Teacher : Ola Svensson
Algorithms CS-250
8 Nov 2021
Duration : 120 minutes

1

Solutions

SCIPER : **111111**

Do not turn the page before the start of the exam. This document is double-sided, has 8 pages, the last ones possibly blank. Do not unstaple.

- Place your student card on your table.
- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- The exam consists of two parts. The first part consists of five multiple choice questions and the second part consists of three open ended questions.
- Use a **black or dark blue ballpen** and clearly erase with **correction fluid** if necessary.
- If a question is wrong, the teacher may decide to nullify it.

Good luck!

First part: multiple choice questions

For each question, mark the box corresponding to the correct answer. Each question has **exactly one** correct answer. You do not need to justify your answers in this part.

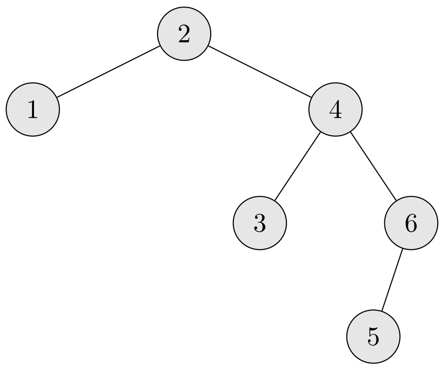
Question 1 *(8 pts)* Consider executing the procedure BUILD-MAX-HEAP(A, n) seen in class on input

$$A = \boxed{6 \mid 7 \mid 2 \mid 9 \mid 4 \mid 8 \mid 5 \mid 3 \mid 1} \qquad \text{and} \qquad n = 9\,.$$

Then the resulting heap A equals

- [✓] $A = \boxed{9 \mid 7 \mid 8 \mid 6 \mid 4 \mid 2 \mid 5 \mid 3 \mid 1}$
- [ ] $A = \boxed{9 \mid 8 \mid 7 \mid 6 \mid 5 \mid 4 \mid 3 \mid 2 \mid 1}$
- [ ] $A = \boxed{9 \mid 6 \mid 8 \mid 7 \mid 4 \mid 2 \mid 5 \mid 3 \mid 1}$
- [ ] $A = \boxed{9 \mid 8 \mid 7 \mid 2 \mid 5 \mid 6 \mid 4 \mid 3 \mid 1}$
- [ ] $A = \boxed{9 \mid 7 \mid 8 \mid 2 \mid 5 \mid 6 \mid 4 \mid 3 \mid 1}$

Question 2 *(8 pts)* Let T be the following binary search tree:



What is the output when executing the following pseudocode?

```
1. Let $S$ be an empty stack
2. PUSH(S, T.head)
3. **while** $S$ is non-empty
4.     $x = $ POP(S)
5.     print $x.key$
6.     **if** x.left $\neq$ NIL
7.         PUSH(S, x.left)
8.     **if** x.right $\neq$ NIL
9.         PUSH(S, x.right)
```

- [✓] The output is $2, 4, 6, 5, 3, 1$.
- ☐ The output is $1, 2, 3, 4, 5, 6$.
- ☐ The output is $6, 5, 4, 3, 2, 1$.
- ☐ The output is $2, 1, 4, 3, 6, 5$.
- ☐ The output is $1, 3, 5, 6, 4, 2$.

Question 3 *(8 pts)* Give tight asymptotic bounds for the following three recurrences

$$F(n) = 81 \cdot F(n/9) + \Theta(\sqrt{n}), \quad G(n) = 9 \cdot G(n/9) + \Theta(n), \quad H(n) = 3 \cdot H(n/9) + \Theta(\sqrt{n}),$$

where we assume $F(n) = \Theta(1), G(n) = \Theta(1), H(n) = \Theta(1)$ for $n \leq 10$.

