CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Midterm Exam #1 Solutions

Problem 1.

(a) Determine the *strongest* asymptotic relation between the functions

$$f(n) = \sqrt{\log n} \quad \text{and} \quad g(n) = \frac{n}{2^{(\log\log n)}},$$

i.e., whether $f(n) = o(g(n))$, $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, or $f(n) = \Theta(g(n))$. Remember to prove the correctness of your choice. **(10 points)**

Solution. The correct answer is $f(n) = o(g(n))$. We now prove this as follows.

Firstly, $\sqrt{\log n} \leq \log n$ so we can instead prove $\log n = o(g(n))$ by transitivity.

Secondly, $2^{\log\log n} = \log n$ (you can do a change of variable $m = \log n$ to see this easily).

So our goal is to prove that $\log n = o(\frac{n}{\log n})$, which require us to show that $\lim_{n\to+\infty} \frac{\log n}{\frac{n}{\log n}} = 0$. We have,

$$\lim_{n\to+\infty} \frac{\log n}{\frac{n}{\log n}} = \lim_{n\to+\infty} \frac{\log^2 n}{n} = 0,$$

where the last equation is because $\log^2(n) = o(n)$ (generally, $\log^c(n) = o(n)$ for all constants $c > 0$).

(b) Use the *recursion tree* method to solve the following recurrence $T(n)$ by finding the *tightest* function $f(n)$ such that $T(n) = O(f(n))$: **(10 points)**

$$T(n) \leq 8 \cdot T(n/2) + O(n^3).$$

(You do *not* have to prove that your function is the tightest one.)

Solution. The correct answer is $T(n) = O(n^3 \cdot \log n)$. We prove this as follows.

We first replace $O(n^3)$ with $C \cdot n^3$ and then draw the following recursion tree:

Solution to part (b) goes here.

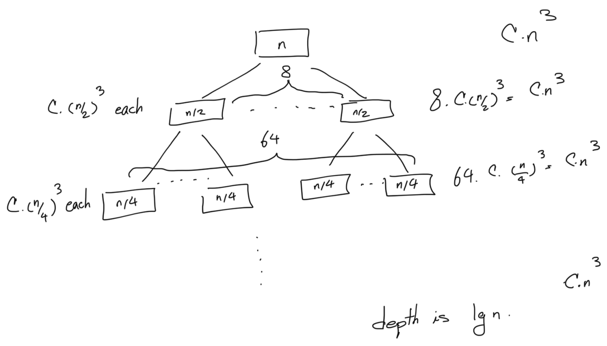

Figure 1: Recursion tree for the recurrence in Problem 1-(b).

The root is at level 1. At level i of the tree, we have 8^{i-1} nodes each with a value of $C \cdot (\frac{n}{2^{i-1}})^3$, hence the total value of level i is $C \cdot n^3$. The total number of levels in this tree is also $\lceil \log_2 n \rceil$. As such, the total value is

$$C \cdot n^3 \cdot \lceil \log_2 n \rceil = O(n^3 \cdot \log n).$$

Hence, $T(n) = O(n^3 \cdot \log n)$, finalizing the proof.

Problem 2. Consider the algorithm below for finding the total sum of the numbers in any array $A[1:n]$.

```
TOTAL-SUM($A[1:n]$):
  1. If $n = 0$: return 0.
  2. If $n = 1$: return $A[1]$.
  3. Otherwise, let $m_1 \leftarrow$ TOTAL-SUM($A[1:\frac{n}{2}]$) and $m_2 \leftarrow$ TOTAL-SUM($A[\frac{n}{2}+1:n]$).
  4. Return $m_1 + m_2$.
```

We analyze `TOTAL-SUM` in this question.

(a) Use *induction* to prove the correctness of this algorithm. **(10 points)**

Solution.

Induction hypothesis: For any integer $n \geq 0$ and array A of size n, `TOTAL-SUM`($A[1:n]$) returns the sum of the numbers in the array A, or in other words, returns the correct answer.

Induction base: For $n = 0$, we assume by convention that sum of an empty array is 0. For $n = 1$, the algorithm returns $A[1]$ which is the only number in the array and is thus equals the sum.

Induction step: Suppose the induction hypothesis is true for all $n \leq i$ for some integer $i \geq 1$ and we prove it for $n = i + 1$.

