Problem 1

I discuss how the runtimes of the operations INSERT, DECREASE-KEY, DELETE-MIN changes in an implementation of multi-level-bucket heaps where I keep the set of a block's nonempty buckets in a standard binary heap instead of maintaining it in an array.

Let k be the depth of the tree and Δ the branching factor, such that $\Delta^k = C$.

INSERT took $O(k)$ time in the array implementation. In the binary heap implementation, it takes $O(k)$ time to find the right binary heap and $O(\log \Delta)$ to insert into it, so the total running time is $O(k + \log \Delta)$.

DECREASE-KEY took $O(k)$ time in the array implementation. In the binary heap implementation, it takes $O(k + \log \Delta)$ time: it takes $O(\log \Delta)$ to delete the item from the binary heap where it used to be (simply decrease its key to $-\infty$ and delete it as the min), then, like for the INSERT operation, it takes $O(k)$ to find the new binary heap for the item and $O(\log \Delta)$ to insert into it.

DELETE-MIN took $O(k\Delta)$ time in the array implementation (plus an overall $O(n\Delta)$). In the binary heap implementation, it takes $O(\log \Delta)$ time to delete the min thanks to the cheap DELETE-MIN operation of the binary heap. However, if the heap becomes empty, the change might need to get cascaded to the top, which costs at most $O(k)$ in additional time. Therefore, in the binary heap implementation, the total running time might reach $O(k + \log \Delta)$.

To balance the running time of $O(k + \log \Delta)$ of the operations in the binary heap implementation, I pick $k = \log \Delta$. Recalling that $\Delta^k = C$ or $\log \Delta = (\log C)/k$, I get $k = \sqrt{\log C}$. Thus, by setting $k = \log \Delta$, I get a runtime of $O(\sqrt{\log C})$ for each INSERT, DECREASE-KEY and DELETE-MIN.

Since Dijkstra's shortest path algorithm makes m calls to DECEREASE-KEY, n calls to INSERT and n calls to DELETE-MIN, its running time using this binary heap implementation of multi-level-bucket heaps is $O((m + n)\sqrt{\log C})$.

Problem 2

I show that for the single-source shortest paths problem on a graph with n nodes and range of edge lengths $\{1, 2, \ldots, C\}$, I can obtain $O(\log \log C)$ time per queue operation.

Even though the range of values is $\{1, 2, \ldots, nC\}$, the trick is to notice that, in Dijkstra's algorithm, only values between x and $x + C$ are active at once. Indeed, if x is the minimum, I will only add values as great at $x + C$ when I process x, as C is the max value of an edge. So before processing x, by a similar argument, I must have only added values $\leq x + C$. Then, after processing x, the range of values remain in an interval $\leq C$. In addition, DECREASE-KEY will only reduce values within the active range.

I maintain two van Emde Boas priority queues, Q_1 and Q_2 which only stores values modulo C. Each queue has an extra field indicating the quotient of the "actual" values stored modulo C in the queue. I start with Q_1 and Q_2 empty, representing the quotients 0 and 1, respectively. When Q_1 becomes empty, I increase its quotient by 2, and switch Q_1 and Q_2. By the trick explained in the previous paragraph, I guarantee that the active range of values can fit in Q_1 and Q_2 at any time.

When inserting an element, I put it in the queue corresponding to its quotient with its value modulo C. When decreasing the key of an element, I either just decrease its key directly in the queue in which it is (if the quotient doesn't change) or (if its quotient decreases) I delete it from Q_2 (necessarily) and insert it in Q_1 with the new key value modulo C. Deleting the min is as simple as deleting the min of Q_1 (and switching the queues as explained in the previous paragraph if Q_1 becomes empty).

Since they only have values in the range $\{0, \ldots, C - 1\}$, my two van Emde Boas priority queues perform each queue operation in $O(\log \log C)$ time. Thus, I achieve the bound of $O(\log \log C)$ per queue operation.

Problem 3

I augment the van Emde Boas priority queue presented in class (which performed the insert and successor queries) to support the following operations on integers in the range $\{0, 1, 2, \ldots, u-1\}$ in $O(\log \log u)$ worse-case time each and $O(u)$ space total:

FIND(x, Q)**:** Report whether the element x is stored in the structure.

```
FIND(x, Q)
1  if x < Q.min or x > Q.max
2      then return False
3  if x = Q.min or x = Q.max
4      then return True
5  return FIND(low(x), Q[high(x)])
```

PREDECESSOR(x, Q)**:** Return x's predecessor, the element of largest value less than x, or null if x is the minimum element.

```
PREDECESSOR(x, Q)
 1  if x ≤ Q.min
 2      then return null
 3  ▷ x is at position low(x) in Q[high(x)]
 4  if low(x) > Q[high(x)].min
 5      then return high(x).√|Q| + PREDECESSOR(low(x), Q[high(x)])
 6      else  ▷ low(x) = Q[high(x)].min
 7            i ← PREDECESSOR(high(x), Q.summary)
 8            if i is not null
 9               then return i.√|Q| + Q[i].max
10               else  return Q.min
```

