6331 - Algorithms, Autumn 2016, CSE, OSU
Summaries of solutions to homework assignments
Instructor: Anastasios Sidiropoulos

The following are only brief summaries of the solutions to the homework problems. You should be able to fill in the details on your own. Note that these are just brief summaries of the correct solutions; you are typically asked to provide a justification of your answer to receive full credit.

Homework 1, Problem 1. $\log(\log(n))$, $\log(n)$, $1000n$, $n\log(n)$, n^2, $n^{\log n}$, 2^n, 2^{2^n}.

Homework 1, Problem 2. $2^{n+1} = O(2^n)$. This can be shown using the definition of O-notation.
2^{2n} is not in $O(2^n)$.

Homework 1, Problem 3.

$$f(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ n+10 & \text{if } n \text{ is even} \end{cases}$$

Homework 1, Problem 4. The answer is $\Theta(n^6 \cdot \log n)$. One way to see that quickly is to observe that the last iteration of the "while" loop dominates the running time. In that iteration, we have $i = \Theta(n^2)$, and thus the running time is $\Theta(n^6 \log n^2) = \Theta(n^6 \log n)$.

Homework 1, Problem 5. Some examples are: $n^{\log n}$, $n^{\log^2 n}$, $2^{\sqrt{n}}$, $2^{n^{1/3}}$, $2^{\sqrt{n}\log n}$.

Homework 2, Problem 1. The solution is via a simple modification of the standard heap data structure. The parent of node i is stored at location i/k. The formula for the locations of the children can be derived from this. In a heap that corresponds to a full k-ary tree, the leaves occupy $(1-1/k)$-fraction of the array, and they are stored at the the last positions.

The depth of the heap now becomes $O(\log_k n)$, which is smaller than $O(\log_2 n)$, when k is large enough. However, the running time of most operations becomes $O(k \cdot \log_k n)$. The additional factor of k is due to the fact that one has to find the right child to recurse at east step; a naive implementation scans through all children until it finds the correct one, and this takes time $O(k)$. There are of course other possible ways one might try to implement this. Any other reasonable implementation is an acceptable answer. Overall, the running time does not increase when k increases because $k \log_k n = k\frac{\log n}{\log k} = \frac{k}{\log k}\log n = \Omega(\log n)$.

Homework 2, Problem 2. (a) Running Heapsort on an array where all the elements have the same value takes time $O(n)$. This is easy to see because each call to Max-Heapify will take time $O(1)$.

(b) Take an array consisting of $n/2$ entries with the value 1, follows by $n/1$ entries with the value 0. Then, during the first $n/4$ calls of the main loop of HeapSort, every call to Max-Heapify will have to "push" the root down for at least $\Omega(\log n)$ levels; thus, the total running time will be $\Omega(n\log n)$.

Homework 3, Problem 1. (a) $1, 2, 3, \ldots, n$.

(b) For $i = 1, \ldots, \log n$, do the following: Insert the numbers $1/2^i, 2/2^i, 3/2^i, \ldots, 2^{i-1}/2^i$, in this order, and omitting any numbers you have already inserted.

Homework 3, Problem 2. The algorithm first shuffles randomly the input set of n keys, and then adds them in the resulting random order to a binary search tree. Then, it uses the resulting tree to sort the keys as follows: It repeatedly finds the minimum key and removes is from the tree (updating the tree accordingly). The cost of each search is at most the height of the tree. Thus, the total running time is at most n times the height of the tree. The expectation of the quantity is $O(n \log n)$, as desired.

One important point is that one needs to show that n numbers can be shuffled quickly. The entries of an array of length n can be randomly permuted in time $O(n)$, so that the resulting permutation is distributed uniformly at random from the set of all possible permutations. This can be done as follows: For $i = 1, \ldots, n-1$, pick j in $\{i, \ldots, n\}$ uniformly at random, and swap the i-th entry with the j-th entry.

Note that there other ways one might try for shuffling an array, but they don't all lead to a permutation that is *uniformly* distributed in the set of all possible permutations.

Homework 4, Problem 1. The solution uses dynamic programming. Note that a palindrome does not have to be a *contiguous* subsequence of the input sequence.

