Chapter 3: Algorithms

Math 275 | Summer 2021

Algorithms

- An algorithm are precise step by step instructions for completing a task.
- Pseudocode is useful for understanding algorithms.
  - Syntax is self-explanatory
- Two big types of algorithms are searching and sorting algorithm

Linear Search

- *Very very* simple and easy searching algorithm
- Very self explanatory
- Given array of numbers and number to search for, iterate through until you find them

```
int linSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i;
        }
    }

    return -1;
}
```

Binary Search

- Requires Sorted array
- Splits array into two and compares number to be searched with max of smaller array
- Binary Searches are best suited for recursive functions by their nature.

Bubble Sort

- Simple sorting algorithm
- Not that efficient
- Works by iterating through array, and comparing element with the next element, and then switching them if they are not in order.
- In practice, this consists of two nested for loops. The outer for loop iterates through the *length of the array* (you don't actually use its index to swap) to guarantee all elements are swapped, and the inner loop loops from the beginning of the array to the last sorted value and does the actual sorting.
- The inner loop does the actual swapping, you have the outer loop there to make sure the inner loop swaps everything the necessary amount of times. If you didn't have the outer loop, and you looped only once, you'd

only guarantee the max element be sorted.

- With each pass of outer for loop, an element is guaranteed to be sorted.
  - Let's say, we are trying to sort the following array: `[2, 3, 4, 1, 5]`.
  - The first outer for loop pass guarantees that the largest element, `5`, is in the correct position.
  - The second outer for loop pass guarantees that the two largest elements, `4`, `5`, are in their correct positions.
  - The third outer for loop guarantees that the three largest elements, `3`, `4`, and `5`, are in their correct positions.
  - > The fourth pass consists of one comparison, namely, the comparison of `1` and `2`. Because `1` < `2`, these elements are in the correct order. This completes the bubble sort.

- This is why the outer loop exists: to guarantee more and more numbers being sorted. Thus you iterate over the length of the array, so you can make sure that length is sorted.
- Because with each pass, we have more and more numbers that are sorted, the inner loop doesn't need to iterate over those sorted values. Thus it just needs to iterate through the beginning to the last unsorted value.
- You can think of each pass of the inner loop, as going over a new smaller array (new and smaller because it's not going over the last element), and pushing that max value to the end. Once it pushes that max value end, it goes over the array again (but without what it just pushed to the end, so it's like it's iterating over a newer smaller array), and then pushes the new max to the end. It then does this again and again until the array's size becomes 0.