The algorithm first runs $m_1 \leftarrow$ `TOTAL-SUM`($A[1:\frac{n}{2}]$) and $m_2 \leftarrow$ `TOTAL-SUM`($A[\frac{n}{2}+1:n]$). Both of these arrays have size $< n$ and thus we can apply the induction hypothesis on them and have that m_1 and m_2 are equal to the sums of numbers in $A[1:\frac{n}{2}]$ and $A[\frac{n}{2}+1:n]$, respectively. Finally, we have that,

$$\sum_{j=1}^{n} A[j] = \Big(\sum_{j=1}^{n/2} A[j]\Big) + \Big(\sum_{j=n/2+1}^{n} A[j]\Big) = m_1 + m_2,$$

exactly as returned by the algorithm. So the algorithm for $n = i + 1$ also outputs the right answer. This concludes the proof of induction hypothesis.

The correctness of the algorithm now follows directly from the induction hypothesis.

(b) Write a recurrence for this algorithm and solve it to obtain a tight upper bound on the worst case runtime of this algorithm. You can use any method you like for solving this recurrence. **(10 points)**

Solution. The algorithm recursively calls itself on two arrays of length $n/2$ each and then spends $O(1)$ additional time to output the solution. Hence, if we define $T(n)$ to be the worst case runtime of the algorithm on any array of length n, we have $T(n) \leq 2 \cdot T(n/2) + O(1)$. We claim that $T(n) = O(n)$ and so the algorithm runs in $O(n)$ time.

To prove this, we first change $O(1)$ with constant C then draw the following recursion tree:

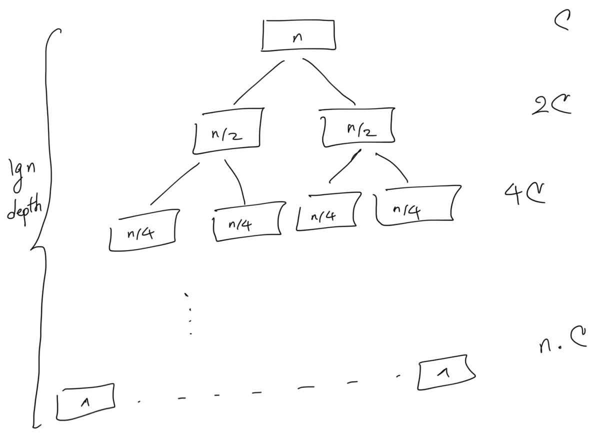

Figure 2: Recursion tree for the algorithm in Problem 2-(b).

The root is at level 1. At level i of the tree, we have 2^{i-1} nodes each with a value of C, hence the total value of level i is $2^{i-1} \cdot C$. The total number of levels in this tree is also $\lceil \log_2 n \rceil$. As such, the total value is

$$\sum_{i=0}^{\log n} 2^i \cdot C = C \cdot \frac{2^{\log n + 1} - 1}{2 - 1} \leq 2 \cdot C \cdot n = O(n).$$

Hence, $T(n) = O(n)$, finalizing the proof.

Problem 3. You are given a collection of n integers $a_1, \ldots, a_n$ with positive weights $w_1, \ldots, w_n$. For any number a_i, we define the *bias* of a_i as:

$$bias(a_i) = |\sum_{j:a_j<a_i} w_j - \sum_{k:a_k \geq a_i} w_k|;$$

i.e., the absolute value of the difference between the weights of elements smaller than a_i and the remaining ones. Design and analyze an algorithm that in $O(n \log n)$ time finds the element that has the *smallest* bias. You can assume that the input is given in two arrays $A[1:n]$ and $W[1:n]$ where $a_i = A[i]$ and $w_i = W[i]$.

Examples:

- When $n = 5$, and $A = [1, 5, 3, 2, 7]$ and $W = [3, 6, 2, 8, 9]$, the smallest biased element is $a_2 = A[2] = 5$ with $bias(a_2) = |(3 + 2 + 8) - (6 + 9)| = 2$.
- When $n = 5$, and $A = [1, 2, 3, 4, 5]$ and $W = [8, 6, 5, 2, 6]$, the smallest biased element is $a_3 = A[3] = 3$ with $bias(a_3) = |(8 + 6) - (5 + 2 + 6)| = 1$.

(a) *Algorithm:* **(7 points)**

Solution. The algorithm is as follows:

(a) Create a new array of pairs $(A[i], W[i])$ and sort these pairs in increasing (non-decreasing) order of their $A[i]$ values using merge sort (moving along the $W[i]$ value naturally).