The dynamic program computes for each $i < j \in \{1, \ldots, n\}$, the value $L(i, j)$, which is the maximum length of a palindrome of the form

$$x_{t_1}, \ldots, x_{t_k}, x_{t_{k+1}}, \ldots, x_{t_{2k}},$$

where $t_k = i$, and $t_{k+1} = j$. That is, the middle two elements of the palindrome are at positions i and j of the input sequence. Using the values $L(i, j)$, you can also compute the actual palindrome.

The rule for computing the values $L(i, j)$ is as follows: If $x_i \neq x_j$, then $L(i, j) = 0$. Otherwise, we have

$$L(i, j) = \max_{i' < i, j' < j} \{L(i', j') + 1\}$$

The boundary condition is $L(0, n+1) = 0$.

Homework 4, Problem 2. Using dynamic programming, you compute a value $C(i, j)$, which equals the maximum number of coins that Indiana Jones can collect so far while he is at position (i, j). The boundary conditions are $C(i, 1) = A[i, 1]$, and $C(1, i) = A[1, i]$, for all $i \in \{1, \ldots, n\}$. For all $i \in \{2, \ldots, n\}$, and all $j \in \{2, \ldots, n\}$, we have

$$C(i, j) = A[i, j] + \max\{C(i-1, j), C(j-1, i)\}.$$

The maximum number of coins you can compute is $C(n, n)$. One can also find the actual path using that Indiana Jones must follow using these values.

Homework 4, Problem 3. We say that i is a *losing* position if when a player has to pick from a pile of i sticks, there is no winning strategy (i.e., a move that is guaranteed to lead to a win if both players are playing optimally). If i is not a losing position, then we say that it is a *winning* position.

For each i, we compute a value $A[i] = \{W, L\}$, with $A[i] = W$ if i is a winning strategy, and $A[i] = L$ if i is a losing strategy. This can be done via dynamic programming as follows. The boundary condition is $A[1] = L$, since when you have to pick from a pile with a single stick, you lose. For any $i \in 1$, we have that $A[i] = W$ if there exists $j \in \{\lceil i/2\rceil, \ldots, i-1\}$, such that $A[j] = L$; otherwise, we have $A[i] = L$. In other words, the values $A[i]$ can be computed "from left to right".

Given the values $A[i]$, one can easily determine what an optimal strategy is: If $A[i] = L$, then the player picks a single stick (because it does not matter how many sticks the player picks). If $A[i] = W$, then there must exist $j \in \{\lceil i/2\rceil, \ldots, i-1\}$, such that $A[j] = L$; the player picks $i-j$ sticks, and the next player ends up at a losing position.

Note that for this game one can also compute whether a position is winning or not via a simple formula: a position i is losing precisely when $i = 2^k - 1$, for some integer k. However, if the game had more complicated rules, it would have been much harder to come up with an explicit formula for the winning/losing positions. In such a scenario, one would have to use a dynamic program as the one described above.

For example, consider the game where the player loses if there the number of sticks left is either 1, or an integer multiple of 5. It is now much harder to come up with an explicit formula for the winning/losing positions.

Homework 5, Problem 1. Let us say that a solution is *greedy* if it is the solution that the greedy algorithm computes.

The idea is to show that given any solution, we can compute a greedy solution without increasing the number of coins. This requires a case analysis.

Let X be the value in cents that we wish to express as a sum of US coins. Let $c_1, \ldots, c_n$ be the coins used in some solution, i.e. $c_1 + \ldots + c_n = X$. We may assume w.l.o.g. that

$$c_1 \geq \ldots \geq c_n.$$

Let also $c'_1, \ldots, c'_m$ be the greedy solution, with

$$c'_1 \geq \ldots \geq c'_m.$$

Let i be the minimum integer such that $c_i \neq c'_i$; that is, i is the first index where the two solutions disagree. We have to consider different cases depending on the value of c'_i.