Example implementation assuming, `swap()` function is defined and implemented:

```
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) { // it's n - 1, because index starts at 0, so its size - 1
        // Last i elements are already in place, so need to iterate over them
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

Insertion Sort

- Simple sorting algorithm, but not efficient.
- Starts at second element, and compares it with first element.
- If second is less than first, inserts it behind first, and first is now second, and second is now first.

From GeekSort: To sort an array of size n in ascending order:

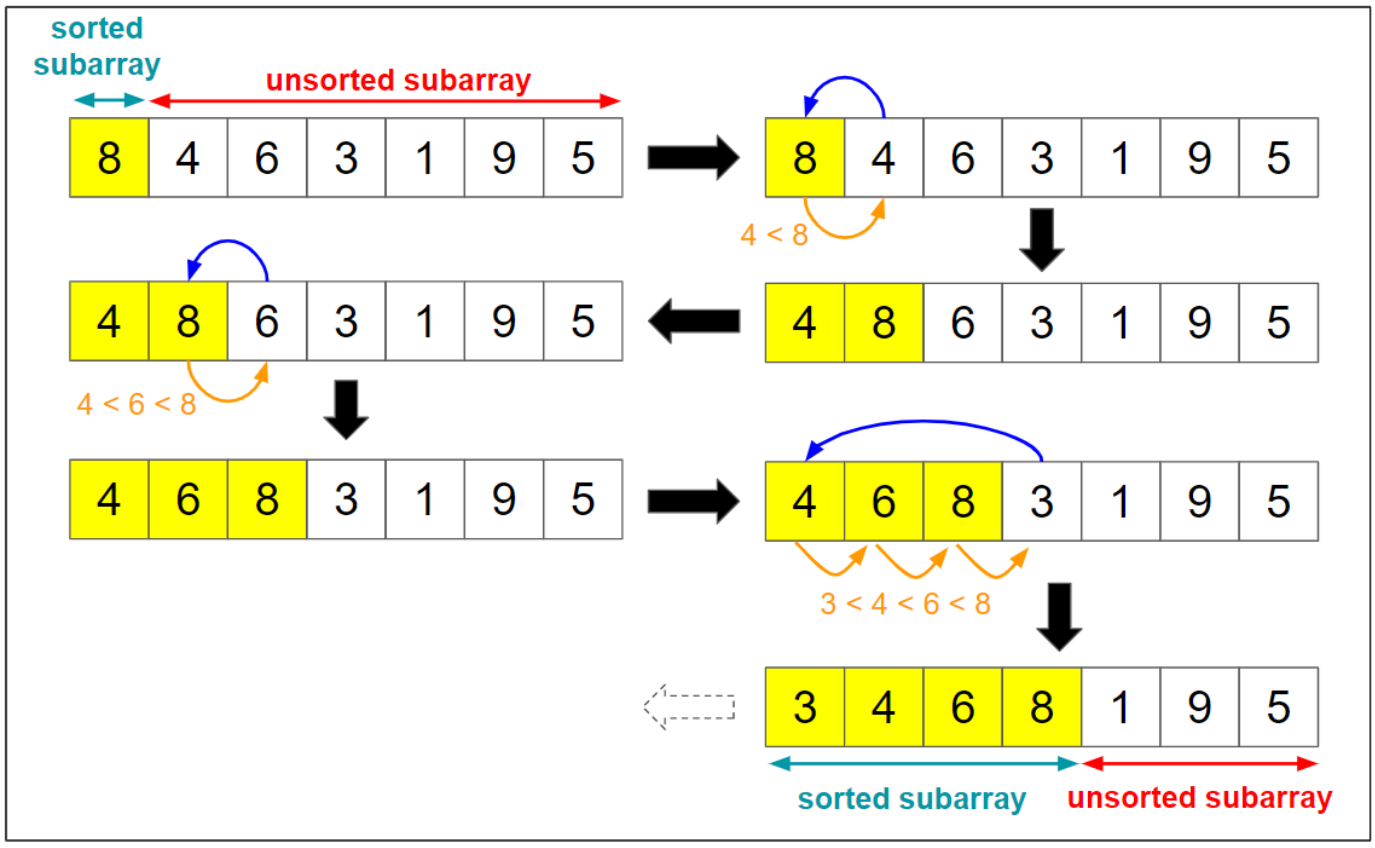
1. Iterate from arr[1] (second element) to arr[n] (last element) over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

- The "compare to the elements before" in the third step is done by a while loop. So this algorithm uses a for loop and a while loop. The for loop iterates *only once*, and stops when it hits that `n + 1` is > `n`. Then the while loop kicks in.

- The while loop then basically swaps the misplaced element with the one behind it, and does that swap again and again, until the misplaced element is in the correct place.
    - When it does that swap, all the elements bigger than it, move up one place, which sorts them in the correct place.

```
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```



Greedy Algorithms

- A greedy algorithm makes the optimal choice at each step.
- However, this does not *necessarily* mean it makes the optimal result.
- We must check if it creates optimal result through proofs.

