Exercises

Please do the exercises on your own.

1. **(2 pt.)** Suppose that $h : \mathcal{U} \to \{0, \ldots, n-1\}$ is a uniformly random function. That is, for each $x \in \mathcal{U}$, $h(x)$ is distributed uniformly at random in the set $\{0, \ldots, n-1\}$ for all i, and the values $\{h(x) : x \in \mathcal{U}\}$ are independent. Prove that for any $x \neq y \in \mathcal{U}$,

$$\mathbb{P}_h\{h(x) = h(y)\} = \frac{1}{n}.$$

Above, notice that x and y are fixed and the probability is over the choice of h.

[**We are expecting:** *A short but rigorous proof.*]

SOLUTION:
The only way for $h(x) = h(y)$ is for there to be some value $i \in \{0, \ldots, n-1\}$ so that $h(x) = i$ and $h(y) = i$. Thus:

$$\mathbb{P}_h\{h(x) = h(y)\} = \sum_{i=0}^{n-1} \mathbb{P}_h\{h(x) = i \wedge h(y) = i\}.$$

Then by independence, we have that $\mathbb{P}_h\{h(x) = i \wedge h(y) = i\} = \mathbb{P}_h\{h(y) = i\} \cdot \mathbb{P}_h\{h(x) = i\}$. Putting all this together, along with the fact that, by uniformity, $\mathbb{P}_h\{h(y) = i\} = 1/n$, we have:

$$\begin{aligned}
&\mathbb{P}_h\{h(x) = h(y)\} \\
&= \sum_{i=0}^{n-1} \mathbb{P}_h\{h(y) = i \wedge h(x) = i\} \\
&= \sum_{i=0}^{n-1} \mathbb{P}_h\{h(y) = i\}\mathbb{P}_h\{h(x) = i\} \\
&= \sum_{i=0}^{n-1} \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n}.
\end{aligned}$$

2. **(4 pt.)** Let $\mathcal{U} = \{000, 001, 002, \ldots, 999\}$ (aka, all of the numbers between 0 and 999, padded so that they are three digits long) and let $n = 10$. For each of the following hash families $\mathcal{H}$ consisting of functions $h : \mathcal{U} \to \{0, \ldots, n-1\}$, decide whether $\mathcal{H}$ is universal or not, and justify your result with a formal proof.

   (a) **(2 pt.)** For $i = 1, 2, 3$, let $h_i(x)$ be the i'th least-significant digit of x. (For example, $h_2(456) = 5$). Define $\mathcal{H} = \{h_1, h_2, h_3\}$. Is $\mathcal{H}$ a universal hash family?

   (b) **(2 pt.)** For $a \in \{1, \ldots, 9\}$, let $h_a(x)$ be the least-significant digit of ax. (For example, $h_2(123)$ is the least-significant digit of $2 \times 123 = 246$, which is 6). Define $\mathcal{H} = \{h_i : i = 1, \ldots, 9\}$. Is $\mathcal{H}$ a universal hash family?

[**HINT:** *To show that something is* not *a universal hash family, you could find two distinct elements $x, y \in \mathcal{U}$ so that the probability that $h(x) = h(y)$ is larger than it's supposed to be.*]

[**We are expecting:** *For each part, a yes/no answer and a rigorous proof using the definition of a universal hash family.*]

SOLUTION:

(a) This is not a universal hash family. To see this, consider $x = 111$ and $y = 112$. Now, when I choose $h \in H$ at random, the probability that $h(x) = h(y)$ is $2/3$, because $h_1(x) = h_1(y) = 1$ and $h_2(x) = h_2(y) = 1$. But for H to be a universal hash family I would need $P(h(x) = h(y)) \leq 1/10$. Since this is not the case, H is not a universal hash family.