$c'_i = 25$: Let j be the minimum integer, with $j \geq i$ such that $c_i + \ldots + c_j \geq c'_i$. There is a constant number of possibilities for the values in the set $\{c_i, \ldots, c_j\}$. Namely, $\{10, 10, 10\}$, $\{10, 10, 5\}$, $\{10, 5, 5, 5\}$, $\{5, 5, 5, 5, 5\}$, $\{5, 5, 5, 5, 1, \ldots, 1\}$, $\{5, 5, 5, 1, \ldots, 1\}$,$\{5, 5, 1, \ldots, 1\}$,$\{5, 1, \ldots, 1\}$. In each of these sub-cases we can modify the given solution so that the sum of its elements remains equal to X, without increasing the number of coins used. For example, if $\{c_i, \ldots, c_j\} = \{10, 10, 10\}$, we can remove the three coins of value 10, and instead add one coin of value 25, and one coin of value 5. All sub-cases can be handled in a similar fashion.

$c'_i = 10$: Arguing as above, there is a constant number of possibilities for $\{c_i, \ldots, c_j\}$; namely, $\{5, 5\}$, $\{5, 1, \ldots, 1\}$, $\{1, \ldots, 1\}$. All sub-cases can be handled as in the previous case.

$c'_i = 5$: The only possibility is $\{c_i, \ldots, c_j\} = \{1, \ldots, 1\}$. We simply replace the coins $c_i, \ldots, c_j$ by a single coin of value 5.

$c'_i = 1$: In this case there can be no disagreement between the greedy and the given solutions; this this case never occurs.

We repeatedly perform the above modification steps until the two solutions become identical. This proves that the greedy solution is optimal.

Homework 5, Problem 2. (a) Sort the elements in non-decreasing order. Let $a_{j_1}, \ldots, a_{j_n}$ be the resulting order. Compute the maximum integer $m \in \{0, \ldots, n\}$ such that

$$a_{j_1} + \ldots + a_{j_m} \leq B.$$

Output the corresponding indices of the elements in the set $\{a_{j_1}, \ldots, a_{j_m}\}$.

(b) Suppose $a_1 = 1, a_2 = 2$, and $B = 2$. Then the above algorithm outputs $\{1\}$, while the optimum solution is $\{2\}$. Thus, the algorithm from (a) is not correct in this case.

(c) Let $A[0, \ldots, 100]$ be a table of boolean values. Initially, all values are set to FALSE. We consider all the elements $a_1, \ldots, a_n$ in this order. We shall maintain the following inductive invariant: After considering elements $a_1, \ldots, a_i$, we have $A[j] =$ TRUE if and only if there exists a subset of the elements in $\{a_1, \ldots, a_i\}$ that sums to j. More precisely, the pseudocode is as follows:

```
for $i = 1$ to $n$
  for $j = n - a_i$ to 0
    if $A[j] =$ TRUE then
      $A[j + a_i] =$ TRUE
    endif
  endfor
endfor
```

The maximum possible achievable sum that is at most 100 is equal to the maximum integer $i \in \{0, \ldots, 100\}$ such that $A[i] =$ TRUE. Furthermore, the actual set of elements that sums to that number can be found via a standard modification of the above dynamic program.

Homework 6, Problem 1. Let $k, \ell \in U$ be two distinct keys. We have

$$|H| = m^{|U|}.$$

For any $i, j \in \{0, \ldots, m-1\}$ let

$$H_{i,j} = \{h \in H : h(k) = i \text{ and } h(\ell) = j\}.$$

We have

$$|H_{i,j}| = m^{|U|-2}.$$

When choosing $h \in H$ uniformly at random, the probability that there is a collision between k and ℓ is

$$\begin{aligned}
\Pr_{h\in H}[h(k) = h(\ell)] &= \sum_{t=0}^{m-1} Pr_{h\in H}[h(k) = t \text{ and } h(\ell) = t] \\
&= \sum_{t=0}^{m-1} \frac{|H_{t,t}|}{|H|} \\
&= \sum_{t=0}^{m-1} \frac{m^{|U|-2}}{m^{|U|}} \\
&= m \cdot \frac{1}{m^2} \\
&= 1/m.
\end{aligned}$$

Thus H is universal.

Homework 6, Problem 2. Let $k = 0$ and $\ell = m$. When choosing $f_i \in H$ uniformly at random, we have

$$\begin{aligned}
\Pr_{f_i\in H}[f_i(k) = f_i(\ell)] &= \Pr_{f_i\in H}[i \bmod m = i + m \bmod m] \\
&= 1 \\
&> 1/m.
\end{aligned}$$

Thus H is not universal.

