

Banner number:

Name:

Final Exam

CSCI 3110: Design and Analysis of Algorithms

Dec 15, 2015

| Group 1 | | Group 2 | | Group 3 | | Σ |
|--------------|--|--------------|--|--------------|--------------------------|----------|
| Question 1.1 | | Question 2.1 | | Question 3.1 | <input type="checkbox"/> | |
| Question 1.2 | | Question 2.2 | | Question 3.2 | <input type="checkbox"/> | |
| Question 1.3 | | Question 2.3 | | Question 3.3 | <input type="checkbox"/> | |
| Σ | | Σ | | Σ | | |

Instructions:

- The questions are divided into three groups: Group 1 (36%), Group 2 (36%), and Group 3 (28%). You have to answer **all questions in Groups 1 and 2** and **exactly two questions in Group 3**. In the above table, put a check mark in the **small** box beside the question in Group 3 you want me to mark. If you select none or both questions, I will randomly choose which one to mark.
- Provide your answer in the box after each question. If you absolutely need extra space, use the backs of the pages; but try to avoid it. Keep your answers short and to the point.
- **You are not allowed to use a cheat sheet.**
- **Make sure your answers are clear and legible. If I can't decipher an answer or follow your train of thought with reasonable effort, you'll receive 0 marks for your answer.**
- If you are asked to design an algorithm and you cannot design one that achieves the desired running time, design a slower algorithm that is correct. A correct and slow algorithm earns you 50% of the marks for the algorithm. A fast and incorrect algorithm earns 0 marks.
- When designing an algorithm, you are allowed to use algorithms and data structures you learned in class as black boxes, without explaining how they work, as long as these algorithms and data structures do not directly answer the question.
- **Read every question carefully before answering. In particular, do not waste time on an analysis if none is asked for, and do not forget to provide one if it is required.**
- **Do not forget to write your banner number and name on the top of this page.**
- **This exam has 12 pages, including this title page. Notify me immediately if your copy has fewer than 12 pages.**

Question 1.1 (Asymptotic growth of functions)**10 marks**(a) *Formally* define the set $o(f(n))$.

$o(f(n))$ is a set of functions. A function $g(n)$ belongs to $o(f(n))$ if, for all $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

(b) *Formally* define the set $\Omega(f(n))$.

$\Omega(f(n))$ is a set of functions. A function $g(n)$ belongs to $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $g(n) \geq c \cdot f(n)$.

Question 1.2 (Average-case analysis and randomization)**6 marks**

We studied three variants of Quick Sort in class. They differ in how they choose the pivot around which they partition the input. Worst-Case Quick Sort uses the same strategy as worst-case linear-time selection to find an approximate median as pivot. Simple Quick Sort uses the last input element as pivot. Randomized Quick Sort randomly picks one of the input elements as pivot. Compare these three algorithms according to the following three properties. Write your answers into the table. The first column asks you to list the worst-case running times of the algorithms. The second column asks you to list their expected running times. The final column asks, for each algorithm, whether it has a worst-case input, that is, an input that forces it to achieve its worst-case running time.

| Algorithm | Worst-case running time | Expected running time | Worst-case input |
|-----------------------|-------------------------|-----------------------|------------------|
| Worst-Case Quick Sort | $O(n \lg n)$ | $O(n \lg n)$ | yes |
| Simple Quick Sort | $O(n^2)$ | $O(n \lg n)$ | yes |
| Randomized Quick Sort | $O(n^2)$ | $O(n \lg n)$ | no |

Question 1.3 (Complexity classes)

9 marks

(a) Formally define the complexity class P

P is the class of all formal languages that can be decided in polynomial time. Formally, a language $L \subseteq \Sigma^$ belongs to P if there exists an algorithm D which, for any string $x \in \Sigma^*$, answers yes if and only if $x \in L$, and the running time of D on any string $x \in \Sigma^*$ is in $O(|x|^c)$, for some constant c.*

(b) Formally define the complexity class NP.