- [✓] $F(n) = \Theta(n^2)$, $G(n) = \Theta(n \log n)$, $H(n) = \Theta(\sqrt{n} \log n)$.
- ☐ $F(n) = \Theta(n^2 \log n)$, $G(n) = \Theta(n)$, $H(n) = \Theta(\sqrt{n})$.
- ☐ $F(n) = \Theta(n^2)$, $G(n) = \Theta(n)$, $H(n) = \Theta(\sqrt{n})$.
- ☐ $F(n) = \Theta(n^2 \log n)$, $G(n) = \Theta(n)$, $H(n) = \Theta(\sqrt{n})$.
- ☐ $F(n) = \Theta(n^2 \log n)$, $G(n) = \Theta(n)$, $H(n) = \Theta(\sqrt{n} \log n)$.
- ☐ $F(n) = \Theta(n^2)$, $G(n) = \Theta(n \log n)$, $H(n) = \Theta(\sqrt{n})$.
- ☐ $F(n) = \Theta(n^2)$, $G(n) = \Theta(n)$, $H(n) = \Theta(\sqrt{n} \log n)$.
- ☐ $F(n) = \Theta(n^2 \log n)$, $G(n) = \Theta(n \log n)$, $H(n) = \Theta(\sqrt{n} \log n)$.

Question 4 *(8 pts)* Consider an array $A[1,\ldots,n]$ consisting of the n distinct numbers $1, 2, \ldots, n$. We are further guaranteed that A is almost sorted in the following sense: $A[i] \neq i$ for at most $\sqrt{n}$ values of i. What are tight asymptotic worst-case running times for Insertion Sort and Merge Sort on such instances?

- [✓] It is $\Theta(n \log n)$ for Merge Sort and $\Theta(n^{3/2})$ for Insertion Sort.
- [ ] It is $\Theta(n + \sqrt{n} \log n)$ for Merge Sort and $\Theta(n^{3/2})$ for Insertion Sort.
- [ ] It is $\Theta(n \log n)$ for Merge Sort and $\Theta(n)$ for Insertion Sort.
- [ ] It is $\Theta(n + \sqrt{n} \log n)$ for Merge Sort and $\Theta(n)$ for Insertion Sort.
- [ ] It is $\Theta(n \log n)$ for Merge Sort and $\Theta(n^2)$ for Insertion Sort.
- [ ] It is $\Theta(n + \sqrt{n} \log n)$ for Merge Sort and $\Theta(n^2)$ for Insertion Sort.

Question 5 *(8 pts)* We consider the change-maker problem that we solved in class with the additional difficulty that the coins now have weights. We are interested in solving a specific instance where we have six different coin types with denominators and weights given by the following table:

	Coin type 1	Coin type 2	Coin type 3	Coin type 4	Coin type 5	Coin type 6
Denominator	1	3	5	7	25	50
Weight	6	10	11	14	47	95

We wish to calculate the minimum weight of coins needed to make the change 102. Formally, if we let x_i denote the number of coins we pick of type i then we wish to minimize $6x_1+10x_2+11x_3+14x_4+47x_5+95x_6$ over all non-negative integers $x_1, \dots, x_6$ that satisfy $x_1 + 3x_2 + 5x_3 + 7x_4 + 25x_5 + 50x_6 = 102$.
We realize that it is a good idea to use dynamic programming and write a recurrence where

$$r[C] = \text{"minimum weight of coins needed to give change } C\text{"}.$$

We are thus interested in the value of $r[102]$. The values of $r[C]$ for $C = 0, 1, \dots, 101$ are as follows:

$r[0] = 0$	$r[17] = 36$	$r[34] = 72$	$r[51] = 100$	$r[68] = 136$	$r[85] = 163$
$r[1] = 6$	$r[18] = 42$	$r[35] = 69$	$r[52] = 105$	$r[69] = 133$	$r[86] = 169$
$r[2] = 12$	$r[19] = 39$	$r[36] = 75$	$r[53] = 103$	$r[70] = 138$	$r[87] = 166$
$r[3] = 10$	$r[20] = 44$	$r[37] = 72$	$r[54] = 108$	$r[71] = 136$	$r[88] = 172$
$r[4] = 16$	$r[21] = 42$	$r[38] = 78$	$r[55] = 105$	$r[72] = 141$	$r[89] = 169$
$r[5] = 11$	$r[22] = 47$	$r[39] = 75$	$r[56] = 111$	$r[73] = 147$	$r[90] = 174$
$r[6] = 17$	$r[23] = 53$	$r[40] = 80$	$r[57] = 108$	$r[74] = 144$	$r[91] = 180$
$r[7] = 14$	$r[24] = 50$	$r[41] = 86$	$r[58] = 114$	$r[75] = 141$	$r[92] = 177$
$r[8] = 20$	$r[25] = 47$	$r[42] = 83$	$r[59] = 119$	$r[76] = 147$	$r[93] = 183$
$r[9] = 26$	$r[26] = 53$	$r[43] = 89$	$r[60] = 116$	$r[77] = 152$	$r[94] = 180$
$r[10] = 22$	$r[27] = 58$	$r[44] = 86$	$r[61] = 122$	$r[78] = 150$	$r[95] = 185$
$r[11] = 28$	$r[28] = 56$	$r[45] = 91$	$r[62] = 119$	$r[79] = 155$	$r[96] = 183$
$r[12] = 25$	$r[29] = 61$	$r[46] = 89$	$r[63] = 125$	$r[80] = 152$	$r[97] = 188$
$r[13] = 31$	$r[30] = 58$	$r[47] = 94$	$r[64] = 122$	$r[81] = 158$	$r[98] = 194$
$r[14] = 28$	$r[31] = 64$	$r[48] = 100$	$r[65] = 127$	$r[82] = 155$	$r[99] = 191$
$r[15] = 33$	$r[32] = 61$	$r[49] = 97$	$r[66] = 133$	$r[83] = 161$	$r[100] = 188$
$r[16] = 39$	$r[33] = 67$	$r[50] = 94$	$r[67] = 130$	$r[84] = 166$	$r[101] = 194$

What is the value of $r[102]$, i.e., the minimum weight of coins needed to give change 102?

- [✓] The value of $r[102]$ is 199.
- [ ] The value of $r[102]$ is 200.
- [ ] The value of $r[102]$ is 198.
- [ ] The value of $r[102]$ is 201.
- [ ] The value of $r[102]$ is 197.
- [ ] The value of $r[102]$ is 196.
- [ ] The value of $r[102]$ is 195.
- [ ] The value of $r[102]$ is 202.
- [ ] The value of $r[102]$ is 194.
- [ ] The value of $r[102]$ is 193.

Second part, open questions

This part consists of three questions, each worth 20 points. Please follow the following instructions:

- Your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudocode without explanations. A good guideline is that a description of an algorithm should be such that a fellow student can easily implement the algorithm following the description.
- You are allowed to refer to material covered in the lectures including algorithms and theorems (without reproving them). You are however *not* allowed to simply refer to material covered in exercises.
- Please answer all questions within the designated boxes (otherwise your answer may not be accurately scanned). At the end of the exam there are five extra pages if you need additional space for your answers.
- Leave the check-boxes empty, they are used for the grading.

Question 6: Train sequences at Lausanne Gare. *(20 pts)*

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10
☐ 11 ☐ 12 ☐ 13 ☐ 14 ☐ 15 ☐ 16 ☐ 17 ☐ 18 ☐ 19 ■ 20

You and your friend are watching the trains at Lausanne Gare. The tracks at Lausanne Gare consist of one main track and one parallel side track as shown in Figure 1. You sit at the left end of the tracks and see that n trains labeled $n, n-1, \ldots, 1$ are lined up so that the first train on the left track is 1, the second train is 2, and so on. Then the trains start moving to the right one by one, first train 1, then 2 and so on. Any train may either turn onto the side track (the side track can hold an infinite number of trains) or keep moving right and leave the station. At any point in time the rightmost train on the sidetrack can choose to leave the station through the right track. However, the trains can never go back to the left track.

