Final Exam

Instructions:

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- Write your name below and circle your recitation at the bottom of this page.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages will be separated for grading.
- **You are allowed three one-sided, letter-sized sheets with your own notes**. No calculators or programmable devices are permitted. No cell phones or other communication devices are permitted.

Advice:

- You have 180 minutes to earn a maximum of 180 points. **Do not spend too much time on any single problem.** Read them all first, and attack them in the order that allows you to make the most progress.
- When writing an algorithm, a **clear** description in English will suffice. Using pseudo-code is not required.
- Do not waste time rederiving facts that we have studied. Simply state and cite them.

Problem	Parts	Points	Grade	Grader
0	2	2		
1	4	8		
2	4	8		
3	4	8		
4	4	12		
5	3	22		
6	1	16		
7	3	24		
8	2	20		
9	2	20		
10	3	20		
11	3	20		
Total		180		

Name: _____

	R01	R02	R03	R04	R05/R07	R06	R08	R09	R10
Circle your recitation:	Brando Miranda **10AM**	Brando Miranda **11AM**	Parker Zhao **12PM**	Alex Jaffe **12PM**	Danil Tyulmankov **1PM 2PM**	Peinan Chen **1PM**	Kevin Tian **2PM**	Allen Park **3PM**	Daniel Manesh **4PM**

Problem 0. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name and circle your recitation section.

(b) [1 point] Write your name on top of each page.

Problem 1. True or False (Asymptotics) [8 points] (4 parts)

Which of the following asymptotic relations are correct? *No justification is needed here.*

(a) **T** **F** If $f(n) = 4n^2 + 5n + 4$,
then $f(n) = O(n^3)$.

Solution: True. $f(n) = \Theta(n^2)$, which is also $O(n^3)$, $O(n^4)$, or any higher power of n.

(b) **T** **F** If $f_1(n) = 5n + \log n$ and $f_2(n) = 2n + 9\sqrt{n}$,
then $f_1(n) \cdot f_2(n) = \Theta(n^2)$.

Solution: True. $f_1(n) \cdot f_2(n) = 10n^2 + 45n\sqrt{n} + 2n \log n + 9\sqrt{n} \log n = \Theta(n^2)$.

(c) **T** **F** If $f_1(n) = \Theta(n)$ and $f_2(n) = \Theta(n)$,
then $\sqrt{f_1(n) \cdot f_2(n)} = \Theta(n)$.

Solution: True. $f_1(n) \cdot f_2(n) = \Theta(n) \cdot \Theta(n) = \Theta(n^2)$ and $\sqrt{f_1(n) \cdot f_2(n)} = \sqrt{\Theta(n^2)} = \Theta(n)$.

(d) **T** **F** If $f(n) = n + 1$,
then $n^{f(n)} = O(n^n)$.

Solution: False. $n^{f(n)} = n^{n+1} = n \cdot n^n = n \cdot \Theta(n^n)$.

Problem 2. True or False (Recurrences) [8 points] (4 parts)

Which of the following are correct solutions to the given recurrences? In all cases, you may ignore roundoffs. *No justification is needed here.*

(a) **T** **F** $T(1) = O(1), T(n) = 3T(n/3) + O(n^2)$.
Proposed solution: $T(n) = O(n^2)$

Solution: True. This is case 3 of the Master Theorem, in which the top level dominates.

(b) **T** **F** $T(1) = O(1), T(n) = 8T(n/2) + O(n^3)$.
Proposed solution: $T(n) = \Theta(n^3)$

Solution: False. This corresponds to case 2 of the Master Theorem, in which all levels contribute similarly, and so $T(n) = O(n^3 \log n)$, which is not necessarily $\Theta(n^3)$.

(c) **T** **F** $T(1) = O(1), T(n) = 9T(n/3) + O(n)$ for $n > 1$.
Proposed solution: $T(n) = \Theta(n^3)$

Solution: False. This corresponds to case 1 of the Master Theorem, in which the bottom level dominates, and so $T(n) = O(n^2)$.

(d) **T** **F** $T(1) = O(1), T(n) = 2T(n/3) + 3T(n/9) + O(n^3)$.
Proposed solution: $T(n) = \Theta(n^3 \log n)$

Solution: False. This case is dominated by the top level, and so $T(n) = O(n^3)$.

Problem 3. True or False (Max Heaps) [8 points] (4 parts)

Which of the following are true statements about every max heap H, when viewed as a tree? *No justification is needed here.*

(a) **T** **F** The tree satisfies the AVL balance property, that is, for every node u in H, the height of the left subtree of u and the height of the right subtree of u differ by at most one.

Solution: True. A max heap is stored as an array but viewed as a nearly complete binary tree. Only the bottom level of the tree may be incomplete, but even so the maximum height difference of any two subtrees of any node is one.

(b) **T** **F** If we add a new key into the tree with the INSERT operation, this key is initially added as a leaf of the tree before being moved to its appropriate location.

Solution: True. A new key is added as the $(n+1)$-st element of the tree (which is a leaf) and then is promoted ("trickled up") to its final location.

(c) **T** **F** To build a new heap of n elements requires $\Theta(n \log n)$ time.

Solution: False. Lecture 4 presented a divide-and-conquer approach to creating a heap from n elements that takes linear time, $\Theta(n)$.

(d) **T** **F** If we increase the value of the key at a single node u of H, then max-heapify restores the heap property in time $O(\log n)$ for a heap of n elements.

