Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains **four** multi-part problems. You have 120 minutes to earn 120 points.
- This quiz booklet contains **18** pages, including this one. An extra sheet of scratch paper is attached.
- This quiz is closed book. You may use one handwritten A4 or $8\frac{1}{2}'' \times 11''$ crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress. Generally, a problem's point value is an indication of how much time 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. Be neat.
- Good luck!

Problem	Points	Grade	Initials
1	30		
2	35		
3	25		
4	30		
Total	120		

Name: Solutions

Circle your recitation:

Brian 11 Brian 12 Jen 12 Jen 1 Brian 2

Problem 1. True or False, and Justify [30 points] (7 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

(a) T F [4 points] A 2-3-4 tree is special case of a B-tree where leaves can have different depths.

Solution: False. In a 2-3-4 tree, all leaves have the same depth.

(b) T F [4 points] Searching on a skip list takes expected $\Theta(\log n)$ time, but could take $\Omega(n)$ time with non-zero probability.

Solution: True. A skip list could be of any height or be a simple linked list, depending on its random choices.

(c) T F [4 points] To determine whether two binary search trees on the same set of keys have identical tree structures, one could perform an inorder tree walk on both and compare the output lists.

Solution: False. An inorder tree walk will simply output the elements of a tree in sorted order. Thus, an inorder tree walk on any binary search tree of the same elements will produce the same output.

(d) T F [4 points] In an undirected weighted graph with distinct edge weights, both the lightest and the second lightest edge are in some MST.

Solution: True. First, since the edge weights are distinct there is only a single MST. Let e_1 and e_2 be the lightest and second lightest edge. In Kruskal's algorithm, e_1 is always the first edge added. Since e_2 cannot possibly create a cycle, it will necessarily be the next edge added by Kruskal's algorithm. So, by the correctness of Kruskal, the two lightest edges are always in the MST.

(e) T F [5 points] Dijkstra's algorithm works correctly on graphs with negative-weight edges, as long as there are no negative-weight cycles.

Solution: False. Consider the directed graph $V = \{A, B, C\}$ and $E = \{AB, AC, CB\}$. Let $w(AB) = 1$, $w(AC) = 2$ and $w(CB) = -2$. Dijkstra will first add the edge AB to the shortest path tree. However, the shortest path tree would be $T = \{AC, CB\}$.

(f) T F [5 points] In a graph $G = (V, E)$, suppose that each edge $e \in E$ has an integer weight $w(e)$ such that $1 \leq w(e) \leq n$. Then there is a an $o(m \log n)$-time algorithm to find a minimum spanning tree in G.

Solution: True. Run Kruskal using Counting sort. Sorting edges will take $O(m + n) = O(m)$. The remainder of Kruskal's algorithm will take $O(m\alpha(m, n))$ time, which is $o(m \log n)$.

(g) T F [4 points] In the comparison model, there is an $\Omega(m \log n)$ lower bound on performing m UNION and FIND-SET operations in any disjoint-set data structure storing n total elements.

Solution: False. Using the Union-Find data structure from class, we can perform m operations on a set of initial size n in $\Theta(m\alpha(m, n))$.

Problem 2. Short Answer [35 points] (3 parts)

Give *brief*, but complete, answers to the following questions.

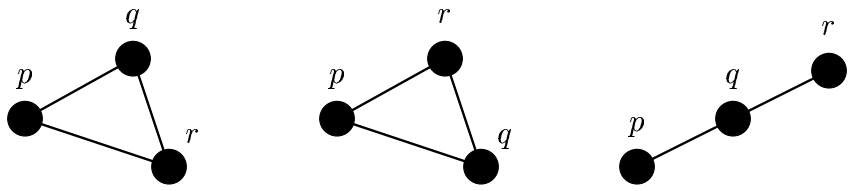
(a) [10 points] We wish to define a universal hash family $\mathcal{H} = \{h_1, h_2, h_3\}$ where $U = \{a, b, c\}$ and $h_i : U \to \{0, 1\}$. Complete the following table to make $\mathcal{H}$ universal. The function h_1 has already been defined for you:

	a	b	c
h_1	0	0	1
h_2			
h_3			

Solution:
We need to ensure that none of the three pairs (a, b), (a, c) or (b, c) collide with more than $1/2$ probability. Since a and b already collide once, we must have that $h_2(a) \neq h_2(b)$ and $h_3(a) \neq h_3(b)$. Then they will collide with probability $1/3$. The value c can collide with each of a and b exactly once, so we can have $h_2(a) = h_2(c)$ and $h_2(b) = h_2(c)$. One valid solution is:

	a	b	c
h_1	0	0	1
h_2	0	1	0
h_3	0	1	1

(b) [10 points] Suppose that you are given a set $\{p_1, p_2, \ldots, p_n\}$ of n points in two dimensions. Give an $O(n^2 \lg n)$-time algorithm to detect whether any three points are *collinear*, that is, whether any three points lie on a common line. You may assume that you have a subroutine that computes in $O(1)$ time whether three given points p, q, r are oriented clockwise, oriented counterclockwise, or collinear, as shown in the figure below. (Such an "orientation test" subroutine was given in lecture.) You may not use hashing.



clockwise **counterclockwise** **collinear**

Solution:

for each point x:
  Use the orientation test as a comparator to sort the other $n - 1$ points radially with respect to x.
  If any two points are collinear with respect to x, **return** True.
return False

As shown in class, we will use the clockwise/counterclockwise/collinear test as a less/greater/equal comparator. We can sort the points with respect to a given x in $O(n \log n)$ time. If any triplet is collinear, it will be detected and the loop will halt. Since we loop through every possible x, this algorithm will take $O(n^2 \log n)$ time.

(c) [15 points] You are given three bags of steel pipes, denoted A, B and C, such that each bag contains n pieces of pipe with lengths in the range $[0, 5n]$. You may assume that each piece of pipe has a unique integer length. You may attach a single pipe $a \in A$ to a single pipe $b \in B$ and to a single pipe $c \in C$ to produce a pipe with length $a + b + c$.

Give an $O(n \log n)$-time algorithm to determine both:

- Every possible length that may be obtained by attaching a triplet of pipes from A, B and C.
- The number of triplets that could be attached to produce a given length.

Solution:
Represent A, B and C as polynomials of degree $5n$ as follows:

$$A(x) = x^{a_1} + x^{a_2} + \ldots + x^{a_n}, B(x) = x^{b_1} + x^{b_2} + \ldots + x^{b_n}, C(x) = x^{c_1} + x^{c_2} + \ldots + x^{c_n},$$

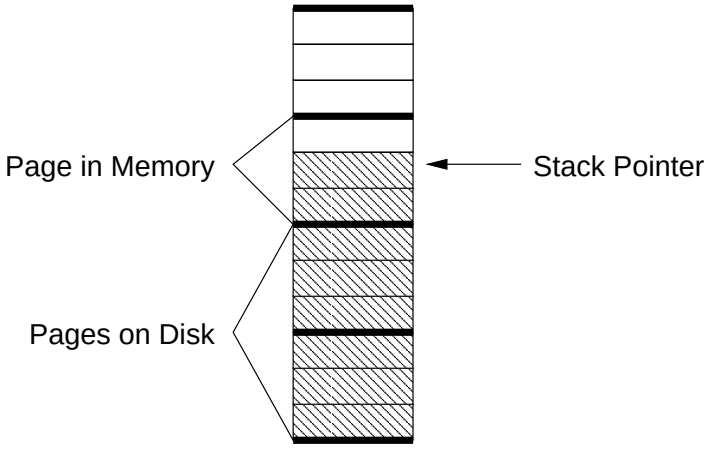
Multiply A, B, and C in time $\Theta(n \log n)$ using the FFT to obtain a coefficient representation $d_0, d_1, \ldots, d_{3n}$. In other words $D(x) = d_0 + d_1 x + d_2 x^2 + \ldots + d_{3n} x^{3n}$. Each triplet a_i, b_j, c_k will account for one term $x^{a_i} x^{b_j} x^{c_k} = x^{a_i + b_j + c_k} = x^m$. Therefore, d_m will be the number of such pairs $a_i + b_j + c_k = m$.

Problem 3. Amortized Stack Analysis [25 points] (2 parts)

Recall the standard implementation of a stack in an array. A *stack pointer* stores the address of the top element of the stack. The PUSH(x) operation increments the stack pointer, then stores a new element x at the top of the stack. The POP operation returns the element at the top of the stack, then decrements the stack pointer.

In this problem, we wish to implement a stack on a computer with a small amount of fast memory and a large amount of slow disk space. The disk space is partitioned into p-element clusters called *pages*. The computer can read or write individual elements in memory, but can read or write only entire pages on disk. Reading or writing an element in memory takes $\Theta(1)$ time while reading or writing a page on disk takes $\Theta(p)$ time.