NP is the class of all formal languages that can be verified in polynomial time. Formally, a language $L \subseteq \Sigma^$ belongs to NP if there exists a language $L' \subseteq \Sigma^* \times \Sigma^*$ such that $L' \in P$ and any string $x \in \Sigma^*$ belongs to L if and only if there exists a string $y \in \Sigma^*$ with $|y| \in O(|x|^c)$ and such that $(x, y) \in L'$.*

(c) Formally define what an NP-hard language is.

A language L is NP-hard if $L \in P$ implies that $P = NP$.

Question 2.1 (What does it do?)

8 marks

Consider the following simple algorithm:

MAGIC(x, y)

```
1  if  $y = 0$ 
2    then return 0
3  if  $y$  is even
4    then return MAGIC( $2 \cdot x, y \text{ div } 2$ )
5    else return MAGIC( $2 \cdot x, y \text{ div } 2$ ) +  $x$ 
```

The input consists of two non-negative integers x and y and the return value is another integer z . **div** denotes integer division: $x \text{ div } y = \lfloor x/y \rfloor$. State what the algorithm computes, that is, state the relationship between x , y , and z . Prove that this is indeed what the algorithm computes.

The algorithm returns the product $x \cdot y$.

To prove this, we use induction on y .

If $y = 0$, the algorithm returns 0, which is equal to $x \cdot y$ in this case.

If $y > 0$ and y is even, then the algorithm returns whatever MAGIC($2 \cdot x, \lfloor y/2 \rfloor$) returns. Since $\lfloor y/2 \rfloor < y$ for $y > 0$, the inductive hypothesis states that this return value is $(2 \cdot x) \cdot \lfloor y/2 \rfloor = (2 \cdot x) \cdot (y/2) = x \cdot y$. The first equality follows because y is even, so $\lfloor y/2 \rfloor = y/2$.

If $y > 0$ and y is odd, then by the same argument as in the previous case, the algorithm returns $x + (2 \cdot x) \cdot \lfloor y/2 \rfloor$. Since y is odd, we have $\lfloor y/2 \rfloor = (y-1)/2$, so $x + (2 \cdot x) \cdot \lfloor y/2 \rfloor = x + (2 \cdot x) \cdot (y-1)/2 = x + x \cdot (y-1) = x + x \cdot y - x = x \cdot y$.

Question 2.2 (Algorithm analysis)

8 marks

Express the running time of the algorithm from Question 2.1 as a function of y . Prove that this is indeed the algorithm's running time by giving a recurrence for the running time and solving this recurrence using the Master Theorem, substitution or a recursion tree. Whichever method you use, show the steps you take to solve the recurrence.

If $y = 0$, the algorithm returns in constant time. Otherwise, it spends constant time and makes one recursive call. The second argument of the recursive call is $\lfloor y/2 \rfloor$, so the running time of the algorithm is given by the recurrence

$$T(y) = T(\lfloor y/2 \rfloor) + \Theta(1).$$

Ignoring floors, as we agreed to do in this course, this becomes

$$T(y) = T(y/2) + \Theta(1).$$

Now, $y^{\log_2 1} = y^0 = 1$, so the second case of the Master Theorem applies to this recurrence and gives the solution, $T(y) \in \Theta(\lg y)$.

Question 2.3 (Polynomial-time reductions)

9 marks

Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ be two formal languages. Assume L_1 is NP-hard and there exists a polynomial-time reduction R from L_1 to L_2 . Prove that this implies that L_2 is also NP-hard.

We need to prove that $L_2 \in P$ implies that $P = NP$. Since $L_1 \in P$ implies that $P = NP$, it suffices to prove that $L_2 \in P$ implies that $L_1 \in P$.