Homework 6, Problem 3. We have

$$\begin{aligned}
E[|S|] &= E[\sum_{k\neq l} X_{kl}] \\
&= \sum_{k\neq l} E[X_{kl}] \\
&= \sum_{k\neq l} \Pr[h(k) = h(l)] \\
&= \sum_{k\neq l} \frac{1}{m} \\
&= \binom{n}{2}\frac{1}{m}.
\end{aligned}$$

Homework 7, Problem 1. Let D be the state of the data structure. Define $\Phi(D) = k \cdot \log n$, where k is the number of elements currently in the heap. Every insertion “pays” an additional $\log n$ “cost”. This cost can then be used to pay for one extract-min operation.

Homework 7, Problem 2.
Insert(m)
Insert($m-1$)
Insert($m-2$)
Delete-Min
Insert($m-3$)
Insert($m-4$)
Insert($m-5$)
Delete-Min
Insert($m-6$)
Insert($m-7$)
Insert($m-8$)
Delete-Min
...

Homework 7, Problem 3. When inserting an element, we simply append it to the end of the array.

When computing an approximate median, we proceed as follows: Let n be the current number of elements in the data strucutre. Let n' be the number of elements in the data structure during the previous call to ApproximateMedian. If $n/n' \leq 2$, then output the same value from the last call. Otherwise, sort the array and output the median.

Remark: Note that there exists an algorithm for computing the median of n elements that runs in time $O(n)$. Using this algorithm instead of sorting, the amortized cost of the above data structure can be improved to $O(1)$.

Homework 8, Problem 1.

(a) A graph $G = (V, E)$ is a tree if and only if it is connected and $|E| = |V| - 1$. If $|E| \neq |V| - 1$ then we ouput "NO". Otherwise we check it G is connected via BFS. The running time is $O(|V| + |E|) = O(|V|)$ (since $|E| = O(|V|)$). If it is connected then we output "YES" and otherwise we output "NO".

(b) A graph is a cycle if and only if it is connected and every vertex has degree 2. We first check if every vertex has degree 2; if not then we output "NO". Otherwise we have

$$|E| = \frac{1}{2}\sum_{v} \deg(v) = \frac{1}{2}|V|2 = |V|,$$

where $\deg(v)$ denotes the degree of v. We check if the graph is connected using BFS (as in part (a). If the graph is connected then we output "YES" and otherwise we output "NO".

Homework 8, Problem 2. First note that a graph G is bipartite if and only if each of each connected components is bipartite. We may thus focus on the case where G is connected; the general case can be addressed by running the same procedure on each connected component.

Let us thus assume for the remainder that G is connected. We build the bipartition U, U', if one exists, using the following procedure: We pick some arbitrary $r \in V$ and we initialize $U = \{r\}$,

$U' = \emptyset$. We perform a BFS starting from r. When we visit a new vertex u for the first time (it is white), if the parent of u is in U, then we add u to U'; otherwise we add u to U. When we visit a vertex w which has already been visited (it is gray), and we are reaching w following some edge (z, w), then if both z and w are in U or both z and w are in U' then we output "NO". If BFS terminates without outputting "NO" then we output "YES".

Homework 8, Problem 3.

(a) We first show that if G has an Euler tour then for all $v \in V$ we have in-degree(v) = out-degree(v). Assume that G has an Euler tour C. Let $v \in V$. The number of times that C enters v is equal to the number of times that C leaves v. Thus in-degree(v) = out-degree(v).

Next we show that if for all $v \in V$ we have in-degree(v) = out-degree(v) then G has an Euler tour. Suppose that for all $v \in V$ we have in-degree(v) = out-degree(v). We will construct an Euler tour in G. Initially consider all edges as being "unmarked". Start from an arbitrary $w \in V$; if w has an unmarked outgoing edge (w, z) then traverse (w, z), add it to C, and update $w = z$. Eventually you arrive at some vertex w' with no outgoing unmarked edges. By induction we have that for all $v \in V$, the number of unmarked incoming edges to v is equal to the number of unmarked outgoing edges from v. Thus the sequence of edges that we traverse must terminate at the initial vertex. We have thus discovered a closed walk. If there are unmarked edges remaining then we pick another vertex with an unmarked outgoing edge and we repeat the same process. Eventually, all edges of G will be marked. At that point we have decomposed E into a collection of disjoint closed walks. Given two walks that share a vertex v we can merge them into a single walk as follows: Suppose that the first closed walk visits the vertices

$$C_1 = x_1, \ldots, x_i, v, x_{i+1}, \ldots, x_n, x_1$$

and the second closed walk visits the vertices

$$C_2 = y_1, \ldots, y_j, v, y_{j+1}, \ldots, y_m, y_1.$$

Then we can form the closed walk

$$C = x_1, \ldots, x_i, v, y_{j+1}, \ldots, y_m, y_1, \ldots, y_j, v, x_{i+1}, \ldots, x_n, x_1.$$