Solution: False. The key increase has the potential to create a violation of the max-heap property, which is repaired by promoting the key upward ("trickling up", similar to what is done in an INCREASE-KEY operation). This is quite different from the MAX-HEAPIFY, which "trickles down".

Problem 4. True or False (Gradient Descent and Newton's Method) [12 points] (4 parts)

Which of the following facts about gradient descent are true? *No justification is needed here.*

(a) T F In order to find the value of e^4, we can run Newton's method on the function $f(x) = \ln x - 4$ with an appropriate choice of starting point $x^{(0)}$.

Solution: True. Newton's method can be used to find the roots of a function. The roots of f are the values of x for which $f(x) = 0$. Setting

$$\begin{aligned} f(x) = 0 &= \ln x - 4 \\ \ln x &= 4 \\ x &= e^4 \end{aligned}$$

Thus, if we can find a value of x for which $f(x) = 0$ by running Newton's method from an appropriate starting value $x^{(0)}$, then that value of x will be equal to (or very close to) e^4.

(b) T F Newton's method is guaranteed to converge, from any starting point $x^{(0)}$.

Solution: False. Newton's method converges for starting points sufficiently close to the root of the function, but it is not guaranteed to converge from *every* starting point.

(c) T F The gradient descent algorithm applied to the function

$$f(x) = (x+1)^2(x-1)(x-2) = x^4 - x^3 - 3x^2 + x + 2$$

is guaranteed to converge to a global minimum of that function.

Solution: False. Note that $f(x)$ is a degree-4 polynomial and has at least one local maximum (between its roots $x_1 = -1$ and $x_2 = 1$, to be precise). So, this maximum is a critical point and if the gradient descent algorithm started from that point it would stay there (and thus not converge to any minimum).

Alternatively, note that the product representation of the polynomial $f(x)$ tells us that it has a double root at $x_1 = -1$, and two roots at $x_2 = 1$ and $x_3 = 2$. As the leading coefficient is 1, x_1 is a local minimum with $f(x_1) = 0$ and the other minimum has to be between roots x_2 and x_3 and be strictly negative. So, x_1 is another critical point that is not a global minimum.

As a result, convergence to the global minimum can only happen for specific starting points and step sizes.

(d) T F Every function that has a single global minimum is convex.

Solution: False. A function can have multiple minima, only one of which is global. In this case it has a single global minimum but is not convex. One example of such function is the function f from part (c).

Problem 5. Sorting Prefixes [22 points] (3 parts)

Suppose that we are given a sequence $A[1 \ldots n]$ of n distinct elements that are *k-prefix unsorted*, for some integer $0 \leq k \leq \frac{n}{2}$. That is, the length-$s$ suffix of A, where $s = n - k$, is already sorted in non-decreasing order. (The elements of the length-k prefix can be arbitrary and in arbitrary order.)

For example, $A = [3, 1, 5, 2, 4, 6]$ is a 3-prefix unsorted sequence of length 6.

(a) [6 points] Provide an asymptotic estimate of the *worst-case* bound (in terms of n and k) on the number of *comparisons* that Insertion Sort makes on such k-prefix unsorted sequences.

Solution:

Observe that we can divide sorting of a k-prefix unsorted sequence via Insertion Sort into two phases.

In the first phase, the length-k prefix is sorted and, as the elements in this prefix can appear in arbitrary order, we know from class that this will take $\Theta(k^2)$ comparisons in the worst case. (Recall that the worst case here corresponds, e.g., to having all these k elements sorted in a non-increasing order.)

In the second phase, the length-s suffix is gradually inserted into the already sorted prefix, with each element being trickled forward to its correct place via a compare-and-swap operation. Observe that as the suffix was sorted in non-decreasing order to begin with, no element of the suffix will swap with any other element of the suffix during such insertions. So, each element of the suffix will need at most k compare-and-swaps before reaching its correct place. In the worst-case, i.e., if all the elements of the initial length-k prefix are larger than all the elements of the length-s suffix, this number of compare-and-swap operations will indeed be k each time, leading to $\Theta()s \cdot k)$ comparisons in this phase.

Overall, the worst-case total number of comparisons will be

$$\Theta(k^2) + \Theta(s \cdot k) = \Theta(k^2) + \Theta((n-k) \cdot k) = \Theta(k^2 + nk) = \Theta(n \cdot k),$$

where we used the fact that $k \leq \frac{n}{2}$. An example of a k-prefix unsorted sequence that triggers this worst-case behavior is

$$A = [n, n-1, \ldots, n-k+1, 1, 2, \ldots, n-k].$$

(b) [8 points] Describe a comparison-based algorithm for sorting any k-prefix unsorted sequence that is efficient with respect to the number of comparisons it makes. (In other words, we care only about the number of comparisons and not the time efficiency here.) Provide an asymptotic worst-case estimate of the number of *comparisons* made by your algorithm. (You can assume that you know the value of k.)

Solution: Consider an algorithm in which each element in the prefix is picked up and inserted into the already sorted suffix via binary search. There is a total of k of such insertions and each one of them corresponds to performing a binary search on the suffix of length s'. (Initially, $s' = s = n - k$ but it grows by one each time we insert a new element.)

Clearly, once this algorithm terminates we have that $s' = n$, i.e., the whole sequence A is sorted. Also, each of the k insertions requires making at most $O(\log s') = O(\log n)$ comparisons. This gives a worst-case bound of the number of comparisons to be $O(k \log n)$.