Consider a stack that stores the top page of the stack array in memory and the remainder on disk, as in the following diagram:



In this diagram, each page has $p = 3$ words. Whenever the stack pointer crosses a page boundary (indicated by the thick lines) there is a disk operation that incurs $\Theta(p)$ cost.

(a) [10 points] What is the worst-case running time to perform n stack operations using this implementation? Express your solution in terms of n and p throughout this problem.

Solution: Consider the alternating sequence

PUSH, PUSH, POP, POP . . .

occurring at a page boundary. The second PUSH requires writing the first word of the next page, and the second POP requires reading in the previous page again.
Thus, n operations can make $\Theta(n)$ disk accesses, taking a total of $\Theta(np)$ time.

(b) [15 points] Suppose you are allowed to store two stack pages in memory. Implement PUSH and POP so that the amortized running time for any stack operation is $O(1)$. You may assume that you have $O(1)$ extra memory for bookkeeping.

Solution:

Keep the page currently needed in memory, as in part (a), but also keep the previously used page. Keep a bit marking which of the two pages in memory has been least recently used (LRU). Whenever a new page must be read in, save the LRU page to disk if modified, and read in the new page. When executing many PUSH operations, memory will first be p_0 (only one page valid), then p_0p_1 (while doing the $(p+1)$st through $(2p)$th pushes), then p_2p_1, and then p_2p_3.

The stack pointer will point to the top of any fresh page read from disk. In other words, there is p data and p free space available in memory for our stack operations. We must either perform p PUSH or p POP operations before the next page must be read.

Therefore each disk access must be preceded by p stack operations. Using the accounting method, pay $1/p$th of our disk access cost, i.e. $\Theta(1)$ for every stack operation. After p stack operations, we will have accumulated $\Theta(p)$ value that can pay for any necessary disk accesses. Therefore, stack operations have a $\Theta(1)$ amortized cost.

Problem 4. Package Shipping [30 points] (3 parts)

You work for a small manufacturing company and have recently been placed in charge of shipping items from the factory, where they are produced, to the warehouse, where they are stored. Every day the factory produces n items which we number from 1 to n in the order that they arrive at the loading dock to be shipped out. As the items arrive at the loading dock over the course of the day they must be packaged up into boxes and shipped out. Items are boxed up in contiguous groups according to their arrival order; for example, items $1 \ldots 6$ might be placed in the first box, items $7 \ldots 10$ in the second, and $11 \ldots 42$ in the third.

Items have two attributes, *value* and *weight*, and you know in advance the values $v_1 \ldots v_n$ and weights $w_1 \ldots w_n$ of the n items. There are two types of shipping options available to you:

Limited-Value Boxes: One of your shipping companies offers insurance on boxes and hence requires that any box shipped through them must contain no more than V units of value. Therefore, if you pack items into such a "limited-value" box, you can place as much weight in the box as you like, as long as the total value in the box is at most V.

Limited-Weight Boxes: Another of your shipping companies lacks the machinery to lift heavy boxes, and hence requires that any box shipped through them must contain no more than W units of weight. Therefore, if you pack items into such a "limited-weight" box, you can place as much value in the box as you like, as long as the total weight inside the box is at most W.

Please assume that every individual item has a value at most V and a weight at most W. You may choose different shipping options for different boxes. Your job is to determine the optimal way to partition the sequence of items into boxes with specified shipping options, so that shipping costs are minimized.

(a) [10 points] Suppose limited-value and limited-weight boxes each cost \$1 to ship. Describe an $O(n)$ greedy algorithm that can determine a minimum-cost set of boxes to use for shipping the n items. Justify why your algorithm produces an optimal solution.

Solution:

We use a greedy algorithm that always attempts to pack the largest possible prefix of the remaining items that still fits into some box, either limited-value or limited-weight. The algorithm scans over the items in sequence, maintaining a running count of the total value and total weight of the items encountered thus far. As long as the running value count is at most V or the running weight count is at most W, the items encountered thus far can be successfully packed into some type of box. Otherwise, if we reach a item j whose value and weight would cause our counts to exceed V and W, then prior to processing item j we first package up the items scanned thus far (up to item $j-1$) into an appropriate box and zero out both counters. Since the algorithm spends only a constant amount of work on each item, its running time is $O(n)$.