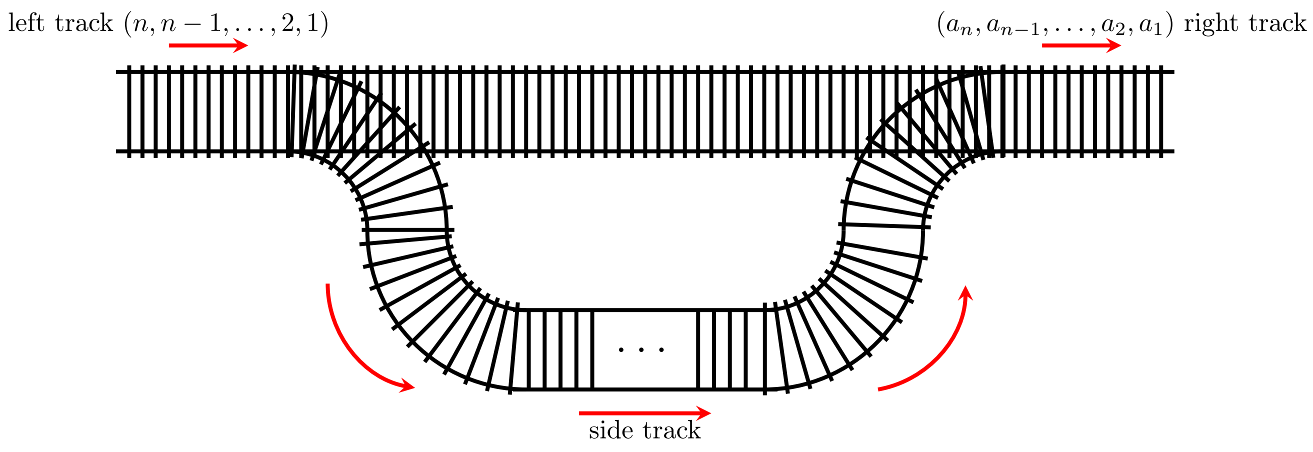

left track $(n, n-1, \ldots, 2, 1)$
$(a_n, a_{n-1}, \ldots, a_2, a_1)$ right track
side track

Figure 1: Lausanne Gare

Your friend sits at the right end of the tracks, observes the trains leaving one by one and writes down their numbers in the order that they leave. Your task is to develop an efficient algorithm that decides, given your friend's notes, if that order of trains is possible or if your friend had made a mistake.

Input: A permutation $a_1, a_2, \ldots, a_n$ of integers between 1 and n.

Output: **YES** if there exists a way for trains $1, 2, \ldots, n$ to leave Lausanne Gare in this order, and **NO** otherwise.

For example, suppose there are $n = 3$ trains. Then order $2, 1, 3$ (i.e., $a_1 = 2, a_2 = 1, a_3 = 3$) is possible: the first train turns onto the side track, the second train leaves, the first train exits the side track and leaves and finally the third train leaves as well. However, the order $3, 2, 1$ is not possible.

Design an efficient algorithm for this problem and **analyze its runtime**.

(In this question you are asked to design and analyze an algorithm for the Lausanne Gare problem that runs in time polynomial in n. Recall that you are allowed to refer to material covered in the lectures. We remark that this question is similar to a question solved in class. However, the same solution does not work for this version as the side track functions differently.)

Solution. This question is similar to the train problem seen in class, except that the side track acts as a **Queue** (Q in pseudocode). Now, we need a variable to keep track of the smallest numbered train still in the main track (left-track), which we call as j in the pseudocode below and we need a variable to iterate through our friend's list i.
For each value of $A[i]$, on of the following holds:

(a) This train might be the current smallest numbered train in the left track (main-track), in which case the train simply goes through without going through the side-track.