(b) This is not a universal hash family. To see this, consider $x = 000$ and $y = 010$. For any $a \in \{1, \ldots, 9\}$, we have $ax = 0$ and $ay = 10 \cdot a$. Thus, the least significant digit of both ax and ay is 0. That means that for any a, $h(x) = h(y) = 0$, so $P_{h \in H}(h(x) = h(y)) = 1$, which is definitely larger than $1/10$.

3. **(4 pt.)** Give one example of a *connected* undirected graph on four vertices, A, B, C, and D, so that both depth-first search and breadth-first search discover the vertices in the same order when started at A. Give one example of a *connected* undirected graph where BFS and DFS discover the vertices in a different order when started at A.

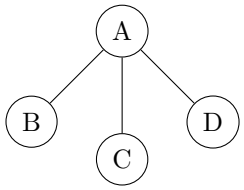
   Above, *discover* means the time that the algorithm first reaches the vertex. Assume that both DFS and BFS iterate over neighbors in alphabetical order.

   Note on drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code. (By default it makes directed graphs; you can add an arrow in both directions to get something that approximates an undirected graph).

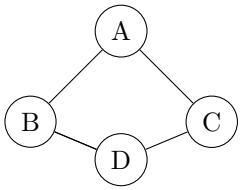
   [**We are expecting:** *A drawing of your two graphs and an ordered list of vertices discovered by BFS and DFS for each of them.*]

   SOLUTION:

   Our graphs are as follows; all searches start at A.



G



G'

   For G, both DFS and BFS, when started at A, recover vertices in order ABCD. For G', DFS starting at A discovers vertices in order ABDC, while BFS starting at A does it in order ABCD.

Problems

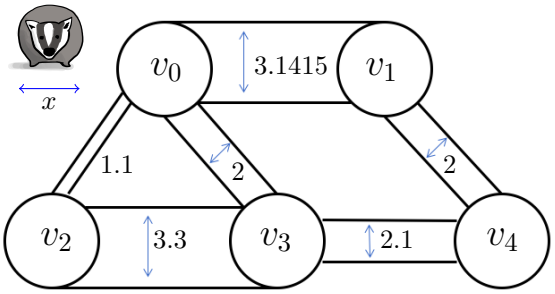
You may talk with your fellow CS161-ers about the problems. However:

- 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.

---

4. **(6 pt.) [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.

[Actual questions on next page.]

(a) **(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 will not be so pedantic about this in the future, but one point of this problem is to make sure you understand how the adjacency-list format works).

[**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, but if you significantly modify them make sure to be precise about how this interacts with the adjacency-list representation.*]

(b) **(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 accomodates 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: The same note about pseudocode holds as in part (a).

[**We are expecting:** *Pseudocode* ***AND*** *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, but if you significantly modify them make sure to be precise about how this interacts with the adjacency-list representation.*]

SOLUTION:

(a) The idea is to remove the too-narrow edges from G, and then run BFS (or DFS works too) to see if there is a path between s and t.

```
def widePath( G, s, t, x ):
     Say that G = V,E
     # first, remove all of the edges that are too narrow
     for i in range(n):
         Ni = V[i]
         for j in Ni:
             if tunnelWidth(i,j) < x:
                 remove j from the linked list
                 \\ time O(1) to remove j (manipulating pointers)
     Run BFS on G starting from s.
     If t is discovered during BFS:
          \\ Fine to use BFS as a black box here
          \\ since I already was pedantic when removing edges.
          return True
     Else:
          return False
```

This algorithm runs in time $O(n+m)$. First, it loops over the and removes the ones that are too narrow. This takes time $O(m+n)$, since for each vertex i we do $O(\deg(i))+O(1)$ work: for each neighbor j of i we check if the edge $\{i,j\}$ is too narrow, and potentially remove it.

$$O\left(\sum_i \deg(i) + \sum_i 1\right) = O(m+n)$$

work. Next, it runs BFS, which takes time $O(n+m)$.

