— You will be graded on *clarity*, *correctness*, and *precision*. When asked to present an algorithm, you should give at least 2-3 sentences explaining your approach, then pseudo-code, then a run-time analysis, and a short explanation for why the algorithm is correct. For example, in the case of dynamic programming problems, be sure to define what the variables in your equation represent in words and give an equation explaining the recursive structure of the problem. Do not write contradictory or ambiguous statements in your answers.

— You may consult the lecture slides as posted on my website under 22s-5800. No other external resources or aids are allowed.

— This is an individual exam. You must not discuss the exam problems with anyone else except the course staff. Do not give or receive any assistance to anyone else in the course. Prepare and type each answer individually without assistance.

— You must tag the pages of your solution when you submit in Gradescope.

PROBLEM 1 *Fewer Routes with 3 hops*

Your second homework asked you to construct a $\Theta(n \log n)$-segment North-South route system (i.e., all of the routes in the system are only allowed to move from i to $j > i$) that allows any commuter to get from stop i to j using at most 2 segments.

In this problem, we design a new North-South system that allows any commuter to get from stop i to $j > i$ in at most 3 "hops", but only requires $\Theta(n \log \log n)$ segments. Hint: the basic idea is to divide the n stops into groups of size $\lceil \sqrt{n} \rceil$. Build a system for these groups recursively so that any travel entirely within these groups can be accomplished in 3 hops. In addition, each group has designated "hubs" which have both incoming and out-going segments to other stops. It is then possible to add $\Theta(n)$ segments to and from these hubs so that a person can get from any stop i to any stop j within another group using at most 3 route segments. Note: these extra routes cannot start at a node j and go backwards to a node $i < j$.

(a) Fully describe this scheme in 3-4 sentences. Clearly specify how to add the $\Theta(n)$ segments to implement the 3-hop property in your route system.

(b) State a recurrence that counts the segments required by your route system.

(c) Prove a tight upper bound on your recurrence using induction.

Ans.

(a) Let the stops be numbered from $(1, \ldots, n)$. Now we divide the stops into groups of size $\lceil \sqrt{n} \rceil$. So first group has $(1, 2, ..., \lceil \sqrt{n} \rceil)$, second group has $(\lceil \sqrt{n} \rceil + 1, \lceil \sqrt{n} \rceil + 2, ..., 2\lceil \sqrt{n} \rceil)$ and so on. There will be total of $\lceil \sqrt{n} \rceil$ such groups.
Let the groups be denoted by numbers $G_1, G_2, ..., G_{\sqrt{n}}$. Within each group we can connect all stops to the last stop(hub). It will take a total of $\lceil \sqrt{n} \rceil - 1$ segments for each group. So for $\lceil \sqrt{n} \rceil$ groups total number of incoming incoming segments = $\lceil \sqrt{n} \rceil * (\lceil \sqrt{n} \rceil - 1) = \Theta(n)$ incoming segments.
We also connect last stop of G_i to first stop of every group G_k such that $i < k \leq \lceil \sqrt{n} \rceil$. Thus total outgoing segments for $\lceil \sqrt{n} \rceil$ such groups = $1 + 2 + ... + (\lceil \sqrt{n} \rceil - 1) = \frac{(\lceil \sqrt{n} \rceil - 1)(\lceil \sqrt{n} \rceil)}{2} = \Theta(n)$ total outgoing segments.
Thus the total number of segments from all these hubs $= \Theta(n)$.

(b) Adding segments in the above mentioned methods leads to a maximum hops of 3 from i to j when i and j are in different Groups. We can recursively add segments within these $\lceil \sqrt{n} \rceil$ groups to get 3 hop segments within each groups. Hence the recurrence relation for count of segments becomes

$$S(n) = \lceil \sqrt{n} \rceil * S(\lceil \sqrt{n} \rceil) + \Theta(n)$$

(c) The total number of incoming and outgoing segments for all the hubs = $\lceil \sqrt{n} \rceil * (\lceil \sqrt{n} \rceil - 1) + \frac{(\lceil \sqrt{n} \rceil - 1)(\lceil \sqrt{n} \rceil)}{2}$ which is upper bounded by $2n$ for large values of n. Thus we can write the recurrence equation as $S(n) \leq \lceil \sqrt{n} \rceil * S(\lceil \sqrt{n} \rceil) + 2n$.
Hypothesis: $S(n) \leq 100 * (n \log \log n)$
Validity for small value of n

$$\begin{aligned} S(4) &\leq 2 * S(2) + 4 \\ &= 2 * 2 + 2 \\ &\leq 100 * (4 \log \log 4) \end{aligned} \tag{1}$$

