Quiz 1 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 7 problems, with multiple parts. You have 120 minutes to earn 120 points.
- This quiz booklet contains 12 pages, including this one.
- This quiz is closed book. You may use one double-sided letter ($8\frac{1}{2}'' \times 11''$) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to "give an algorithm" in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how many minutes to spend on it.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Please be neat.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
1	True or False	40	10		
2	Fast Fourier Transform	5	1		
3	Yellow Brick Road	10	1		
4	Amortized Analysis	15	1		
5	Verifying Polynomial Multiplication	15	4		
6	Dynamic Programming	15	2		
7	Median of Sorted Arrays	20	3		
Total		120			

Name: ______________________________

Circle your recitation:

10am	11am	12pm	12pm	1pm	1pm	2pm	3pm
Akshat	Ling	Chennah	Deepak	Andrea	Shalom	Chongyuan	Albert
	Shankha					Xiangyao	

Problem 1. True or False. [40 points] (10 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false and briefly explain why.

(a) T F [4 points] With all equal-sized intervals, a greedy algorithm based on earliest start time will always select the maximum number of compatible intervals.

Solution: True. The algorithm is equivalent to the earliest finish time algorithm.

(b) T F [4 points] The problem of weighted interval scheduling can be solved in $O(n \log n)$ time using dynamic programming.

Solution: True. The algorithm was covered in recitation.

(c) T F [4 points] If we divide an array into groups of 3, find the median of each group, recursively find the median of those medians, partition, and recurse, then we can obtain a linear-time median-finding algorithm.

Solution: False. $T(n) = T(n/3) + T(2n/3) + O(n)$ does not solve to $T(n) = O(n)$. The array has to be broken up into groups of at least 5 to obtain a linear-time algorithm.

(d) T F [4 points] If we used the obvious $\Theta(n^2)$ merge algorithm in the divide-and-conquer convex-hull algorithm, the overall time complexity would be $\Theta(n^2 \log n)$.

Solution: False. The time complexity would satisfy the recurrence $T(n) = 2T(n/2) + \Theta(n^2)$, which solves to $\Theta(n^2)$ by the Master Theorem.

(e) T F [4 points] Van Emde Boas sort (where we insert all numbers, find the min, and then repeatedly call SUCCESSOR) can be used to sort $n = \lg u$ numbers in $O(\lg u \cdot \lg \lg \lg u)$ time.

Solution: False. Inserting into the tree and then finding all the successors will take $n \lg \lg(u)$ time, which in terms of u is $\lg(u) \cdot \lg \lg(u)$.

(f) T F [4 points] Van Emde Boas on n integers between 0 and $u - 1$ supports successor queries in $O(\lg \lg u)$ worst-case time using $O(n)$ space.

Solution: False. We use $\Theta(u)$ space or do randomization.

(g) T F [4 points] In the potential method for amortized analysis, the potential energy should never go negative.

Solution: True. If the potential function goes negative, then for the sequence of operations up to that point, the total actual cost is smaller than the total amortized cost. (This solution assumes the initial potential is 0, which is usually the case).
Alternative solution: False. The potential function can go negative as long as it's above the initial potential function.

(h) T F [4 points] The quicksort algorithm that uses linear-time median finding to run in worst-case $O(n \log n)$ time requires $\Theta(n)$ auxiliary space.

Solution: False. It can be implemented with $O(\log n)$ auxiliary space.

(i) T F [4 points] Searching in a skip list takes $\Theta(\log n)$ time with high probability, but could take $\Omega(2^n)$ time with nonzero probability.

Solution: True. A skip list could be of any height with nonzero probability, depending on its random choices.
Alternative solution: False. We can limit the height of the skip list to $O(n)$ or $O(\lg n)$ to get $O(n)$ worse-case cost.
Common mistake 1: We go through each element at least once. (Wrong because we also need to "climb up" the skip lists.
Common mistake 2: The worst case is when none of the elements is promoted, or all of the elements are promoted to the same level, in which cases skip lists become a linked list.

(j) T F [3 points] The following collection $\mathcal{H} = \{h_1, h_2, h_3\}$ of hash functions is universal, where each hash function maps the universe $U = \{A, B, C, D\}$ of keys into the range $\{0, 1, 2\}$ according to the following table:

x	A	B	C	D
$h_1(x)$	1	0	1	1
$h_2(x)$	0	1	0	1
$h_3(x)$	2	2	1	0

Solution: False. A and C collide with probability 2/3.