(c) [8 points] Give an asymptotic lower bound (in terms of n and k) on the number of comparisons needed by *any* comparison-based sorting algorithm to sort any k-prefix unsorted sequence A. Your lower bound here in part (c) and the one you obtain in part (b) should match. Briefly justify your result.
Hint: Recall that if $k \leq \frac{n}{2}$ *then* $\binom{n}{k} = \Theta(n^k)$.

Solution:

We use a decision-tree analysis to estimate the number of required comparisons to compute the solution: a non-decreasingly sorted sequence. The leaves of this decision tree represent all possible solutions to a problem, i.e., all possible permutations of the input sequence that can arise as a correct output. Intermediate nodes of this tree correspond to comparisons made. Comparisons produce a binary decision, and so the tree is binary.

To lower bound the number of leaves in the tree, we need to compute the number of permutations that transform *some* k-prefix unsorted sequence into a non-decreasingly sorted sequence. By symmetry, this number is equal to the number of ways we can pick a subset of k elements from an already non-decreasingly sorted sequence and put them as a prefix in some arbitrary order. Note that there are exactly $\binom{n}{k}$ ways to pick these subsets and exactly $k!$ ways to order the k elements pick. As a result, the total number of such permutations is exactly

$$\left[\binom{n}{k} \cdot k!\right].$$

Now, each correct sorting algorithm for k-prefix unsorted sequences has to correspond to one such binary decision tree. Also, we know that the height of such a tree corresponds to the number of comparisons this algorithm makes in the worst case. Furthermore, as this tree is binary, its height has to be lower bounded by the logarithm of the number of its leaves. So, the number of comparisons needed by *any* algorithm in the worst case is at least

$$\begin{aligned}\log\left(\binom{n}{k} \cdot k!\right) &= \log\binom{n}{k} + \log k! \\ &= \log\Theta(n^k) + \Theta(k\log k) = \Theta(k\log n + k\log k) = \Theta(k\log n),\end{aligned}$$

which asymptotically matches the bound we obtained in part (b).

Problem 6. Currency Market Disequilibrium [16 points] (1 part)

Due to market inefficiencies, sometimes there is a short-term possibility for large financial gain in currency exchange markets. For example, imagine the following exchange rates:

Currency Conversion			**Exchange Rate** (1=dollar, 2=yen, 3=euro)
100 yen	=	1 dollar	$R_{1,2} = 100$ yen/dollar
1 euro	=	90 yen	$R_{2,3} = 0.0111111$ euros/yen
9 dollars	=	8 euros	$R_{3,1} = 1.125$ dollars/euro

Starting with 100 dollars, one could purchase 10,000 yen, use all of those yen to purchase euros, and all of the euros in turn to re-purchase dollars. The overall cash at the end of this purchase cycle would be $\$100 \cdot R_{1,2} \cdot R_{2,3} \cdot R_{3,1} = \$125$ (that is, a net profit of 25%; and the cycle can be followed repeatedly for increased profits).

Let R be an $n \times n$ matrix representing the conversion rates between all n currencies, with the element R_{i_1,i_2} being the currency conversion rate from currency i_1 to currency i_2.

Alyssa P. Hacker would like to write an algorithm to monitor the exchange rates represented by the matrix R to detect such profit cycles whenever they occur.

To help her, describe an $O(n^3)$ algorithm to determine whether or not there is a sequence of currency trades that produces a net profit (that is, detects whether there exists a situation in which $R_{i_1,i_2} \cdot R_{i_2,i_3} \cdot \ldots \cdot R_{i_{m-1},i_m} \cdot R_{i_m,i_1} > 1$, for some currencies $i_1, \ldots, i_m$). You don't need to identify the currencies used by this sequence.

Don't forget to provide a running time analysis of your algorithm.

Hint: $\log(a \cdot b) = \log a + \log b$.

Solution: Consider constructing a weighted, directed graph $G(V, E)$ whose vertices are the currencies $1, 2, \ldots, n$. In this graph, for each pair of currencies (i, j), we have an edge (i, j) and another edge (j, i), whose weights are the *(negative)* logarithms of the corresponding exchange rates. That is, $w_{i,j} = -\log R_{i,j}$ and $w_{j,i} = -\log R_{j,i}$, for each pair of currencies (i, j).

Observe that in this setup we have that, for any cycle $i_1, \ldots, i_m$ in G, its total weight is exactly

$$-\log R_{i_1,i_2} - \log R_{i_2,i_3} - \ldots - \log R_{i_{m-1},i_m} - \log R_{i_m,i_1} = -\log \left(R_{i_1,i_2} \cdot R_{i_2,i_3} \cdot \ldots \cdot R_{i_{m-1},i_m} \cdot R_{i_m,i_i} \right).$$

Consequently, the cycle $i_1, i_2, \ldots, i_m, i_1$ corresponds to a sequence of profitable currency trades if and only if its weight is *negative*. (Note that if we had taken a more natural approach and made the weight of each edge correspond to logarithms of the corresponding exchange rates, instead of their negation, then a cycle would be profitable if and only if its weight were positive. This condition, however, would be harder to check with the algorithms we learned in class.)

Our task, therefore, boils down to checking if there exists a negative-weight cycle in the graph G that contains the vertex 1. As we know from the class, this can be detected by running the Bellman-Ford algorithm. This algorithm runs in time $O(V \cdot E)$, which is $O(n^3)$, as our number of vertices is n and the number of edges is $\frac{n(n-1)}{2} = \Theta(n^2)$ here.

