CS 161 (Stanford, Winter 2023) Homework 5

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

1 Exercise: Universality

Plucky the penguin is hosting a free food event for up to 100 Stanford students! To keep track of RSVP's, Plucky wants to build a hash table with 100 buckets using the attendees' 8-digit student ID as keys.

Plucky still needs to choose a hash family $\mathcal{H} = \{h_m : m \in \{1, \ldots, 1000\}\}$, and being a pedantic penguin, wants it to be universal.

Does each of the following formulations result in a universal hash family?

[We are expecting: For each candidate formulation, a proof of universality or a counterexample.**]**

1.1 (2 pt.)

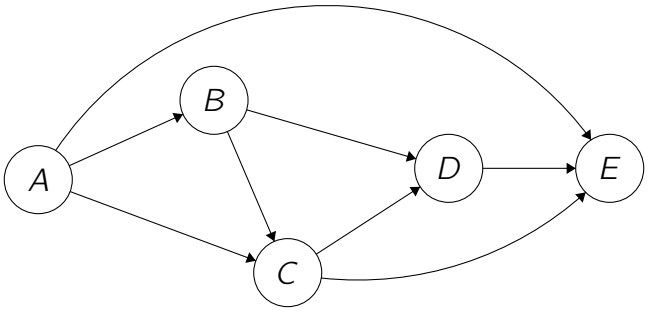
Let $h_m(x)$ be the sum of each digit in input x, plus m, truncated to the final two digits. For example, $h_{100}(01234567) = 28$ because $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 100 = 128$.

1.2 (2 pt.)

Let $h_m(x)$ be the final two digits of mx. For example, $h_4(01234567) =$ last two digits of $4938268 = 68$.

2 Exercise: BFS and DFS Basics

Consider the following directed acyclic graph (DAG):



2.1 (2 pt.)

Run DFS starting at vertex C, breaking any ties by alphabetical order. (For example, if DFS has a choice between B or C, it will always choose B. This includes when DFS is starting a new tree in the DFS forest.) Recall that when you run DFS, if it reached a node with no children (i.e. cant go any further), then it will resume the search at an unvisited vertex.

(a) What do you get when you order the vertices by **ascending** start time?

(b) What do you get when you order the vertices by **descending** finish time?

[We are expecting: An ordering of vertices. No justification is required.**]**

2.2 (2 pt.)

Run DFS starting at vertex D, *this time treating all edges as undirected*. Once again, break any ties by alphabetical order.

(a) What do you get when you order the vertices by **ascending** start time?

(b) What do you get when you order the vertices by **descending** finish time?

[We are expecting: A pair of orderings of vertices. No justification is required.**]**

2.3 (1 pt.)

Run BFS *(not DFS)* starting at vertex D, *treating all edges as undirected*. Break any ties by alphabetical order. What is the order that the nodes are marked by BFS?

[We are expecting: An ordering of vertices. No justification is required.**]**

Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

3 Finding shortest paths for special graphs

3.1 Graphs with few negative edges (5 pt.)

Let $G = (V, E)$ be a directed weighted graph, and $s \in V$ a vertex. Suppose there are no negative-weight cycles, and only the edges going out of s may have negative weights.

As seen in class, Bellman-Ford can solve the shortest path problem on any graph (without negative cycles) including the graph above. However, since the given graph is special and only a specific set of edges may be negative, modify the graph suitably to use a faster algorithm from lecture to solve the shortest paths from s on this graph?

[We are expecting: Algorithm name, English description stating the steps to modify the graph and a concise but clear explanation for correctness.**]**

3.2 Graphs with small integer weights (5 pt.)

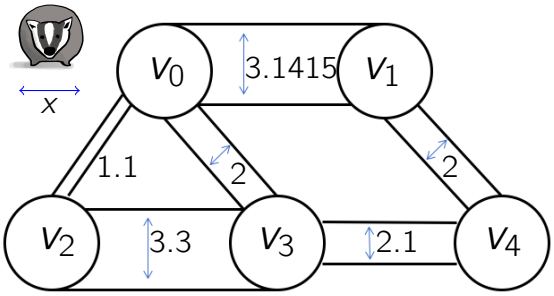
Let $G = (V, E)$ be a directed weighted graph, where all the edge weights are from $\{1, 2, 3\}$. Design an algorithm that modifies the graph suitably and solves the shortest path problem for the given graph asymptotically faster than the algorithm you used in the previous part. Mention the time complexity of both the algorithms.