Halting Problem

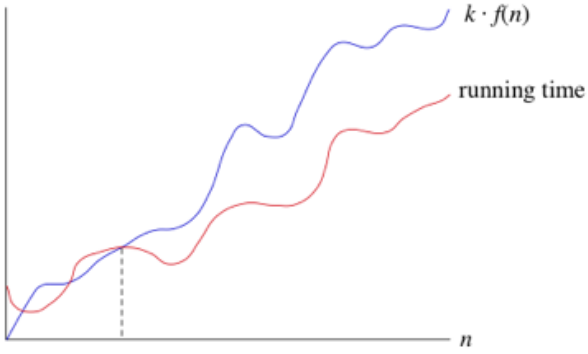
- The halting problem goes like this: Is there an algorithm such that if we give it a computer program and the input to that program, can we determine whether this program will go into an infinite loop or will it stop?
  - Note that we can't actually run the program to see if it runs.
- The answer is no, we can't. Read book for detailed proof.

Growth of Functions

Big-O

- We use Big-O notation to track the growth of functions without having to worry about constant multipliers or smaller order terms.
- Formally, we say that a function $f(x)$ is $O(g(x))$, when: $|f(x)| \leq C|g(x)|$, whenever $x > k$,
- Basically this means a function is big-O of another function g(x), *if* g(x) is greater or equal than it, while multiplied by some constant C, after some point k.
- All this means, is that the O(x) functions acts as an upper bound for f(x). f(x) must be smaller than O(g(x)) for it to be true.

We can visualize Big-O like this:



The blue represents g(x) and the red represents f(x). After some point (marked by the dashed line), g(x) is ALWAYS bigger than f(x). Thus we can say that f(x) is O(g(x)).

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Already, we can guess that f(x) is $O(x^2)$, as they're both the same degree. Using our formal definition of Big-O, we can also further intuitively know that it is $O(x^2)$, because if we were to multiply x^2 by 100, or any other really large number, it would act as C, and we can obviously know it would act as a bigger upper bound than f(x).

However, we can approach the example more systematically as well, but the systematic approach uses the same logic as the intuitive approach. For f(x) to be $O(x^2)$, each term must be less than or equal to x^2, at some point x. For the sake of easiness, let's look at when $x > 1$. The first term, x^2 is obviously less than or equal to x^2, regardless of what x equals to. The second term is $2x$. If we look at the $2x$ term, the $2x$ is greater than x^2 when x is between 1 and 2. But, since we're also allowed to use multiples of x^2, as is stated in the formal definition of big-O, we can look at $2x^2$, and see that $2x^2$ is greater than $2x$ when x > 1. So now we have $2x^2 > 2x$. Note we could have used any multiple of x^2, so long as it made it bigger when x > 1. The final term is 1, and x^2 is greater than 1 when x > 1. Now we can construct this inequality, substituting our new larger terms in:
$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 3x^2$. Since we showed each of the terms were larger than some other value, we can substitute those larger values in to make the inequality. And since $f(x) \leq 3x^2$, which is a multiple of x^2, we

can say that $f(x)$ is $O(x^2)$, when x > 1. We can pick other x values as well to prove that it is $O(x^2)$, but, regardless, we just need to state which x-value we used.

NOTE

This is a very important caveat. Just because $f(x)$ is $O(x^2)$ does not mean, it is not big-O of any other function. Remember O(x) is an upper bound on f(x). f(x) is big-O of a myriad of other functions, not just multiples of quadratics. f(x) is also big-O of x^3 and many exponential and power functions. As long as it is greater than or equal to f(x), f(x) is big-O of that function.