Problem 7. Ben Hashes Again [24 points] (3 parts)

Ben Bitdiddle became a hashing aficionado. His favorite collision resolution scheme is open addressing and he is quite fond of the linear probing technique, that is, using a hash function h_L in which the i-th probe in hashing a key k is given by

$$h_L(k, i) = h(k) + i \pmod{m},$$

where $h(k)$ is a "classic" hash function and m is the size of the hash table.

Still, being the 6.006 expert he is, he came up with a new open addressing hash function h_E that he called "exponential hashing", in which the i-th probe is given by

$$h_E(k, i) = h(k) + (2^i - 1) \pmod{m}.$$

He wants now to demonstrate the superiority of this new hash function over the linear probing hash function.

(a) [8 points] Consider a situation in which the function h is defined as

$$h(k) = k \pmod{m},$$

and two ordered sequences

- $0, 1, \ldots, n/2 - 1$; and
- $m, m + 1, \ldots, m + n/2 - 1$,

were inserted into the initially empty hash table one after another. Here, $n = 2^t$, for some $t \geq 0$, and $n \leq m/4$.

Provide an *asymptotic* estimate (i.e., a Θ-estimate) of the number of probes P_L needed to perform all of these insertions using the hash function h_L. Show the work that led to this estimate.

Solution:

Observe that as the hash table is initially empty and $n \leq m/4$, the insertion of the first sequence of keys will immediately find an empty cell. So, these insertions will take $\Theta(n)$ probes overall.

After this first sequence of insertions, our hash table has all its first $n/2$ cells occupied. This means that when we start inserting the keys from the second sequence, the key $m + i$, for $0 \leq i \leq n/2 - 1$ will initially probe the now occupied cell i and then probe the next $n/2$ cells until finally finding an empty one. (Note that we also take into account here the newly occupied cell corresponding to insertions of earlier elements from the second sequence.)

So, the total number of probes P_L needed for inserting both sequences is

$$P_L = \frac{n}{2} + \frac{n}{2} \cdot \left(\frac{n}{2} + 1\right) = \Theta(n^2).$$

(b) [8 points] Provide an *asymptotic* estimate (i.e., a Θ-estimate) of the number of probes P_E needs to perform all the insertions described in part (a) using the hash function h_E. Show the work that led to this estimate.

Solution:

Similarly to part (a), insertion of the first sequence of keys requires $\Theta(n)$ probes overall and fills the first $n/2$ cells of the hash table.

Now, when we start inserting the keys from the second sequence, say the key $m+i$, for $0 \le i \le n/2 - 1$, then the initial probe will query the occupied cell i, and so will the next $t-1$ probes, until the last $(t+1)$-th probe finds an empty cell $i+2^t-1 = m+n-1$. (Note that as $n \le m/4$, this probe "fits" inside the hash table and also the insertion of the previous elements from the second sequence could not have taken this cell.)

So, the total number of probes P_E needed for inserting both sequences is

$$P_E = \frac{n}{2} + \frac{n}{2} \cdot (t+1) = \Theta(n \cdot t) = \Theta(n \log n).$$

(c) [8 points] Assume that the function h satisfies the simple uniform hashing assumption (SUHA).

Devise an initial configuration of the occupied slots of the hash table such that the expected number P'_L of probes needed to insert a new key k into that hash table when using the hash function h_L is *asymptotically smaller* than the expected number P'_E of probes needed to insert the key k into the hash table using the hash function h_E.

Remember to provide both a precise description of the initial configuration and asymptotic estimates of P'_L and P'_E. Show the work that led to these estimates.

Solution:

Consider a configuration of the hash table in which the only empty cells are the cells i such that either $i > \lfloor \frac{m}{2} \rfloor$ or $i = 2 + 3 \cdot j$, for $j \geq 0$. In other words, we keep unoccupied the cells in the second half of the hash table and every third one in the first half.

Now, as the has function h satisfies SUHA, we know that it maps the key k to a uniformly random cell of the hash table.

Let us focus on bounding P'_L first. Clearly, no matter where the initial probe lands, it will take at most three probes until the linear search performed by the hash function h_L finds one of the unoccupied cells. So, we have that

$$P'_L = \Theta(1).$$

Let us now try to estimate P'_E. Observe that no matter where the initial probe lands, after at most $O(\log m)$ probes one of them will reach the empty half and stop. So, we can conclude that

$$P'_E = O(\log m).$$

To see that this upper bound is actually asymptotically tight, note that with constant probability (at least $\frac{1}{12}$ to be more precise), our initial probe lands at one of the cells of the form $3 \cdot j$, with $0 \leq j \leq \frac{1}{3}(\lceil \frac{m}{4} \rceil)$. That is, it lands in the first quarter of our hash table and on the cell of the form $3 \cdot j$.

Note that all such cells are occupied (as are the cells immediately next to it). In fact, we have that in this case the i-th probe is given by

$$h_E(k, i) = h(k) + (2^i - 1) \pmod m = 3 \cdot j + (2^i - 1) \pmod m \leq \lceil \frac{m}{4} \rceil + 2^i,$$

as long as $2^i \leq m - \lceil \frac{m}{4} \rceil - 1$. However, in our configuration, all these slots are occupied. So, as there are $\Omega(\log m)$ values of i for which this happens, we have that

$$P'_E \geq \frac{1}{12} \cdot \Omega(\log m) = \Omega(\log m).$$

And thus indeed

$$P'_E = \Theta(\log m)$$

and P'_E is asymptotically larger than P'_L.