Thus it is valid for small values of n.
Let there be a positive integer k such that the equation is valid for all $n \leq k$.
Thus, $S(k) \leq 100 * (k \log \log k)$.
Now, for $n = k + 1$, we have

$$\begin{aligned} S(k+1) &\leq \lceil \sqrt{k+1} \rceil * S(\lceil \sqrt{k+1} \rceil) + 2(k+1) \\ &\leq \sqrt{k+1} * S(\sqrt{k+1}) + 2(k+1) \\ &\leq \sqrt{k+1} * (100 * (\sqrt{k+1} \log \log \sqrt{k+1})) + 2(k+1) \\ &= 100 * (k+1) * (\log \log (k+1) - \log 2) + 2(k+1) \\ &= 100 * (k+1) * (\log \log (k+1)) - (100 \log 2 - 2) * (k+1) \\ &\leq 100 * (k+1) * (\log \log (k+1)) \end{aligned} \tag{2}$$

Which is the required RHS. Hence using Principle of Mathematical Induction, we can say that $S(n) \leq 100 * (n \log \log n)$.
Thus we have $\mathbf{S(n) = O(n \log \log n)}$

PROBLEM 2 *Top earnings*

You manage a hedge fund. Suppose you are given an array of numbers $a_1, \ldots, a_n$ that summarizes a trader's earnings (or losses) for n days. Your goal is to present a dynamic programming algorithm that computes the most lucrative trading period for this trader; i.e., your algorithm must find the pair (i, j) where $i \leq j$ that maximizes the sum $s = \sum_{k=i}^{j} a_k$. The algorithm should output (i, j, s) and run in $\Theta(n)$ time.

(a) Define a variable BEST_n in the style of DP solutions that can help solve this problem. Use sentences to explain what the variable represents.

(b) Provide an equation that defines this variable.

$$\text{BEST}_n =$$

(c) Provide pseudo-code for a $\Theta(n)$-time algorithm. Prove your algorithm correct and analyze its running time. (Assume additions are $O(1)$-time operations.) Your pseudo-code should require < 20 lines in total and should output (i, j, s).

Ans.

(a) For the given array of numbers $a_1, \ldots, a_n$, let us consider a variable BEST_i representing the most lucrative trading that includes i^{th} day.
Now, i^{th} day can be included in period ending at $i-1$ or we can start a new period with that day.
If i^{th} day is included in previous period, its earnings is added to previous streak, and if starting a new period, it will be the only earning. The most lucrative trading including i is the maximum of these two possibilities.

(b) Thus we can write a DP-Equation for BEST_n as follows:

$$\text{BEST}_n = \max \begin{cases} \text{BEST}_{n-1} + a_n \\ a_n \end{cases} \tag{3}$$

Based on this, BEST_1 will be a_1.

(c) Since BEST_i represents the most lucrative trading period that includes a_i and ends at i^{th} day, and the most lucrative trading period for the trader could end at any day, the most lucrative trading period for the trader is the maximum value of BEST_i for $i \in \{1, \ldots, n\}$.
We can keep track of the starting indices of the most lucrative trade ending at i^{th} day (START_i), based on which of the two conditions for BEST_i results the maximum.

Pseudocode for TopEarnings:

TopEarnings$(a_1, \ldots, a_n)$

```
1  (BEST_1, START_1) ← (a_1, 1)
2  for i ← {2, ..., n}
3     if BEST_{i-1} > 0
4        (BEST_i, START_i) ← (a_i + BEST_{i-1}, START_{i-1})
5     else
6        (BEST_i, START_i) ← (a_i, i)
7  k ← 1
8  for i ← {2, ..., n}
9     if BEST_i > BEST_k
10       k ← i
11 return (START_k, k, BEST_k)
```

TIME COMPLEXITY ANALYSIS:
Line 2 will be called for $\Theta(n)$ times. So BEST_i and START_i will be computed $\Theta(n)$ times. Best end index(k) is also calculated in $\Theta(n)$. All other lines take $\Theta(1)$.
Hence the overall time complexity of TopEarnings$(a_1, \ldots, a_n)$ is $\Theta(n)$.

