

Banner number:

Name:

Midterm Exam

CSCI 3110: Design and Analysis of Algorithms

June 26, 2018

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"/>	
Σ		Σ		Σ		

Instructions:

- The questions are divided into three groups: Group 1 (35%), Group 2 (43%), and Group 3 (22%). You have to answer **all questions in Groups 1 and 2** and **exactly one question 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 zero or two 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 9 pages, including this title page. Notify me immediately if your copy has fewer than 9 pages.**

Question 1.1 (Worst-case running time)**4 marks**

Define what the worst-case running time of an algorithm is.

The worst-case running time of a (deterministic) algorithm is a function T of the input size n such that $T(n)$ is the maximum running time over all possible inputs of size n . (If the algorithm is not deterministic, the running time may also depend on factors independent of the input, something we ignore here.)

Question 1.2 (Asymptotic growth of functions)**12 marks**

- (a) Formally state the condition that two functions $f(n)$ and $g(n)$ have to satisfy for $f(n)$ to belong to the set $\Theta(g(n))$. (Do not use that $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.)

There must exist three constants $0 < c_1 \leq c_2$ and $n_0 \geq 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

- (b) Formally state the condition that two functions $f(n)$ and $g(n)$ have to satisfy for $f(n)$ to belong to the set $o(g(n))$.

For every $c > 0$, there must exist a constant $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- (c) If a function $f(n)$ is not in $\Omega(g(n))$, does that mean that $f(n) \in o(g(n))$? If so, prove it. If not, provide two functions $f(n)$ and $g(n)$ such that $f(n)$ is neither in $\Omega(g(n))$ nor in $o(g(n))$ and prove that $f(n) \notin \Omega(g(n))$ and $f(n) \notin o(g(n))$.

Consider the functions

$$f(n) = \begin{cases} n & n \text{ is even} \\ 1 & n \text{ is odd} \end{cases}$$

and $g(n) = n$.

Then $f(n) \notin o(g(n))$ because for $c = 1/2$, there exists no n_0 such that $f(n) \leq n/2$ for all $n \geq n_0$: for all even n , we have $f(n) = g(n)$.

Similarly, $f(n) \notin \Omega(g(n))$ because for any $c > 0$, we have $f(n) = 1 < cn = c \cdot g(n)$ for all odd $n \geq \frac{1}{c} + 1$. Thus, there exists no n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Question 2.1 (Asymptotic growth of functions)

10 marks

- (a) Prove that the function $f(n) = 3n \lg n - 2n + 11$ is in $O(n \lg n)$ by explicitly providing constants n_0 and c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

$$\begin{array}{r} f(n) = 3n \lg n - 2n + 11 \quad \forall n \geq 1 \\ 0 \leq \quad \quad \quad 2n \quad \quad \quad \forall n \geq 0 \\ \hline f(n) \leq 3n \lg n + 11 \quad \forall n \geq 1 \\ 0 \leq 5.5n \lg n - 11 \quad \forall n \geq 2 \\ \hline f(n) \leq 8.5n \lg n \quad \forall n \geq 2. \end{array}$$

- (b) Use limits to prove that $n \lg n \in o(n^2)$.

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n \lg n}{n^2} &= \lim_{n \rightarrow \infty} \frac{\lg n}{n} \\ &= \lim_{n \rightarrow \infty} \frac{\frac{1}{n \ln 2}}{1} \quad (\text{by l'Hôpital's rule}) \\ &= \lim_{n \rightarrow \infty} \frac{1}{n \ln 2} \\ &= 0. \end{aligned}$$

Question 2.2 (Amortized analysis)

10 marks

Consider the problem of implementing a stack. The only supported operations are PUSH and POP. If we never store more than m elements on the stack, then we can implement the stack using an array A of size m along with a variable n that equals the number of elements currently stored on the stack. Initially, $n = 0$. A PUSH operation stores the pushed element in the array slot $A[n]$ and increments n . A POP operation decrements n and returns the element $A[n]$. Both operations take $O(1)$ time.