Problem 8. Stock Gain Problem with Dynamic Programming [20 points] (2 parts)

It turns out that the stock gain problem from Lecture 1 can be solved with dynamic programming. In fact, one can use this technique to solve the following, more general, variant of that problem in which we have an ability to execute multiple trades of the same stock, but are constrained to *at most one trade of one share of stock per day*.

Specifically, let $A[1..n]$ be the daily prices of the stock. Our goal is to make as much gain as possible by executing a sequence of buy-and-sell transactions, with each transaction happening on a different day. In other words, we want to find a sequence $(b_1, s_1), \ldots, (b_t, s_t)$ of t buy-and-sell transactions, for some $0 \leq t \leq n/2$, with $b_i < s_i$ being the buy and sell days for the i-th trade and all b_is and s_is being distinct, that maximizes our corresponding gain defined as

$$\sum_{i=1}^{t} (A[s_i] - A[b_i]) .$$

For example, if the daily prices were $A = [1, 10, 5, 2, 8, 4]$ then the maximum gain achievable would be 15 and it would correspond to executing trades $b_1 = 1$, $s_1 = 2$, $b_2 = 4$, and $s_2 = 5$.

On the other hand, if the daily prices were $A = [8, 4, 5, 7, 9]$ then the maximum gain achievable would be 7 and it would correspond to executing trades $b_1 = 2$, $s_1 = 5$, $b_2 = 3$, and $s_2 = 4$. (Note that in this example the trades are nested.)

(a) [10 points] Let $V[i, j]$, for $1 \leq i < j \leq n$, be the maximum gain achievable by executing multiple *complete* buy/sell trades between day i and day j (inclusive).

State a valid recurrence for $V[i, j]$ that can be used to compute $V[i, j]$ with a dynamic programing approach. Analyze the running time of the resulting algorithm.

Hint: Note that without loss of generality we can assume that in the optimal solution for different buy-and-sell transactions do not "cross". That is, we can choose labeling so it can never be the case that $b_i < b_{i'} < s_i < s_{i'}$, for any $i \neq i'$.

(In the second example above, an equivalent solution with "crossed" transaction is $b_1 = 2$, $s_1 = 4$, $b_2 = 3$, and $s_2 = 5$. We will always choose the equivalent "uncrossed" solution for convenience.)

Solution:

Let us consider a subproblem $V[i, j]$. Clearly, if $|i - j| < 2$, then $V[i, j] = 0$, as there is no valid buy-and-sell transaction that we can execute in such a short period of time. (Also, for convenience, we include here the case of $j = i - 1$ here.) Now, to derive the recurrence for the cases when $|i - j| \geq 2$, let us fix some sequence of buy-and-sell transactions between days i and j that results in the maximum possible gain.

Observe that there are two possibilities: either in this sequence there is no sell transaction on the last day j, or there is one. In the former case, clearly, our maximum possible gain can be as well achieved by looking at the period between days i and $j - 1$, so we can look at this smaller subproblem. In the latter case, if there is indeed a sell transaction that occurred on the last day j then there has to be some day i^*, with $i \leq i^* < j$, on which we bought the stock we are selling on day j. Note that by

the no-crossing property of optimal sequences of transaction, it must be the case that: all the stock bought before day i^* has to be also sold before that day; and any stock bought after day i^* must be sold before day j. So, our maximum gain in this case is equal to

$$A[j] - A[i^*] + V[i, i^* - 1] + V[i^* + 1, j - 1],$$

where the first difference corresponds to the gain from the sell operation on day j.

Of course, a priori, we do not know whether the optimal sequence of operations sells a stock on day j and, if so, what the corresponding buy day i^* is. Still, in our recurrence we can just explore all these possibilities. Consequently, our recurrence becomes:

$$V[i,j] = \begin{cases} 0 & \text{if } |i-j| < 2, \\ \max\{V[i,j-1], \max\limits_{i \le i^* < j} (A[j] - A[i^*] + V[i, i^* - 1] + V[i^* + 1, j - 1])\} & \text{otherwise.} \end{cases}$$

Note that our final answer is simply $V[1, n]$.

To analyze the running time of the resulting dynamic programing algorithm, note that we have $\Theta(n^2)$ subproblems, corresponding to all the valid choices of i and j. Also, solving a subproblem $V[i, j]$ (provided all the subproblems this subproblem depends on are already computed) using our recurrence takes $O(|j - i|) = O(n)$ time. (Note that, in principle, our $O(n)$ bound might be loose sometime, but for most of the subproblems $|j - i| = \Omega(n)$, so our bound is asymptotically tight after all.)

As a result the total running time of our algorithm is $O(n^3)$.

(b) [10 points] Now consider a modification of the problem in which you are allowed to own no more than k copies of the stock at any time. Describe a modification of the above dynamic programing approach to tackle this new version of the problem. Don't forget to analyze the running time of the resulting algorithm.

Solution:

Let us modify our solution from part (a) and define our subproblems to be $V[i, j, \ell]$ and denote the maximum gain we are able to achieve using a sequence of buy-and-sell transactions that are fully executed between days i and j and never make us hold more than ℓ copies of the stock in hand.