(b) Create a prefix sum array $S[1:n]$ where $S[1] = W[1]$ and for $i = 2$ to n, $S[i] = S[i-1] + W[i]$.

(c) Let $s = S[n]$ and for $i = 1$ to n:

   i. If $A[i] = A[i-1]$ continue (define $A[0]$ as some number different from $A[1]$).

   ii. Otherwise, let $B[i] = |2 \cdot S[i-1] - s|$ and insert $(A[i], B[i])$ to a linked-list L (define $S[0] = 0$).

(d) Return the pair $(A[j], B[j])$ in the list L with the smallest value of $B[j]$.

(b) *Proof of Correctness:* **(10 points)**

Solution. By the correctness of merge sort, the array of pairs $(A[i], W[i])$ will be sorted correctly. As such, from hereon, we consider these pairs to be sorted and use the indices in their sorted order not the original ones.

Moreover, we can prove inductively that for every $i \in [1 : n]$, $S[i] = \sum_{j=1}^{i} W[j]$, namely, the sum of the first i smallest pairs (according to A-values). This also implies that $s = S[n]$ is the total sum.

Next, note that in the original definition of bias, for any two pairs $a_i = a_j$ (as array A may have duplicates), we have $bias(a_i) = bias(a_j)$ as the formula for the bias for both of them is identical. As a result, we can skip duplicate values in the for-loop. Note that as we shall see soon, this step is actually crucial for guaranteeing the correctness of the algorithm.

We now prove that for every index i where $A[i] > A[i-1]$, the value $B[i] = bias(A[i])$. By definition,

$$
\begin{aligned}
bias(A[i]) &= |\sum_{j:A[j]<A[i]} W[j] - \sum_{k:A[k]\geq A[i]} W[k]| \\
&= |\sum_{j:A[j]<A[i]} W[j] - (S[n] - \sum_{j:A[j]<A[i]} W[j])| \\
&\text{(because } \textstyle\sum_{j:A[j]<A[i]} W[j] + \sum_{k:A[k]\geq A[i]} W[k] \text{ is the sum of all numbers } S[n]) \\
&= |2 \cdot \sum_{j:A[j]<A[i]} W[j] - s| \qquad \text{(because } s = S[n]) \\
&= |2 \cdot \sum_{j=1}^{i-1} W[j] - s| \\
&\text{(because } A[i] > A[i-1] \text{ and thus the set of } j : A[j] < A[i] \text{ is exactly } A[1], \ldots, A[i-1]) \\
&= |2 \cdot S[i-1] - s|. \qquad \text{(since } S[i-1] = \textstyle\sum_{j=1}^{i-1} W[j])
\end{aligned}
$$

Thus $bias(A[i]) = B[i]$ for every $A[i]$ where $A[i] > A[i-1]$. Note that we used the fact that $A[i] > A[i-1]$ crucially in the second to last equality above.

At this point, we have all the different biases stored in a linked list and we are simply returning the minimum one as desired. This concludes the proof of the correctness of the algorithm.

(c) *Runtime Analysis:* **(3 points)**

Solution. Line (a) of the algorithm takes $O(n)$ time to create the pairs and $O(n \log n)$ time to sort them by merge sort.

Line (b) takes $O(n)$ time to create the prefix sum array.

Line (c) involves an n-iteration for-loop, each taking $O(1)$ time, so $O(n)$ time in total.

Line (d) involves finding a minimum of a length list of length at most n by a linear search which takes $O(n)$ time.

So overall the runtime is $O(n \log n)$ as desired.

Problem 4. You are given three arrays $A[1:n]$, $B[1:n]$, and $C[1:n]$ of positive integers. The goal is to decide whether or not there are indices $i, j, k \in [1:n]$ such that $A[i] \cdot B[j] = C[k]$; in other words, is it the case that there are numbers in A and B whose multiplication belongs to C.

Examples:

- When $n = 3$, and $A = [1, 3, 4]$, $B = [2, 3, 5]$, and $C = [1, 3, 5]$, the answer is *Yes*, because for instance we have $A[1] \cdot B[3] = C[3]$ or $A[1] \cdot B[2] = C[2]$.
- When $n = 3$ and $A = [1, 3, 4]$, $B = [2, 4, 6]$, and $C = [7, 9, 11]$, the answer is *No.*

(a) Suppose all the numbers in C belong to the set $\{1, 2, \ldots, n^2\}$. Design and analyze an algorithm with **worst-case runtime** of $O(n^2)$ for the problem in this case. **(10 points)**

Solution. The solution consists of three parts: algorithm, proof of correctness, and runtime analysis.