If we do not know the number of elements we may store on the stack, we need to be able to resize the array to accommodate more elements if necessary, and we want to be able to shrink the array when n is significantly less than m , so as not to waste space. To achieve this, we initially allocate an array of size 4 and set its capacity to $m = 4$ and the number of elements it contains to $n = 0$. A PUSH operation checks whether $n < m$. If so, it behaves exactly like the PUSH operation described above. Otherwise, it first doubles m and allocates a new array of size m . Then it copies the n elements in A to the new array, deallocates A , and lets the new array play the role of A . It then pushes the new element as in the case when $n < m$. A POP operation decreases n by one and stores $A[n]$ in a temporary variable x . Next it checks whether $m \leq \max(4, 4n)$. If $m \leq \max(4, 4n)$, it returns x . If $m > \max(4, 4n)$, it halves m and allocates a new array of size m . Then it copies the n elements in A to the new array, deallocates A , and lets the new array play the role of A . After doing this, it returns the element stored in x .

The actual cost of a PUSH or POP operation is constant if m does not change. If m changes, then the operation spends $\Theta(n)$ time to allocate a new array and copy the $\Theta(n)$ elements from the old array to the new array. Use a potential function to prove that the amortized cost per PUSH or POP operation is still constant.

Hint: Note that each time we resize A , it ends up being exactly half full ($m = 2n$). The chance of an expensive PUSH or POP operation increases the closer A is to being full ($m = n$) or a quarter full ($m = 4n$). This should be all you need to define an appropriate potential function.

The potential function is $\Phi = 0$ if $n < 2$ and $\Phi = c|n - m/2|$ if $n \geq 2$, where $c > 0$ is an appropriate constant to be defined later. Then $\Phi = 0$ initially and $\Phi \geq 0$ at all times, that is, this is a valid potential function.

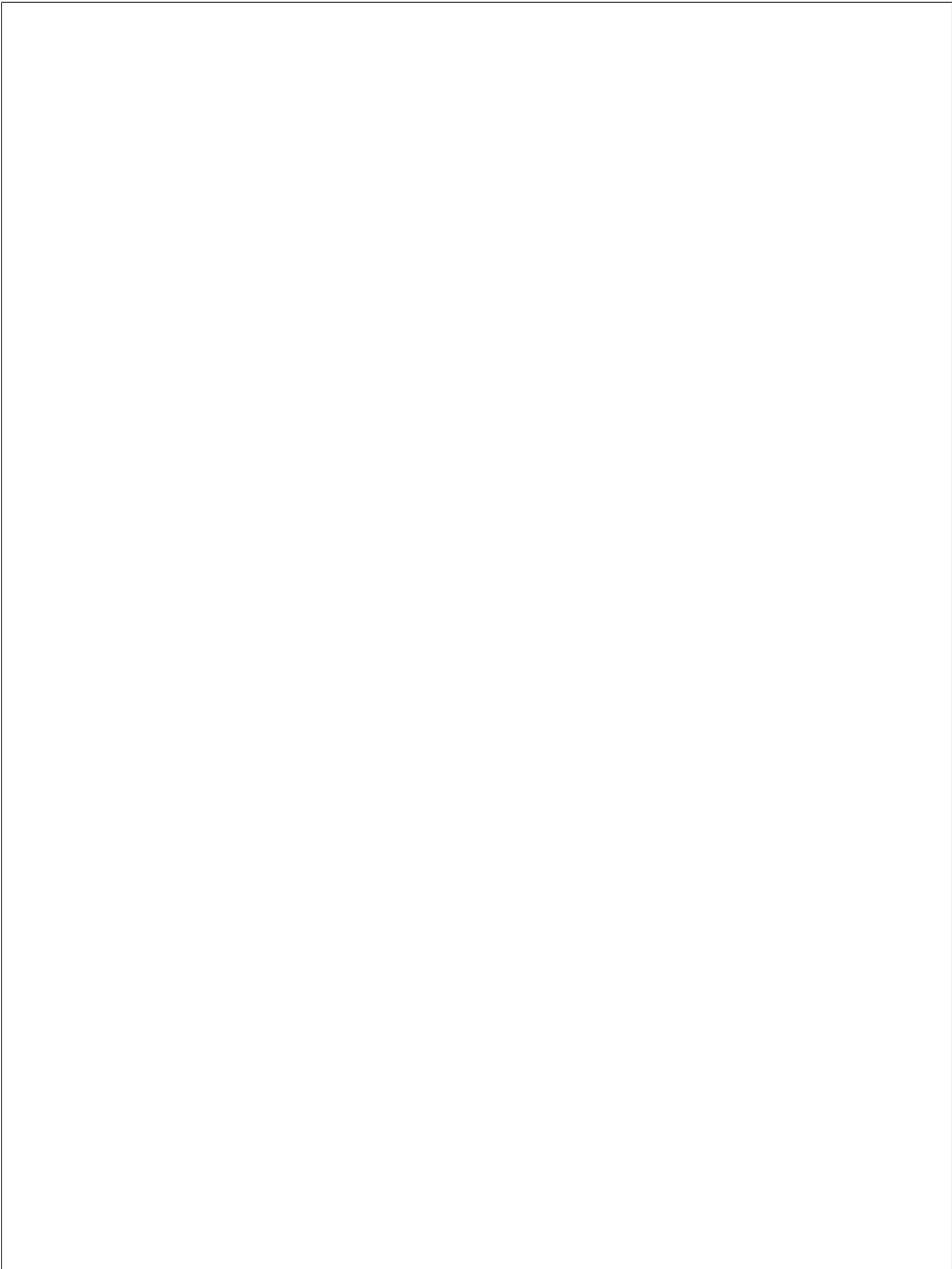
Using this potential function, we can prove that the amortized cost of a PUSH or POP operation excluding the cost of resizing the array is in $O(1)$ and the amortized cost of resizing the array is zero. Thus, the amortized cost per PUSH or POP operation is in $O(1)$.

