

National University of Singapore
School of Computing

CS3230 - Design and Analysis of Algorithms
Final Assessment
(Semester 2 AY2022/23)

Time Allowed: 2 hours

INSTRUCTIONS TO CANDIDATES:

1. Do **NOT** open this assessment paper until you are told to do so.
2. This assessment paper contains **TWO** (2) sections.
It comprises **FOURTEEN** (14) printed pages, including this page.
3. This is an **Open Book** Assessment.
4. For Section A, use the OCR form provided (**use 2B pencil**).
You will still need to hand over the entire paper as the MCQ section will not be archived.
5. For Section B, answer **ALL** questions within the **boxed space**.
If you leave the boxed space blank, you will get automatic 1 mark.
However, if you write at least a single character and it is totally wrong, you will get 0 mark.
You can use either pen or pencil. Just make sure that you write **legibly!**
6. Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
Read all the questions first! Some questions might be easier than they appear.
7. You can assume that all **logarithms are in base 2**.
8. Please write your Student Number only. Do **not** write your name.

A	0								
---	---	--	--	--	--	--	--	--	--

This portion is for examiner's use only

Section	Maximum Marks	Your Marks
A	39	
B	61	
Total	100	

A Multiple Choice Questions ($13 \times 3 = 39$ marks)

Select the **best unique** answer for each question. Each correct answer worth 3 marks.

The default answers (as the MCQ section is not supposed to be archived to open up possibilities of reuse in the future): ecbeb daa[c|e]d ecb. Note that there is a silly one character bug in MCQ Q9 that makes the supposedly correct option c becomes invalid (and thus makes option e becomes the answer). We accept both.

THE REST OF PAGE 2 IS REDACTED.

THE ENTIRE PAGE 3 IS REDACTED.

THE ENTIRE PAGE 4 IS REDACTED.

THE ENTIRE PAGE 5 IS REDACTED.

B Essay Questions (61 marks)

B.1 Strategy Making as Antique Collector (30 marks)

You are a collector of antiques. In recent news, there will be a prestigious once-in-a-lifetime bazaar selling rare antiques you do not want to miss.

In that bazaar, there will be n types of antiques being sold. The i -th type of antique has an initial price of p_i dollars and has a quality grade of q_i stars. While you can assume that each type of antique has an “unlimited” stock of antiques, you actually cannot buy many of them. This is because, due to their rarity, if you buy k many antiques of type i , the organizer will charge you $k^2 \cdot p_i$ dollars (so it gets more expensive the more you buy). Also, it is not possible to buy an item fractionally.

Let’s say your budget is only M dollars and your happiness level is defined as the total quality grade stars you can get by buying the antiques. Your task is to find the maximum possible happiness level! (Assume, all p_i, q_i and M are positive integers.) We refer to this problem as the *Rare Antique Collection* problem.

B.1.1 Test your understanding [4 Marks]

Let’s have an example. Suppose that the bazaar only sells $n = 2$ types of antiques. The first type has an initial price of $p_1 = 30$ dollars and a quality grade of $q_1 = 2$ stars. The second type has an initial price of $p_2 = 100$ dollars and a quality grade of $q_2 = 3$ stars.

If your budget is $M = 125$ dollars, then your maximum possible happiness level will be 4 stars. This is achieved by buying 2 antiques for the first type and 0 antiques for the second type, as the total price will be $2^2 \cdot 30 + 0^2 \cdot 100 = 120$ dollars (within the budget) and the happiness level is $2 \cdot 2 + 0 \cdot 3 = 4$ stars.

If your budget is $M = 500$ dollars instead, then what will be your maximum possible happiness level in this scenario (2 marks)? Write down how many antiques you should buy for the first and the second types to achieve that happiness level (also 2 marks)!

Maximum happiness level is **9** stars.

We should buy **3** antiques of the first type and **1** antique of the second type.

