Rules of the game

- Deadline is 11:59PM, May 6th
- Make directory midterm/ in your repo. LaTeXyour solution or scan it and save it as midterm.pdf in that directory.
- You must work on this alone.
- Any correct answer makes up half your deficit. Suppose, in the actual midterm, you got x points on a question worth y points. Now, suppose you submit the correct answer. I will give you $(y - x)/2$ extra credit points.
- Grading will be pretty strict, so it's rare for any partial points. Since this is a bonus over the midterm, I expect fairly precise well-written answers.
- There is no need to answer questions that you got correct in the exam. You can go to the TAs to see your graded midterm. More details on that on Piazza.
- For the multiple choice questions: I have already revealed the answer. You need to explain why the other answers are wrong. Nothing more than three sentences for each option please. The TAs will explain the correct answer for the multiple choices questions in the discussion sessions, so these are basically free points!
- The old Q3 is not part of this, since almost all of you got it correct.
- I have added more clarifications in some of the questions, so you know what is needed for a complete answer.

Q1 (2 points): Compare the functions $n \log_2 n$, n, and $n^{1.1}$ in $O(\cdot)$ notation. Which of the following is true?

- (A) $n^{1.1} = O(n \log_2 n)$, $n = O(n^{1.1})$, but $n^{1.1}$ is not $\Omega(n)$.
- (B) $n = O(n \log_2 n)$, $n \log_2 n = O(n^{1.1})$, but $n \log_2 n$ is not $\Omega(n^{1.1})$.
- (C) $n = O(n \log_2 n)$ and $n \log_2 n = \Theta(n^{1.1})$.

Answer:

To prove that A and C are not the correct answers, we will have to use the definition of O, Ω, and θ

(A):
$n^{1.1} = O(n \log_2 n) \rightarrow 0 \leq n^{1.1} \leq c * n \log_2 n$ for all n $\geq n_0 \rightarrow \lim_{n \to \infty} \frac{n^{1.1}}{n \log_2 n}$ = ∞
Since $n^{1.1} \neq O(n \log_2 n)$, we can disqualify option A

(C):
We will try and prove $n \log_2 n = \Theta(n^{1.1})$ by using the definition discussed in class $n = O(n \log_2 n)$ and $n \log_2 n = \Theta(n^{1.1}) \rightarrow 0 \leq c_1 n^{1.1} \leq n \log_2 n \leq c_2 n^{1.1} \rightarrow 0 \leq \frac{n \log_2}{n} \leq c_2 \rightarrow \lim_{n \to \infty} \frac{n \log_2 n}{n}$ = ∞
Since $n \log_2 n \neq \Theta(n^{1.1})$, we can disqualify option C.

Therefore the correct option is B

Q2 (2 points): Consider the following sorting algorithm. If the input length is at most 10^9, run insertion sort. Otherwise, run Mergesort. Suppose $T(n)$ denotes the worst-case time complexity of this algorithm. Which is true?

- (A) $T(n) = \Theta(n \log_2 n)$.
- (B) $T(n) = \Theta(n^2)$.
- (C) $T(n) = O(n \log_2 n)$ but not $\Omega(n \log_2 n)$.

Answer:

(B):
$T(n) = \Theta(n^2)$ Is not correct, because it assumes that our input is bounded. It is not, our input is ∞, and because our input keeps increasing, we will pass the if statement of using *Insertion Sort* and go on to use Merge Sort, which gives us a function that is infinitely increasing by $n \log_2 n$

(B):
B is not correct because it implies that out upper bound is $O(n \log_2 n$ but it is not somewhere below it. Which does not make sense. We can use the same logic as mention in part B. Our input is infinitely increasing at a rate of $n \log_2 n$ after it surprasses the input size of 10^9. This option does not make sense.

Therefore the correct option is A

Q4 (4 points): We discussed the notion of stability for sorting algorithms. Prove the Mergesort (with a very simple modification/clarification) is stable. Show that Quicksort is not stable, by providing an example.

For Mergesort, show pseudocode of the modified version, clearly indicating the modification. Give a proof that this version is stable.

For Quicksort, clearly write an example input. Show how Quicksort in unstable, by following the steps of Quicksort and indicating where the instability happens.

(Half credit for only solving one of these.)