Note: It is also okay to write down a modified version of BFS/DFS which just ignores the edges which are too narrow for the badger, as long as the modified version is sufficiently pedantic about the adjacency-list format. (Something similar to the pseudocode from lecture, which talkes about v.neighbors, is okay, although it would be nice to call it $V[v]$ to agree with the problem statement).

Note: It is also okay to say that the algorithm actually runs in time $O(m)$. This is because we only need to explore the connected component of the graph that s is in, so without loss of generality we may assume that G is connected. In that case, $n = O(m)$, so $O(n+m) = O(m)$.

(b) The high-level idea is to use binary search and then use part (a) to find the largest x that works. Notice that the maximum x must be equal to one of the edge weights.

```
def widestPath( G, s, t ):
    if s == t:
        return Infinity  # any badger can fit in a path from s to t, since s=t
        # NOTE: we will not take off points if you didn't handle the s=t case.

    # first, generate a list of the weights that appear, in time O(m + n)
    W = []
    for i in range(n):
        for j in V[i]:
            if j >= i:  # so that we don't add the tunnel twice
                 add tunnelWidth(i,j) to W

     sort W in time O(m log(m)) using MergeSort.

     # now run binary search to see what the largest width we can accomodate.

     a=0, b=n-1 # we are searching in {a,a+1,...,b}
     while b > a:
        mid = ceiling( (a+b)/2 )
        isThereAPath = widePath( G, s, t, W[mid] )
        if isThereAPath:
              # then a badger of size W[mid] could get from s to t;
              # next check a wider badger
              a = mid
        else:
              # then a badger of size W[mid] was too wide to get from s to t;
              # next check a narrower badger
              b = mid - 1

      # now we should have a=b = index of widest possible value in W.
      return W[a]
```

The running time is $O((n+m)log(m))$. This is because it takes time $O(mlog(m))$ to

sort the list. Then we call `widePath` $O(\log(m))$ times during the binary search, and each time it takes time $O(n + m)$.

5. **(8 pt.)** [**Painted Penguins.**] A large flock of T painted penguins will be waddling past the Stanford campus next week as part of their annual migration from Monterey Bay Aquarium to the Sausalito Cetacean Institute. Painted Penguins (not to be confused with pedantic penguins) are an interesting species. They can come in a huge number of colors—say, M colors—but each flock only has m colors represented, where $m < T$. The penguins will waddle by one at a time, and after they have waddled by they won't come back again. You'd like to design a randomized data structure to keep track of the penguin colors so that, after all the penguins have gone, you'll be able to answer queries about what colors of penguins appeared in the flock; you'd like your answers to these queries to probably be correct.

   For example, if $T = 7$, $M = 100000$ and $m = 3$, then a flock of T painted penguins might look like:

`seabreeze, seabreeze, indigo, ultraviolet, indigo, ultraviolet, seabreeze`

   You'll see this sequence in order, and only once. After the penguins have gone, you'll be asked questions like "How many `indigo` penguins were there?" (Answer: 2), or "How many `neon orange` penguins were there?" (Answer: 0).

   You know m, M and T in advance, and you have access to a universal hash family $\mathcal{H}$, so that each function $h \in \mathcal{H}$ maps the set of M colors into the set $\{0, \ldots, n-1\}$, for some integer n. For example, one function $h \in \mathcal{H}$ might have $h(\texttt{seabreeze}) = 5$.

   (a) **(5 pt.)** Suppose that $n = 10m$. Suppose also that you only have space to store:

   - An array B of length n, consisting of numbers in the set $\{0, \ldots, T\}$, and
   - one function h from $\mathcal{H}$.

   Use the universal hash family $\mathcal{H}$ to create a randomized data structure that fits in this space and that supports the following operations in time $O(1)$ in the worst case, assuming that you can evaluate $h \in \mathcal{H}$ in time $O(1)$.