[We are expecting: Short English description stating the steps to modify the graph, algorithm name (from class) to solve shortest path problem on the modified graph, time complexity of the proposed algorithm along with a short justification, and time complexity of the algorithm from previous part. (No pseudocode necessary)**]**

4 Badger badger badger

A family of badgers lives in a network of tunnels; the network is modeled by a connected, undirected graph G with n vertices and m edges (see below). Each of the tunnels have different widths, and a badger of width x can only pass through tunnels of width $\geq x$.

For example, in the graph below, a badger with width $x = 2$ could get from v_0 to v_4 (either by $v_0 \to v_1 \to v_4$ or by $v_0 \to v_3 \to v_4$). However, a badger of width 3 could not get from v_0 to v_4.



The graph is stored in the adjacency-list format we discussed in class. More precisely, G has vertices $v_0, \ldots, v_{n-1}$ and is stored as an array V of length n, so that $V[i]$ is a pointer to the head of a linked list N_i which stores integers. An integer $j \in \{0, \ldots, n-1\}$ is in N_i if and only if there is an edge between the vertices v_i and v_j in G.

You have access to a function `tunnelWidth` which runs in time $O(1)$ so that if $\{v_i, v_j\}$ is an edge in G, then `tunnelWidth(i,j)` returns the width of the tunnel between v_i and v_j. (Notice that `tunnelWidth(i,j)`=`tunnelWidth(j,i)` since the graph is G undirected). If $\{v_i, v_j\}$ is not an edge in G, then you have no guarantee about what `tunnelWidth(i,j)` returns.

4.1 Is there a path for a given badger? (3 pt.)

Design a deterministic algorithm which takes as input G in the format above, integers $s, t \in \{0, \ldots, n-1\}$, and a desired badger width $x > 0$; the algorithm should return **True** if there is a path from v_s to v_t that a badger of width x could fit through, or **False** if no such path exists.

(For example, in the example above we have $s = 0$ and $t = 4$. Your algorithm should return **True** if $0 < x \leq 2$ and **False** if $x > 2$).

Your algorithm should run in time $O(n + m)$. You may use any algorithm we have seen in class as a subroutine.

Note: In your pseudocode, make sure you use the adjacency-list format for G described above. For example, your pseudocode should *not* say something like "iterate over all edges in the graph." Instead it should more explicitly show how to do that with the format described.

[We are expecting: Pseudocode **AND** an English description of your algorithm, and a short justification of the running time. You should make sure to use the adjacency-list representation of G described above in your pseudocode. You can use any algorithms we have seen from class as a subroutine.**]**

4.2 Find the largest fitting badger (3 pt.)

Design a deterministic algorithm which takes as input G in the format above and integers $s, t \in \{0, \ldots, n-1\}$; the algorithm should return the largest real number x so that there exists a path from v_s to v_t which accommodates a badger of width x. Your algorithm should run in time $O((n+m)\log(m))$. You may use any algorithm we have seen in class as a subroutine. (Hint, use part (a)).

Note: Don't assume that you know anything about the tunnel widths ahead of time. (e.g., they are not necessarily bounded integers).

Note: In your pseudocode, make sure you use the adjacency-list format for G described above. For example, your pseudocode should *not* say something like "iterate over all edges in the graph." Instead it should more explicitly show how to do that with the format described.

[We are expecting: Pseudocode **AND** an English description of your algorithm, and a short justification of the running time. You should make sure to use the adjacency-list representation of G described above in your pseudocode. You can use any algorithms we have seen from class as a subroutine.**]**

4.3 Ethics (5 pt.)

Suppose you want to design a network of tunnels that can accommodate the widest of badgers. City planners do something similar: the tunnels are roads, and the badgers represent traffic. Imagine you are tasked with designing a road system that avoids congestion by optimizing for the widest roads possible to accommodate the heaviest amount of traffic the route can get. However, you realized that wider roads actually do not help with traffic congestion (see Building Bigger Roads Actually Makes Traffic Worse). What are some other considerations that are overlooked when we optimize the roads for cars? Here are a few articles for some inspiration:

- Speed kills, so why do we keep designing for it?
- Places and non-places
- Life in the Slow Lane
- Widening Highways Doesn't Really Help Traffic

[We are expecting: four to six sentences explaining (1) what other groups of people who share the road we overlook when we only consider car users; (2) what are the consequences they face when traversing a city planned for cars; and (3) what are the consequences that everyone faces when car traffic is encouraged.**]**

5 Perfect hashing

Ollie the overachieving ostrich has just read about hash tables and wants to learn more!

Recall from lecture 8 that a hash table supports the following operations:

- INSERT(k): Insert key k into the hash table.
- SEARCH(k): Check if key k is present in the table.
- DELETE(k): Delete the key k from the table.

