Quiz 2

You will have 2 hours to complete this exam. No notes or other resources are allowed. Unless otherwise specified, full credit will only be given to a correct answer which is described clearly and concisely.

Do not discuss this exam with anyone who has not yet taken it.

Problem	Points	Grade	Initials
Name	1		
1	24		
2	18		
3	12		
4	25		
5	20		
Total	100		

Name: [1 point] _____

R01	R02	R03	R04	R05	R06	R07
WF10	WF11	WF12	WF1	WF2	WF3	WF3
Shaunak	Shaunak	Alan	Jeff	Rafael	Henrique	Dragos

Problem 1. True/False [24 points]

Note: Correct answers are worth 2 points, blanks are worth 0 points, and incorrect answers are worth -3 points. You will not be graded on any explanation.

(a) Depth-first search can be modified to check if there are cycles in an undirected graph.

True

(b) Breadth-first search can be modified to check if there are cycles in an undirected graph.

True

(c) If we represent a graph with $|V|$ vertices and $\Theta(|V|)$ edges as an adjacency matrix, the worst-case running time of breadth-first search is $\Theta(|V|^2)$.

True

(d) In this problem, suppose that G is a directed graph and that u and v are vertices of this graph such that there is a path from u to v in G but no path from v to u.

i. Any depth-first search in G that discovers both u and v must discover u before it discovers v. [1]

False

ii. Any depth-first search in G that discovers both u and v must finish u before it finishes v. [2]

False

iii. Any depth-first search in G that discovers u and ***later*** discovers v must finish u before it finishes v.

False

[1] In the terminology of CLRS, the discovery time is the time it is colored grey.

[2] The finishing time is, in the terminology of CLRS, the time in which a vertex is colored black.

(e) The strongly-connected components of a directed graph are preserved if you reverse every edge — that is, if you replace every edge (u, v) of the graph with the edge (v, u).[3]

True

(f) We can use Dijkstra's algorithm to find the shortest path between two vertices in a graph with arbitrary edge weights.

False

(g) The worst-case running time of A* is asymptotically better than the worst-case running time of Dijkstra's algorithm.

False

(h) Suppose that s and t are vertices in a weighted graph G that does not contain negative cycles, and suppose that there is a path from s to t. We run Bellman-Ford on G with starting vertex s.

i. If there is a shortest path from s to t consisting of k edges, then after the k^{th} iteration, then Bellman-Ford's estimate of the distance to t will be correct.

True

ii. If Bellman-Ford's estimate of the distance to t is correct after the k^{th} iteration, then there is a shortest path from s to t consisting of at most k edges.

False

(i) If we draw out the full recursion tree of a problem that can be sped up by memoization, the same subproblem might appear multiple times in the tree.

True

[3]Recall that u and v are in the same strongly connected component if there is a path from u to v and a path from v to u.

Problem 2. Short Answers [18 points]

(a) [6 points] Mark the entries of the following table that correspond to properties that are true of counting and radix sort, as described in lecture.

Solution:

Property	Counting sort	Radix sort
Can be implemented so it is stable	X	X
Can be implemented so it is in-place		
Sorts n integers in the range $\{0, 1, \ldots, n^c\}$ in $O(n)$ time, for any constant $c > 0$.		X

Note: We gave everyone full credit for the in-place question. This is because, while what we did in classl

(b) [8 points] On which of the following undirected graphs does bi-directional breadth-first search perform asymptotically better than regular breadth-first search? Circle the numbers of **all that apply**.

i. A path graph on n vertices, in which s and t are connected by a path of length $n - 1$ (and there are no other edges).

ii. A complete graph, in which there is an edge between every pair of vertices.

ⓘⓘ. A star graph, in which s, t, and $n - 3$ other vertices are all connected to a central nth vertex (and there are no other edges).

ⓘⓥ. A balanced binary tree on n vertices in which s and t are leaves.

(c) [4 points] Ben Bitdiddle thinks it is possible to find s-t shortest paths on any weighted graph using the following algorithm:

1. Find the minimum weight, m, of any edge.
2. Subtract m from the weight of every edge - that is, let $w'(i, j)$ equal $w(i, j) - m$.
3. Run Dijkstra on the transformed graph.