Algorithm: We create the intermediate array of counting sort for entries of C and then enumerate all pairs $A[i] \cdot B[j]$ to check whether they appear in C or not using this array. Formally,

```
(a) Let $T[1 : n^2]$ be an array initialized to zero.
(b) For $i = 1$ to $n$ increase $T[C[i]]$ by one.
(c) For $i = 1$ to $n$:
      i. For $j = 1$ to $n$: if $T[A[i] \cdot B[j]] \neq 0$ return Yes.
(d) If never returned Yes in the above for-loops, return No.
```

Proof of Correctness: As proved in the counting sort also, after Line (b) of the algorithm, an entry $T[c] \neq 0$ if and only if c appears at least once in the array C.

Suppose first that the correct answer to the problem is *Yes.* Then, there must exist at least one pair a, b such that $A[a] \cdot B[b] = c$ for some c that belongs to C. When iterating the nested for-loops, for $i = a$ and $j = b$, the algorithm checks $T[A[a] \cdot B[b]] = T[c]$ and we know $T[c] \neq 0$ as argued above. So, the algorithm in this case correctly outputs *Yes.*

On the other hand suppose the correct answer to the problem is *No.* Then, there is no pair i, j such that $A[i] \cdot B[j]$ belongs to C, so when doing the search, we never return *Yes* in the algorithm. At the end, the algorithm returns *No* which is the correct answer for this case.

Runtime analysis:

In Line (a), initializing the array T takes $O(n^2)$ time.

Line (b) takes $O(n)$ time to iterate over the array C.

Line (c) involves two nested for-loop of length n each, and $O(1)$ time per each iteration, so $O(n^2)$ time.

The worst-case runtime of the algorithm is thus $O(n^2)$.

(*b*) Now suppose C can be any arbitrary array of n integers. Design and analyze a **randomized** algorithm with **expected worst-case runtime** of $O(n^2)$ for the problem in this case. **(10 points)**

Note: Actually, this problem also has a deterministic algorithm that runs in worst-case $O(n^2)$ time. But you do not need to design such an algorithm for this problem (although if you do, you will receive the full credit for both parts (*a*) and (*b*)).

Solution. The solution consists of three parts: algorithm, proof of correctness, and runtime analysis.

Algorithm: We simply switch the role of array T in part (a) with a hash table T instead as follows:

(a) Create a hash table T of size $m = n$ using a near-universal random hash family, and by handling collisions using the chaining method.

(b) Insert all elements of C into the hash table T.

(c) For $i = 1$ to n:

  i. For $j = 1$ to n: search $(A[i] \cdot B[j]])$ in the hash table T and if it exists return *Yes*.

(d) If never returned *Yes* in the above for-loops, return *No*.

Proof of Correctness: By correctness of hash table T, we know that every element of C that is hashed to T will be found via search operation. As such, the proof of correctness is verbatim as in part (a) as we are simply changing the search operation from the array T of part (a) to a hash table.

Runtime analysis: Creating the hash table T and inserting the elements of C into it, using chaining, takes $O(n + m) = O(n)$ time deterministically. For any $i, j \in \{1, \ldots, n\}$ define $X_{i,j}$ as the random variable denoting the time it takes to search for $(A[i] \cdot B[j])$ in T (this runtime is a random variable depending on the choice of hash function). As such, the expected worst-case runtime of the nested for-loops is:

$$O(\mathbf{E}[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{i,j}]) = O(\sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{E}[X_{i,j}]),$$

by linearity of expectation. As we are using a near-universal hash family, each single search takes $O(1 + n/m) = O(1)$ time in expectation, thus $\mathbf{E}[X_{i,j}] = O(1)$.

This means that the expected worst-case runtime of the algorithm is $O(n^2)$, as desired.

Alternative solution for both parts (a),(b). The following is the deterministic solution that the "*Note*" in the problem statement referred to.

Algorithm: The algorithm is as follows:

(a) Sort each of the arrays A and B and C in increasing order using merge sort.

(b) For $i = 1$ to n:

    i. Let $b = c = 0$.
    ii. While $b \leq n$ AND $c \leq n$: if $A[i] \cdot B[b] = C[c]$, return *Yes*; if $A[i] \cdot B[b] > C[c]$, increase c by one, otherwise increase b by one.