Solution:

The modification of Mergesort in order to make it stable will occur within the "Merge" portion. Assuming we know the Mergesort(A) function we discussed in class on April 4rth, here is the modifiedMerge(L, R). Which we all also show is correct. Most of this psuedo code is taken from lecture as well:

1.) modifiedMerge(L, R)
2.) $b[m+1] = C[m+1] = \infty$
3.) Initialize D as empty array
4.) i = j = k = 1
5.) while B[i] $< \infty$ or C[j] $< \infty$
6.) if B[i] $\leq$ C[j] // stabilize algorithm
7.) D[k] = B[i]
8.) i = i + 1
9.) else
10.) D[k] = C[j]
11.) j = j + 1
12.) k = k + 1
13.) return D

As we can see from line Line 6, we need to change the less than sign to less than or equal too. This is because if it weren't for the equal sign, the merge function (when comparing two equal elements in sorted arrays B and C) would completely skip over the first element, putting the second element in D first. This ruins stability. The equal sign would fix this case, and put the elements in order ass they appear in the original array. We can formally prove this by using the loop invariant - *At the beginning of each while loop, D contains all of the elements of* $B[1..i-1]$ *and* $C[1..j-1]$ *in sorted order, and as they appear. When the loop terminates, i = j = m + 1. So by loop invariant, D is the desired output, as well as stable.*

Quick sort is stable for one simple reason: *It swaps elements that are non adjacent*. I am going to refer to the Quicksort pseudo code that was discussed in class on April 12, 2016. Let us say that our example input is an array $A = [3, 3, 1, 4]$. Let's create a permutation array that allows us to track the elements in our code and visualize how the elements shift when sorted. Let's call this array $A' = [*, !, **, ***]$. We can clearly see here that the two values that are equal, are distinguished by a $*$ at index 1 and a ! at index two. In line 9 of the psuedo code that was presented in class (Swap $A[i]$ and $A[j]$), the algorithm will completely ignore the order, and see that i meets the conditions of being less than or *equal* to j, and will swap the two values. This makes the sorted array $A' = [**, \underline{!}, \underline{*}, ***]$ instead of $A' = [**, \underline{*}, \underline{!}, ***]$ like we would want it to be, therefore making the algorithm Quicksort, unstable.

Q5 (4 points): You're a financial analyst, and have another stock market problem to solve. The input is A, an array of stock prices. For day i, the best trade is the maximum profit that can be achieved by buying at day i and selling on a subsequent day. For convenience, you can define the best trade for the last day to simply be $-A[n]$ (because if you buy on the last day, you cannot sell and you lost all your money).

Give the pseudocode of an algorithm that returns an array containing the maximum profit for every day in A. Prove that your algorithm is correct, and do a complexity analysis. I'm looking for an $O(n \log_2 n)$ algorithm using divide and conquer. In class, we solve the problem of finding the best trade over all possible buying days. You might want to revisit that algorithm, and modify it to get the algorithm for this problem.

There is a completely different $O(n)$ algorithm for this problem. If you can find that, I'll give you two bonus extra credit points. (If you didn't get this in the exam, you still have to give the $O(n \log_2 n)$ algorithm.)

$O(n \log(n))$ Divide and Conquer Solution:

I will refer to the pseudo code discussed in class in April, and modify it appropriately so that it returns an array of the correct stock prices, instead of just the pair of days it returns now. The pseudo code for modMax-Diff is as follows -

1.) modMax-Diff(A)
2.) n = A.length
3.) if n ==2, return A[$(A[[2] - A[1])$, $(-A[2])$] // Our solution is trivial
4.) L = A[1 ... $\frac{n}{2}$]
5.) R = A[$\frac{n}{2} + 1$,... n]
6.) $L_{maxProfit}$ = Max-Diff(L) // Recursively give us a sub array that contains max profit for each day
7.) $R_{maxProfit}$ = Max-Diff(R)
8.) Concatenate $L_{maxProfit}$ and $R_{maxProfit}$ and call it A // A now contains max profits in L and R subarrays
9.) for i = 0 to L.length // loop to check if max is in original Right sub array
10.) if a max to all the values in L, is contained in R
11.) Swap A[i] with (MaxNumInR - L[i]) // subrtact new max with original L value
// We do not check for max in right sub array, as comapred to left because we are only looking for the max profit on subsequent days

$O(n\log(n))$ Time Complexity Analysis:

Since we only added constant running time to our algorithm, our recurrence relation will be that of what we discussed in class: $T(n) \leq 2T(\frac{n}{2}) + cn$. We will use the Master Theorem to find the time complexity of modMax-Diff. The Master THM tells us -

Theorem (Master Theorem)

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

We see a = 2, b = 2, and c = 1
From the Master Theorem, we see that $2 = 2^1$, $2 = 2$
This matches case three in the theorem, meaning our time complexity is $T(n) = \theta(n^d \log n)$
.
The running time or worst case time complexity of modMax-Diff is $T(n) = \theta(n \log n)$
.