Clearly, again, if $|i - j| < 2$, then $V[i, j, \ell] = 0$. Similarly, if $\ell = 0$, $V[i, j, \ell] = 0$, too. Now, to derive the recurrence for the cases when $|i - j| \geq 2$ and $\ell > 0$, we proceed exactly as in part (a), except now, if in the case of selling the stock on the last day j and buying it on some earlier day i^*, we know that our maximum possible gain will be

$$A[j] - A[i^*] + V[i, i^* - 1, \ell] + V[i^* + 1, j - 1, \ell - 1],$$

where we take into account the fact that between days i^* and j we are already keeping one extra copy of the stock in hand, so our transactions between these two days need to have a capacity constraint that is smaller by one.

The resulting updated recurrence becomes

$$V[i, j, \ell] = \begin{cases} 0 & \text{if } |i - j| < 2 \text{ or } \ell = 0, \\ \max\{V[i, j - 1, \ell], & \\ \max\limits_{i \leq i^* < j} (A[j] - A[i^*] + V[i, i^* - 1, \ell] + V[i^* + 1, j - 1, \ell - 1])\} & \text{otherwise.} \end{cases}$$

Note that our final answer will be $V[1, n, k]$.

To analyze the running time, note that this time we have $\Theta(n^2 k)$ subproblems, as we have the additional parametrization by k. Also, solving each subproblem $V[i, j, \ell]$ using our recurrence takes $O(|j - i|) = O(n)$ time. So, our overall running time becomes $O(n^3 k)$.

Problem 9. Cookie Quality Control [20 points] (2 parts)

To support the increasing need for high-volume, high-quality, fast-response-time cookie deliveries, Prof. Madry has established the 6.006 bakery in the Stata Center. Excellence, integrity, and above all asymptotic efficiency are the hallmarks of the new facility.

To ensure cookie perfection, a special, highly sensitive testing apparatus has been installed that analyzes cookies by weight two-at-a-time and determines which is heavier. (It is so sensitive that no two cookies weigh exactly the same. We assume here that each use of the apparatus takes constant time.)

Prof. Madry finds that he spends more and more time at the bakery, and that it and the testing apparatus are an inspiration for many advances in algorithm design. For instance, he was able to develop an $O(n)$ (linear time) algorithm that finds from among a set of n cookies, one that is both heavier than at least a quarter of the cookies and lighter than at least (another) quarter. (Note that this algorithm returns only the cookie and *not* these two quarters.)

(a) [10 points] Using the "quartering" algorithm from above, design an algorithm that identifies the k-th heaviest cookie in a set of n cookies. Give and solve the recurrence that describes the running time of your algorithm.

Solution: Let us apply a divide-and-conquer approach. In each iteration, given $n \geq 4$ cookies, we use the quartering algorithm to identify, in $O(n)$ time, a reference cookie that is in the "middle half" of the weight ranking. (If $n \leq 4$ then we can just "brute force" the base case by sorting all four cookies by weight using the apparatus. This will take $O(1)$ time.)

Next, we use the apparatus to compare the weight of the reference cookie against the weight of each other cookie in the set. (This uses $n-1$ comparisons and thus can be done in $O(n)$ time.) Consequently, we partition these $n-1$ cookies into a "heavier" and "lighter" subsets of size h and ℓ, respectively.

At this point, we have three groups in total: the "heavy" set of h cookies; the reference cookie; and the "light" set of ℓ cookies. Observe that $h+1+\ell = n$ and also, by the definition of the quartering algorithm, $\max\{h, \ell\} \leq \frac{3}{4}n$.

Now, if $k \leq h$, then we recurse on the heavier subset, and if $k = h+1$, we can return the reference cookie as it has to be exactly the k-th heaviest cookie. Finally, if $k > h+1$ then we recurse on the lighter subset with a new value of $k' = k-(h+1)$. This algorithm will eventually return the k-th heaviest cookie.

To bound the overall running time $T(n)$ of the resulting algorithm, note that $T(n)$ obeys the following recurrence

$$T(n) \leq T(\max\{h, \ell\}) + O(n) \leq T\left(\frac{3n}{4}\right) + O(n).$$

By case 3 of the Master Theorem, the solution to this recurrence is $O(n)$. So, our algorithm runs in linear time (which is the best possible here).

(b) [10 points] If one did not have the quartering algorithm, one could solve the problem of finding the k-th heaviest cookie by using a comparison-based sorting method, e.g., Merge sort, which would run in $\Theta(n \log n)$ time. This seems to be doing too much work, as it solves the problem more thoroughly than needed. (It sorts all the cookies, and at worst we only need the top k.)

Design an alternative approach to finding the k-th heaviest cookie that does *not* use the quartering algorithm and outperforms the $\Theta(n \log n)$ sorting bound, for sufficiently small k. Your algorithm should run in $O(n + k \log n)$ time.

Solution:

Note that our apparatus enables us to implement any comparison-based algorithm for finding the k-th heaviest cookie. So, let us use heaps to help us with that.

Specifically, let us start by building a Max Heap that contains all the cookies, with their weight being their key values. As we know, this takes $O(n)$ time in total. (Recall that each comparison takes constant time.)

Now, we simply extract the maximum key element from the heap, i.e., remove the maximum-weight cookie, by calling EXTRACT-MAX, k times. The last cookie returned has to be exactly the k-th heaviest cookie we seek. The total time to perform these k operations is $O(k \log n)$.

This gives the overall time of $O(n + k \log n)$, as desired.