B.1.2 How happy you can be? (6 marks)

Suppose Q be the maximum stars any item can give, i.e., $Q = \max_i q_i$. Show that the maximum possible happiness level you can attain is at most $Q\sqrt{nM}$. (You may use the following inequality without proving it: For any two sequences of numbers (a_1, a_2, \dots, a_n) and (b_1, b_2, \dots, b_n) , $(\sum_{i=1}^n a_i b_i)^2 \leq (\sum_{i=1}^n a_i^2) \cdot (\sum_{i=1}^n b_i^2)$.)

You get a partial marks of 3 if you could only show an upper bound of $Qn\sqrt{M}$ (instead of $Q\sqrt{nM}$).

For the partial marks of 3, note that each type can only be bought at most \sqrt{M} times, because the next purchase will be charged at $(\sqrt{M} + 1)^2 \cdot p_i$ dollars, and this already exceeds M as $p_i \geq 1$.

Therefore, if we buy \sqrt{M} times for each type of item, the happiness level will be at most $\sqrt{M} \cdot n \cdot Q$.

To get a full mark, let k_i be the number of antiques we buy for the i -th type. We want to maximize $\sum_{i=1}^n k_i \cdot q_i$ with respect to $\sum_{i=1}^n k_i^2 \cdot p_i \leq M$.

Note that as $p_i \geq 1$, we get $\sum_{i=1}^n k_i^2 \leq M$. Using this fact and the inequality given:

$$\sum_{i=1}^n k_i \cdot q_i = \sqrt{\left(\sum_{i=1}^n k_i \cdot q_i\right)^2} \leq \sqrt{\sum_{i=1}^n k_i^2 \cdot \sum_{i=1}^n q_i^2} \leq \sqrt{M \cdot \sum_{i=1}^n Q^2} \leq \sqrt{M \cdot n \cdot Q^2} \leq Q\sqrt{nM}$$

B.1.3 Design an algorithm for the Rare Antique Collection problem (10 Marks)

Design an algorithm to compute the value of the maximum possible happiness level. To get the full marks, your algorithm should run in $O(Qn^{3/2}M)$ time. You may assume the statement given in Question B.1.2 even if you can't prove it. (**Hints:** Try to find a similarity with the Knapsack problem discussed in the lecture, and try to come up with a Dynamic Programming algorithm.)

Provide a clear description of your algorithm, including the optimal substructure, and a detailed running time analysis.

You get a partial marks of 7 if your algorithm runs in $O(nM^{3/2})$ time, and a partial marks of 3 if your algorithm runs in $O(M^{n/2})$ time.

For the partial marks of 3, note that from Question B.1.2, each type can only be bought at most \sqrt{M} times. Therefore, we could bruteforce how many antiques do we buy for each type. Efficiently, this can be done in $O((\sqrt{M})^n) = O(M^{n/2})$.

For the partial marks of 7, we could modify the Knapsack algorithm from the lecture. In the original Knapsack algorithm, we either take or leave the item. For this problem, we try every possibility of the number of antiques we should buy instead. Previously, we know that this is upperbounded by \sqrt{M} .

Let $m[i, j]$ be the **maximum happiness** that can be obtained using antiques of types $1, 2, \dots, i$ with a budget of j . We can define that:

- $m[i, j] = 0$, if $i = 0$ or $j = 0$;
- $m[i, j] = \text{MAX}_k (k \cdot q_i + m[i - 1, j - k^2 \cdot p_i])$, for all $k \geq 0$ such that $k^2 \cdot p_i \leq j$, otherwise.

Using above m , the answer will be the value of $m[n, M]$. Visibly, there are $O(nM)$ different subproblems, which can be computed once using dynamic programming. Each of them is computed in $O(\sqrt{M})$, thus the total runtime will be $O(nM^{3/2})$.

