Assignment 3
Data Structures and Algorithms for AI
Computer Science Department, VU Amsterdam

Caspar Grevelhoerster

October 24, 2021

Instructions: This is the third assignment of the course. The assignment consists of three questions and they appear below. Your job is to answer the questions and submit your solutions in PDF format via Canvas.

Note 1. If you like to discuss something regarding the assignment, you are only allowed to discuss with your instructor.
Note 2. The deadline to submit your file is before the **end of Sunday October 24, 2021**. As stated in Section 7 of the syllabus of the course, late submissions will not be accepted.
Note 3. You can **use only** the algorithms and/or data structures that we have discussed in the course and **before publishing the assignment, which includes lectures 1–11**. Moreover, if you need, you can use such algorithms and/or data structures without discussing their details.
Note 4. If you prefer to present your algorithms in code, rather than in text, then you are required to present them in pseudocode and not an actual code (e.g., Python).

QUESTION 1. [6 POINTS] Please answer the following six short questions.

1. Define the following terms in your own words: *connected graph* and *strongly connected digraph*.

   A *connected graph* is a type of graph where it is possible to start at any node of that graph and from there by traversing over vertices to visit every other node. A *strongly connected digraph* (or strongly connected *directed graph*) is a type of graph where all edges have a direction and just like in a connected graph, it is possible to visit every other node in a strongly connected digraph after starting at any node of that graph.

2. Suppose the array 2, 1, 3, 5, 4, 6, 7, 9 has been obtained after finishing the first step of QUICK SORT on an (unknown) input array. Which of the keys in the given array could be the pivot? Write all the keys that could be the pivot. Also, briefly explain your answer.

   QUICK SORT takes the last element of an array and makes it a pivot. Afterwards, it splits the array into two sub-arrays, one with all values larger than the pivot and another with all values smaller than it. The pivot is placed in between those two sub-arrays and the quick sort algorithm initiates the next recursive iteration for both of the two sub-arrays. Following this definition, it becomes obvious that after the first iteration of the algorithm applied to an unknown array, the pivot can be any element n with a value x that is in a place where the set of all values left to the element S_L have values $< x$ and the set of all values S_R right to the element n have values $> x$. The possible elements that could have been pivots in the previous iteration are therefore: 3, 6, 7, and 9.

3. Is HEAP SORT a stable sorting algorithm or not? Briefly explain your answer. (You may assume that equal keys can be stored in a binary min-heap with the following slight modification. The key of each node has to be less than or equal to the keys of its children.)

   Heap Sort without modifications is not stable because in a heap structure, keys of the same value can be switched and might therefore not be in the same order after the algorithm has sorted the given array. This switching of nodes of the same values might happen because during a dequeue operation in a min-heap structure, a leaf is taken as the new root and then $heapify()$ is applied. Given a min-heap structure with $[1, 8_1, 8_2]$ where 1 is the root node and 8 are the two children of eight, then in the initial array, 8_1 comes before 8_2 but after dequeuing the root 1, the min-heap structure will be $[8_2, 8_1]$ - which proves that elements of the same value can be returned in switched respective positions. Even with the mentioned modification of the min-heap structure, this does not change.

4. Is the following statement TRUE or FALSE? Briefly explain your answer. COUNTING SORT can be easily modified so that it can handle singly linked-lists as its inputs, instead of arrays, without changing its asymptotic running time.

   In counting sort, the input (length n) as well as the counting array (length k) are for-looped over twice during execution of the algorithm. This makes for a time complexity of $O(n+k+n+k)$ which reduces to $O(n+k)$. If the input array is changed to a singly linked list, then the for loop needs to be adapted. Instead of iterating using an index i, the for-loop must loop using node.next and thereby visit the n elements in the input. The two differing for-loops adopted for a singly-linked list and an array, however, have the same time complexity. Thus, the overall time complexity of counting sort is unaffected and stays $O(n+k)$.

5. Is the following statement TRUE or FALSE? Briefly explain your answer. Let $G = (V, E)$ be an edge-weighted, connected and undirected graph with n vertices and m edges. If $m = \Theta(n)$, then Kruskal's algorithm finds a minimum spanning tree of G in $O(n \log n)$ time. If $m = \Theta(n^2)$, then Kruskal's algorithm finds a minimum spanning tree of G in $O(n^2)$ time.

   KRUSKAL'S ALGORITHM finds a minimum spanning tree in $m * log_2(n)$ time for a graph with m edges and n vertices. Thus, if m is tightly bounded by $\Theta(n)$, the running time is $O(n * log_2(n))$. If, however, m is tightly bounded by $\Theta(n^2)$, then it finds it in $O(n^2 * log_2(n))$ which is not further reducible. Since $O(n^2 * log_2(n)) \notin O(n^2)$, the statement is FALSE.