Problem 10. Winning a Game of Strategy [20 points] (3 parts)

Consider a one-player game in which there are r red chips and b blue chips on a table, with $r = R$ and $b = B$ initially. In each move, you must remove either one red chip or one blue chip. At the end of your move, you add to the running sum (which is initially zero) a number $f(r, b)$, where r and b are the number of red and blue chips, respectively, remaining on the table at the end of that move, and f is a certain function known to you in advance. The game continues until there are no chips remaining on the table. Your goal is to maximize the value of the running sum at the end of the game.

(a) [8 points] Describe an algorithm that computes, for a given function f and the initial number R and B of red and blue chips, the maximum value of the sum achievable. What is the running time of your algorithm?

Solution: We use a dynamic programming approach. We let $F(r, b)$ represent the maximum value of the sum that can be achieved starting from the position with r red chips and b blue chips remaining.

Now, we use the recurrence

$$F(r,b) = \begin{cases} 0 & \text{if } r, b = 0, \\ -\infty & \text{if } r < 0 \text{ or } b < 0, \\ \max\{F(r-1,b) + f(r-1,b), F(r,b-1) + f(r,b-1)\} & \text{otherwise.} \end{cases}$$

In the last case, the first term in the $\max$ function represents the maximum value achievable if a red chip is removed, and the second term if a blue chip is removed.

Note that our final solution will be $F(R, B)$.

The number of subproblems is $R \cdot B$. Also, in the recurrence, only constant work is done in solving each subproblem (providing the needed subproblems are already computed). Additionally, outputting the final answer takes only $O(1)$ time.

So, the overall running time of this algorithm is $O(R \cdot B)$.

(b) [4 points] A new rule has been added to the game, and now you lose automatically if the product of the number of red chips and blue chips remaining at the end of one of your moves is 6006. How would you modify your algorithm to correctly adapt to this new rule?

Solution: It suffices to replace all occurrences of $f(r, b)$ in the recurrence in our solution to part (a) with $f'(r, b)$ where

$$f'(r,b) = \begin{cases} -\infty & \text{if } r \cdot b = 6006, \\ f(r,b) & \text{otherwise.} \end{cases}$$

(c) [8 points] Let us discard the new rule from part (b) now and consider a two-player version of the game, in which you alternate turns with an adversary. On his or her turn, the adversary also removes either a red or a blue chip, and we add to the running sum the value of $f(r, b)$ corresponding to the number of red and blue chips remaining on the table after the move. The adversary, however, chooses his or her moves so as to make the *final* sum as *small* as possible.

Describe an algorithm that computes, for a given function f and the initial number R and B of red and blue chips, the largest final value that you can guarantee that you will at least be able to achieve in the two-player game against an adversary that plays optimally. What is the running time of this algorithm? Assume that you make the first move and that $R + B$ is even.

Solution: We use a very similar dynamic programming approach to the one from part (a). This time, the player chooses a move that will be optimal assuming that the adversary chooses a move that will minimize the final sum. The new recurrence becomes

$$F(r,b) = \begin{cases} 0 & \text{if } r, b = 0, \\ -\infty & \text{if } r < 0 \text{ or } b < 0, \\ \max\{best\_resp(r-1,b) + f(r-1,b), & \\ \quad best\_resp(r,b-1) + f(r,b-1)\} & \text{otherwise.} \end{cases}$$

In the last (and, again, the most interesting) case above, the max corresponds to deciding whether we should remove a red chip or a blue chip and $best\_resp(r, b)$ denotes the maximum value of the sum achievable if we start from the configuration (r, b) and the adversary moves first.

However, as the adversary moves optimally, it is clear how to express $best\_resp(r, b)$ in terms of our subproblems $F(r, b)$. Specifically, he/she has one of two choices – remove a red chip or a blue chip – and will choose the one that minimizes the maximum value of the sum that we can achieve. Specifically, we have that

$$best\_resp(r,b) = \min\{F(r-1,b) + f(r-1,b), F(r,b-1) + f(r,b-1)\},$$

where the first option corresponds to the adversary removing a red chip and the other one to removing a blue chip.

Plugging the above recurrence for $best\_resp(r, b)$ into our recurrence for $F(r, b)$, we get a recurrence that refers solely to subproblems $F(r, b)$, as desired. Also, the final answer is simply $F(R, B)$.

To bound the running time of this algorithm, note again that the number of subproblems is $R \cdot B$. Also, again, in each recurrence, only constant time work is involved in solving the new subproblem based on the existing ones. (We just need to evaluate a max of two options where each option is a min of two things.) So, the overall running time is $O(R \cdot B)$ again.

Problem 11. Retrieving the 6.006 Crown [20 points] (3 parts)

You embark on a quest to retrieve a legendary treasure, the 6.006 Crown, from a maze. The good news is that you already have a map of the maze, with the location of the treasure marked on it. The not-so-good news is that there is a guardian, a particularly nasty ogre, that patrols the maze—and you definitely don't want to encounter him before recovering the crown. (Once you have the crown in hand, its algorithmic powers will let you deal with the ogre handily.) You need to find a route through the maze that reaches the crown while making certain to avoid the ogre (if such a route exists).

The maze can be modeled as an *undirected* graph $G = (V, E)$, with vertices representing the entrance s, the location of the treasure t, and the starting location of the ogre g (which is distinct from s and from t), marked on it. You start at the entrance s and, in each time step, you can traverse one of the edges incident on the vertex you are currently at. Similarly, the ogre starts at the vertex g and, in each time step, can either traverse one of the incident edges or stay put.