Repeating this process we can merge pairs of intersecting closed walks. Since G is strongly connected the process terminates when there is a single close walk left. Since the initial collection of walks contains all edges, it follows by the definition of the merging process that the final walk will also contain all edges and thus it is an Euler walk.

(b) The procedure described from the second part of (a) can clearly be implemented in time $O(|E|)$.

Homework 9, Problem 1.

(a) No. Consider the graph $G = (V, E)$ with $V = \{s, a, b\}$, $E = \{\{s, a\}, \{s, b\}, \{a, b\}\}$, and edge weights $w(\{s, a\}) = 10$, $w(\{s, b\}) = 10$, $w(\{a, b\}) = 1$.

(b) No. The example from (a) also works here.

Homework 9, Problem 2. T' is a spanning tree. This can be proven via induction on the execution of any standard MST algorithm.

Homework 9, Problem 3.

(a) This is the special case of part (b) for $k = 1$.

(b) We construct a graph $G = (V, E)$. Each vertex of G corresponds to a possible state of the set of k robots. Formally we set

$$V = (\{1, \ldots, n\}^2 \times \{N, W, E, S\})^k.$$

Each vertex in V is a tuple $(x_1, y_1, A_1, \ldots, x_k, y_k, A_k)$. The meaning is that the i-th robot is at row y_i and at column x_i, and has orientation A_i (where $A_i = N$ means that the orientation is "North", and so on). Clearly, we have $|V| = n^{O(k)}$. We add an edge (u, v) if it is possible to transition from state u to state v in one step. Let s be the vertex that corresponds to the starting state of all robots, and let t be the vertex that corresponds to the final state. We compute a shortest path from s to t in G using any standard algorithm (e.g. Dijkstra's Algorithm). The sequence of edges traversed by the shortest path gives the desired sequence of steps.

Homework 9, Problem 4. We construct an auxiliary graph $G' = (V', E')$. We set

$$V' = V \times T.$$

That is, each vertex in G' is an ordered pair (w, t), where $w \in V$ and $t \in T$. For each flight that starts from some city x at time t, terminates at some city y, and has cost c, we add the edge $((x, t), (y, t + d(x, y))$ to E' with weight c (corresponding to taking a flight from x to y). We also add for each $z \in V$ and for each $t \in \{0, \ldots, T - 1\}$ the edge $((z, t), (z, t + 1))$ with weight 0 (corresponding to the fact that you can wait at city z for the next flight).

Let u be the starting city and let v be the destination. Let $u' = (u, 0)$ and $v' = (v, T)$. We compute the shortest path from u' to v' in G' using any standard algorithm (e.g. Disjkstra's algorithm).

Homework 9, Problem 5.

(a) For each $(u, v) \in E$ let $\alpha(u, v)$ be tte weight of (u, v) in G. Let also $\beta(u, v) = -\log(\alpha(u, v))$. The goal is to find a cycle $C = v_1, \ldots, v_k, v_1$ in G such that

$$\alpha(v_1, v_2) \cdot \alpha(v_2, v_3) \cdot \ldots \cdot \alpha(v_k, v_1) > 1.$$

This is equivalent to

$$-\log(\alpha(v_1, v_2)) - \log(\alpha(v_2, v_3)) - \ldots - \log(\alpha(v_k, v_1))) < 0.$$

That is, the goal is to decide whether G contains a negative cycle with respect to the weights β. This can be done using the Bellman-Ford algorithm.

(b) This can be done via a simple modification of the Bellmand-Ford algorithm.

Midterm 1, Problem 1. (a) Insert(1), Insert(2), ..., Insert(n).