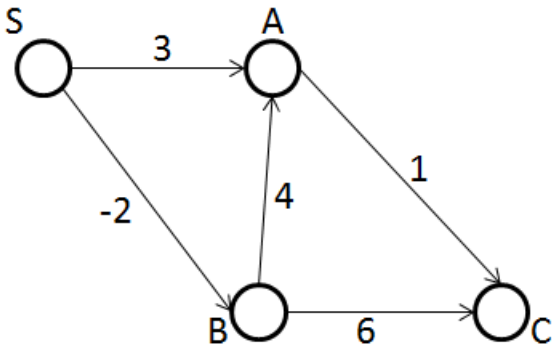
Draw a three-vertex directed graph with vertices s, t, and u on which Ben's algorithm does **not** find the shortest path from s to t. Label the vertices and assign **non-negative** weights to the edges to construct your counterexample.

Solution:

Any graph where the shortest $s \to t$ path goes directly from s to t,, yet goes through u after transforming. For example, have $w(s, u) = w(u, t) = 2$ and $w(s, t) = 3$.

Problem 3. Bellman-Ford [12 points]

In this problem, you must run Bellman-Ford manually on the directed graph provided below, starting at the source vertex S. In each iteration, the edges will be relaxed in the following order: BC, AC, BA, SA, and SB.



(a) [8 points] Fill in the table with the distance estimates for each vertex after each iteration. Note that **all** the edges are relaxed in each iteration. For example, after the first iteration, you should find that the distance estimate for B is -2.

Solution:

Vertex	Iteration 0	Iteration 1	Iteration 2	Iteration 3
S	0	0	0	0
A	∞	3	2	2
B	∞	-2	-2	-2
C	∞	∞	4	3

(b) [4 points] In the worst case, the Bellman-Ford algorithm runs for $|V| - 1$ iterations, where $|V|$ is the number of vertices. However, for this particular graph, there exists an ordering of the edges such that for **any** edge weights, the Bellman-Ford algorithm will terminate after a **single** iteration.

Give one such edge ordering, and briefly explain why it works.

Solution: Since the graph is a DAG, we can take any ordering which will process all of the edges of a path in the correct order. In particular, a sequence works if and only if for every node, the incoming edges are relaxed before any of the outgoing edges. There are actually many such sequences.

Problem 4. Optimal Travel Plans [25 points]

In this problem, your goal is to determine a sequence of flights between airports which will get you from your current location to a target destination as quickly as possible.

Each airport is represented by a vertex on a directed graph $G = (V, E)$. Each directed edge $e = (u, v)$ in the graph has an associated array, $e.flights$. The array $e.flights$ has at most k entries. The i^{th} entry in the array is a pair (dep_i, arr_i), which means that a direct flight leaves u at time dep_i and arrives at v at time arr_i.

For each edge e, the array $e.flights$ satisfies the following constraints:

1. No flight can arrive before it departs:

   $$\text{For each } i,\ dep_i < arr_i.$$

2. The array is sorted by increasing departure time:

   $$\text{For each } i,\ dep_i < dep_{i+1}.$$

3. Flights that depart later also arrive later:

   $$\text{For each } i,\ arr_i < arr_{i+1}.$$

Given a source node s and an initial starting time, your goal is to determine a sequence of flights that can be taken to reach a target node t as early as possible. Of course, in order for you to take a flight, you must be at airport that the flight departs from by the time that it departs.

Continue to the next page.

(a) [20 points] Design and analyze an efficient algorithm to compute a sequence of flights which arrives at t as early as possible. Be sure to explicitly state your algorithm's running time in terms of $|V|$, $|E|$, and k.

Note: You will receive 6 points for a blank answer to this question. You will get more than 6 points for progress towards a correct solution, but any other text will count against you.

Solution:

We solve this problem by running a slightly modified variant of Dijkstra's algorithm (using a fibonacci heap, for optimal running time.) In our algorithm, for each node v we will keep track of $d[v]$ which is earliest time for which we know it is possible to arrive at v, when we leave from s at $t = 0$. (We initialize $d[v] = \infty$ for all $v \neq s$ and $d[s] = 0$.) We now run Dijkstra's algorithm using these d values. The only difference is in how we relax an edge.

