— 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.

Solution: If $n < 4$, then fully connect the bus stops using at most 3 segments.

For larger n, divide the stops into $\lceil\sqrt{n}\rceil$ groups of size $\lceil\sqrt{n}\rceil$. (The last group may have fewer stops.) In each group, identify the last node in the group as the "hub" for the group. Add a route from every node in the group to its hub. (This takes a total of at most $\lceil\sqrt{n}\rceil \cdot (\sqrt{n}-1) \le n$ segments.) Now add a node from the hub of a group to every stop (excluding the hub) in the next group. (This again takes $\lceil\sqrt{n}\rceil \cdot (\sqrt{n}-1) \le n$ segments.) Connect each of the $\sqrt{n}$ hubs to each other. In particular, connect the first hub to the other $\sqrt{n}-1$ other hubs. Connect the second hub to the $\sqrt{n}-2$ remaining hubs, etc. This take a total of $(\sqrt{n}-1)+(\sqrt{n}-2)+(\sqrt{n}-3)+\cdots+1 \leq n$ segments.

Finally, in each group of $\sqrt{n}$ nodes, recursively apply this construction. The construction is illustrated in Fig. 1.

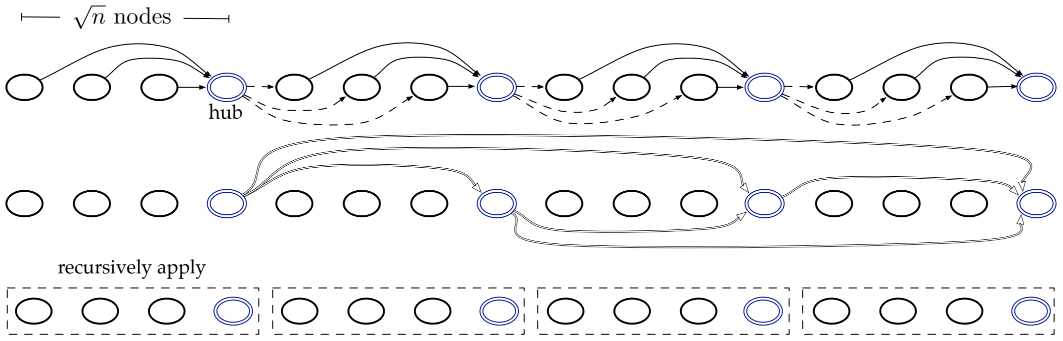

Figure 1: The top diagram illustrates how each hub is connected to the $\sqrt{n}$ nodes immediately to the left and right of the hub. The middle diagram shows the segments connecting the $\sqrt{n}$ hubs. The bottom diagram illustrates how the entire construction is recursively applied to each group of $\sqrt{n}$ stops.

Connectivity: Let us first argue that the connectivity property holds. Any commuter traveling from one stop to another in the same group of $\sqrt{n}$ stops uses the recursive solution. Suppose now that i and j are in different groups. Take a route from i to its hub, then from that hub to the hub before node j, then take a segment from that hub to j. At most three hops are needed.

(b) State a recurrence that counts the segments required by your route system.
Solution: The number of segments in our scheme is $T(n) < \sqrt{n}T(\sqrt{n}) + 3n$.

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

Solution: We show the upper bound using the guess and check method. The lower bound follows similar reasoning. Suppose that $T(n) < 4n \log\log n$ for small n. For larger n, we have that

$$\begin{aligned}
T(n) &< \sqrt{n}T(\sqrt{n}) + 3n \\
&< \sqrt{n}\left[4\sqrt{n}\log\log\sqrt{n}\right] + 3n \\
&= 4n\left[\log(1/2\log n)\right] + 3n \\
&= 4n\left[\log(1/2) + \log\log n\right] + 3n \\
&= 4n\log\log n - 4n + 3n \\
&< 4n\log\log n
\end{aligned}$$

which completes the upper-bound.

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.

Solution: The best streak must end at some day j. Define BEST_j to hold the value of the best streak that ends at j: it will either be the streak that starts and ends on day j and thus has a value of a_j, or it will be a continuation of the best streak that ends at $j - 1$.

Observations: Rather giving what literally

$$\text{BEST}_n$$

means, some students have just explained the DP equation by saying something like, "we will add a_i to $best_{i-1}$ or just take a_i" which does not explain why.