(b) This train might be the head of the Queue (in front of the side-track), in which case, it can be Dequeued.

(c) If neither of the above conditions are satisfied, we can only send the current smallest train in the left-track to the side-track, i.e, Enqueue it to the side-track.

(d) Finally, if we have no trains left on the main track, and we are not able to Dequeue from the side-track, then we run into a contradiction (return FALSE).

If we go through the friend's list without running into a contradiction, the sequence was possible.

Algorithm 1 Lausanne-Gare(A)

```
$i \leftarrow 1$
$j \leftarrow 1$
$var \leftarrow TRUE$
Initialize Empty Queue Q.
while $i \leq n$ and var == TRUE do
    if j == A[i] then
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if !Q.isEmpty and Q.head == A[i] then
        Dequeue(Q)
        $i \leftarrow i + 1$
    else if $j \leq n$ then
        Enqueue(Q,j)
        $j \leftarrow j + 1$
    else
        $var \leftarrow FALSE$
    end if
end while
return var
```

Each train is sent directly from the main-track or Enqueued into the side-track. This requires $\Theta(n)$ operations. Now, the trains in the side-track are Dequeued once if it matches with our friend's list or we return False and this is again an additional $\Theta(n)$ operations. Thus in total we have that the running time is $\Theta(n)$.

Question 7: Common elements. *(20 pts)*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9 ☐10
☐11 ☐12 ☐13 ☐14 ☐15 ☐16 ☐17 ☐18 ☐19 ■20

An element x is said to be *common* in an array $A[1,\ldots,n]$ if at appears in at least $\lceil n/3 \rceil$ entries of A. In other words, at least 1/3 of the elements of A equals x. For example, the common elements in the array

$\bigoplus$	$\bigotimes$	$\bigoplus$	$\bigotimes$	$\bigotimes$	$\bigodot$

are $\bigoplus$ and $\bigotimes$. In the arrays we consider, the elements are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as images, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time. **Design** and **analyze** an efficient algorithm for the following problem:

Input: An array $A[1,\ldots,n]$ of n elements.

Output: All common elements of A or "None" if A has no common elements.

Your algorithm should run in time $O(n \log n)$; it cannot perform comparisons of the form "is $A[i] > A[j]$?" but it can check equality "is $A[i] = A[j]$?' in constant time. Furthermore, your algorithm should not use randomness so you cannot use e.g. hash functions/tables (if you do not understand this sentence, don't worry: you can use all techniques we have seen so far in the course).

Hint: use divide-and-conquer.

(In this question you are asked to design and analyze an algorithm that runs in time $O(n \log n)$ for the problem of outputting the common elements of a given array A (or output None if no common element exists). You are not allowed to perform comparisons of the form "is A[i] > A[j]?" so you cannot sort the elements. Moreover, your algorithm should not use randomness so you cannot use hash functions/tables. Recall that you are allowed to refer to material covered in the lectures.)

Solution.
We proceed via a divide and conquer approach. If $n \geq 2$, we split the instance in half and recursively solve the resulting smaller sub-instances. If $n = 1$, then the element the array contains is trivially common. Note that any given sub-array has at most 3 common elements. Thus for each sub-array we maintain a list containing the common elements. Note that if we are merging two sub-arrays, a common element of the resulting array must be common in at least one of the two arrays. Thus to merge two sub-instances, we count the number of occurrences of the at most 6 elements that form the set elements that are common in either sub-array. This procedure has a runtime of $\Theta(n)$. Noting that the base case and divide steps have runtime $\Theta(1)$, we get the recurrence relation

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n \geq 2 \\ O(1) & \text{otherwise} \end{cases}.$$

We then apply the Master Theorem to obtain a runtime of $O(n \log n)$ as desired.