(b) Perform $n/2$ left rotations on the root. Recurse on both children of the root. When recursing on a right child, perform left rotations. When recursing on a left child, perform right rotations.

Midterm 1, Problem 2. Due to a typographical error, this problem has two possible solutions. If you assume that all the 0s are stored as the $\sqrt{n}$ right-most leafs at the last level of the heap, then there will be $\Theta(n)$ extract-min operations with cost $\Theta(\log n)$; thus the running time is $\Theta(n \log n)$.

If you assume that all the 0s are stored in a small subtree towards the left side of the heap, then the total running time becomes $\Theta(n)$.

Midterm 1, Problem 3. (a) Scan the beach from left to right; when meeting someone at position x who is currently in the sun, place an umbrella covering the interval $[x, x + L]$.

(b) Observe that there exists an optimal solution where the umbrellas are non-overlapping. Thus it suffices to find such an optimal solution.

We may assume w.l.o.g. that
$$x_1 \leq \ldots \leq x_n.$$

Intuitively, the dynamic program computes for each possible location x_i, and for all possible values of ℓ, the maximum number of people we can cover with ℓ umbrellas, such that the right-most umbrella covers the interval $[x_i, x_i + L]$.

For any $j \in \{0, \ldots, n\}$, and $\ell \in \{1, \ldots, k\}$, let $S(j, \ell)$ be the maximum number of people that we can cover with at most ℓ umbrellas, such that the right-most umbrella is covers the interval $[x_j, x_j + L]$. For the boundary conditions we have

$$S(j, 1) = N_j, \quad \text{for all } j,$$

where N_j is the number of people contained in the interval $[x_j, x_j + L]$. For all $\ell > 1$, and for all j, we have
$$S(j, \ell) = \max \left\{ S(j, \ell - 1), \max_{j' \in \{1,\ldots,j''\}} \{S(j', \ell - 1) + N_j\} \right\},$$

where j'' is the maximum integer such that $x_{j''} + L < x_j$. Note that the term $S(j, \ell - 1)$ above is needed for the case where it is possible to cover all people with strictly less than k umbrellas.

The above recursion relation can be immediately turned into a dynamic program that computes the value of the optimum. Using the standard modification of dynamic programs, we may also compute the optimal placement for umbrellas.

Remark: A common mistake in the midterm was to use the following greedy algorithm: For $i = 1$ to k, find the interval the covers the maximum number of yet uncovered people and add it to the solution. The following example proves that this algorithm is incorrect: $x_1 = 1$, $x_2 = 3$, $x_3 = 4$, $x_4 = 5$, $x_5 = 6$, $x_6 = 8$, $L = 3$, $k = 2$. The set of intervals $\{[1, 4], [5, 8]\}$ covers everyone, thus this is an optimal solution. The greedy algorithm will first find the interval $[3, 6]$, since this contains the maximum number of people (initially, everyone is uncovered). Thus, the final solution of the greedy algorithm will contain three intervals; e.g. a possible solution of the greedy algorthm is $\{[3, 6], [1, 4], [8, 11]\}$.

Midterm 2, Problem 1. The expected time is $O(n^{2/3})$. The solution can be found in the book.

Midterm 2, Problem 2.

(a) $O(n \log n)$.

(b) We modify the potential function so that each insertion "pays" $O(\log n)$, where n is the number of elements currently in the Heap. Formally, we set

$$\Phi = t + 2m + \sum_{i=1}^{n} \log(1+i),$$

where t is the number of trees and m is the number of marked nodes. The additional factor of $O(\log n)$ can be used to pay for each extraction.

Midterm 2, Problem 3.

(a) Yes. This follows almost immediately from the definition.

(b) This is precisely the problem of computing a topological sort (see the relevant chapter in the book).

Final, Problem 1. Every spanning tree has $|V| - 1$ edges. Thus if we decrease the weight of each edge by 1, the weight of each spanning tree decreases by $|V| - 1$. Thus the minimum spanning tree remains the same.

Final, Problem 2. For each edge $(u, v) \in E$ set its weight to be $w(u, v) = -\log(p(u, v))$. Then the problem reduces to computing the shortest path from s to t with weights w.

Final, Problem 3.