PROBLEM 3 *COVID testing*

During this pandemic, epidemiologists have been faced with many challenging problems. One issue that arose was tracking the dominant variant of the virus that was infecting the population.

Suppose a testing lab is given n viral samples from a population on a given day. Using specialized test equipment, it is possible to use a COMPARE test to determine whether two samples are the same or different variants. Although this test is expensive, it is faster and cheaper than running full genetic sequencing on both strands and then comparing the sequences; however, the test only returns whether the two samples are the same or not.

On input the n samples $(s_1, \ldots, s_n)$, give the pseudo-code of a divide and conquer algorithm that uses only $\Theta(n)$ calls to COMPARE(s_i, s_j) to determine if there is one variant that is a majority among the day's samples. Your algorithm simply outputs "yes" or "no" if there is a majority variant.

In this case, a majority sample is one that occurs *more* than $\lfloor n/2 \rfloor$ times. For example, if $n = 100$, then the majority item occurs at least 51 times. If $n = 99$, the majority item occurs at least 50 times.

Explain why your algorithm is correct. Analyze the number of tests your algorithms performs using a recurrence.

Hint: this problem is similar to the counting problem from our first lecture. Be careful to handle all base cases properly. Your pseudo-code should require no more than 15 lines in total.

Ans. Let us assume the given samples $(S_1, \ldots, S_n)$ has a majority variant S_m. If we divide the samples in pairs, we can find that there is atleast one pair of samples, both of which are same as S_m. Now for any given pair (S_i, S_j) the following cases arise:

(a) COMPARE(S_i, S_j) returns SAME, and COMPARE(S_m, S_i) returns SAME

(b) COMPARE(S_i, S_j) returns SAME, and COMPARE(S_m, S_i) returns DIFFERENT

(c) COMPARE(S_i, S_j) returns DIFFERENT, COMPARE(S_m, S_i) returns SAME, and COMPARE(S_m, S_j) returns DIFFERENT

(d) COMPARE(S_i, S_j) returns DIFFERENT, COMPARE(S_m, S_i) returns DIFFERENT, and COMPARE(S_m, S_j) returns SAME

(e) COMPARE(S_i, S_j) returns DIFFERENT, COMPARE(S_m, S_i) returns DIFFERENT, and COMPARE(S_m, S_j) returns DIFFERENT

For cases c,d, and e, if we remove both S_i and S_j from the list of sample, the majority among the remaining samples still remains S_m, since removing 0 or 1 samples of S_m from 2 samples, still makes S_m as the majority.

For cases a and b, we can replace S_i and S_j with a single sample of S_i with frequency of 2. Since a single sample can exist (last sample if n is odd), we add it as S_n with frequency 1.

We can recursively do these operations and add frequencies of SAME pairs and

replacing with single instance till atmost one sample remains. For cases c,d, and e, we can remove one of the sample having smaller frequency as removing it still wont change the majority sample S_m. Since we removed many samples, the sample we get is the potential majority sample. We can check the majority sample with every other sample to get its actual frequency and confirm if the given samples contain a majority sample or not. Based on the above explanation, we can write Pseudocode for MajorityTesting as follows:

MajorityTesting$(S_1, \ldots, S_n)$

1 Initialize $Freq_i \leftarrow 1$ for $i \leftarrow \{1, ..., n\}$
2 S_m = PotentialMajority$(S, Freq)$
3 $ActualFreq \leftarrow 0$
4 Add 1 to $ActualFreq$ whenever Compare$(S_i, S_m) =$ Same **for** $i \leftarrow \{1, \ldots, n\}$
5 **if** $ActualFreq > \lfloor \frac{n}{2} \rfloor$ **return** Yes
6 **else return** No