So assume $L_2 \in P$, that is, there exists a decision algorithm D_2 such that, for any $x \in \Sigma^$, $D_2(x) = \text{true}$ if and only if $x \in L_2$; the running time of D_2 on input x is in $O(|x|^{c_2})$ for some constant c_2 .*

We construct a decision algorithm D_1 for L_1 as $D_1(x) = D_2(R(x))$, that is, we first apply R to the input of D_1 , then pass the result $R(x)$ to D_2 , and return the answer this invocation of D_2 returns. Since $x \in L_1 \Leftrightarrow R(x) \in L_2$ (R is a reduction from L_1 to L_2) and $R(x) \in L_2 \Leftrightarrow D_2(R(x)) = \text{true}$ (D_2 decides L_2), we have $x \in L_1 \Leftrightarrow D_2(R(x)) = D_1(x) = \text{true}$, that is, D_1 decides L_1 .

The running time of D_1 on input x is the cost of running R on x plus the cost of running D_2 on $R(x)$. Since R is a polynomial-time reduction, its running time on input x is in $O(|x|^c)$ for some constant c . In time $O(|x|^c)$, R can produce an output of size at most $O(|x|^c)$, so $|R(x)| \in O(|x|^c)$. The running time of D_2 on $R(x)$ is in $O(|R(x)|^{c_2}) \subseteq O(|x|^{cc_2})$. Thus, the total cost of D_1 is in $O(|x|^c + |x|^{cc_2})$, which is polynomial in $|x|$.

Since D_1 decides L_1 and its running time on any input x is polynomial in $|x|$, $L_1 \in P$, which is what we had to show.

Question 3.1 (Greedy algorithms)

10 marks

Consider a set of n jobs you want to run on a single computer. Let d_i be the *duration* of the i th job, that is, the amount of time it takes to run this job to completion. If you choose some permutation π of the jobs and you start each job immediately after the previous job in the permutation finishes, then the finish time of job $\pi(i)$ is $t_\pi(\pi(i)) = \sum_{j=1}^i d_{\pi(j)}$. We call $t_\pi(i)$ the *completion time* of job i . Your goal is to find the permutation π that minimizes the average completion time $\bar{t}_\pi = \frac{1}{n} \sum_{i=1}^n t_\pi(i)$. Develop an algorithm that solves this problem in $O(n \lg n)$ time, argue briefly that this is indeed the running time of your algorithm, and prove that the permutation π it produces does indeed minimize \bar{t}_π .

The algorithm is ridiculously simple: We sort the jobs by increasing duration. This clearly takes $O(n \lg n)$ time if we use an $O(n \lg n)$ sorting algorithm, such as Merge Sort.

To prove that this algorithm minimizes the average completion time \bar{t}_π , we consider a permutation π^ such that $\bar{t}_{\pi^*} \leq \bar{t}_\pi$ for all possible permutations π , that is, π^* minimizes \bar{t}_{π^*} .*

If the i th job in π^ is the i th shortest job for all $1 \leq i \leq n$, then the jobs are sorted by increasing duration, so π^* is exactly the permutation we produce (modulo rearrangement of jobs of equal length, which has no impact on the average completion time). In other words, the permutation π our algorithm produces minimizes \bar{t}_π .*

If there exists an index i such that job $\pi^(i)$ is not the i th shortest job, we consider the smallest such index i . In other words, for $1 \leq j < i$, job $\pi^*(j)$ is the j th shortest job. This implies that job $\pi^*(i)$ is the i' th shortest job for some $i' > i$. In particular, job $\pi^*(i)$ is at least as long as the i th shortest job. Now let i'' be the index such that $\pi^*(i'')$ is the i th shortest job. Again, since job $\pi^*(j)$ is the j th shortest job for all $1 \leq j < i$ and job $\pi^*(i)$ is not the i th shortest job, $i'' > i$. Finally, we define a new permutation π' as*