6. Let G be a connected graph with n vertices. What can be the maximum number of edges of G? Briefly explain your answer.

   A graph with n vertices can have a maximum number of edges of $\frac{n*(n-1)}{2}$. This formula arises because each vertex n can connect to every other vertex but itself, hence to $n-1$ vertices. If this connected graph is assumed to be an undirected graph where two opposite directed edges are counted as one edge ($A \rightarrow B$, $B \rightarrow A$ is counted as one edge), this result needs to be halved (thus be multiplied with $\frac{1}{2}$).

QUESTION 2. [2 POINTS] A thief robbing a store finds n items. The i-th item is worth v_i dollars and weights w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (This problem is known as $0/1$-knapsack because for each item, the thief must either take it or leave it behind—he cannot take a fractional amount of an item or take an item more than once.)

A brute-force algorithm for the problem works as follows. Let $\mathcal{S}$ be the set of all subsets of the n items. For each set $S \in \mathcal{S}$, compute the sum of the weights of its items and let $\mathcal{S}_{\leq W}$ be the set of all subsets in $\mathcal{S}$ that the sum of the weights of their items is less than or equal to W. Clearly, $\mathcal{S}_{\leq W}$ is the set of only subsets that the thief can carry, but which one should he carry? Well, for each of them, compute the sum of the values of its items and pick the one that gives the maximum value. This algorithm works, but its running time is in $\Omega(2^n)$.

The goal of this exercise is to develop a dynamic-programming algorithm for this problem. To this end, we do the followings. Let $S = \{a_1, a_2, \ldots, a_n\}$ be the sets of items and further let define the following subproblem. For any two variables $1 \leq i \leq n$ and $1 \leq j \leq W$, let $c(i, j)$ be the value of an optimal solution for the 0/1-knapsack problem in which the set of items are the first i items in S, namely $S_i = \{a_i, a_2, \ldots, a_i\}$, and that the capacity of the knapsack is only j pounds. Ultimately, we wish to compute the value of $c[n, W]$.

QUESTION 2-1. Write a recursive formula for computing $c[i, j]$. (Hint. You can think of c as a table with n rows and W columns and that when you want to compute the value of $c[i, j]$, an optimal solution to all subproblems of smaller sizes have already been computed and stored in c. That is, all rows before row i, and in row i all columns before column j have already been computed and filled in. In order to compute $c[i, j]$, consider whether to include item a_i into the knapsack or not.)

$$c[i,w]) = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w < w_i \\ max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } 0 < i \text{ and } w_i \leq w \end{cases}$$

QUESTION 2-2. Consider converting the recursive formula in **QUESTION 2-1** into an algorithm that computes all entries of table c and thus it finally computes the value of $c[n, W]$. What is the running time of this algorithm? Briefly explain your answer.

The running time of an algorithm using this function that computes the integer matrix values would have $\Theta(W * n)$ time. This is because there are W columns and n rows and each item in the matrix needs to be computed. Afterwards, the result needs to be traversed in the matrix which takes $O(n)$ time, but this is ignored in the tight bound of $\Theta(W * n)$.

QUESTION 3. [2 POINTS] Alice has an unweighted and connected graph with n vertices and m edges, in which $m = O(n \log n)$. For two vertices s and v in G, Alice wants to find the shortest path from s to v in G. She has two algorithms in mind to achieve this:

- **Algorithm 1.** Do a BFS traversal on G, starting from s.
- **Algorithm 2.** Run Dijkstra's algorithm on G, starting from s, in which the algorithm enjoys a Fibonacci heap in its implementation.

Present the running times of each of the two algorithms separately, and select the algorithm (out of the two algorithms mentioned above) that you would suggest to Alice by determining which one is asymptotically faster. (You do not need to analyze the running times of BFS and Dijkstra's algorithms, as we have already done in the course.)

The running time of BFS traversal is $\Theta(m + n)$. Now, if $m = O(n * log_2(n))$, then the running time of BFS will be $\Theta(n + n * log_2(n))$ which reduces to $\Theta(n * log_2(n))$.
The running time of Dijkstra's algorithm with a Fibonacci heap is $O(m + n * log_2(n))$. Now, if $m = O(n * log_2(n))$, then the running time of Dijkstra's algorithm is $O(n * log_2(n) + n * log_2(n))$ which reduces to $O(n * log_2(n))$.

At first glance, the two running times of $\Theta(n * log_2(n))$ and $O(n * log_2(n))$ look like they are the same but the difference lies in O and the tight bound Θ. Since O of Dijkstra's algorithm is in the worst case and might be faster than that but the tight bound Θ of BFS will never be faster, it is advisable to use Dijkstra's algorithm with a Fibonacci heap implementation for this problem.