PotentialMajority$((S_1, \ldots, S_n), (F_1, \ldots, F_n))$

1 **if** $n = 0$ **return** None
2 **if** $n = 1$ **return** S_n
3 **for** $i \leftarrow \{1, 3, 5, ..., \leq n\}$
4 **if** Compare$(S_i, S_{i+1}) =$ Same
5 Add S_i in *SampleList*, Add $F_i + F_j$ in *FreqList*
6 **else if** $F_i \neq F_j$
7 Add $\max_i^{i+1}\{F_k\}$ and corresponding S_k in *FreqList* and *SampleList*.
8 **if** n is odd, Add S_n in *SampleList* and F_n in *FreqList*
9 **return** PotentialMajority(SampleList, FreqList)

ANALYSIS ON NUMBER OF CALLS TO Compare(S_i, S_j)
In PotentialMajority, Compare is called in line 4 for maximum of $\lfloor n/2 \rfloor$ times. Also In line 9, the maximum number of samples for next call to PotentialMajority will be $\lceil n/2 \rceil$ since minimum one sample is removed during each iteration. Along with it, Compare is called n times in MajorityTesting line 4. Based on these, we can write the following recurrence relationship:

$$C(n) = C(\lceil n/2 \rceil) + \lfloor n/2 \rfloor + n$$

Where $C(n)$ represents the number of calls to Compare for n samples. Let $a = 1$, $b = 2$ and $f(n) = \lfloor n/2 \rfloor + n = \Theta(n)$. With this, we can write above equations in the form of $C(n) = aC(n/b) + f(n)$.

Since $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Also we have $f(n) = \Theta(n)$. Hence, by using Case 3 of Master's Theorem, we have

$$C(n) = \Theta(n)$$

PROBLEM 4 *Day Trader's Decisions*

A day trader wants to trade $n \cdot 100$ shares of the stock AAPL in blocks of 100. Temporary market rules prevent her from short-selling; in other words, she must own a block of shares before selling them. To avoid price slippage from large orders, she can only buy or sell 100 shares, i.e., 1 block, at a time. At the end of the day, she wants to own zero shares of AAPL.

How many ways can she do her trading? For example, when $n = 2$, she can do "BUY, SELL, BUY, SELL" or "BUY, BUY, SELL, SELL" and so there are 2 ways. Present your answer as a recurrence and clearly explain why the recurrence captures this number.

Ans. A valid sequence of trading will have number of BUYs $\geq$ number of SELLs at any point in the sequence and number of BUYs = number of SELLs in the end. Based on this, we can pair a BUY_i with its corresponding SELL_i. For example, "BUY_1, SELL_1, BUY_2, SELL_2" or "BUY_1, BUY_2, SELL_2, SELL_1".

Now, for any i, if we remove both BUY_1 and SELL_1, two sequences are formed before and after SELL_1. Since the original sequence is valid, the new sequences made should also be valid. If the left sequence is of i trades then the right sequence will be of $n - i - 1$ trades as we removed one pair of trade(BUY_1 and SELL_1) to divide into two sequences. Since both are instances of valid sequences, we can calculate number of different sequences formed with left and right sequence trade sizes i and $n - i - 1$ by multiplying number of valid sequences formed by these two sequences. Iterating over i will give the number of ways to trade n trades.

It can also be the case that either of the sequence remains empty. Thus size = 0, It is the case when the sequence is like "BUY_1, SELL_1, ..." (if left sequence is empty) or "BUY_1, ..., SELL_1"(if right sequence is empty). For this we can simply add the number of ways trading size $n - 1$ is valid.

Based on the above observations, we can write the recurrence relation as:

$$T(n) = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ 2 * T(n-1) + \sum_{i=1}^{n-2} T(i) * T(n-1-i) \end{cases} \tag{4}$$

PROBLEM 5 *Faster Price Run (Extra Credit)*

This question develops the challenge problem from Price Run in Homework 3. In the problem you are given a list of closing stock ticker prices $p_1, p_2, \ldots, p_n$ and the goal is to find the length of the longest (not necessarily consecutive) streak of prices that increase or stays the same. For example, given the prices $2, 5, 2, 6, 3, 3, 6, 7, 4, 5$, there is the streak $2, 5, 6, 6, 7$ of prices that increase or stay the same, but an even longer streak is $2, 2, 3, 3, 4, 5$. Thus, the answer is 6.