(a) [8 points] Design an $O(V + E)$ time algorithm that given G either:

- returns an $s \rightsquigarrow t$ path P in G such that if you follow it you are guaranteed to avoid being intercepted by the ogre. That is, *no matter* what sequence of moves the ogre makes from g, he will not reach any vertex of the path (including the last vertex t) before or at the same time that you do;
- or (correctly) concludes that no such path P exists in G.

Solution:

Let P be a hypothetical $s \rightsquigarrow t$ path in G that we might use to get to the crown.

Observe that if the ogre is able to intercept us at some vertex v of the path P, then by following the $v \rightsquigarrow t$ portion of this path, he is able to reach the crown room t no later than we would if we just followed the path P without any interception. (Recall that the ogre moves at the same speed as us and can traverse here exactly the same edges that we can.)

So, the ogre is able to intercept you if and only if he is able to reach the crown room t no later than us. (The "if" direction here follows by the fact that if he is able to reach t no later than us then he can just intercept us at the vertex t.)

But to check if the ogre can reach t no later than us, we just need to run BFS twice in G (once from s and once from g) to compare the shortest-path distance from s to t and from g to t. If the former is smaller then the corresponding shortest $s \rightsquigarrow t$ path is the "safe" path P to return. Otherwise, no such "safe" path exists.

Clearly, this algorithm works in $O(V + E)$ time, as needed.

(b) [6 points] Imagine now that the ogre, due to his size, cannot pass through certain narrow corridors. That is, the graph G has some subset $\widehat{E}$ of edges marked as narrow and the ogre cannot traverse them (but you still can). Provide an $O(V + E)$ time algorithm that solves the task defined in part (a) while taking the existence of narrow edges into account.

Solution:

Once the narrow corridors exist, the assertion from the solution to part (a) that we can focus on estimating the ability of the ogre to reach the crown room t before us, is no longer valid.

Instead, let us compute for each vertex v the earliest time step g_v when the ogre can reach that vertex. To compute g_v, for all vertices v, we just need to run a BFS from the vertex g in the subgraph of G obtained after removing the narrow corridors $\widehat{E}$ from it. This takes $O(V + E)$ time.

Now, in order to avoid interception, we need to make sure that we never visit a vertex v at time step g_v or later. So, to check if such a "safe" $s \rightsquigarrow t$ path that never visits any vertex v at or after time g_v exists in G, we need to run a modified version of BFS from the vertex s.

In this modified version, whenever we discover a vertex v at level l of the BFS tree we construct and $l \geq g_v$ then we immediately backtrack and remove this vertex and all the edges incident to it from our graph, and then continue the algorithm. (Note that if we discover a vertex v at the l-th level of the tree created by this modified BFS procedure it means that there is no "safe" $s \rightsquigarrow v$ path of length smaller than l in G.)

Clearly, this modified version of BFS still runs in $O(V + E)$ time. Also, if it finds an $s \rightsquigarrow t$ path then we can return this path as our safe path P. Otherwise, we report that no safe path exists.

(c) [6 points] Suppose next that, thanks to your mastery of the 6.006 material, you possess the power to teleport yourself to an adjacent room, and thus traverse the corresponding edge instantaneously.

Clearly, as long as there exists a path from s to t that avoids the ogre's starting room g, by using this teleportation power sufficiently many times, you can always reach the 6.006 crown while avoiding interception by the ogre. (We assume here that the ogre has good enough reflexes that it is impossible to "teleport through him".) However, your 6.006 instructors taught you that such power should be used only as often as is absolutely necessary.

Design an $O(V + E)$ algorithm that computes the minimal number ℓ^* of uses of that power needed to evade the ogre. (You should ignore here the existence of narrow corridors introduced in part (b).)

Solution:

Observe that if we need ℓ uses of the teleportation power then we might as well make all ℓ of these uses occur at time step 0. Furthermore, this initial use of ℓ teleportations enables us to effectively move our starting position to any vertex v that is within distance of ℓ of s in graph G *with the vertex* g *removed*. (We need to remove g to ensure that we do not "teleport through" the ogre.) Let us denote by S_ℓ the set of all such possible starting vertices v.

In light of the comments above, our task is to determine the smallest value of ℓ for which the set S_ℓ contains a vertex s' such that if we started from it (and did not use our teleportation power anymore) we were guaranteed to reach the crown room t without being intercepted by the ogre.

Note that as there are no more narrow corridors, the analysis we made in part (a) allows us to conclude that we are able to avoid interception when starting from a vertex s' if and only if the distance in G of s' to t is smaller than the distance D of the vertex g to t.

Now, to compute that minimal value of ℓ it suffices to:

- Compute the distances ℓ_v from the vertex s to all the vertices v in the graph G with the vertex g removed;
- compute all the distances d_v from all vertices v to the vertex t in the graph G; and then
- return $\ell^* = \min_{s' :\, d_{s'} < D} \ell_{s'}$. (Note that if $\ell^* = +\infty$ then it means there is no $s \rightsquigarrow t$ paths in G that do not pass through g.)

It is not hard to see that computing all ℓ_vs, all d_vs, and the $g \rightsquigarrow t$ distance D can be done with three runs of the BFS algorithm on appropriate versions of the graph G. And the final minimization can be executed in $O(V)$ time. So, the whole algorithm runs in $O(V + E)$ time, as desired.