For simplicity, Ollie is examining *static hash tables*, a more restricted problem where we know all the keys to be inserted ahead of time. Specifically, a static hash table supports the following operations:

- BUILD($k_1, \ldots, k_n$): Construct a static hash table from a set of n unique keys.
- SEARCH(k): Check if key k is present in the table.

To distinguish static hash tables from the more general hash tables presented in lecture, we will refer to the latter as *dynamic hash tables* for the remainder of this problem.

Notes from the pedantic penguin: For this problem you can assume:

- You have access to a universal hash family $\mathcal{H}$ with a size greater than the number of possible keys.
- Hash functions in family $\mathcal{H}$ are independent of each other.
- INSERT(k) runs in deterministic $O(1)$ time for dynamic hash tables, as long as the key being inserted is guaranteed to be unique. (If the key is unique, we can just append it to a bucket without having to scan through the bucket.)
- Initializing an empty dynamic table with n buckets takes deterministic $O(n)$ time.

5.1 Simple static tables (0 pt.)

Ollie first looks at this simple implementation of static hash tables using dynamic hash tables:

- BUILD($k_1, \ldots, k_n$): Construct a dynamic hash table with n buckets with some hash function $h_m \in \mathcal{H}$. Then, run INSERT(k_i) for each k_i.
- SEARCH(k): Run SEARCH(k) on the dynamic hash table.

What are the asymptotic runtimes of BUILD and SEARCH for this implementation? What is the asymptotic size of this table, in terms of the total number of buckets?

Solution (provided)

BUILD runs in deterministic $O(n)$ time, since it consists of n unique calls to INSERT, which is a deterministic $O(1)$ operation.
SEARCH runs in expected $O(1)$ time since SEARCH for dynamic hash tables is expected $O(1)$.

Since the dynamic table uses $O(n)$ buckets, this implementation uses $O(n)$ buckets.

5.2 Expected vs. deterministic (1 pt.)

Why does an expected $O(1)$ runtime for SEARCH not imply a deterministic worst-case $O(1)$ runtime?

[We are expecting: A brief explanation in plain English.**]**

5.3 Expected collisions (1 pt.)

Ollie is despondent upon learning of the lack of deterministic search. Being an overachieving ostrich, Ollie searches for a new implementation with better performance.

Ollie now looks at hashing n keys into a table with n^2 buckets. What is the expected total number of collisions in such a table?

Hint: You may cite equations from lecture notes.

[We are expecting: A mathematical derivation.**]**

5.4 Collision probability (1 pt.)

When hashing n keys into a table with n^2 buckets, show that there is at least a $1/2$ probability of having no collisions in the table. (In other words, show that there is at most a $1/2$ probability of having any collisions in the table.)

Hint: Markov's inequality may be useful. For a random variable X and a constant a:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

[We are expecting: A mathematical derivation.**]**

5.5 Big fast tables (4 pt.)

With this knowledge, help Ollie implement a static hash table with the following properties:

- BUILD$(k_1, \ldots, k_n)$ runs in *expected* $O(n^2)$ time.
- SEARCH(k) runs in *deterministic* $O(1)$ time.
- The size of the data structure is n^2 buckets.

[We are expecting: A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties.**]**

5.6 Small slow tables (6 pt.)

Although we have satisfied Ollie's runtime requirements, Ollie is still worried about the $O(n^2)$ size.

We now turn to more space-efficient tables with deterministic search times. Help Ollie implement a static hash table with the following properties:

- BUILD($k_1, \ldots, k_n$) runs in *expected* $O(n)$ time.
- SEARCH(k) runs in *deterministic* $O(n)$ time.
- The size of the data structure is n buckets.

[We are expecting: A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties.**]**

5.7 Interlude (1 pt.)

For a table with n unique keys, m buckets, and k total collisions, let s_i be the size of bucket i, where $i \in [1, \ldots, m]$. Show the following relation:

$$\sum_{i=1}^{m} s_i^2 = 2k + n$$

[We are expecting: A mathematical derivation.**]**

5.8 Small fast tables (6 pt.)

Now, we are ready to define a data structure that fulfills Ollie's runtime requirements *and* space requirements.

Help Ollie implement a data structure with the following properties:

- BUILD($k_1, \ldots, k_n$) runs in *expected* $O(n)$ time.
- SEARCH(k) runs in *deterministic* $O(1)$ time.
- The size of the data structure is $O(n)$ buckets.

Hint: Can we combine static hash tables with different tradeoffs to get the best of both worlds?

[We are expecting: A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties. Feel free to reference previous portions of the problem.**]**