To get a full mark, we actually need to “flip” the definition of above m . :)

Let $w[i, j]$ be the **minimum budget** that we need to obtain a happiness level of exactly j , using antiques of types $1, 2, \dots, i$. We can define that:

- $w[i, j] = 0$, if $j = 0$;
- $w[i, j] = \infty$, if $i = 0$ and $j \neq 0$;
- $w[i, j] = \text{MIN}_k (k^2 \cdot p_i + w[i - 1, j - k \cdot q_i])$, for all $k \geq 0$ such that $k \cdot q_i \leq j$, otherwise.

Using above w , the answer will be the maximum possible r where the value of $w[n, r] \leq M$. From Question B.1.2, we know that the maximum possible happiness level is at most $Q\sqrt{nM}$. Therefore, the number of different subproblems is $O(n \cdot (Q\sqrt{nM})) = O(Qn^{3/2}\sqrt{M})$.

However, note that each subproblem may be computed in $O(j/q_i) = O(Q\sqrt{nM})$ as $k \cdot q_i \leq j$ (this is crucially different from m where the k is squared). We need to optimize by limiting the k by the observation on Question B.1.2, so we have the following redefinition:

- $w[i, j] = \text{MIN}_k (k^2 \cdot p_i + w[i - 1, j - k \cdot q_i])$, for all $0 \leq k \leq \sqrt{M}$ such that $k \cdot q_i \leq j$, otherwise.

Using the optimization, each subproblem can be computed in at most $O(\sqrt{M})$, thus the total runtime will be $O((Qn^{3/2}\sqrt{M}) \cdot \sqrt{M} + Q\sqrt{nM}) = O(Qn^{3/2}M)$. Note that the $+Q\sqrt{nM}$ is from finding the answer r .

B.1.4 The rare antique collection problem is NP-complete! (10 Marks)

Prove that the decision version of the Rare Antique Collection problem is NP-complete. (You may show a reduction from any of the NP-complete problems introduced in the lectures/tutorials/assignments/practice sets, including Circuit Satisfiability, CNF-SAT, 3-SAT, Vertex Cover, Independent Set, Max-Clique, Hamiltonian Cycle, Traveling Sales Person, Knapsack, Partition, Subset-sum Problem.)

[**Hints:** You have perhaps already observed some sort of similarity between the Knapsack and the Rare Antique Collection (RAC) problem. So try a reduction from the Knapsack problem. More specifically, for an instance of the Knapsack with n items with value, weight pairs as $(v_1, w_1), \dots, (v_n, w_n)$, and a capacity W , you may try creating an instance of the Rare Antique Collection problem using the following:

- Consider $B = n \cdot \max\{W, v_i, w_i\} + 1$.

- For each item of Knapsack, create an antique type with $p_i = B + w_i$ and $q_i = B + v_i$.
- Create n additional “dummy” antique types each with $p_i = B$ and $q_i = B$.
- Set the budget $M = nB + W$.

]

Consider the decision version of Knapsack problem with the threshold value $\geq T_v$. Correspondingly, consider the reduction from Knapsack to Rare Antique Collection (RAC) problem given in the hints, and transform the threshold value to $T_q = nB + T_v$ for RAC (this completes the reduction of an instance of the decision version of Knapsack problem to that of the RAC problem). For proving that the RAC problem is NP-Complete by considering the reduction from the Knapsack problem, we can proceed as follows:

1. The RAC problem is in NP (choice of antique types is the certificate). This verification can be done in polynomial time.
2. The reduction runs in polynomial time in input size.
3. YES-instance of Knapsack implies YES-instance of RAC.
 - We can directly use the antique types corresponding to the items chosen for the Knapsack problem (say k items), along with $n - k$ “dummy” antique types (one of each type).
 - This choice gives us exactly n antiques, where no antique type is chosen more than once.
 - This choice will satisfy the budget constraint $M = nB + W$, as well as the quality threshold $T_q = nB + T_v$ (direct implication of the corresponding constraints for the Knapsack problem).
4. Next, we prove that YES-instance of RAC implies YES-instance of Knapsack.
 - Suppose, we buy k_i antiques of type i .
 - We cannot buy $> n$ antiques, since the price will exceed $(n + 1)B > nB + W = M$. Thus, $\sum_i k_i \leq n$.
 - If we buy $< n$ antiques, the total quality is $< nB < T_q$. Thus, $\sum_i k_i \geq n - 1$.
 - Combining, we have $\sum_i k_i = n$, i.e., we must buy exactly n antiques.
 - Next, if $k_j > 1$ for any $1 \leq j \leq 2n$ (i.e., we buy more than 1 antique of any type), we must pay price $> k_j^2 B$ for the antique type j . Therefore, remaining budget is $< M - k_j^2 B = (n - k_j^2)B + W$, and we can buy a maximum of $n - k_j^2$ antiques with this budget ($W < B$).
 - In this case, total quality grade stars is $< (n - k_j^2)B + B + k_j B = (n + 1)B - (k_j^2 - k_j)B$.
 - Since $k_j > 1$, $k_j^2 - k_j > 2$ (note that k_j is a non-negative integer). Thus, total quality grade stars is $< (n - 1)B < T_q$.
 - Therefore, for any YES-instance of RAC problem, every antique type must be chosen at most once, i.e., $k_i \in \{0, 1\} \forall i$.
 - Finally, if $\sum_{i=1}^{2n} k_i p_i \leq M = nB + W$, then $\sum_{i=1}^n k_i w_i \leq W$.
 - Similarly, if $\sum_{i=1}^{2n} k_i q_i \geq T_q = nB + T_v$, then $\sum_{i=1}^n k_i v_i \geq T_v$.
 - Thus, we can consider the antique types chosen out of the first n types in the YES-instance of

RAC problem to get a YES-solution to the Knapsack problem.

Alternative Solution: Apparently, there is a **simpler** proof for the NP-completeness, which is (deliberately not given as hints) by the reduction from Subset-sum problem instead! For a set S with a target sum of T_s , we create an instance of RAC with $n = |S|$, $p_i = q_i = S_i$, and $M = T_q = T_s$. Note that it is no use to buy more than one type of antique, as the price will then be greater than the quality, while the budget is just exactly equal to the target. The rest of the proof follows directly from the classical proof of the reduction from Subset-sum to Knapsack problem, in our Tutorial 09. :)

B.2 Act Greedily in Transporting Hazardous Items (16 marks)

Suppose a pharmaceutical company has a warehouse where several chemical compounds are stored. However, due to a certain space crunch in the current location, the company has decided to relocate the warehouse to a new location slightly far from the current one. So it is required to transport all the chemicals in the warehouse to the new location, and the company has hired a truck for that purpose. Unfortunately, it is impossible to transport certain compounds simultaneously due to security reasons. We call two compounds X and Y *hazard-pair* if they can react with each other and may create an explosion. Thus while transporting chemicals, it is not allowed to put a hazard-pair in the same truck. Now, of course, to reduce the transport cost, the owner of the company wants to have as few trips (by the transport truck) as possible to relocate all the compounds, and thus wants to maximize the number of items transported in each trip (while avoiding transport of a hazard-pair). After knowing that you have learned a lot of algorithms in CS3230, suppose the owner has hired you.

Suppose there are n chemicals present. After seeing the hazard-pairs, you build a *hazard-graph* as follows: The set of vertices is the same as the set of chemicals. Between two vertices, you draw an edge if and only if the corresponding chemicals form a hazard-pair. By carefully looking into the graph, you have observed that no cycle is present in your hazard-graph. Your objective is to design an $O(n)$ time *greedy algorithm* that outputs the maximum number of chemicals so that no hazard-pair is present. Answer the following questions to achieve that.

B.2.1 What is the greedy-choice property (I)? (7 marks)

State a greedy strategy that works for the above problem. Also, provide a formal proof why your greedy strategy leads to an optimal solution to the above problem. (*No marks* will be given without the correct proof.)