Important Functions

- $n!$ is big-O of n^n and $n \log n$

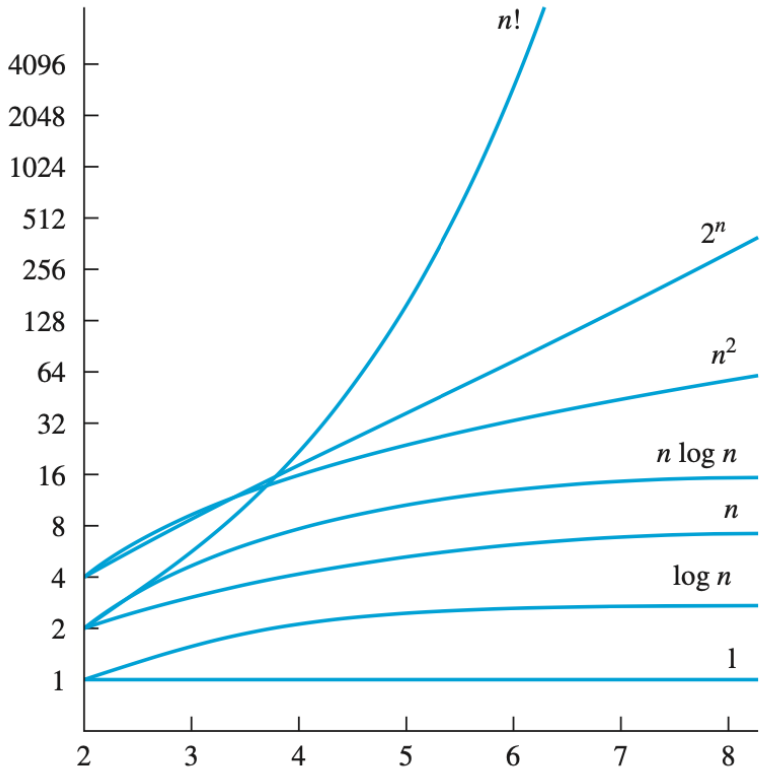
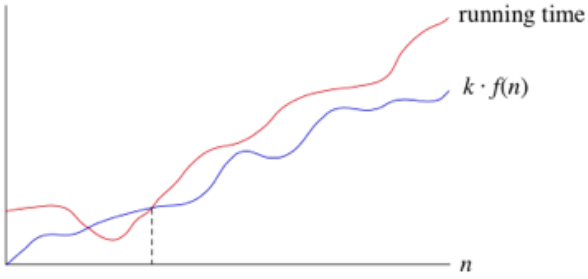


FIGURE 3 A display of the growth of functions commonly used in big-*O* estimates.

Big-Omega

- Big-Omega notation is like big-O notation, but instead of an upper bound, it gives a lower bound.
  > We say that f (x) is $\Omega(g(x))$ if there are constants C and k with C positive such that $|f(x)| \geq C|g(x)|$ whenever $x > k$.

This is what big-omega notation looks like:



Blue is is Big-Omega of f(x) after that dashed line.

- $f(x)$ is $\Omega(g(x))$ **if and only if** $g(x)$ is $O(f(x))$.
  - This is because you're basically switching the bounds, and doing so gives you big-O.

> The function $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$, where g(x) is the function $g(x) = x^3$. This is easy to see because $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers x. This is equivalent to saying that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$, which can be established directly by turning the inequality around.

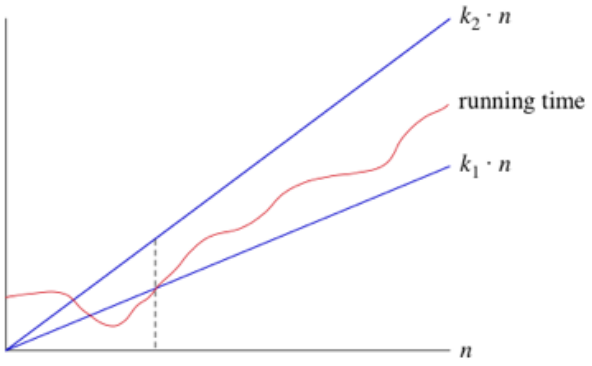
Big-Theta

- Big-Theta is like a combination of big-O and big-Omega. It gives both an upper limit and lower limit on a function. If a function fits between the bounds, it is Big-Theta of the bounds.
- If $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$, then it is $\Theta(g(x))$.
- If it is both big O and big Omega of g(x), that means the only thing that's changing $g(x)$ is a constant, so that it looks like this:

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

- The constants (C1 and C2) are the only thing that's changing $g(x)$ from one bound to another.
- If $f(x)$ is $\Theta(g(x))$, that means $f(x)$ and $g(x)$ are in the same order.
- For polynomials, the leading term determines its order and thus its Big-Theta.