$O(n)$ Linear Solution:

We can find all of the information we need in linear time, by simply starting at the end of the array A, and decrementing indices as me move backwards and collect max prices. The time complexity is $T(n) = O(n)$ because this algorithm only makes one pass through the array and uses only one for loop. The psuedo code for stockPrices(A) is as follows -

1.) stockPrices(A)
2.) Initialize an array called $profit$ of length A.length
3.) max = 0 // To keep track of the current maximum number in the array
4.) For i = A.length to n // Start at the end of the array
5.) profit[i] = max - A[i] // subtract because largest profit is always current - subsequent max
6.) max = $maximum$(max, A[i]) // Find the maximum of the two, and set to new current max

$O(n)$ Time Complexity Analysis:

The analysis for this algorithm is very simple. We are iterating through the input only one time. The number will always be fixed and constant. For every iteration, you are doing a fixed number of elementary operations, or the cost is always cn. This cost will never change with a given input. Therefor, we know that the time complexity is simply $T(n) = O(n)$.

Q6 (4 points): Given an array A, give pseudocode and big-Oh time-complexity analysis for an algorithm that determines which permutation of the *stable* sorted order is A. Meaning, output an array P, such that $P[i]$ is the index of the element $A[i]$ after a stable sort of A. (You can assume that all elements in A are distinct.)

For example, given $A = [4\ 3\ 10\ 8\ 4]$, the output should be $P = [2\ 1\ 5\ 4\ 3]$. So $A[1] = 4$ and it is the second element in the sorted order. Thus, $P[1] = 2$. Since $A[2] = 3$ and is the minimum element, $P[2] = 1$. So on and so forth. Note that because of the stable sorting, the last 4 goes to the third position.

You may assume access to a stable sorting algorithm, like the version of Mergesort you designed in Q4. (If the whole discussion about stability is confusing you, go ahead and solve this problem assuming all elements are distinct. You will get partial credit.) You might want to revisit the lecture on Lower Bounds, and look at the permutation problem discussed there.

If you solve this assuming all elements are distinct, you get half extra credit points.

Solution:

Since we are allowed to assume access to a stable sort, I will use the algorithm of modified Merge I figured out in Q4, as well as the original mergesort discussed in class during the month of April. I will modify it to determine the permutation of the stable sorted order of array. We simply replicate A, and then use this replication to swap the indices of our sorted array D. Modifications to Merge are highlighted in red

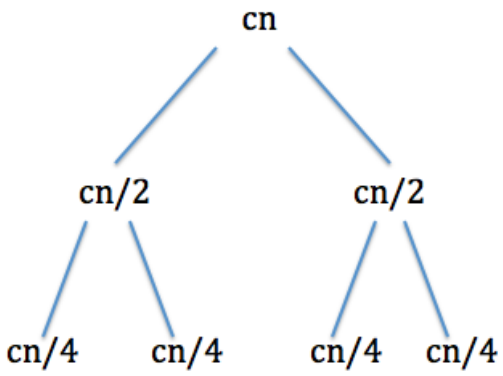
1.) modifiedMergeSort(A)
2.) Create an array A' that has is identical to A // A' will help us track our indices in Merge fcn
3.) n = A.length
4.) if n == 1, return A
5.) $L = A[1...\frac{n}{2}]$
6.) $R = A[\frac{n}{2} + 1...n]$
7.) L_{sort} = modifiedMergeSort(L)
8.) R_{sort} = modifiedMergeSort(R)
9.) Return modifiedMergeSort(L_{sort},R_{sort})

1.) modifiedMerge(L, R, A') // We will keep track of our original perm array by passing it along as well
2.) $B[m + 1] = C[m + 1] = \infty$
3.) Initialize D as empty array
4.) i = j = k = 1
5.) while B[i] $< \infty$ or C[j] $< \infty$
6.) if B[i] $\leq$ C[j] // stabilize algorithm
7.) D[k] = B[i]
8.) i = i + 1
9.) else
10.) D[k] = C[j]
11.) j = j + 1
12.) k = k + 1
// At this point, D is sorted
13.) x = y = 1 // our index trackers
14.) If D[x] = A'[y] // If the value in our Sorted D is equal to a value in A'
15.) D[x] = y // That value in D is now equal to the index of that value in A'

See next page for Time Complexity Analysis of Mergesort

We can see from modifiedMergeSort (MMS) and modifiedMerge, that not much was changed. We added an array, at the very beginning, which would add to the time complexity by a constant factor. We don't consider constant factors when analyzing the running time of algorithms. At the very end of MM, we used one loop to swap the values in our array D, both of which have time complexity O(1). From this, we know that the time complexity for modifiedMergeSort is $O(n \log(n))$, which we learned in class on April. A formal proof of modifiedMergeSort is as follows -

An example of a recursion tree for modifiedMergeSort -



From structure of the recursion tree, we can see come up with the recurrence relation of $T(n) \leq 2T(\frac{n}{2}) + cn$. Using the Master thm which is -

Theorem (Master Theorem)

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

We see a = 2, b = 2, and c = 1
From the Master Theorem, we see that $2 = 2^1$, $2 = 2$
This matches case three in the thm, meaning our time complexity is $T(n) = \theta(n^d \log n)$
.
The running time or worst case time complexity of modifiedMergeSort is $T(n) = \theta(n \log n)$
.