PUSH (excluding resizing): *This operation has an actual cost in $O(1)$, does not alter m , and increases n by one. Thus, Φ increases by at most c and its amortized cost is $O(1) + \Delta\Phi \leq O(1) + c = O(1)$.*

POP (excluding resizing): *This operation has an actual cost in $O(1)$, does not alter m , and decreases n by one. Thus, Φ increases by at most c and its amortized cost is $O(1) + \Delta\Phi \leq O(1) + c = O(1)$.*

Resizing: *The array can be resized only if $n \geq 2$, so $\Phi = |n - m/2|$. We resize A either because $n = m$ or $n = m/4$. In the former case, $\Phi = c|m - m/2| = cm/2 = cn/2$. In the latter case, $\Phi = c|m/4 - m/2| = cm/4 = cn$. After the resize operation, $m = 2n$ (if $n = m$, we double m , so $m = 2n$; if $n = m/4$, we halve m , so $m = 2n$). Thus, $\Phi = 0$ after the resize operation and $\Delta\Phi \leq -cn/2$. The actual cost of the resize operation is bounded by an for some constant a . Thus, its amortized cost is at most $an + \Delta\Phi \leq an - cn/2$, which is non-positive as long as we choose the constant c in the potential function to be at least $2a$.*

Extra space for Question 2.2



Extra space for Question 3.1

Analysis: The construction of H obviously takes linear time in the number of vertices and edges added to H . H has m vertices, but we observed already that $m \leq 4n$. Similarly, every vertex of H has at most 6 neighbours, up to 3 per endpoint of the corresponding edge of G . Thus, H has at most $6m \leq 24n$ edges. This shows that H can be constructed in $O(n)$ time. Running Dijkstra's algorithm on H 4 times to compute shortest paths takes $O(n \lg n)$ time. (By exploiting that all edge weights are 0 or 1, we can in fact compute shortest paths in H in $O(n)$ time, but this takes more effort.)

Correctness: Consider any path Q between two nodes e_1 and e_2 in H . Since there is an edge between two nodes in H if and only if the two corresponding edges in G share an endpoint, Q corresponds to a path P in G and the edge weights in H are chosen so that the length of Q is exactly the number of turns P makes.

Conversely, any path P in G that starts with e_1 and e_2 corresponds to a path Q from e_1 to e_2 in H whose length is the number of turns P makes. Thus, $\text{dist}_H(e_1, e_2)$ is exactly the minimum number of turns in any path in G that starts with e_1 and ends with e_2 . By considering all paths starting with an edge e_1 incident to A and ending with an edge e_2 incident to B , we find the minimum-turn path from A to B in G .

Question 3.2 (Greedy algorithms)

10 marks

You are given n jobs J_1, \dots, J_n . Each job J_i takes one unit of time to complete and has a deadline d_i before which it must be completed. Develop an algorithm that takes $O(n \lg n)$ time to compute an ordering in which to complete the jobs so that the number of jobs completed before their deadlines is maximized. Argue briefly that its running time is indeed in $O(n \lg n)$ and prove that there is no ordering of the jobs that ensures that more jobs are completed before their deadlines than using the ordering output by your algorithm.