   - `Update(color)`: Update the data structure when you see a penguin with color `color`.
   - `Query(color)`: Return the number of penguins of color `color` that you have seen so far. For each query, your query should be correct with probability at least 9/10. That is, for all colors `color`,

$$\mathbb{P}\{\texttt{Query(color)} = \text{ the true number of penguins with color } \texttt{color}\ \} \geq \frac{9}{10}.$$

   To describe your data structure:

   i. Describe how the array B and the function h are initialized.
   ii. Give pseudocode for `Query`.
   iii. Give pseudocode for `Update`.

   [**We are expecting:** *A description following the outline above (including pseudocode), and a short but rigorous proof that your data structure meets the requirements. Make sure you clearly indicate where you are using the property of universal hash families.*]

   (b) **(3 pt.)** Suppose that you now have k times the space you had in part (a). That is, you can store k arrays $B_1, \ldots, B_k$ and k functions $h_1, \ldots, h_k$ from $\mathcal{H}$. Adapt your data structure from part (a) so that all operations run in time $O(k)$, and the `Query` operation is correct with probability at least $1 - \frac{1}{10^k}$.

[**We are expecting:** *As in part (a), a description following the outline above (except say how all arrays B_i and functions h_i are initialized), and a short but rigorous proof that your data structure meets the requirements. Make sure you clearly indicate where you are using the property of universal hash families.*]

SOLUTION:

(a) Here is the description of our data structure:

- Our data structure stores an array B of length n, where each bucket stores a number in $\{0, \ldots, T\}$ and is initialized to zero. Before the flock waddles by, we choose a random $h \in \mathcal{H}$ and store that too.
- `Update(color): B[h(color)] ++`
- `Query(color): Return B[h(color)]`.

Each of these operations takes time $O(1)$. The probability that a single `Query` option fails is the probability that any of the m (or $m-1$ other) colors which did appear collided with the color that was queried. That is, we want

$$\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}, \text{ so that } h(x) = h(\texttt{color})\}$$

to be small. By the universal hash family property, we have for each color x,

$$\mathbb{P}\{h(x) = h(\texttt{color})\} \leq \frac{1}{n}.$$

Thus, by the union bound, the probability that there exists an x which appeared that collides with `color` is at most

$$\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}, \text{ so that } h(x) = h(\texttt{color})\}$$
$$\leq m \cdot \mathbb{P}\{h(x) = h(\texttt{color})\} \leq \frac{m}{n} = \frac{1}{10}.$$

(b) We will basically just keep k copies of our data structure from part (a). More precisely, our data structure stores:

- k arrays $B_1, \ldots, B_k$, initialized to zero.
- k hash functions $h_1, \ldots h_k \in \mathcal{H}$, chosen uniformly at random and independently. (With replacement).

Then our update strategy is:

```
Update(color):
    for i = 1,...,k:
        B_i[ h_i(color) ] ++

Query(color):
    return min_{i = 1,...,k} B_i[ h_i(color) ]
```

Both of these operations take time $O(k)$, since they both loop over k things.
To compute the success probability of `Query`, notice that this returns the correct value as long as the color `color` is isolated in *any* of the k tables. Since each of these k hash functions are independent, we have:

$$\mathbb{P}\{\text{for all } i, \text{ there is a color } x \text{ which appeared, not the same as } \texttt{color}, \text{ so that } h_i(x) = h_i(\texttt{color})\}$$
$$= \left(\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}, \text{ so that } h_i(x) = h_i(\texttt{color})\}\right)^k$$
$$\leq \left(m \cdot \mathbb{P}\{h(x) = h(\texttt{color})\}\right)^k$$
$$\leq \left(\frac{m}{n}\right)^k$$
$$= \frac{1}{10^k}.$$

Thus, with probability at least $1 - 1/10^k$, there is at least one i so that $B_i[h_i(\texttt{color})]]$ is equal to the number of times that that `color` appeared, and `Query(color)` returns the right thing.