One solution to this problem is to define a variable $\textsc{long}_i$ to be the length of the longest sub-sequence of prices that only increase or stay the same in height among the prices from $p_1, \ldots, p_i$ *that ends with price* p_i.

First observe that $\textsc{long}_1 = 1$. Next, note that if p_i is larger than p_j, then i can be added to the longest subsequence of prices that ended at j to form a longer subsequence. In other words, if $p_i \geq p_j$, then one can form a sequence of length $1 + \textsc{long}_j$ by considering ..., p_j, p_i where the ... represents the prices in the longest sequence that ends at price j. Thus we have

$$\textsc{long}_i = \max_{j=1}^{i-1} \begin{cases} 1 & \text{if } p_j > p_i \\ \textsc{long}_j + 1 & \text{if } p_j \leq p_i \end{cases} \tag{5}$$

Finally, the longest overall sub-sequence is simply $\max_{i=1}^{n}\{\textsc{long}_i\}$. Computing each value $\textsc{long}_i$ takes $O(n)$. There are n such values; thus, the overall running time is $\Theta(n^2)$ because the last step of finding the max takes $\Theta(n)$.

PRICES$(h_1, \ldots, h_n)$
1 $\textsc{long}_1 \leftarrow 1$
2 **for** $i = 2$ to n
3 $\quad \textsc{long}_i = \max_{j=1}^{i-1} \begin{cases} 1 & \text{if } p_j > p_i \\ \textsc{long}_j + 1 & \text{if } p_j \leq p_i \end{cases}$
4 Output $\max\{\textsc{long}_1, \ldots, \textsc{long}_n\}$

In this problem you will improve the solution to an $O(n \log n)$ time algorithm. Notice that line 3 uses a linear scan requiring $\Theta(i)$ steps to find the price that is smaller than p_i and currently has the largest length.

We can improve our solution by maintaining a slightly different DP variable that allows us to use binary search instead of linear scan for this step. The idea is to consider the variable $\textsc{fast}_i$ which maintains "the *index* of the *smallest* price that ends a sequence of length i that increases or stays the same." If no such value exists, we define it to be ∞.

(a) (1 pt) In a problem of size 1, what is $\textsc{fast}_1$?

(b) (5 pt) Assuming we have computed $\textsc{fast}_i$ for $i = 1, \ldots, j$, explain in words how to update the values given the next price p_{j+1}.

(c) (4 pt) (Why we maintain indicies?) Assuming we have computed this $\textsc{fast}_i$ variable for all $i = 1, \ldots, n$, and we know that the longest price run is j, how can we print the prices which correspond to this longest run?

(a) Since FAST_i maintains the *index* of the *smallest* price that ends a sequence of length i that increases or stays the same, for a problem size of 1, $\text{FAST}_1 = 1$.

(b) Given FAST_i for $i = 1, \ldots, k$, the values $p(\text{FAST}_i)$ for $i = 1, \ldots, k$ is in increasing order since $p(\text{FAST}_i)$ represents the smallest price that ends the sequence of length i that increases or stays the same.
Now, next price $p(j+1)$ can either reduce one of $p(\text{FAST}_i)$ for $i = 1, \ldots, k$ when there is a length i for which $p(\text{FAST}_i) > P(j+1)$ and update $\text{FAST}_i = j+1$ or increase the maximum length to $k+1$ if no such i exists and represent $\text{FAST}_{k+1} = j+1$.
We can binary search for for $i = 1, \ldots, k$ and find the length i such that $p(\text{FAST}_{i-1}) \leq p(j+1) < p(\text{FAST}_i)$, if such i exists, we can change $\text{FAST}_i = j+1$. If $p(j+1) \geq p(\text{FAST}_k)$, then it increase the maximum length of subsequence by 1 and hence, $\text{FAST}_{k+1} = j+1$.

(c) Given we have calculated FAST_i for $i = 1, \ldots, j$ for all prices $p(1, ..., n)$ and have j as the length of maximum length of a sequence that increases or stays the same and FAST_j representing the index of such sequence. Thus $p(\text{FAST}_j)$ for $i = n$ will be the last element in this sequence. Now we need to find second to last, which represents FAST_{j-1} calculated for $i = \text{FAST}_j - 1$. We can backtrack it till $j > 0$ and reverse the sequence to get the longest subsequence of prices that increases or remains the same.