(c) Return *No*

Proof of Correctness: Let us fix any choice of $i \in \{1, \ldots, n\}$. In the algorithm, for some triples (i, j, k), we may never check whether $A[i] \cdot B[j] = C[k]$, depending on the values of b and c. We are going to prove that none of these choices can ever result in $A[i] \cdot B[j] = C[k]$. This immediately implies the correctness as for the remaining triples, we are checking the equality explicitly.

Now consider every step we are updating b or c:

- If $A[i] \cdot B[b] > C[c]$, we will increase c; this means that we never check triples (i, j, k) for $j > b$ and $k = c$. But these can never form an answer since the array B is sorted and thus $A[i] \cdot B[j] \geq A[i] \cdot B[b] > C[c]$. So we are good in this case.
- Similarly, if $A[i] \cdot B[b] < C[c]$, we will increase b; this means that we never check triples (i, j, k) for $j = b$ and $k > c$. But these can never form an answer since the array C is sorted and thus $C[k] \geq C[c] > A[i] \cdot B[b]$. So we are good in this case also.

The above argument now shows that any triple not explicitly checked by the algorithm cannot ever form an equal triple, finalizing the proof.

Runtime analysis: Sorting the arrays takes $O(n \log n)$ time.

The while-loop can take $2n$ iterations at most before finishing: in each iteration we either output the answer or increase (exactly) one of b or c by one and either can only go up to n. The outer for-loop can also have n iterations at most, so the total runtime is $O(n^2)$.

Problem 5. Please solve the problems in **exactly one** of the two parts below.

Part 1 (dynamic programming): We want to purchase an item of price M and for that we have a collection of n different coins in an array $C[1:n]$ where coin i has value $C[i]$ (we only have one copy of each coin). Our goal is to purchase this item using the *smallest* possible number of coins or outputting that the item cannot be purchased with these coins. Design a **dynamic programming** algorithm for this problem with worst-case runtime of $O(n \cdot M)$. **(20 points)**

Examples:

- Given $M = 15$, $n = 7$, and $C = [4, 9, 3, 2, 7, 5, 6]$, the correct answer is 2 by picking $C[2] = 9$ and $C[7] = 6$ which add up to 15.
- Given $M = 11$, $n = 4$, and $C = [4, 3, 5, 9]$, the correct answer is that 'the item cannot be purchased' as no combination of these coins adds up to a value of 11 (recall that we can only use each coin once).

(a) *Specification of recursive formula for the problem (in plain English)*: **(5 points)**

Solution. For any integer $0 \leq i \leq n$ and $0 \leq j \leq M$, we define:

- $S(i, j)$: the smallest possible number of coins among $C[1:i]$ for purchasing an item of value j; if it is not possible to purchase the item using these coins, $S(i, j) = +\infty$.

The solution to the problem can be obtained by returning $S(n, M)$.

(b) *Recursive solution for the formula and its proof of correctness:* **(7 points)**

Solution. The recursive formula for $S(i, j)$ is as follows:

$$S(i,j) = \begin{cases} 0 & \text{if } j = 0 \\ +\infty & \text{if } j > 0 \text{ but } i = 0 \\ S(i-1, j) & \text{if } C[i] > j \\ \min\{S(i-1,j), S(i-1, j - C[i]) + 1\} & \text{otherwise} \end{cases}.$$

Proof of correctness: We consider each case separately:

- If $j = 0$, there is no price to pay and thus we do not need to use any coins either. Thus the correct value is zero.
- If $j > 0$ but $i = 0$, we ran out of coins but still have to pay something for the item which we cannot; so we cannot afford the item and the answer should be $+\infty$.
- If $C[i] > j$, we can only ignore the coin i as we cannot use it at all and thus work with the coins $C[1 : i-1]$. In this case, the best option is $S(i-1, j)$ by definition as chosen by the formula.
- Otherwise, we have two options: (1) we do not pick coin i: in this case, we still have to pay a price of j for the item but we now we can only pick from $C[1 : i-1]$; in this option, the best way to pay using the remaining coins is to use $S(i-1, j)$ coins by definition. (2) we pick coin i: in this case, we have to pay a price of $j - C[i]$ using the coins $C[1 : i-1]$ but now also used one more coin; so the "cost" of this option is $S(i-1, j - C[i]) + 1$ as chosen by the algorithm. As our goal is to use the smallest number of coins, taking the minimum ensures the correctness.