(b) Provide an equation that defines this variable.

Solution:

$$\text{BEST}_j = \max \begin{cases} \text{BEST}_{j-1} + a_j \\ a_j \end{cases}$$

We noticed that some students did not give a DP equation that can be used for all $i \leq n$. Some students are thinking it to be of last day n and taking an extra max on all Best computed to assign it to $Best_n$, furthermore I have seen some students comparing all j for all $0 \leq j \leq i$ which causes unnecessary comparisons.

(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).

Solution:

```
1  $\text{BEST}_1 = a_1$
2  for $i = 2$ to $n$
3      $\text{BEST}_i = \max\{\text{BEST}_{i-1} + a_i,\ a_i\}$
4      $\text{CHOICE}_i = \text{argmax}\{\text{BEST}_{i-1} + a_i,\ a_i\}$
5  $s \leftarrow \max\{\text{BEST}_1, \text{BEST}_2, \ldots, \text{BEST}_n\}$
```

After computing all $\textsc{best}_j$ for $j = 1, \ldots, n$, find the largest one to compute the j, s component of the answer. One can now use backtracking to determine the i (or conversely, save i during the computation of $\textsc{best}_j$.) Notice if all values are negative, then the best streak consists of the largest value.

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.

Solution: This problem is similar in pattern to person-counting from L1 and Greeks from the practice midterm. As a base case, if there is one sample, then it is a majority element.

Pair each of the samples (like we did in class when we counted the number of people) and run the Compare test. Discard all pairs that report "different." From each of the remaining pairs, we choose one representative (either one) and discard the other.

It is important to handle the extra sample in the case that n is odd. Always discarding that extra sample does not work. For example, when $n = 5$, and the matching is AA, BB, A, then discarding the extra sample results in the set A, B for which there is no majority. On the other hand, always keeping the extra sample also does not work. For example, when $n = 3$, then if we test AA, B, the result would again be A, B. This leads us to the following approach:

```
PAIR($I$)
1  Initialize set $S$ to the empty set
2  Pair the input samples in $I$ arbitrarily and run the Compare test on them
3  Discard any pair that does not respond "Same"
4  For every pair that responds "Same", add one of the two to the set $S$.
5  At the end, if $|S|$ is even and one untested sample remains, add the untested one to $S$
6  Output set $S$
```

```
MAJORITY($I = \{s_1, \ldots, s_n\}$)
1 Set $S = I$.
2 while $|S| > 1$
3   $S \leftarrow$ PAIR($S$)
4 if $|S| = 1$
5   Compare the element in $S$ with each element in $I$.
6   If more than $|I|/2$ are equal, output “Majority”
7 Otherwise, output "No majority"
```

First, this algorithm has no false positives, i.e., it never incorrectly outputs “Majority” because Majority only outputs “Majority” in line 5 after testing a sample against all of the inputs and verifying it is the same as a majority of the input.

The question is, can it have a false negative and output “No majority" even if a majority exists? To answer this question, we show that if a majority variant exists in I, then the same majority element must also exist in S.

Claim 1 *If the input I has a majority variant, the set S at the end of one iteration of* Pair *has a majority variant.*

Proof. Let A stand for the majority variant and let a be the number of majority variants in the input, and let b be the number of non-majority variants (some of which can be the same) such that $n = a + b$. By definition of majority, it follows that $a > b$, or equivalently, $a \geq b + 1$.

The first step is to randomly pair the samples. Some of these random pairs will contain different variants and both will be discarded. Without loss of generality, lets assume that all of these “different" pairs are compared first. After both such samples are discarded, the set of remaining uncompared samples still has majority A. This is because such a comparison pair either consists of one element of A and a different one in which case, the remaining samples are $a - 1 > b - 1$, or the sample consists of elements of the non-majority variant, and $a > b - 2$.

After this point, all of the remaining comparisons occur between samples that are the same variant and one of each pair gets added to S. As before, let $a > b$ respectively represent the number of remaining majority and non-majority samples.

Suppose $a = 2k + 1$ is odd, and thus we have $b \leq 2k$ non-majority elements. After the A variants pair up, S contains k counts of majority variant and one unpaired A variant is leftover. In the case of strict inequality $2k > b$, then at most $k - 1$ non-majority samples are added to S, and A remains the majority in S. On the other hand, if $b = 2k$, then k non-majority variants are added to S, but at this point, $|S| = 2k$ is even, and thus the leftover sample of type A is added to S to create a majority of $k + 1$ samples of A in S.