Problem 2. Fast Fourier Transform (FFT). [5 points] (1 part)

Ben Bitdiddle is trying to multiply two polynomials using the FFT. In his trivial example, Ben sets $a = (0, 1)$ and $b = (0, 1)$, both representing $0 + x$, and calculates:

$$A = \mathcal{F}(a) = B = \mathcal{F}(b) = (1, -1),$$

$$C = A * B = (1, 1),$$
$$c = \mathcal{F}^{-1}(C) = (1, 0).$$

So c represents $1 + 0 \cdot x$, which is clearly wrong. Point out Ben's mistake in one sentence; no calculation needed. (Ben swears he has calculated FFT $\mathcal{F}$ and inverse FFT $\mathcal{F}^{-1}$ correctly.)

Solution: The resulting polynomial is of degree 2, so Ben need to pad a and b with zeroes. (Or Ben need at least 3 samples to do FFT).

Here is the correct calculaton (not required in the solution). Let $a = b = (0, 1, 0, 0)$; then,

$$A = \mathcal{F}(a) = B = \mathcal{F}(b) = (1, -i, -1, i)$$

$$C = A * B = (1, -1, 1, -1)$$
$$c = \mathcal{F}^{-1}(C) = (0, 0, 1, 0)$$

which represents x^2. It is also OK to set $a = b = (0, 1, 0)$.

Common mistake 1: $A * B$ should be convolution.

Common mistake 2: Ben should reverse b.

Both mistakes confuse the relation between convolution, polynomial multiplication and FFT calculation. If one computes convolution directly (without FFT), one needs to reverse the second vector. FFT provides a faster way to compute convolution and polynomial multiplication (these two are the same thing). Using the FFT, one should perform FFT on the two original vectors (no reversal). Then, after the FFT, one only needs to do element-wise multiplication (as opposed to convolution), which Ben performed correctly.

Problem 3. Yellow Brick Road. [10 points] (1 part)

Prof. Gale is developing a new Facebook app called "Yellow Brick Road" for maintaining a user's timeline, here represented as a time-ordered list $e_0, e_1, \ldots, e_{n-1}$ of n (unchanging) events. (In Facebook, events can never be deleted, and for the purposes of this problem, don't worry about insertions either.) The app allows the user to mark an event e_i as ***yellow*** (important) or ***grey*** (unimportant); initially all events are grey. The app also allows the user to jump to the next yellow event that comes after the event e_i currently on the screen (which may be yellow or grey). More formally, you must support the following operations:

1. MARK-YELLOW(i): Mark e_i yellow.
2. MARK-GREY(i): Mark e_i grey.
3. NEXT-YELLOW(i): Find the smallest $j > i$ such that e_j is yellow.

Give the fastest data structure you can for this problem, measured according to *worst-case time*. The faster your data structure, the better.

Hint: Use a data structure you have seen in either 6.006 or 6.046 as a building block.

Solution: Initialization takes $O(n \lg(\lg(n)))$ time to insert all the yellow elements into a VEB tree, V.

More importantly, each operation takes $O(\lg\lg(n))$ time. When a user asks to MARK-YELLOW(i), then call $V.insert(i)$ which takes $O(\lg\lg(n))$ time. When a user asks to MARK-GREY(i), then call $V.delete(i)$ which takes $O(\lg\lg(n))$ time. When a user asks to NEXT-YELLOW(i), then call $V.successor(i)$ which takes $O(\lg\lg(n))$ time.

Another slower solution used an AVL tree in place of a vEB for an $O(\lg(n))$ runtime for the operations.

Common mistake 1: Claiming operations took $O(\lg\lg(u))$. The universe size is exactly n, and the term u was undefined.

Common mistake 2: Inserting both yellow and grey elements into the same data structure without an augmentation to keep track of whether any yellow elements existed within children. Next-Yellow could take $O(n)$ in the worst case.

Common mistake 3: Use a skip list to keep track of all yellow elements. Some operations would take $O(\lg(n))$ with high probability, but $O(n)$ worst case.

Common mistake 4: Using a skip list capped at 2 levels, doubly linked list, or hash table. Some operations would take $O(n)$ worst case.

Problem 4. Amortized Analysis. [15 points] (1 part)

Design a data structure to maintain a set S of n distinct integers that supports the following two operations:

1. INSERT(x, S): insert integer x into S.
2. REMOVE-BOTTOM-HALF(S): remove the smallest $\lceil \frac{n}{2} \rceil$ integers from S.

Describe your algorithm and give the worse-case time complexity of the two operations. Then carry out an amortized analysis to make INSERT(x, S) run in amortized $O(1)$ time, and REMOVE-BOTTOM-HALF(S) run in amortized 0 time.