(c) *Algorithm (either memoization or bottom-up dynamic programming):* **(3 points)**

Solution. Let $T[1:n][1:M]$ be a 2D-array initialized with 'undefined' originally. We design the following memoization algorithm:

`MemS`(i, j):

(a) if $j = 0$ return 0.
(b) if $j > 0$ but $i = 0$, return $+\infty$.
(c) if $T[i][j] \neq$ 'undefined' return $T[i][j]$.
(d) if $C[i] > j$, let $T[i][j] = \texttt{MemS}(i-1, j)$ otherwise let $T[i][j] = \min\{\texttt{MemS}(i-1, j), \texttt{MemS}(i-1, j-C[i]) + 1\}$.
(e) Return $T[i][j]$.

The answer to our problem is then `MemS`(n, M).

(d) *Runtime Analysis:* **(5 points)**

Solution. There are a total of $O(nM)$ subproblems each of them taking $O(1)$ time to compute, so the runtime is $O(nM)$ in general.

Part 2 (greedy): We want to purchase an item of price M and for that we have an unlimited (!) supply of $\lceil \log M \rceil$ types of coins with value $1, 2, 4, \cdots, 2^i, \cdots, 2^{\lceil \log M \rceil}$. Our goal is to purchase this item using the *smallest* possible number of coins (it is always possible to buy this item by picking M coins of value 1). Design and analyze a **greedy** algorithm for this problem with $O(\log M)$ runtime. **(20 points)**

Examples:

- Given $M = 15$ (and so $\lceil \log M \rceil = 4$), the correct answer is 4 coins by picking one copy of each of the coins $8, 4, 2, 1$. Note that here we cannot pick the coin of value $2^{\lceil \log M \rceil} = 2^4 = 16$.
- Given $M = 32$ (and so $\lceil \log M \rceil = 5$), the correct answer is 1 coin by picking one copy of the coin $32 = 2^5 = 2^{\lceil \log M \rceil}$.

(a) *Algorithm*: **(5 points)**

Solution. The algorithm goes over the coins from largest value to smallest and greedily pick each coin smaller than M, update the remaining price of M, and continues. Formally,

```
(a) For $i = \lceil \log M \rceil$ down to 0:
      • If $M \geq 2^i$, pick the $i$-th coin and update $M \leftarrow M - 2^i$.
(b) Return the set of picked coins.
```

Note that the algorithm is picking each coin at most once.

(b) *Proof of Correctness*: **(10 points)**

Solution. We first prove that by the time the algorithm finishes, the value of chosen coins is exactly equal to the original value of M.

Define M_i as the value of M at the beginning of iteration i. We prove by induction that for $i = \lceil \log M \rceil$ down to 0 (in this order), $M_i < 2^{i+1}$.

For the base case when $i = \lceil \log M \rceil$, by definition $M < 2^{\lceil \log M \rceil + 1}$ and thus we are done.

Now suppose this is true for some iteration i and we prove it for iteration $i - 1$ (again, notice that we are reducing the value of i in our induction instead of increasing because our for-loop is in the decreasing order not increasing).

By induction, we know that at the beginning of iteration i, $M_i < 2^{i+1}$. If $M_i \geq 2^i$, then in that iteration we update M and have $M_{i-1} = M_i - 2^i < 2^{i+1} - 2^i = 2^i$. So $M_{i-1} < 2^i$ in this case as desired. Otherwise, $M_i < 2^i$ already and we have $M_{i-1} = M_i < 2^i$ as well, proving the induction step and hypothesis.

Now consider the last iteration of the algorithm for $i = 0$. In this case, by our induction statement $M_0 < 2^1$ and so $M_0 \in \{0, 1\}$. As such, either in this iteration we purchase a coin of value 1 and or purchase nothing; in either case the value of M becomes zero after this iteration. This means we exactly pay for the price of the item.

We now prove the optimality of the greedy algorithm. Let $G = \{g_0, g_1, \ldots, g_{\lceil \log M \rceil}\}$ be the greedy solution where $g_i = 1$ means we picked the i-th coin and $g_i = 0$ means we did not. Similarly, let $O = \{o_0, \ldots, o_{\lceil \log M \rceil}\}$.