(a) There exists infinitely many paths from s to t in G if and only if there exists a directed cycle C in G such that some vertex s' in C is reachable from s and t is reachable from some vertex t' in C. The algorithm for deciding whether there exist infinitely many paths from s to t is as follows: First we check whether t is reachable from s; if not, then the answer is clearly NO. We compute the set S of vertices that are reachable from s. This can be done by performing BFS starting from s. We also compute the set T of vertices u such that t is reachable from u. This can be done by reversing the directions of all edges and performing BFS starting from t. Let G' be the graph obtained from G by deleting all the vertices not in $S \cap T$. There are infinitely many paths from s to t if and only if there exists a cycle in G'; this can be checked by performing BFS starting from s.

Remark: A common mistake was to claim that there exist infinitely many paths from s to t if and only if there exists a cycle in G; this is clearly false.

(b) Computing the number of paths from s to t can be done via dynamic programming. For each $v \in V$ let $\alpha(v)$ be the number of paths from v to t. Initially we set $\alpha(t) = 1$. For each $u \in V$ with outgoing edges $(u, v_1), \ldots, (u, v_k)$, we have

$$\alpha(u) = \sum_{i=1}^{k} \alpha(v_i).$$

We can compute all these values by considering the vertices in the order that they are visited in a BFS starting from t with all the edges of G reversed.

Remark: A common mistake was to perform BFS from s and increment a counter every time you reach t (or something along these lines). This approach fails for the following reason: It is easy to construct graphs where there are exponentially many paths from s to t; however, BFS runs in polynomial time and thus cannot create a counter value grater than some polynomial.

Final, Problem 4. Negate all the edge weights and compute the all-pairs shortest paths. Output the shortest path found.

Final, Problem 5. Let $G = (V, E)$ be the input graph. We first construct a directed graph $G' = (V, E')$. For every undirected edge $\{u, v\} \in E$ we have two oppositely directed edges (u, v) and (v, u) in E'. Next we construct a new directed graph $G'' = (V'', E'')$. For each $v \in V$ we have two vertices $v_1, v_2 \in V''$. We also add a directed edge (v_1, v_2) in E'' with weight $w(v)$. For each $(x, y) \in E'$ we add an edge $(x_2, y_1) \in E''$. Finally we compute a shortest path in G'' from s_1 to t_2.

Final, Problem 6. We first check whether t is reachable from s; if not, then the answer is ∞. Thus we may assume that t is reachable from s. This implies that there exists a path from s to t that contains at most $|V| - 1$ edges. Clearly, all people can reach t in at most $k \cdot (|V| - 1)$ time steps (in fact, $k + |V| - 2$ steps are always sufficient). We will describe an algorithm which given some integer $M \geq 0$ decides whether all people can move to room t in at most M time steps. Then the minimum possible number of time steps can be computed by running this algorithm for $M = 0, 1, 2, \ldots, k \cdot (|V| - 1)$.

We construct a flow network $G' = (V', E')$ as follows. We set

$$V' = V \times \{0, \ldots, M\}.$$

That is, each vertex in G' is a pair (v, i) where v is a room and i is an integer in $\{0, \ldots, M\}$. Intuitively, this pair represents room v at time i. For each $(u, v) \in E$, and for all $i \in \{0, \ldots, M-1\}$, we add the edge $((u, i), (v, i + 1))$ in E' with capacity $c_{u,v}$. Intuitively, this edge encodes the fact that at most $c_{u,v}$ people that are located in room u at time i can move to room v at time $i + 1$ by traversing the corridor (u, v). Also, for each $v \in V$, and for each $i \in \{0, \ldots, M - 1\}$, we add the edge $((v, i), (v, i+1))$ with infinite capacity. Intuitively, this edge encodes the fact that an arbitrary number of people can stay in room v at step i. Finally, we compute a max-flow in G' from $(s, 0)$ to (t, M). If the value of the max-flow is at least k, then we can send at least k people from s to t in at most M steps, and thus the answer is YES; otherwise the answer is NO.

Remark: A common mistake was to claim that it is enough to compute a max-flow in the original graph. This is clearly not enough. Some attempts of a proof claimed this to be the case by arguing that the Edmonds-Karp algorithm augments the flow along shortest paths in the residual network. Note that this affects the running time of the Edmonds-Karp algorithm, but the final output is still just a max-flow. Thus this line of reasoning is also incorrect.