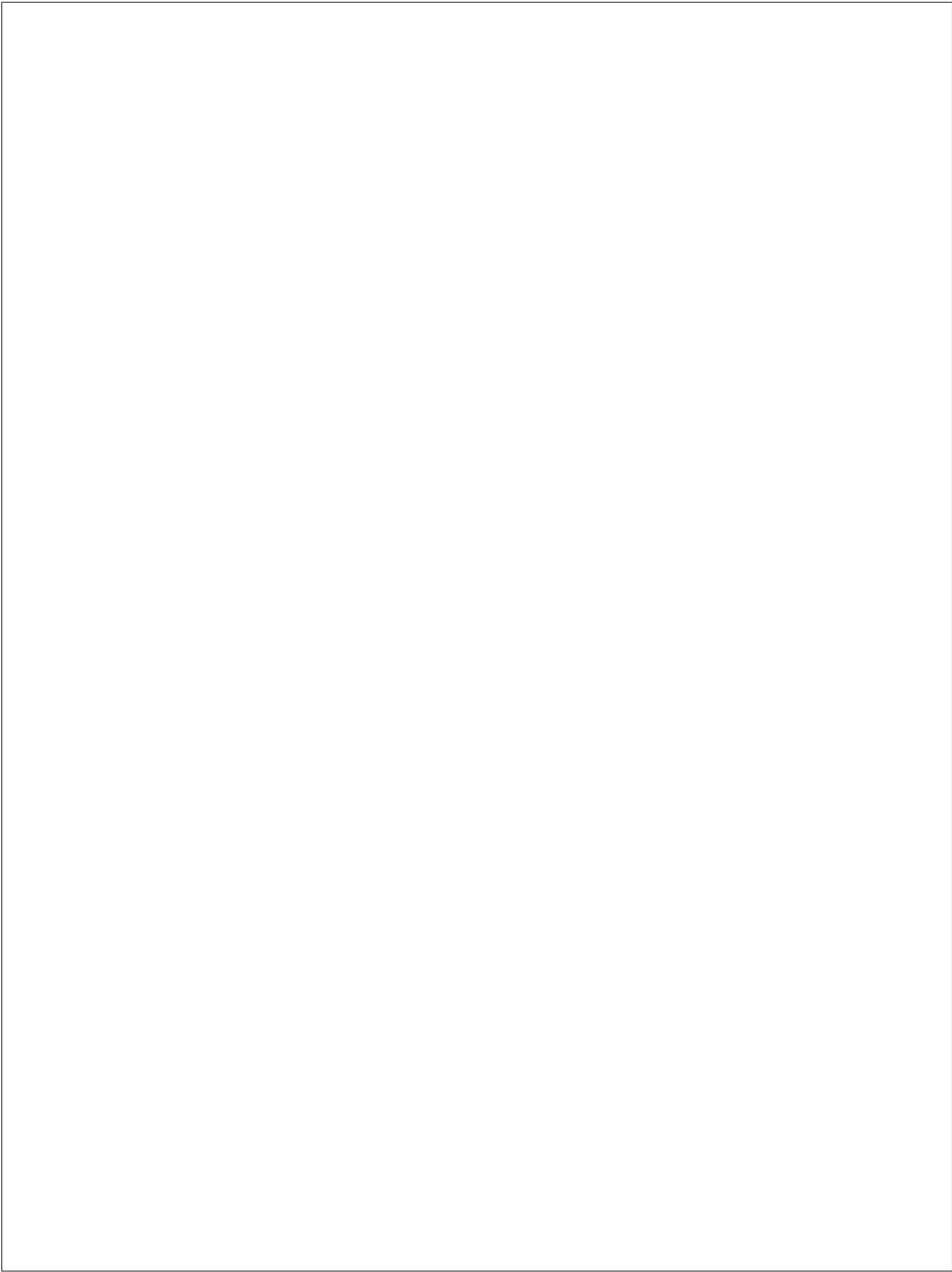
$$\pi'(j) = \begin{cases} \pi^*(i) & j = i'' \\ \pi^*(i'') & j = i \\ \pi^*(j) & \text{otherwise} \end{cases}.$$