Consider relaxing edge $e = (u, v)$. Because of the properties of e.flights, we know that leaving u for v as early as possible will be at least as good as having a longer layover in u and leaving for the direct $u \to v$ flight later. Therefore, we look for the smallest i^* such that the corresponding depart_{i^*} in e.flights is greater than or equal to $d[u]$. We can use binary search to find this i in time $O(\log k)$.

To relax the edge $e = (u, v)$, we set $d[v]$ to be the minimum of the current $d[v]$ and $d[v] + \text{arrive}_{i^*}$.

The correctness of this algorithm follows from the correctness of Dijkstra's algorithm. The running time, using a fibonacci heap for the Dijkstra priority queue, is $O(V \log V + E \log k)$. (The $O(\log k)$ term comes from needing to find the earlier flight to take on a given edge.)

Partial credit was given for the $O(V \log V + kE)$ solution which did a linear scan through the departure times. Also, note that using binary heaps instead of a fibonacci heap gives running time $O(\log kE \log V)$.

Note that this problem was a modified (harder) version of an idea from

$http://www.csl.mtu.edu/cs2321/www/newLectures/30\_More\_Dijkstra.htm$

Notes on grading:

- **Getting runtimes using binary heaps instead of Fibonacci heaps lost you no points.**
- **Not getting the binary search step caused you to lose 2 points.**
- **There were other solutions which involved modifying the graph, which got a worse running-time, and thus 16 or 12 points, depending on whether the transformation obtains Ek or VEk edges in the new graph.**

(b) [5 points] Suppose now that we now have additional geographic information about the graph. In particular, we model the Earth as a plane, and we determine the location (x_u, y_u) of each airport in the plane. We also know that the speed of any plane is bounded by a top speed, s.

Explain how to use a heuristic to speed up your search algorithm in practice. Be sure to explicitly define any heuristic functions that you use.

Note: Use at most four sentences. You should only need half of this page.

Solution: We can use a heuristic of $h(u) = \sqrt{(x_u - x_t)^2 + (y_u - y_t)^2}/c$, which is clearly a lower bound on the minimum time it will take to get from u to t. We modify our Dijkstra algorithm analogously to A^*: instead of sorting our queue by d values, we sort by $d + h$ values.

Problem 5. Longest Football Subsequence [20 points]

In the game of football, teams can score 2, 3, or 7 points at a time. A *football sequence* is a sequence of valid scores for the two teams in a football game. That is, it is a sequence of pairs of nonnegative integers (a_i, b_i) that satisfies the following properties:

1. The initial scores are 0:

$$(a_0, b_0) = (0, 0).$$

2. Exactly one team scores at a time:

$$\text{For all } i, \text{ either } a_{i+1} = a_i \text{ or } b_{i+1} = b_i, \text{ but not both.}$$

3. Teams score in the correct increments:

$$\text{If } a_{i+1} \neq a_i, \text{ then } a_{i+1} - a_i \text{ is either 2, 3 or 7, and}$$

$$\text{If } b_{i+1} \neq b_i, \text{ then } b_{i+1} - b_i \text{ is either 2, 3 or 7}$$

For example, the following sequence is a football sequence:

$$(0, 0), (0, 3), (0, 5), (7, 5), (7, 12), (9, 12).$$

In this problem, your goal is to determine the length of the longest football subsequence of a given n-element sequence S of pairs of nonnegative integers (x_i, y_i). (Note that your subsequence may include non-consecutive elements of S, as long as their relative order is preserved.)

For example, if

$$S = (2, 7), (0, 0), (7, 0), (0, 3), (0, 6), (7, 6)$$

then the longest football subsequence is

$$(0, 0), (0, 3), (0, 6), (7, 6),$$

so your algorithm should return 4.

Continue to the next page.

(a) [10 points] Give a simple dynamic programming algorithm which finds the size of the largest football subsequence in time $O(n^2)$. Briefly prove your algorithm's runtime and argue its correctness.