Greedily select a leaf (or an isolated vertex) into our independent set and delete that leaf (and its immediate neighbor, if any) from the graph. The exchange argument proof is in Kleinberg-Tardos book.

B.2.2 What is the optimal substructure (II)? (4 marks)

State an optimal substructure property that you would like to use together with your greedy-choice (in B.2.1) to get an optimal solution. Again, provide a formal proof. (*No marks* will be given without the correct proof.)

See Algorithm Design (Kleinberg and Tardos) page 559, Point (10.5).

B.2.3 Design an algorithm (5 marks)

Combine the greedy-choice (in B.2.1) and optimal substructure (in B.2.2) to design an $O(n)$ time algorithm that always outputs an optimal solution. (Provide a pseudocode of your algorithm and a complete running time analysis.)

This task (B.2) is from Algorithm Design (Kleinberg and Tardos, Chapter 10.2 - Solving NP-Hard Problems on Trees).

Note that the pseudo-code given there is not 100% correct, i.e., there could be isolated vertices (vertices with degree 0/chemical that has no hazard-pair) that is not taken care of by the given pseudo-code.

B.3 Dynamic Table, Stack, and Queue (15 marks)

Consider a standard (FIFO) queue that supports the following operations:

- PUSH(x): Add item x at the end of the queue.
- PULL(): Remove and return the first item present in the queue.
- SIZE(): Return the number of elements present in the queue.

We can easily implement such a queue using a doubly-linked list so that PUSH, PULL, and SIZE operations take at most c_1, c_2, c_3 worst-case time for some constants $c_1, c_2, c_3 > 0$.

Now suppose that you are asked to implement this (FIFO) queue using Python, but you are too lazy to code your own singly-linked list. Instead, you want to just use Python's built-in `list` (which, if you are not aware, implements Dynamic Table (i.e., a resizable array) as discussed in week 7 lecture). Even if you do not know Python, you just need the following information to solve this question:

- You can create an empty Python list L in $O(1)$ by calling `L = []`.
- You can append a new item x to the back of a Python list L in (amortized) $O(1)$ by calling `L.append(x)`. You can assume that in the event Python has to enlarge the internal list/array to two times of the original size because the internal list/array is full, it can do it automatically and we have analyzed this to be amortized $O(1)$ in week 7 lecture.
- You can remove (and return) the last item x from the back of a Python list L in by calling `L.pop()`. Again, you can assume that this operation also runs in (amortized) $O(1)$ even when intermixed with the append operation earlier.
- You can check the length of a Python list L in $O(1)$ by calling `len(L)`.

B.3.1 Python list as Efficient Stack (2 marks)

In pseudo-code (a few short sentences), explain how you can simply use a Python `list` to implement a (LIFO) stack efficiently, i.e., both of these two operations in (amortized) $O(1)$.

- PUSH(x): Add item x at the top of the stack.
- PULL(): Remove and return the top item present in the stack.

For PUSH(x), just do `L.append(x)`. The question already states that this is amortized $O(1)$.

For PULL(), just return `L.pop()`. The question already states that this is amortized $O(1)$ too.

B.3.2 Two Python lists as Efficient Queue (6 marks)

Now the actual question: In pseudo-code (a few short sentences), explain how you can use *two* Python `lists` (technically, two stacks, see Section B.3.1) to implement a (FIFO) queue efficiently, i.e., PUSH, PULL, and SIZE operations remains (amortized) $O(1)$. Please reserve this Section for explaining your

ideas and only explain the amortized analysis in Section B.3.3. (**Hints:** PUSH(x) to one of the stack, PULL() from the other stack.) **Ok, let's lower the difficulty by mentioning this hint...**

Let's label the two Python lists (stacks) as `s1` and `s2`.

For PUSH(x), just do `s1.append(x)`.

For SIZE(), just do `return len(s1)+len(s2)`.