In other words, π' is obtained from π^ by swapping jobs $\pi^*(i)$ and $\pi^*(i'')$. We prove that $\bar{t}_{\pi'} < \bar{t}_{\pi^*}$, a contradiction because $\bar{t}_{\pi^*} \leq \bar{t}_\pi$ for every permutation π . Thus, there cannot be any index i such that job $\pi^*(i)$ is not the i th shortest job. As we argued above, this implies that the permutation π our algorithm produces minimizes \bar{t}_π .*

To prove that $\bar{t}_{\pi'} < \bar{t}_{\pi^}$, observe that, for $j < i$ or $j \geq i''$, the sets $\{\pi^*(1), \pi^*(2), \dots, \pi^*(j)\}$ and $\{\pi'(1), \pi'(2), \dots, \pi'(j)\}$ are the same. Thus, $t_{\pi^*}(\pi^*(j)) = \sum_{h=1}^j d_{\pi^*(h)} = \sum_{h=1}^j d_{\pi'(h)} = t_{\pi'}(\pi'(j))$ for each such value of j .*

For $i \leq j < i''$, we have $\{\pi'(1), \pi'(2), \dots, \pi'(j)\} = \{\pi^(1), \pi^*(2), \dots, \pi^*(j)\} \setminus \{\pi^*(i)\} \cup \{\pi^*(i'')\}$. Thus, $t_{\pi'}(\pi'(j)) = \sum_{h=1}^j d_{\pi'(h)} = \sum_{h=1}^j d_{\pi^*(h)} - d_{\pi^*(i)} + d_{\pi^*(i'')} < \sum_{h=1}^j d_{\pi^*(h)} = t_{\pi^*}(\pi^*(j))$ because job $\pi^*(i)$ is longer than job $\pi^*(i'')$. This shows that $\sum_{j=1}^n t_{\pi'}(j) = \sum_{j=1}^n t_{\pi^*}(\pi'(j)) < \sum_{j=1}^n t_{\pi^*}(\pi^*(j)) = \sum_{j=1}^n t_{\pi^*}(j)$ and, hence, $\bar{t}_{\pi'} = \frac{1}{n} \sum_{j=1}^n t_{\pi'}(j) < \frac{1}{n} \sum_{j=1}^n t_{\pi^*}(j) = \bar{t}_{\pi^*}$, as claimed.*

Extra space for Question 3.1



Question 3.2 (Dynamic programming)

10 marks

Recall the Subset Sum problem from class: Given a set S of n positive numbers and a target number t , the problem asks us to decide whether there exists a subset $S' \subseteq S$ such that $\sum_{x \in S'} x = t$. We also proved in class that this problem is NP-hard, but the proof required that the numbers were exponentially large (exponential numbers can be represented in a linear number of bits). This is not just a caveat of the proof, as you will show here: Develop an algorithm that solves the Subset Sum problem in $O(nt)$ time (which is polynomial in n if t is polynomial in n). Argue briefly that your algorithm is correct and that it achieves the desired running time.