The other case is when $a = 2k$ is even, and $b \leq 2k - 1 = 2(k - 1) + 1$. After pairing, S contains k samples of the majority variant A. In the case $b = 2k - 1$, then $k - 1$ non-majority variants are added to S, and the leftover is *always* discarded because the size of $|S| = k + (k - 1)$ will be odd. In the case of strict inequality,

$b < 2k-1$, then at most $k-1$ non-majority variants are added to S. In both cases, A remains the majority in S. □

Thus, if I begins with a majority, then each call to Pair returns a set S that also contains a majority. Observe, that the size of S decreases by half each time.

Claim 2 *After one call to* Pair, *the size of* $|S| \leq \lceil |I|/2 \rceil$.

There are at most $\lfloor n/2 \rfloor$ pairs that can respond "Same". From these pairs, exactly one of the two samples can be added to S. Finally, line 5 in Pair can possibly add one person to S if $|S|$ was otherwise even. Thus, $|S| < \lfloor n/2 \rfloor + 1 \leq \lceil n/2 \rceil$.

Thus, by induction, it follows that if a majority variant exists in I, then in line 4 in Majority the loop terminates with S containing only this element.

Claim 3 Majority *uses* $\Theta(n)$ *tests.*

First note that Pair uses exactly $\lfloor n/2 \rfloor$ tests. This follows because tests are only performed on the second line, and there are only $\lfloor n/2 \rfloor$ pairs. If $|I|$ is odd, then one sample remains untested.

Based on the simple recurrence that $T(n) = T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$, by the Master's theorem, it follows that $T(n) = \Theta(n)$. Furthermore, the last line of Majority uses n comparison tests. Overall, this results in $\Theta(n)$ calls to the Compare procedure.

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.

Solution: The solution to this problem is the same as the number of different ways to multiply a chain of matricies as given in the homework. Imagine a sequence of n matricies $A_1, \ldots, A_n$ that are parenthesized to indicate the order in which they are multiplied. Replace the left parenthesis with "BUY" and the right one with "SELL." This is a legitimate trading schedule since every block of shares is bought (left paren) before it is sold (right paren). Thus, the solution, as presented in class is

$$T(n) = \sum_{i=0}^{n-1} T(i)T(n-i-1)$$

with $T(1) = T(0) = 1$.

Another way to see this is to associate every SELL with the closest prior unassociated BUY operation. Now that the operations have been paired, identify the SELL operation associated with the first BUY.

$$B\langle \cdots \rangle S\langle \cdots \rangle$$

Suppose there are i BUY-SELL operations that occur between this first BUY and its associated SELL. In this case, there are $T(i)$ ways to order this middle sequence of i trades, followed by $T(n-i-1)$ ways to order the remaining trades in the suffix of the sequence. Thus, there are a total of $T(i)T(n-i-1)$ ways for the first BUY to be associated with a sell that appears after i intermediate trades. This gives us the recurrence stated above.

One has to be careful with the indexing of the summation to ensure that each trade sequence is counted only once. If instead, we use a method that results in the term $T(i)T(n-i)$, then the counting method double-counts certain sequences of trades.

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 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 $\text{LONG}_1 = 1$. 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 $1 + \text{LONG}_j$ by considering ..., p_j, p_i where the ... represents the prices in the longest sequence that ends at price j. Thus we have