For PULL(), we need to do the transformation from a LIFO data structure into a FIFO one:

If (and only if) `s2` is empty, we have to transfer *all* content of `s1` into `s2`, effectively reversing the order from LIFO to FIFO. After this `s1` is empty and `s2` contains the required elements in FIFO order. We can then `return s2.pop()`. Subsequent PUSH(x) operations will only affect `s1` and we do not transfer `s1` into `s2` until `s2` is empty again. This transfer process from `s1` to `s2` looks like a heavy $O(n)$ operation... But we shall see that this PULL operation remains amortized $O(1)$ in the next section. Btw, we do not yet take care of what happens if one calls PULL() when SIZE() == 0.

ChatGPT (GPT-4) answer:

```
class Queue:
    def __init__(self):
        self.s1 = [] # Python list 1 as stack 1
        self.s2 = [] # Python list 2 as stack 2

    def PUSH(self, x): # O(1)
        self.s1.append(x)

    def PULL(self):
        if not self.s2: # only executed if s2 is empty
            while self.s1: # is this O(len(s1)) -- heavy? or... amortized O(1)?
                self.s2.append(self.s1.pop())
        return self.s2.pop() # O(1)

    def SIZE(self): # O(1)
        return len(self.s1) + len(self.s2)

# Example usage:
q = Queue()
q.PUSH(1)
q.PUSH(2)
q.PUSH(3)
print(q.PULL()) # Output: 1
q.PUSH(4)
```

```
print(q.PULL()) # Output: 2
```

B.3.3 Amortized Analysis (7 marks)

Use **potential method** to show that in any intermixed sequence of PUSH, PULL, and SIZE operations of (FIFO) queue implemented using two Python lists/(LIFO) stacks as answered in Section B.3.2, the amortized cost of each of them is $O(1)$. (Assume, you start with an empty queue).

Suppose at any point of time the number of elements present in the $s1$ is l , which is $\text{len}(s1)$ (actual length of $s2$ is not actually needed although it is OK to include this too – the analysis will be a bit different). Then the worst-case (actual) running time of PUSH(x), PULL(), and SIZE() are c_1 , c_2l , c_3 , respectively, for some constant $c_1, c_2, c_3 > 0$. Now, we use the potential method to establish the desired bound on the amortized costs of the above three operations.

Let us consider the following potential function: $\phi = c_2 \cdot l$ (1 mark).

Alternative potential functions that also work are OK.

$\phi(D_0) = 0$, an empty queue (actually optional, asked in the redacted MCQ for this semester).

$\phi(D_i) \geq 0$, as the length of the queue is always non-negative (1 mark).

The easiest one first:

The amortized cost of SIZE(): $c_3 + \phi(D_i) - \phi(D_{i-1}) = c_3 + c_2 \cdot l - c_2 \cdot l = c_3$ (1 mark).

Basically, l does not change (so no change of potential at all).

Then the next easiest:

The amortized cost of PUSH(x): $c_1 + c_2 \cdot (l + 1) - c_2 \cdot l = c_1 + c_2$ (1 mark).

Basically, l (of $s1$) will increase by 1 upon each PUSH(x) to $s1$ only.

Now the most interesting part, the one that seems to do the most amount of work...

Two subcases:

Easier subcase: If $s2$ is not empty.

The amortized cost of PULL (case 1): $c_2 + c_2 \cdot l - c_2 \cdot l = c_2 + 0 = c_2$ (1 mark).

Harder subcase: If $s2$ is not empty, we transfer l items from $s1$ to $s2$ so LIFO becomes FIFO.

But by then, our potential value is already sufficient for this operation :).

The amortized cost of PULL (case 2): $c_2 + c_2 \cdot (l - 1) - c_2 \cdot l = c_2 - c_2 = 0$ (1 mark).

The conclusion: We have shown that all three operations have amortized cost of $O(1)$ (1 mark).

– END OF PAPER; All the Best –