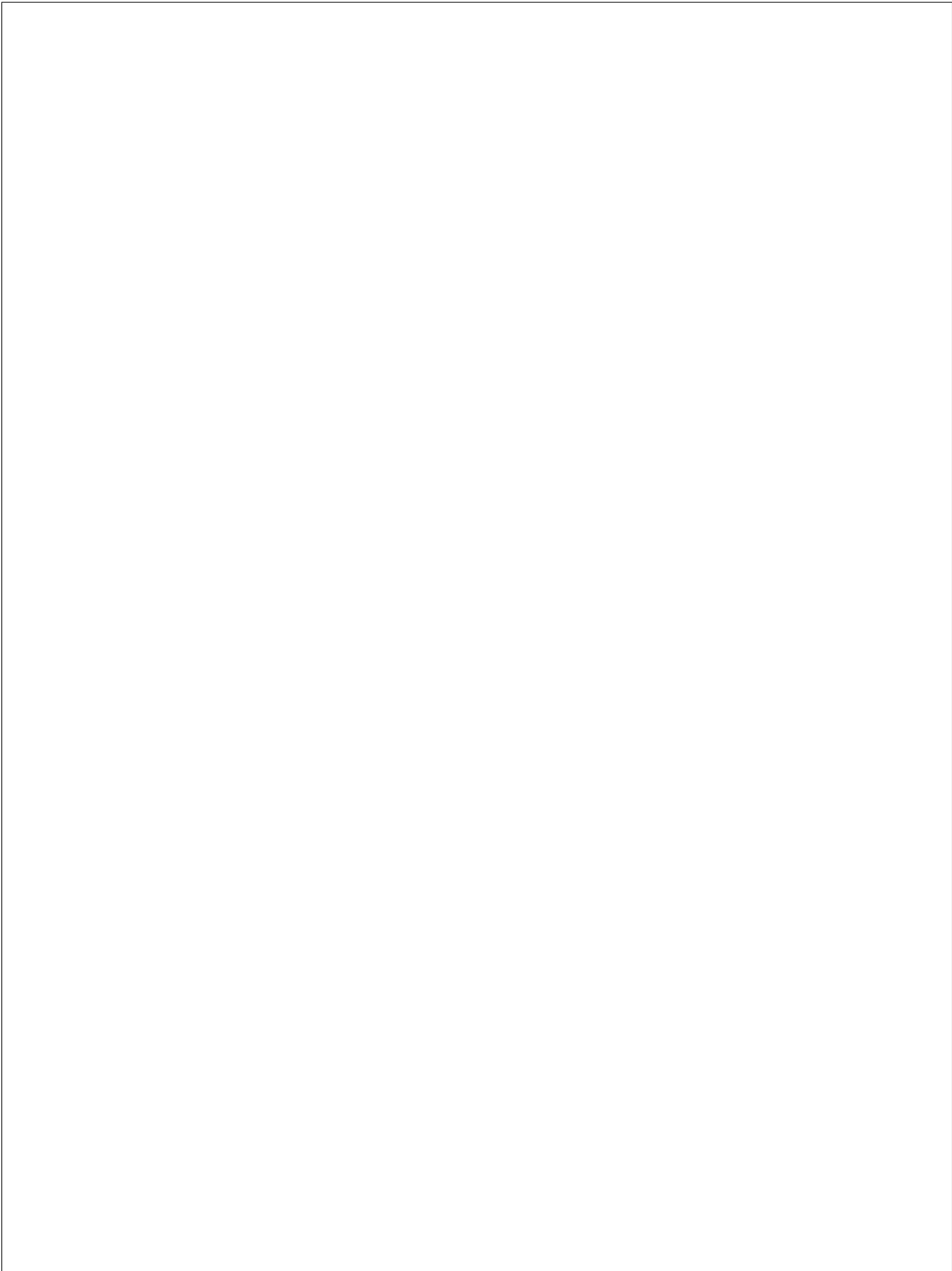
First we preprocess S in linear time by removing all elements that are greater than t . Since all numbers in S are positive, no element $x > t$ can be part of a subset $S' \subseteq S$ such that $\sum_{x \in S'} x = t$. From here on, we can therefore assume that all elements in S are no greater than t .

Let us denote the elements in S by x_1, x_2, \dots, x_n and let us define $S_i = \{x_1, x_2, \dots, x_i\}$. We build a Boolean $(n+1) \times 2t$ table T with index range $[0..n] \times [-t+1..t]$ such that $T[i, v] = \text{true}$ if and only if there exists a subset $S' \subseteq S_i$ such that $\sum_{x \in S'} x = v$. After constructing this table, we can then simply report $S[n, t]$ as the answer to the question whether there exists a subset $S' \subseteq S = S_n$ such that $\sum_{x \in S'} x = t$.