Example: Given 5 jobs J_1, \dots, J_5 with deadlines 3, 2, 2, 1, 5, you can complete 4 of them before their deadlines using the following two orderings (and a few others):

Job	J_4	J_2	J_1	J_3	J_5
Completion time	1	2	3	4	5
Deadline	1	2	3	2	5

Job	J_4	J_3	J_1	J_5	J_2
Completion time	1	2	3	4	5
Deadline	1	2	3	5	2

No order allows you to complete more than 4 jobs before their deadlines, so these orders are optimal.

Hint: Your algorithm should make a very natural greedy choice for each job. If you fail to complete job J_i before its deadline d_i , then you should have more than d_i jobs with deadlines before d_i . This should be the key to your correctness proof.

We can assume that the deadlines are integers because a job can be completed before its deadline d_i if and only if it can be completed before the earlier deadline $\lfloor d_i \rfloor$.

The algorithm: We sort the jobs by their deadlines. Let J_1, \dots, J_n be the resulting ordering. Now we inspect the jobs in order and add them to a schedule S . We also keep a list L of late jobs that we cannot schedule before their deadlines. Initially, both S and L are empty. For the i th job J_i , if S already contains d_i jobs, then the earliest completion time for job J_i would be $d_i + 1$. Thus, we append J_i to L . If S contains less than d_i jobs, we append J_i to S . Once we have inspected all jobs, we append all jobs in L to S .

Analysis: The algorithm sorts the jobs in $O(n \lg n)$ time and then inspects each job spending constant time for each job to decide whether to add it to S or L . Thus, the construction of S and L takes $O(n)$ time. Concatenating S and L at the end takes no more than $O(n)$ time even if we represent S and L as an array (which is what I would do in an actual implementation). Thus, the running time of the algorithm is dominated by the initial sorting step and thus takes $O(n \lg n)$ time. (By exploiting that the deadlines are integers and that jobs with deadlines greater than n are completed before their deadlines no matter the order in which we schedule jobs, we can accomplish the sorting step in linear time using Counting Sort. This reduces the running time to $O(n)$.)

Correctness: We prove by induction on i that (1) all jobs we add to S are completed before their deadlines and (2) if we schedule n_i of the jobs in $\{J_1, \dots, J_i\}$, before their deadlines, then it is impossible to schedule more than n_i jobs in this set before their deadlines. For $i = n$, this shows that it is impossible to schedule more jobs before their deadlines than using the final ordering produced by our algorithm.

For the base case, $i = 0$, $S = \emptyset$, so (1) holds trivially. It is clearly not possible to schedule more than 0 jobs in the empty set before their deadlines, so (2) also holds.

For the inductive step, $i > 0$, we distinguish two cases:

Extra space for Question 3.2

If we append J_i to S , then S contains at most d_i jobs after we append J_i to S , so J_i is completed before its deadline d_i and (1) continues to hold. We have $n_i = n_{i-1} + 1$ in this case. Any ordering that completes more than n_i of the jobs in $\{J_1, \dots, J_i\}$ before their deadlines would have to complete more than $n_i - 1 = n_{i-1}$ of the jobs $\{J_1, \dots, J_{i-1}\}$ before their deadlines. By the inductive hypothesis, this is impossible. Thus, it is impossible to complete more than n_i of the jobs in $\{J_1, \dots, J_i\}$ before their deadlines, that is, (2) holds.

If we do not append J_i to S , then (1) trivially continues to hold. Since we do not append J_i to S , S already contains d_i jobs from $\{J_1, \dots, J_{i-1}\}$, that is, $n_i = n_{i-1} = d_i$. All jobs in $\{J_1, \dots, J_i\}$ have deadlines no later than d_i because we inspect jobs by increasing deadlines. Thus, even if all jobs in $\{J_1, \dots, J_i\}$ had deadline d_i , it would not be possible to complete more than d_i of them before their deadline. Since $n_i = d_i$, this shows that it is impossible to complete more than n_i of the jobs in $\{J_1, \dots, J_i\}$ before their deadlines; (2) holds.