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

Finally, the longest overall sub-sequence is simply $\max_{i=1}^{n}\{\text{LONG}_i\}$. Computing each value 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  $\text{LONG}_1 \leftarrow 1$
2  for $i = 2$ to $n$
3      $\text{LONG}_i = \max_{j=1}^{i-1} \begin{cases} 1 & \text{if } p_j > p_i \\ \text{LONG}_j + 1 & \text{if } p_j \leq p_i \end{cases}$
4  Output $\max\{\text{LONG}_1, \ldots, \text{LONG}_n\}$
```

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 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 FAST_1 ?

Solution: With an input of 1 element, the index of the smallest price that ends a sequence of length 1 is the index of the first element. With zero-indexing, that is $\text{FAST}_1 = 0$.

(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}.

Solution: Perform a binary search using price p_{j+1} on the array containing $\textsc{fast}[1 \ldots j]$, but instead of comparing p_{j+1} to $\textsc{fast}_i$, it is compared to $p_{\textsc{fast}_i}$. We use this search to return the largest number $k \leq j$ such that we have $\textsc{fast}_k \neq \infty$ and $p_{j+1} \geq p_{\textsc{fast}_k}$. If no such k exists, meaning that $p_{j+1} < p_{\text{FAST}_1}$, we just use $k = 0$. Then we have identified a price streak of length $k+1$, obtained by appending p_{j+1} to the price streak of length k that ends at $p_{\textsc{fast}_k}$ (or in the case where $k = 0$, just p_{j+1} itself). Now we have two cases:

- If $\textsc{fast}_{k+1}$ is ∞, then this new price streak is longer than what we previously have (length k). So we should just update $\textsc{fast}_{k+1}$ to be $j+1$.
- Otherwise, since k is the largest number such that $p_{j+1} \geq p_{\textsc{fast}_k}$, we have $p_{j+1} < p_{\textsc{fast}_{k+1}}$. The definition of $\textsc{fast}_{k+1}$ says that it should be the *index* of the *minimum* price that ends a price streak of length $k+1$. Since the one we have previously ($p_{\textsc{fast}_{k+1}}$) is larger than the one for our new price streak (p_{j+1}), we should also update $\textsc{fast}_{k+1}$ to be $j+1$.

Therefore we can just update $\textsc{fast}_{k+1}$ to be $j+1$ regardless of what it used to be.

Alternatively, we can use binary search to search for the *smallest* number k' with $p_{j+1} < p_{\textsc{fast}_{k'}}$ (set k' to be $j+1$ if no such k' exists) and then update $\textsc{fast}_{k'}$ to be $j+1$.

One has to be careful about what we are searching for in the binary search part: we want the largest k such that $p_{j+1} \geq p_{\textsc{fast}_k}$ instead of $p_{j+1} > p_{\textsc{fast}_k}$, and then we do update on $\textsc{fast}_{k+1}$, since our price streaks are sequences that increase or stay the same. In other words, they are non-decreasing sequences instead of increasing sequences.

(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?

Solution: We need to use another variable $\textsc{prev}_i$ to store the *index* of the second last element of the longest price streak that ends with price p_i. This variable can be calculated along with the process we described above in part (b): for price p_{j+1}, after finding the number k as in part (b), we update $\textsc{prev}_{j+1}$ to be $\textsc{fast}_k$ if $k \neq 0$, and -1 if $k = 0$. This is because when $k \neq 0$, $p_{\textsc{fast}_k}$ is the second last price of the price streak of length $k+1$ that we have identified as in part (b), and when $k = 0$, there is no second last element.

Now we can do backtracking to print the prices in the end, assuming that we know the longest price run has length j (which is indeed the largest number j such that $\textsc{fast}_j \neq \infty$ in the end):

```
Backtracking(prev_1, ..., prev_n, fast_1, ..., fast_n, j)
1 Initialize array $L[1..j]$
2 $k \leftarrow \text{fast}_j$
3 for $i = j$ downto 1
4     $L[i] \leftarrow p_k$
5     $k \leftarrow \text{prev}_k$
6 Output $L$
```

One cannot use fast_i directly to print the price run in the end, since $p_{\text{fast}_1}, p_{\text{fast}_2}, \ldots, p_{\text{fast}_j}$ is not necessarily a price streak of length j in $p_1, p_2, \ldots, p_n$. For example, for $n = 3$ and $p_1 = 2$, $p_2 = 3$, $p_3 = 1$, in the end we will have $\text{fast}_1 = 3$, $\text{fast}_2 = 2$, and $\text{fast}_3 = \infty$, but $p_{\text{fast}_1}, p_{\text{fast}_2} = 1, 3$ is not a price streak in $2, 3, 1$. However, recording indices instead of values in fast_i enables us to easily calculate prev_i and do backtracking as we have shown above.

Another common mistake is to find the price run greedily from p_1, p_2, ..., p_{fast_j}, by adding p_{fast_j} first and then scanning backwards from $p_{\text{fast}_j - 1}$ and add numbers to the solution if it is less than or equal to the last number added. As we have been using DP to solve this problem, there is no way that we can print the actual run greedily. Consider $2, 3, 1, 4$, the greedy way to backtrack will get stuck after getting $1, 4$.