To build the table, we need a recurrence: First observe that $S[i, 0] = \text{true}$ for all $0 \leq i \leq n$ because $\emptyset \subseteq S_i$ and $\sum_{x \in \emptyset} x = 0$. Similarly, $S[i, v] = \text{false}$ for all $0 \leq i \leq n$ and $-t+1 \leq v < 0$ because S_i contains only positive elements, so there is no subset $S' \subseteq S_i$ such that $\sum_{x \in S'} x = v < 0$. Finally, $S[0, v] = \text{false}$ for all $0 < v \leq t$ because $S_0 = \emptyset$, so there is no subset of S_0 whose elements sum to a positive value. Our algorithm initializes these $(n+2) \times t$ values in $O(nt)$ time because each can be computed in constant time.

For $i > 0$ and $v > 0$, we observe that, if there exists a subset $S' \subseteq S_i$ such that $\sum_{x \in S'} x = v$, then either $S' \subseteq S_{i-1}$ or $x_i \in S'$ and $\sum_{x \in S''} x = v - x_i$, where $S'' = S' \setminus \{x_i\}$. Thus, $T[i, v] = T[i-1, v] \vee T[i-1, v-x_i]$. (Note that these are valid table indices because $v > 0$ and $x_i \leq t$, so $v-x_i \geq -t+1$. That's why I chose to extend the index range of T to include negative values of v .) If we fill T in column by column, then all $T[i-1, v']$ values are computed before $T[i, v]$, so $T[i, v]$ can be computed in constant time by looking up $T[i-1, v]$ and $T[i-1, v-x_i]$ and taking the logical or. Thus, each of the $n \times t$ values where $i > 0$ and $v > 0$ can also be computed in $O(nt)$ time. In summary, we can fill in the entire table T in constant time per entry, $O(nt)$ time in total. Reporting the answer whether $\sum_{x \in S'} x = t$ for some subset $S' \subseteq S$ then takes a single lookup of $T[n, t]$ and thus takes constant time.

Extra space for Question 3.2



Question 3.3 (Data structures)

10 marks

Describe a data structure that stores a set S of n numbers and supports $\text{MINPAIR}(S)$ queries. A $\text{MINPAIR}(S)$ query reports a pair $(x, y) \in \binom{S}{2}$ such that $|x - y| = \min_{(u,v) \in \binom{S}{2}} |u - v|$, where $\binom{S}{2} = \{(x, y) \in S \times S \mid x \neq y\}$. In other words, a $\text{MINPAIR}(S)$ query reports the two elements in S with the smallest difference between them. If $|S| < 2$, $\text{MINPAIR}(S)$ reports nil . The cost of a MINPAIR query should be in $O(1)$. The data structure should also support insertions of new elements into S and deletions of elements from S . These update operations should take $O(\lg n)$ time. Argue briefly that your implementations of INSERT , DELETE , and MINPAIR queries are correct and take $O(\lg n)$, $O(\lg n)$, and $O(1)$ time, respectively.

The data structure is an (a, b) -tree T augmented with some extra information stored at its nodes. For every node v , let S_v denote the subset of S stored in the descendant leaves of v . Every node v of T already stores the minimum element $m_v = \min S_v$ as its search key. Now we also make it store $M_v = \max S_v$ as well as a pair $(x_v, y_v) = \text{MINPAIR}(S_v)$.

Let r be the root of T . Since $S_r = S$ and r stores the answer (x_r, y_r) to a $\text{MINPAIR}(S_r)$ query, answering a $\text{MINPAIR}(S)$ query takes constant time, just by reporting the pair (x_r, y_r) . This is obviously correct.

