```
For (int a : arrA) {
    print(a);
}
For (int b : arrB) {
    print(b);
}
```

#we're doing A chunks of work & then B chunks of work so we add them

ii) Multiply
- When algo is in form "do this, for each time you do that

- Ex: O(A*B)

```
for (int a : arrA) {
    for (int b : arrB) {
        print(a + "," + b);
    }
}
```

* We're doing B chunks of work for each element in A so we multiply them

b) Amortized Time pg. 43

- Is used to describe that the worst case happens every once in a while. But once it does, it won't happen again for so long that it doesn't matter much

H) Log N Runtimes

- log n runtimes happen a lot when the number of elements in the problem space gets halfed each iteration.

- Ex:
  Binary search => n log(n) right?

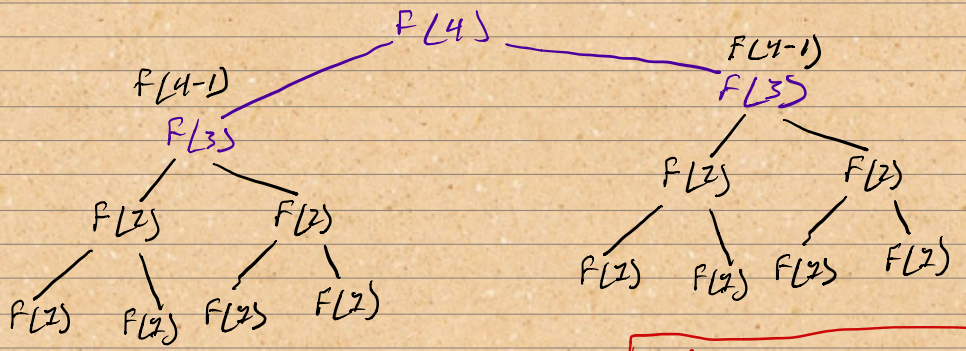
- Bases of logs don't matter much because logs of different bases only differ by a constant and we drop constants in Big-O

I) Recursive Runtimes pg. 45

- Ex: What's the runtime?

```
int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n-1) + f(n-1);
}
```

Suppose $n = 4$



* Runtime $= 2^N$ where 2 is the number of branches for each recursive call & N is the depth

Note:
Look up recursion again. I have forgotten it.

- Recursive functions that make multiple calls will often, but not always, have a runtime of $O(branches^{depth})$

- Space complexity for above algorithms O(N) since only O(N) nodes exist at any given time

Question:
What does it mean to have a log(N) runtime? How do we know when an algorithm is that?

Answer:
It's probably a log(N) runtime when the number of elements in the problem space get halved each time (binary search)

Note:
Review log identities & permutations

$2^p = Q \Rightarrow \log_2(Q) = p$

Logarithms

$2^7 = 128$ $\quad 2^N = 4096 \Rightarrow \log_2 4096 = N$