Solution:

Use a singly linked list to store those integers. To implement INSERT(x, S), we append the new integer to the end of the linked list. This takes $\Theta(1)$ time. To implement REMOVE-BOTTOM-HALF(S), we use the median finding algorithm taught in class to find the median number, and then go through the list again to delete all the numbers smaller or equal than the median. This takes $\Theta(n)$ time.

Suppose the runtime of REMOVE-BOTTOM-HALF(S) is bounded by cn for some constant c. For amortized analysis, use $\Phi = 2cn$ as our potential function. Therefore, the amortized cost of an insertion is $1 + \Delta\Phi = 1 + 2c = \Theta(1)$. The amortized cost of REMOVE-BOTTOM-HALF(S) is $cn + \Delta\Phi = cn + (-2c \times \frac{n}{2}) = 0$.

Common mistake 1: Use a sorted list and binary search for insertion.

If it is an array, you need to shift all the elements after the inserting point, making it $O(n)$. If it is a linked list, you cannot do binary search.

Common mistake 2: Use a perfecly balanced BST such that insertion is $O(\lg n)$ and REMOVE-BOTTOM-HALF is $O(1)$.

You cannot dynamically maintain a perfectly balanced BST in $O(\lg n)$. If it is not perfectly balanced, REMOVE-BOTTOM-HALF is not $O(1)$ because the left half the tree may not contain exactly half of the elements. A much better solution is to augment a balanced BST with the size of its subtree. That allows you to implement REMOVE-BOTTOM-HALF in $O(\lg n)$, yielding a slower (amortized logarithmic) but functionally correct solution.

Common mistake 3: Each REMOVE-BOTTOM-HALF charges all the n elements to pay for the $O(n)$ actual cost.

You may only charge the $\lceil \frac{n}{2} \rceil$ elements that are removed. If you charge all the n elements, the remaining ones will be charged again later, and you cannot argue each element is charged only by a constant number of times.

Problem 5. Verifying Polynomial Multiplication. [15 points] (4 parts)

This problem will explore how to check the product of two polynomials. Specifically, we are given three polynomials:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0,$$
$$q(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_0,$$
$$r(x) = c_{2n} x^{2n} + c_{2n-1} x^{2n-1} + \cdots + c_0.$$

We want to check whether $p(x) \cdot q(x) = r(x)$ (for all values x). Via FFT, we could simply compute $p(x) \cdot q(x)$ and check in $O(n \log n)$ time. Instead, we aim to achieve $O(n)$ time via randomization.

(a) [5 points] Describe an $O(n)$-time randomized algorithm for testing whether $p(x) \cdot q(x) = r(x)$ that satisfies the following properties:

1. If the two sides are equal, the algorithm outputs YES.
2. If the two sides are unequal, the algorithm outputs NO with probability at least $\frac{1}{2}$.

Solution: Pick a value $a \in [1, 4n]$, and check whether $p(a)q(a) = r(a)$. The algorithm outputs YES if the two sides are equal, and NO otherwise. It takes $O(n)$ time to evaluate the three polynomials of degree $O(n)$. Thus the overall running time of the algorithm is $O(n)$.

(b) [2 points] Prove that your algorithm satisfies Property 1.

Solution: If $p(x) \cdot q(x) = r(x)$, then both sides will evaluate to the same thing for any input.

(c) [3 points] Prove that your algorithm satisfies Property 2.

Hint: Recall the Fundamental Theorem of Algebra: A degree-d polynomial has (at most) d roots.

Solution: $s(x) = r(x) - p(x) \cdot q(x)$ is a degree-$2n$ polynomial, and thus has at most $2n$ roots. Then

$$\Pr\{s(a) = 0\} \leq \frac{2n}{4n} = \frac{1}{2}$$

since a was picked from a set of size $4n$.

(d) [5 points] Design a randomized algorithm to check whether $p(x) \cdot q(x) = r(x)$ that is correct with probability at least $1 - \varepsilon$. Analyze your algorithm in terms of n and $1/\varepsilon$.

Solution: We run part a m times, and output YES if and only if all answers output YES. In other words, we amplify the probability of success via repetition.

Our test works with probability $\geq 1 - \left(\frac{1}{2}\right)^m$. Thus we need

$$\left(\frac{1}{2}\right)^m \leq \varepsilon$$

$$\Rightarrow m \geq \log \frac{1}{\varepsilon}.$$

Problem 6. Dynamic Programming. [15 points] (2 parts)

Prof. Child is cooking from her garden, which is arranged in grid with n rows and m columns. Each cell (i, j) ($1 \leq i \leq n, 1 \leq j \leq m$) has an ingredient growing in it, with ***tastiness*** given by a positive value $T_{i,j}$. Prof. Child doesn't like cooking "by the book". To prepare dinner, she will stand at a cell (i, j) and pick one ingredient from each quadrant relative to that cell. The tastiness of her dish is the product of the tastiness of the four ingredients she chooses. Help Prof. Child find an $O(nm)$ dynamic programming algorithm to maximize the tastiness of her dish.