If $G = O$ we are done, otherwise find the *largest* index i where $g_i \neq o_i$ (i.e., $g_{i+1} = o_{i+1}, g_{i+2} = o_{i+2}, \ldots$). Note that it cannot be the case $g_i = 0$ and $o_i = 1$ as the greedy algorithm would have picked coin i also if optimal could have picked it (since value of M_i is the same between G and O at this point). So we have $g_i = 1$ and $o_i = 0$. We now claim that there should exist some index $j < i$ such that $o_j > 1$. Suppose not, then the total value of remaining coins in O is

$$\sum_{k=0}^{i-1} o_k \cdot 2^k \leq \sum_{k=0}^{i-1} 2^k = 2^i - 1,$$

which is less than g_i; but since greedy picked g_i, it means $M_i \geq g_i$, so that would be a contradiction as it means O could have not paid for the item.

Finally, since there is some $o_j \geq 2$, we can instead reduce o_j to $o_j - 2$ and increase o_{j+1} to $o_{j+1} + 1$ and get exactly the same value, but with one less coin. This contradicts the optimality of O and finalizes the proof.

(c) *Runtime Analysis:* **(5 points)**

Solution. The algorithm consists of one for-loop with $O(\log M)$ iterations each taking $O(1)$ time, thus has worst-case runtime of $O(\log M)$.

Problem 6. [**Extra credit**] You are given two *unsorted* arrays $A[1:n]$ and $B[1:n]$ consisting of $2n$ *distinct* numbers such that $A[1] < B[1]$ but $A[n] > B[n]$. Design and analyze an algorithm that in $O(\log n)$ time finds an index $j \in [1:n]$ such that $A[j] < B[j]$ but $A[j+1] > B[j+1]$.

Hint: Start by convincing yourself that such an index j always exist in the first place. **(+10 points)**

Solution. It helps to start by observing that why such an index j always exist. The proof is simply as follows: Suppose such an index does not exist at all. Then, we should have $A[1] < B[1]$, $A[2] < B[2]$, $A[3] < B[3]$, all the way to $A[n] < B[n]$, but this is a contradiction with $A[n] > B[n]$ in the problem statement. Note that this proves actually shows that for any subarray $A[i:k]$, if $A[i] < B[i]$ and $A[k] > B[k]$, then there exists an index j such that $A[j] < B[j]$ and $A[j+1] > B[j+1]$; we use this in our algorithm in the following.

The solution consists of three parts: algorithm, proof of correctness, and runtime analysis.

Algorithm: We design an algorithm for this problem in the spirit of the binary search algorithm. Note however that we have to arrays here and neither of them is sorted.

1. If $n \leq 4$ do a linear search over the arrays to find the desired index j.
2. Let $m = \lfloor n/2 \rfloor$:
   - if $A[m] > B[m]$, recursively solve the problem in arrays $A_{\leq m} := A[1:m]$ and $B_{\leq m} := B[1:m]$.
   - if $A[m] < B[m]$, recursively solve the problem in arrays $A_{\leq m} := A[m:n]$ and $B_{\geq m} := B[m:n]$ (return $m-1+$ that index to adjust for change of the range in the array).

Proof of Correctness: The proof is by induction on size of the array A, i.e., n. The base case of the induction is when $n \leq 4$ for which the algorithm is correct as we are just checking all options and we already argued that any such type of array definitely has a solution index.

Now suppose by induction that the algorithm correctly solves the problem on every array of size up to i and we prove it for $i+1$.

Consider an array A of size $n = i+1$ and define $m = \lfloor n/2 \rfloor$ as in the algorithm. Notice that in this case both pairs of arrays $(A_{\leq m}, B_{\leq m})$ and $(A_{\geq m}, B_{\geq m})$ have size strictly less than n. Moreover, if $A[m] > B[m]$, then we have $A_{\leq m}[1] < B_{\leq m}[1]$ while $A_{\leq m}[m] > B_{\leq m}[1]$; so this pair has a solution and by recursively running the algorithm over them we will be able to find it by the induction hypothesis. The proof for $A[m] < B[m]$ case is also symmetric (and note that we add $m-1$ to the answer to the index will be relative to the original $[1:n]$ not the range $[m:n]$).

This concludes the correctness of the algorithm.

Runtime Analysis. Let $T(n)$ denote the worst-case runtime of the algorithm on any input of size n. We have $T(n) \leq T(n/2) + O(1)$. This is the same recurrence as binary search and so it solves to $T(n) = O(\log n)$. So the runtime of our algorithm is $O(\log n)$.