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

Solution:

Let $C[i]$ be the length of the longest football subsequence which ends at $S[i]$. We now do a scan of S from left to right to compute the C values. To compute $C[i]$, we need only look at the values between $C[1], \ldots, C[i-1]$ and look at the maximum value $C[j]$ for which we could append $S[i]$ after $S[j]$. The correctness of this algorithm follows trivially by induction: Given that we have computed the first $i-1$ values of C correctly, it follows that any football subsequence ending with $S[i]$ is constructed by taking a football subsequence ending before i and (if valid) appending $S[i]$ to the end. (If $S[i] = (0,0)$, then we can instead have the length-1 subsequence consisting of $S[i]$. Since football subsequences are increasing, $(0,0)$ can only appear at the beginning of a subsequence.)

This gives us the following pseudocode algorithm:

- Initialize $C[i] = -\infty$ for all i
- For $i = 1$ to n:
  - Let c^* be the maximum $C[j]$ value for $j < i$ such that $S[i]$ can immediately follow $S[j]$ in a valid football sequence (that is, the first components are equal and the second component increases by 2, 3, or 7, or instead the second components are equal and the first component increases by 2, 3, or 7.) If no such j exists, set c^* to $-\infty$.
  - If $S[i] == (0,0)$, set $C[i] \leftarrow 1$.
  - Otherwise, set $C[i] \leftarrow c^* + 1$.
- Return $\max\{\max_i C[i], 0\}$.

Notice that in the final step, we return $\max\{\max_i C[i], 0\}$. This deals with the case that $(0,0)$ does not appear in S.

Our algorithm has running time $O(n + 1 + 2 + 3 + \cdots + (n-1)) = O(n^2)$.

(b) [10 points] Design and analyze the most efficient algorithm you can for this problem. Be sure to explicitly state your algorithm's running time in terms of n, and briefly argue its correctness.

Note: You will receive 3 points for a blank answer to this question. You will get more than 3 points for progress towards a correct solution, but any other text will count against you.

Solution:

We can solve this problem by doing a single linear scan through S. As before, we let $C[i]$ be the length of the longest football subsequence ending at $S[i]$. We note that, for $S[i]$ to appear in a sequence, there are at most 6 possible terms that could immediately proceed $S[i]$ (corresponding to subtracting 2, 3, or 7 from either the first or second component.)

As we scan S, we will hash each pair (a, b) for which we have found an j with $S[j] = (a, b)$ and $C[j] > 0$. The value of (a, b) will be the length of the longest football subsequence we have found thus far which ends at (a, b). We will compute $C[i]$ by searching for all six possible proceeding (a, b) pairs in the hash table, and setting $C[i]$ to be 1 more than the maximum of the corresponding C values for these six pairs. We give pseudocode for this algorithm below. Correctness follows from a trivial induction argument, similar to that above.

- Create a hash table d.
- For $i = 1$ to n:
  - Denote by (a, b) the term $S[i]$.
  - If $(a, b) == (0, 0)$, set $C[i] \leftarrow 1$ and $d[(0, 0)] \leftarrow 1$.
  - Else:
    - Let P be the set of the (at most 6) pairs which could possibly proceed (a, b) in a football sequence. (i.e., P consists of the terms $(a - 7, b), (a - 3, b), (a - 2, b), (a, b - 7), (a, b - 3), (a, b - 2)$ in which both values are nonnegative.)
    - If there exists a $p \in P$ for which $d.has\_key(p)$, let c^* be $\max_{p \in P} d[p]$ (if any p is not in d, initialize its value to $-\infty$). Set $C[i] \leftarrow c^*$ and $d[(a, b)] \leftarrow \max\{d[(a, b)], c^* + 1\}$.
    - Otherwise, set $c[i] \leftarrow -\infty$.
- Return $\max\{\max_i C[i], 0\}$.

By resizing our hash table appropriately, the amortized running time for our hash operations will be $O(1)$ per operation. Therefore, each iteration of the inner loop takes $O(1)$ time, and therefore the overall running time is $O(n)$. (Notice that there are several slight variations of this algorithm which also achieve $O(n)$ running time.)