2.2-2

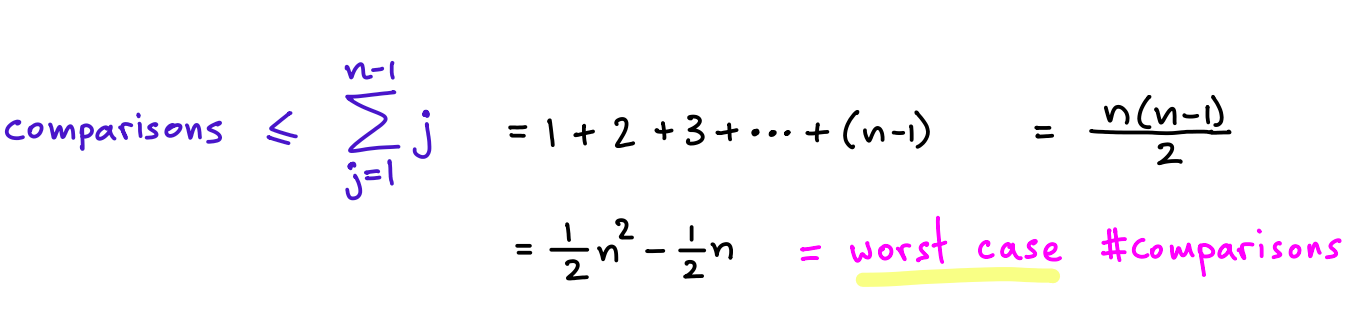
Understand what Selection sort does, and then determine what the worst case time complexity is, using big-O notation. What is the best case time complexity? Ignore pseudocode.

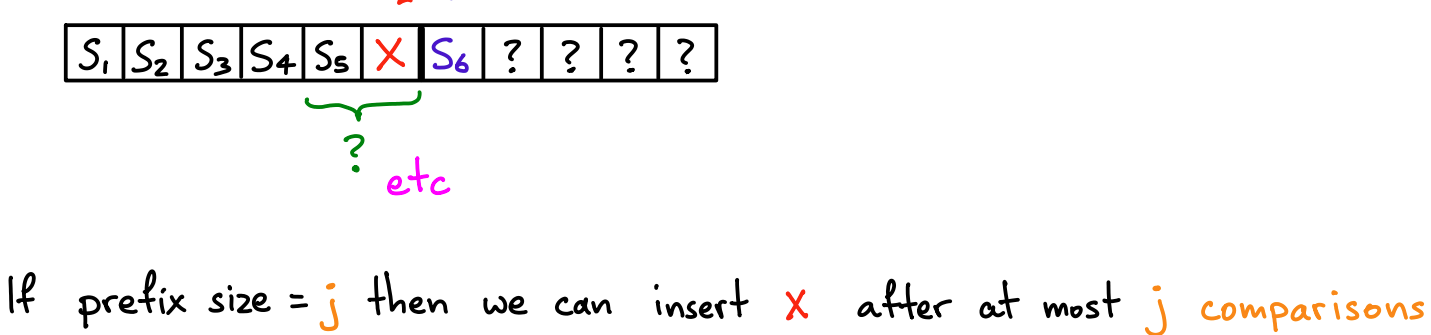
Selection sort establishes and maintains an array of ordered numbers beginning from the pivot of n = 0. It starts at the beginning of an unsorted list and establishes the first element as the smallest element. It will look to the element right next to the array and compare the two elements. If the next element is smaller than the element in the array, then the two elements swap and the array pivot is incremented.

If the next element is larger, then the array is incremented because it would mean that they’re already in order.

Upon checking the next element outside of the array, it will check the two elements to determine which one is smaller. If the next element is smaller than the element in the array, then the two swap. The element will check the next item in the array to see if it needs to be swapped further. The element will keep swapping until it either reaches the end or reaches an element that is smaller than the swapped element. This means the array is now in-order. This process repeats until the entire array has been checked and swapped.

Worst Cas time complexity and best-case time complexity is O(n2). The entire array has to be iterated once. Because the sorting algorithm terminates when prefix size = n, that means checking the entire array, adding a time complexity of n. For each element checked at n, it will have to check each element up to the n prefix.





Scanning the minimum number requires scanning all n elements. Finding where to swap the element to requires scanning the remaining n-1 elements. ½ (n2-n) comparisons through a geometric series.

**2-2**

Understand what Bubble sort does, and then determine what the worst case time complexity is, using big-O notation. What is the best case time complexity? How would you explain to someone that this algorithm is correct?

Bubble sort takes two elements next to each other and compare them. If the two elements need to be swapped, then they are swapped before incrementing to the next two elements. The sort checks the next two elements and swaps them if needed and repeats. The process will loop over the entire array until the entire array is sorted.

Worst case time complexity of Bubble sort would be O(n2) where it would have to loop over the entire array multiple times in a geometric series.

In a best case scenario, the algorithm goes over the entire array once and confirms that it is sorted, giving O(n) time complexity.

**3.1-2**

Show that (n + a) b = Θ(n b ), as long as a and b are constants, and b is positive.

**2.2-4**

Here’s why in general best-case analysis is not as interesting as worst-case analysis, in other words if you claim that your algorithm has a really good best-case performance, most likely people will not care: We can slightly alter any algorithm, so that it will have a really good best-case analysis. How?

You can change any sorting algorithm to have O(N) time complexity by checking if the match is perfect. It will iterate over the array once to verify, but I doesn’t affect the worst case or average cases of the time complexity.

**3.1-3**

What’s wrong with saying “the time complexity is at least O(n2 )”? What would be an equivalent incorrect statement involving Ω?

Because Big-O(n2) means that the time complexity will go only up to c\*g(n). It can intersect but never go above n2\*c.

The equivalent statement would be at most Ω(n2)

**3.2-4** (relatively difficult)

Assume values of n such that log log n is integer. (That’s why I didn’t include the ceiling function). (a) Is (log n)! = O(n k ) for some constant k? (b) Is (log log n)! = O(n k ) for some constant k? I’ll give a hint on the next page.

Hint for 3.2-4: You will probably want to take a log, and use exaggeration / underestimation. There is some similarity to Hw1p3.

2.3-5 (Ignoring the part about pseudocode, and expanded a bit)

What recurrence describes the time complexity of binary search?

How do you determine the time complexity using a tree, induction, and master method?

* Induction method: Make a guess for the solution then use induction to prove if the guess is correct or incorrect
* Tree method: By constructing a recurrence relation and forming a tree of what the recurrence relation would produce as you reiterate
* Master method: Transform the solution into T(n) = aT(n/b) + f(n)
  + Requires a recurrence relation of a specific size. Otherwise, it works similar as the Tree method and picks a specific case and derive from that.
  + Find the case of f(N) = O(nc) where c is < Logba
    - Then T(n)=O(nlogb a)
  + Find the case of f(N) = O(nc) where c is = Logba
    - Then T(n)=O(nc log n)
  + Find the case of f(N) = O(nc) where c is > Logba
    - T(n)=O(f(n))

**2.3-6**

In Insertion sort can we get an improvement by using binary search instead of a linear scan from right

to left?

Binary search has n log n time complexity. Worst case time complexity of binary search is O(n2) due to the number of comparisons. This worst case scenario is due to the process of having to find and sort every number in the array. Binary search doesn’t change the possibility that each individual element has to search for and swap with another element. It would remain as O(n2)

**2.3-7** (wording is directly from CLRS)

Describe a Θ(n log n)-time algorithm that, given a set S of n integers and another integer x, determines

whether or not there exist two elements in S whose sum is exactly x.

**2.1**

Lengthy description. See textbook. Ignore (d).

Short version, without the hints in the book:

a) In Merge sort if we use base cases of size k, and on each base case we use Insertion sort, what is the

time complexity of dealing with all the base cases? Don’t assume k = O(1), use it as a parameter.

b) What is the time complexity of the rest of Merge sort? (Everything except the base cases)

c) How large can k be without exceeding the time complexity of normal Merge sort?

**4.3-1** (More than what the book asks, and no hint)

Solve T(n) = T(n − 1) + n. Use a tree. Confirm by induction. Try upper and lower bounds.

**4.3-2** (without the hint)

Derive an upper bound for T(n) = T(dn/2e) + 1

**4.3-8** (I don’t really like this problem, but someone asked about it)

Solve T(n) = 2T(bn/2c + 17) + n

**4.3-8** (page 87)

What’s wrong with the problem statement?

4.3-9 (after you do the similar homework problem)

Solve T(n) = 3T(

√

n) + log n

4.4-2

T(n) = T(n/2) + n

2

. Get an upper bound with a tree and by induction. Why is the lower bound easy?

4.4-3

T(n) = 4T(n/2 + 2) +n. For the upper bound, use a tree, and induction. Why is the lower bound easy?

4.4-4

T(n) = 2T(n − 1) + 1. Get an upper bound with a tree and by induction.

4.4-6

T(n) = T(n/3) + T(2n/3) + cn. Why is this intuitively Ω(n log n)?

Verify using a tree, and by induction. Get a matching upper bound.

Note: 4.4-9 is a generalization of this.

4.1

See page 107 of the book. Try to see if you can figure these out without writing anything.