Question 8: Number of ways to partition a rod. *(20 pts)*

0 1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 [✓ 20]

Design and analyze an algorithm for the following problem:

Input: A rod of length n where n is a non-negative integer.

Output: The number

$$\left|\left\{(x_1, x_2, \ldots, x_n) \mid \; x_i\text{'s are non-negative integers such that } \sum_{i=1}^{n} i \cdot x_i = n \;\right\}\right|$$

of distinct ways to partition the rod by potentially cutting it into smaller pieces.

The running time of your algorithm should be polynomial in n. While your algorithm should run in polynomial time, it is not important that you achieve an optimal running time in this question.

We remark that the number that you should output does not care in which order you cut the rod (only the resulting sizes). For example, a rod of length 4 can be partitioned in five distinct ways:

$$\begin{gathered} 4 \\ 3+1 \\ 2+2 \\ 2+1+1 \\ 1+1+1+1 \end{gathered}$$

So the output of your algorithm should be 5 on input 4.

(In this question you are asked to design and analyze an algorithm that runs in time polynomial in n for the problem of calculating the number of distinct ways to partition a rod of length n. Recall that you are allowed to refer to material covered in the lectures.)

Solution.

Solution-1

We will use dynamic programming to solve this question. Let $r[\ell, i]$ denote the number of different ways to cut a rod of length ℓ such that the length of largest cut is atmost i. We can make cuts of length i as many as $0, 1, \ldots, \lfloor \frac{\ell}{i} \rfloor$ times. After cutting pieces of length i, we can cut the remaining rod such that the length of largest cut is atmost $i-1$ instead of i. This gives us the following recurrence relation:

$$r[l, i] = \sum_{k=0}^{\lfloor \ell/i \rfloor} r[\ell - k * i, i-1]$$

Runtime Analysis Since we need to fill in a table of dimensions $L \times L$, the two outer for loops take $\Theta(L^2)$ time. The while loop runs atmost in $\mathcal{O}(L)$ time, together the time complexity is $\mathcal{O}(L^3)$. With a more careful analysis, we can show that it is infact $\mathcal{O}(L^2 \log L)$ since the while loop iterates only as many as $\lfloor L/i \rfloor$ times (except when $i = 1$ where it runs L times).

```
Algorithm: COUNT-RODS($L$)
    for $i = 0$ to $L$ do
        r[0, i] = 1
    end
    for $\ell = 1$ to $L$ do
        r[$\ell$, 0] = 0
    end
    for $i = 1$ to $L$ do
        for $\ell = 1$ to $L$ do
            $K = \lfloor \ell / i \rfloor$
            $k = 0$
            $ans = 0$
            while $k \leq K$ do
                $ans = ans + r[\ell - k * i, i - 1]$
                $k = k + 1$
            end
            $r[\ell, i] = ans$
        end
    end
    return $r[L, L]$
```

Solution-2

Another way to count distinct cuts of the rod is to count them either in increasing or decreasing order. Consider again $r[\ell, i]$ as defined above. Then we have the following recurrence relation if we count cuts only in decreasing order:

$$r[l, i] = \sum_{k=1}^{i} r[\ell - k, k]$$

where we make the leftmost cut of size k and force the remaining rod of length $\ell - k$ to use pieces of at most size k.

Runtime Analysis This solution runs in $\mathcal{O}(L^3)$.

Solution-3

An alternative but elegant way to perform the same counting as above is the following recurrence:

$$r[l, i] = r[\ell - i, i] + r[\ell, i - 1]$$

This solution is more difficult to be proved of correctness. An approach to this would be to verify that 1) it counts all combinations at least once 2) verify that it does not double count anything (disjointness of 2 terms in the recurrence).

Runtime Analysis Since there is not summation in the recurrence, this solution runs in $\mathcal{O}(L^2)$ and is the fastest amongst the three.