Here the four ***quadrants*** relative to a cell (i, j) are defined as follows:

$$\begin{aligned}
\text{top-left} &= \{\text{all cells } (a, b) \mid a < i, b < j\},\\
\text{bottom-left} &= \{\text{all cells } (a, b) \mid a > i, b < j\},\\
\text{top-right} &= \{\text{all cells } (a, b) \mid a < i, b > j\},\\
\text{bottom-right} &= \{\text{all cells } (a, b) \mid a > i, b > j\}.
\end{aligned}$$

Because Prof. Child needs all four quadrants to be non-empty, she can only stand on cells (i, j) where $1 < i < n$ and $1 < j < m$.

(a) [10 points] Define $TL_{i,j}$ to be maximum tastiness value in the top-left quadrant of cell (i, j): $TL_{i,j} = \max\{T_{a,b} \mid 1 \leq a \leq i, 1 \leq b \leq j\}$. Find a dynamic programming algorithm to compute $TL_{i,j}$, for all $1 < i < n$ and $1 < j < m$, in $O(nm)$ time.

Solution: When trying to calculate $TL_{i,j}$, we see that the maximum can be at cell (i, j). If not, it must lie either in the rectangle from $(1, 1)$ to $(i, j - 1)$, or the rectangle from $(1, 1)$ to $(i - 1, j)$, or both. These three overlapping cases cover our required rectangle. We have then,

$$TL_{i,j} = \max\{T_{i,j}, TL_{i-1,j}, TL_{i,j-1}\}$$

For the base cases, we can just set $TL_{0,j} = TL_{i,0} = 0$ for all valid values of i and j. We can compute the DP value for each state in $O(1)$ time. There are nm states, so our algorithm is $O(nm)$.

(b) [5 points] Use the idea in part (a) to obtain an $O(nm)$ algorithm to find the tastiest dish.

Solution: In part (a) we calculated range maximum for the top-left quadrant. We can similarly define range maximums for the other quadrants. Let $BL_{i,j} = \max\{T_{a,b} \mid i \leq a \leq n, 1 \leq b \leq j\}$, $TR_{i,j} = \max\{T_{a,b} \mid 1 \leq a \leq i, j \leq b \leq m\}$, and $BR_{i,j} = \max\{T_{a,b} \mid i \leq a \leq n, j \leq b \leq m\}$. Each of these can be computed in $O(nm)$ time similar to TL.

To calculate the tastiest dish Prof. Child can cook when she stands at cell (i, j) ($1 < i < n$ and $1 < j < m$), we now just need to compute the product $TL_{i-1,j-1} BL_{i+1,j-1} TR_{i-1,j+1} BR_{i+1,j+1}$ and pick the maximum product. This can be done in $O(nm)$ time.

Problem 7. Median of two sorted arrays. [20 points] (3 parts)

Finding the median of a sorted array is easy: return the middle element. But what if you are given two sorted arrays A and B, of size m and n respectively, and you want to find the median of all the numbers in A and B? You may assume that A and B are disjoint.

(a) [3 points] Give a naïve algorithm running in $\Theta(m + n)$ time.

Solution: Merge the two sorted arrays (which takes $O(m + n)$ time) and find the median using linear-time selection.

(b) [10 points] If $m = n$, give an algorithm that runs in $\Theta(\lg n)$ time.

Solution: Pick the median m_1 for A and median m_2 for B. If $m_1 = m_2$, return m_1. If $m_1 > m_2$, remove the second half of A and the first half of B. Then we get two subarrays with size $n/2$. Repeat until both arrays are smaller than a constant. $m_1 < m_2$ is symmetric.

(c) [7 points] Give an algorithm that runs in $O(\lg(\min\{m, n\}))$ time, for any m and n. *Don't spend too much time on this question!*

Solution: Without loss of generality, assume $|A| = m > n = |B|$. We can safely remove elements $A[0 : \frac{m-n}{2}]$ and $A[\frac{m+n}{2} : m-1]$ because none of these elements can be the median of $A + B$. After this process, we get two arrays of size approximately n. Then we can run part (b). The complexity is $\Theta(\lg(\min(m, n)))$