Here's what big-Theta looks like:



Interesting Note From Book:

> Unfortunately, as Knuth observed, big-O notation is often used by careless writers and speakers as if it had the same meaning as big-Theta notation. Keep this in mind when you see big-O notation used. The recent trend has been to use big-Theta notation whenever both upper and lower bounds on the size of a function are needed.

Complexity of Algorithms

- The computational complexity of algorithms is usually measured with either time complexity or space complexity. We'll be focusing on time complexity.

Time Complexity

- Can be expressed in terms of the number of operations used by the algorithm when the input has a particular size.
- The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation.
- Basically, every time you perform an operation (whatever it may be), we add to it's time complexity.
- We use number of operations rather than actual computing time for time complexity because each computer has different times to actually compute the basic operations.
- In this way, it gives us a standard way to measure algorithms, without worrying about the specifics of every computer in existence.
- So remember **Time Complexity = Number of Operations Done** in an algorithm, and input is a some size that which algorithm is doing.

For example in a search algorithm that iterates over an array to find the largest value:

```
int findMax(int arr[], int size) {
    int max = arr[0];
    for {int i = 1; i < size; i++} {
        if arr[i] > max {
            max = arr[i];
        }
    }

    return max;
}
```

Let's identify all the operations that are going on, and let's say `n = size`:

1. Comparing whether `i < size`
    i. This happens `n` times.
    ii. The first `n-1` times, it actually enters the loop. And the last `nth` time to exit the loop.
2. Add to `i` with `i++`
    i. This happens `n-1` times.
    ii. At the last element of the array, it increments `i` by `1`, making it greater than size.
    iii. This would have been `n` times, but we started at `i = 1` rather than `i = 0`.
3. Comparing `arr[i]` with `max`
    i. This also happens `n-1` times.
    ii. Logically, this would be the same amount as `i++`, because `i++` comes after the last line of the function, and this is before it.

So overall our complexity of this function is `2(n-1) + n = 3n - 2`. This means we do `3n - 2` operations when we use this function.

Worst Case Complexity

- What we just did with the earlier example was the worst case complexity.

- The worst case complexity is the largest amount of operartions needed to guarantee to solve the problem.
- In the previous one, though, it ALWAYS runs `3n - 2` times, so saying worst case analysis is kind of redundant.
- Worst case analysis comes more into play, when the amount of times an algorithm needs to run is variable, like a searching algorithm.
- Take the binary search algorithm for example, what would the worst case scenario be if it has $n = 2^k$ elements?
- Every time we do two comparisons, we have 2^{k-1} elements, thus we continue this until we have 2^1 elements left. Then we do two more comparisons: when one term is left in the list, one comparison tells us that there are no additional terms left, and one more comparison is used to determine if this term is x.
- So the number of comparisons we do is: `2k + 2`, which is $2 \log n + 2$.
- However this is the MAX amount of comparisons we need to do.
- We could have just as easily found our eleemnt with our first search.

Complexity of Algorithms

Complexity	Terminology
$\Theta(1)$	Constant Complexity
$\Theta(logn)$	Logarithmic complexity
$\Theta(n)$	Linear Complexity
$\Theta(nlogn)$	Linearithmic Complexity
$\Theta(n^b)$	Polynomial Complexity
$\Theta(b^n)$, where $b > 1$	Exponential Complexity
$\Theta(n!)$	Factorial Complexity

- Binary search has logarithmic complexity.
- Many important algorithms have n log n, or linearithmic (worst-case) complexity

Tractability and Solvability

- A problem that is solvable using an algorithm with polynomial (or better) worst-case complexity is called tractable.
  - Provided n is not very large
- Intractable otherwise.
- If an algorithm cannot solve a problem, that problem is unsolvable.

Computer Time Used by Algorithms

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*