To insert a new element into S , we perform a standard (a, b) -tree insertion and ensure that the information associated with each node v (m_v , M_v , and (x_v, y_v)) in the tree remains valid. Since an insertion adds a single leaf to T and then performs up to $\lg n$ node splits, we have to show that leaf additions can be supported in $O(\lg n)$ time and node splits can be supported in $O(1)$ time.

Similarly, to delete an element from S , we perform a standard (a, b) -tree deletion and ensure that the information associated with each node remains valid. Since a deletion removes a single leaf and then performs up to $\lg n$ node fusions and possibly one node split, we have to show that leaf removal can be supported in $O(\lg n)$ time and node splits and node fusions can be supported in $O(1)$ time.

Bottom-up propagation. *To maintain the information associated with the nodes of T as part of leaf additions, leaf removals, node splits, and node fusions, we repeatedly apply the same primitive: computing the information associated with a node v from the information associated with its children. Let w_1, w_2, \dots, w_k be the list of children of v ordered from left to right. If v is a leaf, then $m_v = M_v$ is the element stored at v ; $(x_v, y_v) = \text{nil}$ because $|S_v| = 1$.*

If v is not a leaf, then observe that $m_v = m_{w_1}$ and $M_v = m_{w_k}$, that is, they can be computed in constant time from the information stored at w_1 and w_k . To determine (x_v, y_v) , observe that x_v and y_v must be neighbours in the sorted sequence of elements in S_v ; otherwise, their difference couldn't be minimal. Thus, either $(x_v, y_v) \in S_{w_i}$ for some child w_i of v or $x_v = M_{w_i}$ and $y_v = m_{w_{i+1}}$ for some index $1 \leq i < k$. (x_v, y_v) thus is the pair among $(x_{w_1}, y_{w_1}), (x_{w_2}, y_{w_2}), \dots, (x_{w_k}, y_{w_k}), (M_{w_1}, m_{w_2}), (M_{w_2}, m_{w_3}), \dots, (M_{w_{k-1}}, m_{w_k})$ with the least difference. There are at most $2b - 1 \in O(1)$ such candidate pairs and each can be accessed in constant time. Thus, computing (x_v, y_v) also takes constant time, given the information associated with v 's children.

Extra space for Question 3.3

Leaf addition. After creating a new leaf ℓ storing some element x , observe that the set S_v has changed only if v is an ancestor of ℓ . We process these ancestors bottom-up from the ℓ to the root of T and, for each, update m_v , M_v , and (x_v, y_v) in constant time as described above. Since there are $O(\lg n)$ ancestors of ℓ , this takes $O(\lg n)$ time.

Leaf removal. After removing a leaf ℓ , S_v once again changes only if v is an ancestor of ℓ . Thus, we update m_v , M_v , (x_v, y_v) for each such ancestor as after a leaf addition. This takes $O(\lg n)$ time.

Node split. When splitting a node v into two nodes v_1 and v_2 , note that $S_{v'}$ does not change for any node $v' \notin \{v, v_1, v_2\}$. Thus, $m_{v'}$, $M_{v'}$, and $(x_{v'}, y_{v'})$ do not change. For v_1 and v_2 , we can compute m_{v_1} , M_{v_1} , (x_{v_1}, y_{v_1}) , m_{v_2} , M_{v_2} , and (x_{v_2}, y_{v_2}) from the information stored at v_1 's and v_2 's children. Since this takes constant time per node, the cost of a node split remains constant.

Node fusion. When fusing two nodes v_1 and v_2 into a single node v , $S_{v'}$ once again does not change for any node $v' \notin \{v, v_1, v_2\}$. Thus, $m_{v'}$, $M_{v'}$, and $(x_{v'}, y_{v'})$ do not change. For v , we can compute m_v , M_v , and (x_v, y_v) from the information stored at v 's children. Since this takes constant time, the cost of a node fusion remains constant.