DELETE(x, Q)**:** Delete x from the queue while preserving the structure of the priority queue.

```
DELETE(x, Q)
 1  if Q.min is null or x < Q.min:
 2      then return
 3  if Q.min = x
 4      then i ← Q.summary.min
 5           if i is null
 6              then Q.min ← null
 7                   Q.max ← null
 8                   return
 9              else  x ← i.√|Q| + Q[i].min ▷ overwriting x
10                    Q.min ← x
11
12  DELETE(low(x), Q[high(x)])
13  if Q[high(x)].min is null
14      then DELETE(high(x), Q.summary)
15  ▷ Update the max if necessary
16  if Q.max = x
17      then if Q[high(x)].max is null
18              then i ← Q.summary.max
19              else  i ← high(x)
20           if i is null
21              then Q.max ← null
22              else  Q.max ← i.√|Q| + Q[i].max
23
```

Analysis

FIND, PREDECESSOR and DELETE do at most 1 costly recursive call of size $\sqrt{u}$. DELETE might do two recursive calls, but, when it does, the first call is cheap: it takes $\Theta(1)$ time as it simply nulls the minimum (and unique) element of $Q[\text{high}(x)]$. Therefore, the running time of each operation is $T(u) = T(\sqrt{u}) + \Theta(1) = O(\log \log u)$. The total space remains $O(u)$ with these added operations as they don't change the structure in any way.

Problem 4

Problem 5

I suppose I have already computed the maximum flow in a network with m edges and integral capacities using an augmenting-paths algorithm.

Part (a)

I show how to update the maximum flow in $O(m)$ time after increasing a specified capacity by 1.

I construct the residual graph of the old maximum flow and find an augmenting path. Since only a specified capacity was increased by 1, if there is an augmenting path, it must involve this increased capacity (as the flow used to be maximum) and once the augmentation is performed, there cannot be any other augmenting paths (as the increased capacity must now be completely used).

The residual graph takes $O(m)$ time to construct. Using a Breath First Search, I can find the augmenting path in $O(m)$ time. Since the path has $\leq n$ edges (as each node can only appear once), the augmentation can be done in $O(n)$ time. Therefore, the running time of the update is $O(m)$.

Part (b)

I show how to update the maximum flow in $O(m)$ time after decreasing a specified (positive) capacity by 1.

If the old maximum flow did not saturate the decreased capacity, then it can be decreased by 1 without affecting the maximum flow.

Otherwise, I choose a flow path of value 1 that uses this capacity, and substract it from the old maximum flow. This modified flow has a value decreased by 1 compared to the old maximum flow. I construct the residual graph of this modified flow and find an augmenting path. If there is an augmenting path, once the augmentation is performed, there cannot be any other augmenting paths because otherwise the new maximum flow would have a greater value than the old maximum flow, a contradiction since the network is more constrained.

Using a Breath First Search, I can find a flow path of value 1 and substract it in $O(m)$. Like in part (a), finding an augmenting path using a Breath First Search takes $O(m)$ and the augmentation takes $O(n)$. Therefore, the running time of the update is $O(m)$.

Problem 6

Part (a)

Supposing all people are initially in a single room s, and that the building has a single exit t, I show how to use maximum flow to find a fastest way to get everyone out of the building.

For time i, I construct an auxiliary graph G'_i as explained below. The value of maximum flow of G'_i from s_0 to t_i tells me how many people I can get out of the building in time i. So I try time $i = 1$, $i = 2$, ... until I reach a time $i_{\min}$ where the value of the maximum flow is greater than the number of people initially in s. The maximum flow in the graph $G'_{i_{\min}}$ tells the fastest way to get everyone out of the building.

The auxiliary graph G'_i for time i is constructed as follow.

- For each vertex v in G, add vertices $v_0, \ldots, v_i$ in G'_i and edges $v_0 \rightarrow v_1, \ldots, v_{i-1} \rightarrow v_i$ with capacity ∞.
- For each edge (v, w) with capacity c in G, add edges $v_0 \rightarrow w_1, \ldots, v_{i-1} \rightarrow w_i$ with capacity c in G'_i.

Part (b)

I show that the same technique can be used when people are initially in multiple locations and there are multiple exists.

I construct the auxiliary graph G'_i for time i as before, except that I add a super-source s and a super-sink t in G'_i. For each location v in which x people are initially, I add an edge $s \rightarrow v_0$ with capacity x. For each exit v, I add an edge $v_i \rightarrow t$ with capacity ∞. Now, when I look for the maximum flow in G'_i, the source is s and the sink t.

Part (c)

I generalize the approach to where different corridors have different (integral) transit times.

When I construct the auxiliary graph G'_i, I take edges of G into account as follows:

- For each edge (v, w) with capacity c in G and transit time k, add edges $v_a \rightarrow w_{a+k}$ with capacity c, $\forall a$ such that $0 \leq a \leq i - k$.