Why does the greedy algorithm generate an optimal solution (minimizing the total number of boxes)? Suppose that it did not, and that there exists an optimal solution different from the greedy solution that uses fewer boxes. Consider, among all optimal solutions, one which agrees with the greedy solution in a maximal prefix of its boxes. Let us now examine the sequence of boxes produced by both solutions, and consider the first box where the greedy and optimal solutions differ. The greedy box includes items $i \ldots j$ and the optimal box includes items $i \ldots k$, where $k < j$ (since the greedy algorithm always places the maximum possible number of items into a box). In the optimal solution, let us now remove items $k+1 \ldots j$ from the boxes in which they currently reside and place them in the box we are considering, so now it contains the same set of items as the corresponding greedy box. In so doing, we clearly still have a feasible packing of items into boxes and since the number of boxes has not changed, this must still be an optimal solution; however, it now agrees with the greedy solution in one more box, contradicting the fact that we started with an optimal solution agreeing maximally with the greedy solution.

(b) [20 points] Suppose limited-value boxes cost $\$C_v$ and limited-weight boxes cost $\$C_w$. Give an $O(n^2)$ algorithm that can determine the minimum cost required to ship the n items, and briefly justify its correctness.

Solution:

We use dynamic programming. Let $A[j]$ denote the optimal cost for packing just items $j \ldots n$ into boxes. We compute the sequence of subproblem solutions $A[n] \ldots A[1]$ in reverse order as follows (we take $A[n+1] = 0$ as a base case). For every item i, let $V(i)$ be the largest item index such that $v_i + v_{i+1} + \ldots + v_{V(i)} \leq V$, and let $W(i)$ be the largest item index such that $w_i + w_{i+1} + \ldots + w_{W(i)} \leq W$. We can compute $V(i)$ and $W(i)$ for a item i by scanning forward from i and maintaining a running count of the total value and weight from item i onward. Applying this to every item, we can compute $V(i)$ and $W(i)$ for all items in $O(n^2)$ time. We can now easily compute solutions to our DP subproblems via the following formula:

$$A[i] = \min(A[V(i)+1] + C_v, A[W(i)+1] + C_w)$$

The two cases above correspond to the decision of whether or not to use a limited-value or limited-weight box as the first box when packing items $i \ldots n$ in sequence. Regardless of the type of box we select, we clearly want to place as many items as possible in that box — this follows from the greedy proof above. The DP algorithm requires only $O(n)$ time after computing the $V(i)$'s and $W(i)$'s, which requires $O(n^2)$ time.

Some student suggested a greedy solution in this part: find the largest prefices for each the limited-value box and limited-weight box, and choose the option that results in the lowest cost per item. However, this greedy strategy does not give an optimal solution in the following example.

Suppose $W = 5, V = 5, C_v = 2$ and $C_w = 5$. Consider the 5 items with the following (v_i, w_i) values:

$$(5,3), (5,2), (3,2), (1,1), (1,2)$$

The greedy algorithm will pick VLB, WLB and VLB in the above order, costing a total of \$9. On the other hand, the optimal solution should choose a WLB followed by a VLB, with a total cost of \$7. Therefore, the greedy algorithm does not always give an optimal solution.

(c) [3 points] **(Extra Credit)** Can you reduce the running time of the algorithm from (b) to $O(n)$?

Solution:

We can speed up the algorithm above by computing the $V(i)$'s and $W(i)$'s in $O(n)$ total time. Note that $V(i)$ and $W(i)$ are both monotonically non-decreasing functions of i. Therefore, as we scan forward for $i = 1 \ldots n$, the values of $V(i)$ and $W(i)$ can only move forward as well. The following pseudocode illustrates how we can exploit this monotonicity to speed up the total computation:

```
1   V(0) ← 0, TotalValue ← 0
2   W(0) ← 0, TotalWeight ← 0
3   for i ← 1 to n:
4       V(i) ← V(i − 1), TotalValue ← TotalValue − v_{i−1}
5       W(i) ← W(i − 1), TotalWeight ← TotalWeight − w_{i−1}
6       while TotalValue + v_{V(i)+1} ≤ V:
7         V(i) ← V(i) + 1, TotalValue ← TotalValue + v_{V(i)}
8       while TotalWeight + w_{W(i)+1} ≤ W:
9         W(i) ← W(i) + 1, TotalWeight ← TotalWeight + w_{W(i)}
```

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER