HKUST – Department of Computer Science and Engineering
COMP3711: Design and Analysis of Algorithms – Fall 2021

#

Final Examination – Solution Key

Date: Dec. 7, 2021 Time: 08:30 -11:30 Venue: LG4 Multi-purpose Room,

Name: _____	Student ID: _____
Email: _____	Lecture L1 / L2

Instructions

- This is a closed book exam. It consists of 8 questions on 29 pages.
- Please write your name, student ID and ITSC email at the top of this page.
- At the top of every subsequent page that you use, please write your student ID in the space provided.
- Please sign the honor code statement on page 2.
- Answer all the questions within the space provided on the examination paper. The last 2 pages are scrap paper and may be detached from the booklet if you wish. You may use the backs of pages and the last 2 pages for your rough work.
- Each question is on a separate page and most have at least one extra page for writing answers. These are provided for clarity and are not meant to imply that each question requires all of the blank pages. Many can be answered using much less space.
- Do NOT write final solutions on the back side of the paper. Anything written on the back of the paper will be ignored.

Questions	1	2	3	4	5	6	7	8	Total
Points	10	15	14	12	10	13	14	12	100
Score									

Student ID: _____

As part of HKUST's introduction of an honor code, the HKUST Senate has recommended that all students be asked to sign a brief declaration printed on examination answer books that their answers are their own work, and that they are aware of the regulations relating to academic integrity. Following this, please read and sign the declaration below.

I declare that the answers submitted for this examination are my own work.

I understand that sanctions will be imposed, if I am found to have violated the University regulations governing academic integrity.

Student's Name: _____

Student's Signature: _____

Student ID: _____

1. **Asymptotic Analysis [10 pts]**

(a) We have two algorithms, A and B. Let $T_A(n)$ and $T_B(n)$ denote the time complexities of algorithm A and B respectively, with respect to the input size n. In the table shown below, there are 5 different cases of time complexities for each algorithm. Complete the last column of the following table with "A", "B", or "U", where:

- "A" means that for large enough n, algorithm A is always faster;
- "B" means that for large enough n, algorithm B is always faster;
- "U" means that the information provided is not enough to justify stating that, for large enough n, one algorithm is always faster than the other.
- We fill in the first line for you as an example.

Case	$T_A(n)$	$T_B(n)$	Faster
0	$\Theta(n)$	$\Theta\left(n^2\right)$	A
1	$\Theta\left(n^{1.5}\log n\right)$	$\Theta\left(n^2/\log n\right)$	
2	$\Theta\left(2^{\sqrt{n}}\right)$	$\Theta\left(4^{\sqrt{n}}\right)$	
3	$O(\log n)$	$\Omega(\log\log n)$	
4	$\Theta(4^n)$	$\Theta(\sqrt{16^n})$	
5	$\Theta(n^2)$	$T_B(n) = \begin{cases} 5T_B(n/2)+n & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$	

(b) Derive the exact asymptotic solution for the recurrence $T(1) = 1$ and

$$\forall n > 1, \quad T(n) = 16T\left(\frac{n}{4}\right) + n^2.$$

You may assume that n is always an integral power of 4. Show your derivation from first principles (you may NOT use the Master Theorem for this part). Show all your steps.

Write your final result in the form $T(n) = \Theta\left(n^\alpha(\log n)^\beta\right)$ for appropriate values of α and β.

Solution

(a)

Case	$T_A(n)$	$T_B(n)$	Faster
0	$\Theta(n)$	$\Theta\left(n^2\right)$	A
1	$\Theta\left(n^{1.5}\log n\right)$	$\Theta\left(n^2/\log n\right)$	**A**
2	$\Theta\left(2^{\sqrt{n}}\right)$	$\Theta\left(4^{\sqrt{n}}\right)$	**A**
3	$O(\log n)$	$\Omega(\log\log n)$	**U**
4	$\Theta(4^n)$	$\Theta(\sqrt{16^n})$	**U**
5	$\Theta(n^2)$	$= 5T_B(n/2) + n$	**A**

Notes

(1)
$$\frac{T_A(n)}{T_B(n)} = \Theta\left(\frac{\log^2 n}{\sqrt{n}}\right) \to 0.$$

(2)
$$\frac{T_A(n)}{T_B(n)} = \Theta\left(\frac{1}{2^{\sqrt{n}}}\right) \to 0.$$

(3) Note that $T_A(n) = \log_2 n$ and $T_B(n) = \log_2\log_2 n$ satisfy the requirements and for this pair of functions B is faster than A.

But, setting $T_B(n) = \log_2 n$ and $T_A(n) = \log_2\log_2 n$ ALSO satisfy the requirements, and for this pair of functions A is faster than B.

(4) $\sqrt{16^n} = 4^n$ so A and B have the same asymptotic running time and we can't know which, if either, is faster.

(5) From the Master Theorem, $T_B(n) = \Theta\left(n^{\log_2 5}\right)$. Since $\log_2 5 > 2$, A is faster than B.

(b) Let $h = \log_4 n$ so $n = 4^h$. Then

$$\begin{aligned}
T(n) &= 16T\left(\frac{n}{4}\right) + n^2 \\
&= 16\left(16T\left(\frac{n}{4^2}\right) + \left(\frac{n}{4}\right)^2\right) + n^2 \\
&= 16^2 T\left(\frac{n}{4^2}\right) + 2n^2 \\
&= 16^2\left(16T\left(\frac{n}{4^3}\right) + \left(\frac{n}{4^2}\right)^2\right) + 2n^2 \\
&= 16^3 T\left(\frac{n}{4^3}\right) + 3n^2 \\
&\cdots \\
&= 16^i T\left(\frac{n}{4^i}\right) + in^2 \\
&\cdots \\
&= 16^h T\left(\frac{n}{4^h}\right) + hn^2 \\
&= \left(4^h\right)^2 T(1) + n^2 \log_4 n \\
&= n^2 T(1) + \Theta(n^2 \log n) = \Theta(n^2 \log n)
\end{aligned}$$

2. **Dynamic Programming I [15 pts]**

In what follows, the input is an array $A[1..n]$ that stores n positive real values. A *subsequence* $S = \langle s_1, s_2, \ldots, s_k \rangle$ of A is given by a sequence of increasing indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ where $s_j = A[i_j]$. The *value* of S is $V(S) = \prod_{j=1}^{|S|} s_j = s_1 \times s_2 \times s_3 \cdots \times s_k$.

Given a subsequence S of A, we call S *stable* if $|S| = 1$ or, if $|S| > 1$ then

$$\forall j,\ 2 \leq j \leq |S|, \quad \frac{1}{2} \leq \frac{s_{j-1}}{s_j} \leq 2.$$

This just means that the ratio of two items in the sequence do not differ by too much. See the next page for examples illustrating all the definitions above as well as instructions for how to present your solutions.

We say that one stable subsequence S of A is *smaller* than another stable subsequence S' of A if the value of S is less than the value of S'.

Describe an $O(n^2)$-time dynamic programming algorithm to find the value of a smallest stable subsequence of A.

Note that there may not be a unique smallest stable subsequence of A. You only need to return the value of a smallest stable subsequence. You do not need to return the subsequence.

(a) Give the recurrence relation upon which you will base the logic of your algorithm. Do not forget to provide your initial conditions. Explain why your recurrence relation is correct.

(b) Describe your algorithm (either in words or pseudocode) and explain why it takes $O(n^2)$ time

Rules and Examples:

(i) When giving a recurrence, first *define* the items in your table. Recall that the items in a DP table are the minimal or maximum values of some subproblems.

When defining the items in your table, you need to explain what the corresponding subproblems are and what values are being calculated. If you have a one-parameter DP, call this item $c[i]$ and define the meaning of $c[i]$. If you have a two parameter DP, call it $c[i, j]$ and define the meaning of $c[i, j]$.

After defining the meaning, write the recurrence relation that $c[i]$ or $c[i, j]$ satisfies.

(ii) Examples. Consider $A = \left[5,\ \frac{3}{4},\ 7,\ 1,\ 6,\ \frac{1}{2},\ 5,\ 20, 10, 30\right]$.

$S_1 = \langle 5,\ 7, 20\rangle$ is NOT a stable subsequence of A.

$S_2 = \langle 5,\ 7,\ 6,\ 5,\ 10\rangle$, $S_3 = \left\langle \frac{3}{4},\ 1,\ \frac{1}{2}\right\rangle$ and $S_4 = \langle 6\rangle$ are all stable sequences of A.

$V(S_2) = 10,500$. $V(S_3) = \frac{3}{8}$. $V(S_4) = 6$.

S_3 is the smallest stable subsequence of A.

Solution:

In the tutorial you learned an algorithm for finding the longest *increasing* subsequence. The technique for finding the least value stable one one is very similar.

Let $A_i = A[1 \ldots i]$ denote the prefix of A consisting of the first i items.

Define $c[i]$ to be
the value of the smallest value (cheapest) stable subsequence in A_i that ends with $A[i]$.

It is clear that the value of the cheapest stable subsequence in A is given by

$$\min_{1 \le i \le n} c[i].$$

It is possible that the cheapest subsequence is just $A[i]$ itself.

If the cheapest subsequence has length two or more, then it is equal to $< Z, A[i] >$ where Z is a stable subsequence that ends with $A[r]$ for some $r < i$ satisfying $\frac{1}{2} \le \frac{A[r]}{A[i]} \le 2$.

Since we are looking for the *cheapest* stable subsequence ending with x_i,

- Z has to be the *cheapest* stable subsequence ending in x_r so
- Z has value $c[r]$ and
- $< Z, A[i] >$ then has value $A[i] \cdot c[r]$.

Since r could be *any* index $< i$ this gives the recurrence

$$c[i] = \begin{cases} A[i] & \text{if } i = 1 \\ A[i] & \text{if } i > 1 \text{ and } J_i = \emptyset \\ A[i] \cdot \min\left(1, \min_{r \in J_i} c[r]\right) & \text{if } i > 1 \text{ and } J_i \neq \emptyset \end{cases}$$

where

$$J_i = \left\{ i \,:\, 1 \le r < i \text{ AND } A[r] \in \left[\frac{A[i]}{2}, 2A[i] \right] \right\}$$

The following pseudocode calculates $c[i]$ directly from the recurrence and then returns the final answer.

For $i = 1$ to n
    $c[i] \leftarrow A[i]$
    For $r = 1$ to $i - 1$
        If $\left(A[r] \in \left[\frac{A[i]}{2}, 2A[i]\right] \quad \text{AND} \quad c[i] > A[i] \cdot c[i]\right)$ Then
            $c[i] \leftarrow A[i] \cdot c[i]$
Report $\min_{1 \le i \le n} c[i]$.

Filling in the table uses $O(n^2)$ time because the code does $O(1)$ work for each (r, i) pair with $r < i \le n$ and there are $O(n^2)$ such pairs.

As mentioned at the beginning, the final value will be

$$\min_{1 \le i \le n} c[i].$$

which can be calculated using another $O(n)$ time from the $c[i]$ table, so the entire algorithm requires $O(n^2)$ time.

Marking Note.

Many students had trouble with this question. Please note that the solution given (or a very minor variant) was the only acceptable solution All other "solutions" we saw were wrong.

If you believe that you had a different solution that was correct, first read all of the below. Then try to see if your recurrence works on the input

$$A' = [2^{-10}, 2^{-10}, 2^{-6}, 2^{-6}, \ 2^{-6}, , \ 2^{-6}, \ ]$$

Only appeal if you can guarantee that your recurrence works on this A' and you do not have any of the issues below.

Here are some of the incorrect solutions we saw.

- **$c[i]$ was replaced by $pi]$ where $p[i]$ was defined as**
  the value of the smallest value (cheapest) stable subsequence in A_i.

  Note that this differs from $c[i]$ in that the stable sequence no longer is required to *end* at $A[i]$. The sequence only has to be *contained* within A_i. Without this ending requirement the recurrence totally fails.

  With this approach, students often wrote a recurrence like

  $$p[i] = \min\left(p[i-1],\ A[i], A[i] \cdot \min_{r \in J_i} p[r]\right).$$

  Try running this recurrence on the A' given above to see why it fails.

  The problem is that this recurrence is forgetting important information about what the best stable sequence ending at a location is.

  Some students tries augmenting this by saying that they remembered the location at which the best stable sequence in A_i ended. That didn't help. Try running that algorithm on A' above to see why.

- **Some students modified this to define $p[i, j]$ to be**
  the value of the smallest value (cheapest) stable subsequence in $A[i \ldots j]$.

This had the same issue as the above.

- **Others defined $p[i, j]$ to be**
  the value of the smallest value (cheapest) stable subsequence of length j in A_i.

  This again had the same issue as the above.

- **Others defined $p[i, j]$ to be**
  the value of the smallest value (cheapest) stable subsequence of length j in A_i that ends at $A[i]$.

  This would work, but would require $O(n)$ time to calculate each $p[i, j]$ so the algorithm would need $O(n^3)$ time and not $O(n^2)$.

If your recurrence was marked wrong and you believe that it is correct then you can file an appeal. When you file the appeal, (i) type up your solution from the exam.

Then (ii) type up a formal mathematical proof of correctness that would yield an $O(n^2)$ time algorithm.

A precondition to your appeal being processed is that your recurrence must be well-defined and your proof of correctness must be mathematically rigorous.

When marking, we spent a lot of time working through ill-defined recurrences that were ambiguous. Those will not be accepted for appeals. Recurrences that are not well-defined and do not have an accompanying proof will not be checked.

3. **Dynamic Programming II [14 pts]**

In what follows, the input consists of two items. The first input item is an array $A[1..n]$ that stores n non-zero digits, i.e., for $1 \leq i \leq n$, $A[i] \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The second input item is a positive integer m that is less than n.

We want to insert m "+"s into A to turn it into a summation which we call an *m-sum*. As an example, if $m = 2$ and $A = [7, 9, 8, 4, 6]$, there are six valid ways of inserting two "+"s to form six 2-sums:

$$\begin{array}{lll} 7 + 9 + 846, & 7 + 98 + 46, & 7 + 984 + 6, \\ 79 + 8 + 46, & 79 + 84 + 6, & 798 + 4 + 6. \end{array}$$

Your task is to design an algorithm to find a smallest m-sum, that is, an m-sum that gives the smallest summation value. You only need to output the value of a smallest m-sum. As an illustration, in the above example, the output should be 133 because the smallest 2-sum is $79 + 8 + 46$.

Describe an $O(mn^3)$-time dynamic programming algorithm to find the value of a smallest m-sum.

(a) Give the recurrence relation upon which you will base the logic of your algorithm. Do not forget to provide your initial conditions. Explain why your recurrence relation is correct. Follow the instructions described in notes (i) on page 7 for giving a recurrence.

(b) Describe your algorithm (either in words or pseudocode) and explain why it takes $O(mn^3)$ time.

Your DP/code may use the function $atoi(i, j)$, $1 \leq i \leq j \leq n$, which converts $A[i \ldots j]$ into an integer. As an illustration, $atoi(2, 4) = 984$ in the above example. You may assume that a call to $atoi(i, j)$ takes $O(j - i + 1) = O(n)$ time.

Note: Original exam printing stated that $atoi(2, 4) = 948$; this was corrected during the exam.

Solution

(a) Define $c[i, j]$ to be the minimum value obtainable by inserting j "+"s into the subarray $A[1 \ldots i]$, i.e., the minimum value j-sum for $A[1 \ldots i]$.

Note that $c[i, j]$ is only defined if $j < i$.

If $j = 0$, no "+" sign is inserted, which means that the digits in $A[1..i]$ form the integer $atoi(1, i)$. Suppose that $j \geq 1$. Then, the rightmost "+" sign must lie between $A[k]$ and $A[k + 1]$ for some $k \leq i - 1$. The other $j - 1$ "+" signs must be placed among the digits in $A[1..k]$. It follows that k must be greater than or equal to j. That is, k must be chosen from the range $[j, i - 1]$. The rightmost "+" sign splits $A[1..i]$ into two parts, namely $A[k + 1..i]$ and $A[1..k]$. The digits in $A[k + 1..i]$ form a single integer with value $atoi(k + 1, i)$. The digits in $A[1..k]$ together with the $j - 1$ "+" signs form a $(j - 1)$-sum of $A[1..k]$. Therefore, given a particular choice of k, the value of the corresponding j-sum of $A[1..i]$ is equal to the value of the $(j - 1)$-sum of $A[1..k]$ plus $atoi(k + 1, i)$. It follows that we should obtain the smallest $(j - 1)$-sum of $A[1..k]$ in order to obtain the smallest j-sum of $A[1..i]$ for this particular choice of k. Therefore, $c[i, j] \leq c[k, j - 1] + atoi(k + 1, i)$. Clearly, one of the choices of k in $[j, i-1]$ would give the value of $c[i, j]$. Hence, we obtain the following recurrence and boundary condition.

$$c[i, j] = \begin{cases} atoi(1, i) & \text{if } j = 0 \\ \min_{j \leq k < i} \left(c[k, j - 1] + atoi(k + 1, i) \right) & \text{otherwise.} \end{cases} \tag{1}$$

Alternatively, we can define $c[i, j] = \infty$ for the impossible cases and write

$$c[i, j] = \begin{cases} atoi(1, i) & \text{if } j = 0 \\ \infty & \text{if } j \geq i \\ \min_{1 \leq k < i} \left(c[k, j - 1] + atoi(k + 1, i) \right) & \text{otherwise} \end{cases} \tag{2}$$

(b) We do a simple loop using the recurrence to fill in the table

1. for $i = 1$ to n do

2. $c[i,0] \leftarrow atoi(1,i)$
3. for $j = 1$ to m do
4. for $i = j+1$ to n do
5. $c[i,j] \leftarrow c[j,j-1] + atoi(j+1,i)$
6. for $k = j+1$ to $i-1$ do
7. if $c[i,j] > c[k,j-1] + atoi(k+1,i)$ then
8. $c[i,j] \leftarrow c[k,j-1] + atoi(k+1,i)$

The final answer is $c[n,m]$.

Lines 1-2 use n iterations and $O(n^2)$ time because each call of $atoi(\cdot)$ takes $O(n)$ time.

Line 5 is executed $O(mn)$ times. Since each call to $atoi()$ requires $O(n)$ time, this is $O(n^2m)$ time in total

Lines 7-8 are executed $O(mn^2)$ times, requiring $O(mn^3)$ time as each call of $atoi(\cdot)$ takes $O(n)$ time.

So, the entire algorithm requires $O(mn^3)$ time.

Marking Note 3a. For full marks it may be necessary to define the meaning of $c[i,j]$ explicitly, depending on whether things are clear from the context.

Marking Note 3b. It was necessary to deal with the case $i \leq j$ either by only defining $c[i,j]$ for $i > j$ or using something like $c[i,j] = \infty$ to denote that impossible case.

Marking Note 3c:

There were two major types of DP design errors that appeared.

The first was to define a 1D $V[i]$, usually the "cost of summation" on $A[1 \ldots i]$. The "cost of summation"' was often not well defined so we assumed it meant the min cost of a summation using m "+'s on $A[1 \ldots i]$. This could not work because, after adding the $(i+1)$st, there was no way to know how to readjust the '+'s to its left.

Another was to define a 2D $V[i,j]$ to be the minimum sum achievable using m "+"'s on array $A[i \ldots j]$. Again, such a recurrence can not yield a correct solution. This is for the same reason. $V[i,j]$ does not store any information about where the m "+"'s are located so there is no efficient way of constructing $V[i,j]$ from earlier calculated values $V[i',j']$.

We did not provide counterexamples for most such submissions because most of the recurrences were not well-defined (this means that the recurrence was not structured to give one and exactly one $V[i,j]$ value for every legal i,j pair.

If your recurrence was marked wrong and you believe that it is correct then you can file an appeal. When you file the appeal, (i) type up your solution from the exam.

Then (ii) type up a formal mathematical proof of correctness that would yield an $O(n^3m)$ time algorithm.

A precondition to your appeal being processed is that your recurrence must be well-defined and your proof of correctness must be mathematically rigorous. Recurrences that are not well-defined and do not have an accompanying proof will not be checked.

You also (iii) need to show why this leads to an $O(n^3m)$ time algorithm.

4. **Sorting [12 pts]**

The input is an array $A[1..n]$ of n numbers in (a) and (b) below.

(a) The items in $A[1..n]$ are arbitrary distinct real numbers. **Prove that any comparison based algorithm for sorting A requires at least $\Omega(n \log n)$ comparisons.**

For the purposes of this question, you can only manipulate the elements of the array A using the following two subroutines:

- compare(A, i, j): This function returns -1 if $A[i] < A[j]$ and 1 if $A[i] > A[j]$.
- swap(A, i, j): This procedure swaps the contents of $A[i]$ and $A[j]$.

The only mathematical facts you may assume without proof are the ones provided in the mathematical handout provided along with the exam. If you do use such facts, you must explicitly state which fact you are using and that it is from the math handout before using it. All other statements that you use must be proven from scratch.

(b) The items in $A[1..n]$ are integers from the range $[1 \ldots k]$ and the items in $A[1..n]$ may not be distinct. A *range query* specifies two integers x and y that form an interval $(x, y]$. The answer to such a range query is the number of elements in A that are greater than x and less than or equal to y. As an example if $A = [4, 5, 2, 6, 8, 3, 4, 4]$, the range query for the interval $(3, 6]$ would return 5 because there are five elements in A that lie in $(3, 6]$, namely $A[1]$, $A[2]$, $A[4]$, $A[7]$, and $A[8]$.

Describe an $O(n + k)$-time algorithm that generates a new array C such that, using C, you can answer any range query in constant time.

You should clearly describe your algorithm for constructing C (either in words or pseudocode) and why your algorithm runs in $O(n + k)$ time. Then explain how to use C to solve range queries.

Hint. Consider the intermediate arrays built in Counting Sort. How could you use that to solve range queries for intervals of the form $(0, y]$?

Solution

(a) The proof is the one from the "Sorting in Linear Time" slides.

(i) Sorting n items can produce $n!$ different outputs.

(ii) This means that the decision- tree modeling any sorting algorithm has at least $n!$ leaves, corresponding to different outputs of the algorithm.

(iii) The math handout states that **A binary tree of height h has at most 2^h leaves.** Thus, the height of any comparison tree modelling a sorting algorithm must have height h, satisfying

$$2^h \geq n! \quad \Rightarrow h \geq \log_2 n!.$$

(iv) The math handout also states that $\log(\mathbf{n!}) = \mathbf{\Theta(n \log n)}$.

(v) Then $h = \Omega(n \log n)$, which is the same thing as stating that every sorting algorithm must use at least $\Omega(n \log n)$ comparisons on at least one input.

Marking Note: To get full credit for this solution your proof needed versions of all 5 of the statements above (although not necessarily in that order).

Also, as per the explicit instructions, for each mathematical statement that you used it was necessary to either
(i) prove it from scratch or
(ii) explicitly state the exact fact that you used from the mathematical handout BEFORE USING IT and be explicit that the fact is in the handout.
In particular you needed to explicitly state that the A binary tree of height h has at most 2^h leaves **and that** $\log(\mathrm{n!}) = \Theta(\mathrm{n} \log \mathrm{n})$.

If your proof missed this, e.g., you used a nonproven fact without having a preface explicitly stating it before use, points were deducted.

Finally, some students confused binary search trees **with** decision trees**. Those are two totally different things.**

(b) This is essentially just the first part of counting sort.

Create an array C such that $C[j]$ contains the number of elements in A with entry values $\leq j$.

1. For $j = 1$ to k % Intialize
2. $\quad C[j] \leftarrow 0$

3. For $i = 1$ to n % Set $C[j] = |\{i : A[i] = j\}|$
4. $\quad C[j] \leftarrow C[A[i]] + 1$

5. For $j = 2$ to k % Set $C[j] = |\{i : A[i] \leq j\}|$
6. $\quad C[j] \leftarrow C[j] + C[j-1]$

The algorithm above takes $O(n + k)$ time.

Given this C, $Range(i, j) = C[j] - C[i]$ can be found in $O(1)$ time.

Marking Note. Any constant time solution required using a similar technique. More explicitly, to preprocess in $\Theta(n + k)$ time and answer range queries in $O(1)$ time, it is necessary to use something like counting sort.

Many students only included something like lines 3-4 and not 5-6. Without lines 5-6, the range search would not be constant, e.g., if $|x - y| = k$ it would need $\Theta(k)$ time to calculate the answer. Solutions with this error had points deducted.

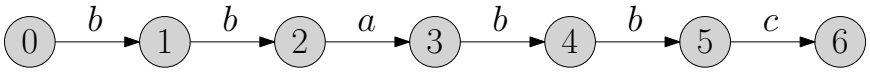
Appeals for this question, both parts A and B must contain a clear typed version of what you submitted so that we can read and understand it.

Student ID: _____

5. **A pattern matching automaton [10 pts]**

Your alphabet is $\Sigma = \{a, b, c\}$. Let P be the pattern string $\mathbf{P} =$ **bbabbc**. Describe the transition function for the pattern matching automaton for P by answering parts (a), (b), and (c) in the following. (Part (c) is on the next page.)

(a) In the diagram below we have drawn the 7 states of the pattern matching automaton for P and some of the arrows corresponding to its transition function. Draw all of the missing arrows that do NOT point to the state with label 0. Do not forget to label each arrow you draw with the proper letter from the alphabet Σ.



(b) Fill in all the missing entries in the transition function table of the automaton in part (a). We have already filled in the entries corresponding to the pre-drawn arrows.

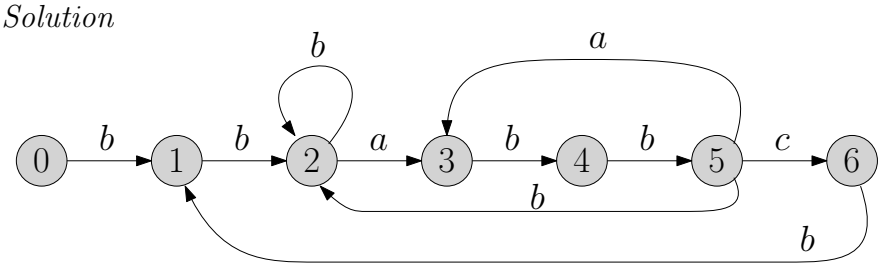
state	a	b	c	P
0		1		b
1		2		b
2	3			a
3		4		b
4		5		b
5			6	c
6				

Note: In the original exam the final column of the table stated a, a, b, a, a , c. This was corrected to b, b, a, b, b , c.

(c) What is/are the accepting state(s) in the automaton drawn in part (a)? Fill in your answer below

$$\text{Accepting State(s)} = \Big\{ \qquad\qquad \Big\}.$$

Solution



Marking Note: Many students forgot the edge from state 6 to state 1.

state	a	b	c	P
0	0	1	0	b
1	0	2	0	b
2	3	2	0	a
3	0	4	0	b
4	0	5	0	b
5	3	2	6	c
6	0	1	0	

Marking Note: Many students did not fill in row 6.

C) What is/are the accepting state(s) in the automaton drawn in Part (A)? Fill in your answer below

$$\text{Accepting State(s)} = \Big\{ \quad \mathbf{6} \quad \Big\}.$$

6. **Divide and Conquer [13 pts]**

In what follows, the input is an array $A[1..n]$ for some integer $n > 1$ that stores n real numbers with the following properties:

- Every $A[i]$ is non-zero.
- $A[1] < 0$ and $A[n] > 0$.
- For every $2 \leq i \leq n-1$, if $A[i] > 0$, then $A[j] > 0$ for all $i+1 \leq j \leq n$.

Design an $O(\log n)$-time algorithm for finding the index of the first positive number in A.

As an example, if

$$A = [-1, -30, -5, -7, 100, 20, 1, 200]$$

the algorithm should return 5 since $A[5] = 100$ is the first positive number in the array.

(a) Give a documented pseudocode for your algorithm.

(b) Briefly explain why your algorithm is correct.

(c) Explain why your algorithm runs in $O(\log n)$ time.

Solution:

Note that, from the problem definition, you may assume that $n \geq 2$.

(a) The solution is just a minor modification of binary search

Call with $Find(1, n)$

$Find(i, j)$ will maintain the invariant that $i < j$ with $A[i]$ being negative and $A[j]$ being positive.

$Find(i, j)$ % Finds the correct answer within range $A[i..j]$
1. If $j = i + 1$ $\quad \%A[i] < 0$ and $A[i + 1] > 0$
2. $\quad Return(j)$
3. Else $\quad$ % search in left half or right half
4. $\quad m \leftarrow \lfloor \frac{i+j}{2} \rfloor$
5. $\quad$ If $A[m] < 0$ then
6. $\quad\quad Find(m, j)$
7. $\quad$ Else
8. $\quad\quad Find(i, m)$

(b) The algorithm maintains the invariant that, when $Find(i, j)$ is called, $i < j$, $A[i]$ is negative, and $A[j]$ is positive.

At each step, it reduces the size of the range $[i, j]$. So, the algorithm must terminate when $j = i + 1$, $A[i] < 0$ and $A[i + 1] > 0$.

But, in this case, the answer must be $j = i + 1$.

Marking Note 6a: A correct justification had to somehow, explicitly or implicitly, specifically identify the fact that the current range called by the algorithm contains the answer. This is THE reason that the recursion works. Note that the solution above identified this fact implicitly, by identifying the invariant.

A solution that did not identify this fact had points deducted since it was missing an essential part of the justification

Marking Note. Some students were not careful in their code and referenced indices that did not exist, e.g., $A[m]$ where $m = 0$ or $m = n + 1$. These had some points deducted.

Others recursed improperly. For example, their recursion on $Find(i, j)$ might recurse into one of $Find(i, q)$ or $Find(q + 1, j)$ (instead of $Find(q, j)$. The problem is that this sometimes led to situations in which ALL of the items in the new subarray were all negative or all positive and their code did not work properly because of that.

(c) This is the standard analysis of binary search. Running time is $T(2) = 1$ and

$$T(n) = T(n/2) + 1$$

which we know, by the Master Theorem (with $a = 1$, $b = 2$ and $c = 0$), is $O(logn)$. This is also the same recurrences we had for binary search

Note that because it is already included in the Master Theorem, it is not necessary to deal with floors and ceilings of $n/2$.

Many students wrote unclear or ambiguous code with ambiguous explanations so we had no idea what was going on.

Appeals for this problem must include a typed version of what you wrote on the exam paper, cleanly formatted to be readable.

7. **Interval Scheduling [14 pts]**

In this problem you will describe and explain the correctness of the interval scheduling algorithm that was taught in class.

> *The Interval Scheduling Problem*: Suppose that you are given n intervals $I_1, I_2, \ldots, I_{n-1}, I_n$ that are sorted from left to right according to some criterion. For $1 \leq i \leq n$, the start and finish times of I_i are denoted by s_i and f_i, respectively. Two intervals I_i and I_j are *compatible* if their interior do not overlap, i.e., $s_i \geq f_j$ or $s_j \geq f_i$. The *interval scheduling problem* is to select a subset of mutually compatible intervals that has the maximum size.

> *The Greedy Algorithm*: The following pseudocode describes a greedy algorithm for the interval scheduling problem.
>
> 1. $A \leftarrow \{I_1\}$;
> 2. for $j = 2$ to n
> 3. if I_j is compatible with the intervals in A
> 4. $A \leftarrow A \cup \{I_j\}$; /* add I_j to the set A */
> 5. output the set A

We have not specified HOW the intervals are sorted. Consider the following two ways of sorting:

(a) By increasing starting time, i.e., $s_1 \leq s_2 \leq \cdots \leq s_n$.

(b) By increasing finishing time, i.e., $f_1 \leq f_2 \leq \cdots \leq f_n$.

Answer the following question separately for each of the two sorting orders in (a) and (b) above.

Will the greedy algorithm return a subset of mutually compatible intervals that has the maximum size? Explain.

If the answer is yes, provide a formal proof of correctness.

If the answer is no, provide a counterexample. Your counterexample should specify a set of input intervals. You should state the subset returned by the greedy algorithm and explain why that subset is not a correct solution.

Solution:

(a) When the intervals are sorted by increasing start time the solution does not have to be correct. Consider this counterexample:

$$I_1 = (1, 6),\ I_2 = (2, 3),\ I_3 = (4, 5).$$

The optimal solution is $\{I_2, I_3\}$ while the greedy algorithm will return just the one interval $\{I_1\}$.

(b) When the intervals are sorted by increasing finish time the solution us optimal.

This is the analysis from the class notes:

(a) Assume greedy is different than OPT

(b) Let $i_1, i_2, \ldots, i_k$ denote the set of intervals selected by greedy
We are ordering them so that $i_1 < i_2 < \cdots$

(c) Let $j_1, j_2, \ldots, j_m$ denote the set of intervals selected by OPT
Again we are ordering them so that $j_1 < j_2 < \cdots$
Since an interval must end before the next one starts this means $f_{j_r} \leq s_{j_{r+1}}$ for all r

(d) Note that by the definition of OPT, $m \geq k$.

(e) Find largest possible value of r such that $i_1 = j_1$, $i_2 = j_2$, $\ldots$, $i_r = j_r$.
This means that we can write OPT as

$$OPT = i_1, i_2, \ldots, i_r, \mathbf{j_{r+1}}, j_{r+2}, \ldots, j_m.$$

Note that it's possible that $r = 0$.

(f) By the definition of the greedy algorithm j_{r+1} can not finish before i_{r+1}, otherwise the greedy algorithm would have chosen j_{r+1}. This means that $f_{i_{r+1}} \leq f_{j_{r+1}}$

(g) From the comment in (c) above, this automatically implies that $f_{i_{r+1}} \leq f_{j_{r+1}} \leq s_{j_{r+2}}$.
This simply means that we can replace j_{r+1} with i_{r+1} in OPT and still have a set of compatable coverings. This gives us

$$OPT' = i_1, i_2, \ldots, i_r, \mathbf{i_{r+1}}, j_{r+2}, \ldots, j_m.$$

Now note that $size(OPT') = m = size(OPT)$ so OPT' is also optimal.

(h) We have therefore found an optimal solution that agrees with OPT with at least $r + 1$ places instead of r places.

(i) We can repeat lines (a)–(g) until we have found an optimal solution OPT that agrees with greedy on all of its first k places

(j) If $m > k$ then greedy could have chosen j_{k+1} after choosing i_k. Since greedy did not choose it, j_{k+1} does not exist so $m = k$.

(k) Since $m = k$, greedy is optimal

Marking Notes:

(a) required a "No" answer + a correct counter example illustrating that the size (number of intervals) of a Greedy solution can be smaller than that of an optimal solution.

(b) required a "Yes" answer. Key steps in the lecture proof were as follows:
Assume that Greedy is different from Optimal and define the two solutions.
Define the distance between Greedy and Optimal, i.e., the first place that the two solution differs.
Define the one-step modification of Optimal into OPT*.
Justify why the modification is legal (compatible with the remaining intervals chosen by optimal).
Justify why the modified solution OPT* is still optimal (number of intervals selected remains the same after the modification).
Repeat the process until OPT* is converted into Greedy. Greedy thus is optimal.

Note that an induction proof was also possible.

Common mistakes encountered were:

(1) Mistaking the problem for interval partitioning and discussing the classroom allocation problem instead.

(2) The problem asks for "a maximum size set of mutually compatible intervals"; size here means the number of intervals, *not* the total length of the intervals.

(3) A counterexample was drawn but there was no EXPLANATION as to why it showed that greedy is not optimal.

(4) Providing pseudo code or algorithm description rather than the formal proof as shown in the lecture note.

(5) Not using the ordering given in the problem at all in the proof. (Without using the ordering it is impossible to differentiate between (a) and (b)).

(6) In the proof by incremental modification, not giving a clear definition of the position (say r) in which the two algorithms differ – only stating that there is "a location" in the solution. Also, some notational mistakes showing that sizes of Greedy solution and optimal solution are the same.

(7) In the proof by incremental modification, providing no explanation as to why that the finishing time of the Greedy choice is smaller or equal to that of the optimal solution. Some got the direction of the proof wrong.

(8) In the proof by incremental modification, saying the modification is legal without explanation. Some proved that it is legal for $r-1$, but do not prove it for $r+1$.

(9) In the proof by incremental modification, not explaining why after modification OPT* is still optimal.

(10) In the proof by incremental modification, saying that Greedy is optimal immediately after the one-step modification. It is necessary to *repeat* the process (not by induction here) until OPT* is completely converted into Greedy.

(11) In the proof by induction, not providing the base case.

(12) In the proof by induction it was necessary to assume that Greedy is optimal for *any* input of a size *smaller* than n (for inductive proof on n), not just a particular

problem of size $n-1$. Not making this explicit was an error.

(13) In the proof by induction, understanding that Greedy is optimal means understanding that the *size* of the Greedy output is the same as the size of the optimal output, but their actual choices can be different.

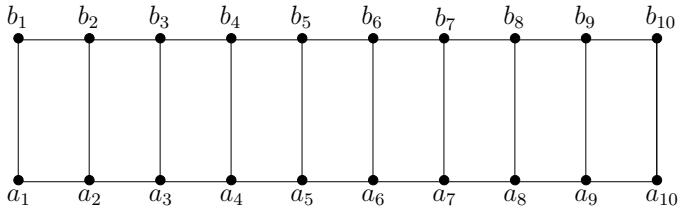
Many students wrote unclear or ambiguous proofs so we had no idea what was going on.

Appeals for this problem must include a typed version of what you wrote on the exam paper, cleanly formatted to be readable.

Student ID: _____

8. **BFS & DFS [12 pts]**

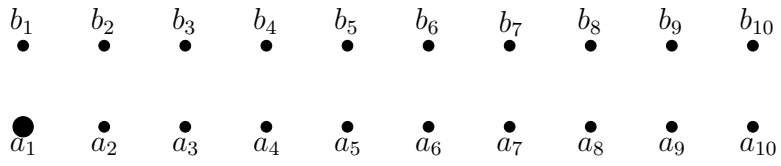
Consider the undirected graph G below.



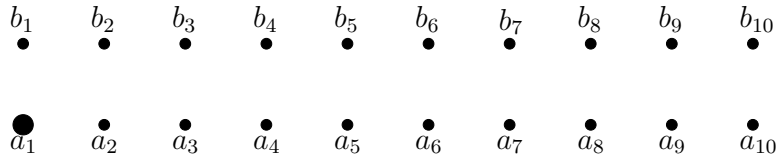
Its adjacency lists at each vertex are created by going up, right, down, and left (for the edges that exist). Explicitly, the adjacency lists are

$$\begin{array}{lll} & a_1 : b_1 \to a_2; & b_1 : b_2 \to a_1; \\ \forall i,\ 1 < i < 10, & a_i : b_i \to a_{i+1} \to a_{i-1}; & b_i : b_{i+1} \to a_i \to b_{i-1}; \\ & a_{10} : b_{10} \to a_9; & b_{10} : a_{10} \to b_9 \end{array}$$

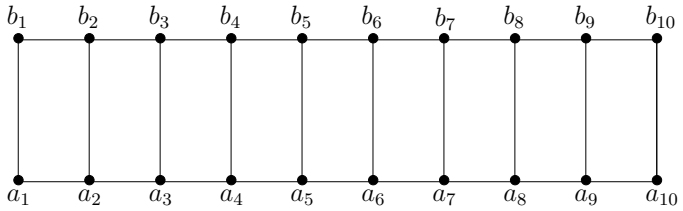
(a) Using the vertices below, draw the edges in the **Breadth First Search (BFS)** tree that results when running BFS on G **starting from node** a_1**.**



(b) Using the vertices below, draw the edges in the **Depth First Search (DFS)** tree that results when running DFS on G **starting from node** a_1**.**

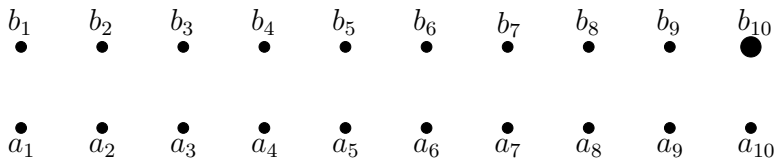


Take the same graph G and adjacency lists as on the previous page, which we reproduce below for your ease of reference.

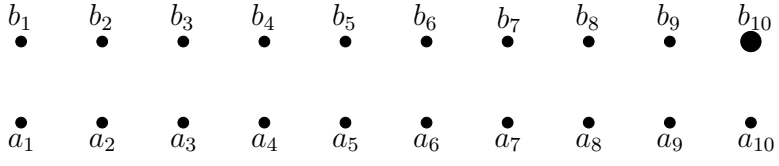


$$
\begin{array}{lll}
 & a_1 : b_1 \to a_2; & b_1 : b_2 \to a_1; \\
\forall i,\ 1 < i < 10, & a_i : b_i \to a_{i+1} \to a_{i-1}; & b_i : b_{i+1} \to a_i \to b_{i-1}; \\
 & a_{10} : b_{10} \to a_9; & b_{10} : a_{10} \to b_9
\end{array}
$$

(c) Using the vertices below, draw the edges in the **Breadth First Search (BFS)** tree that results when running BFS on G **starting from node** b_{10}**.**

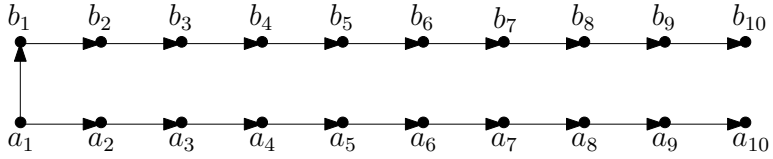


(d) Using the vertices below, draw the edges in the **Depth First Search (DFS)** tree that results when running DFS on G **starting from node** b_{10}**.**

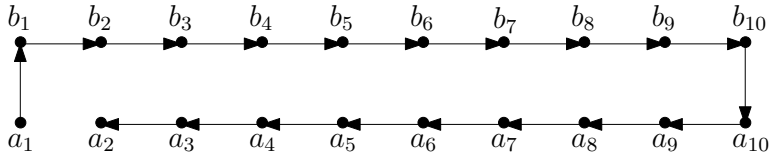


Solution

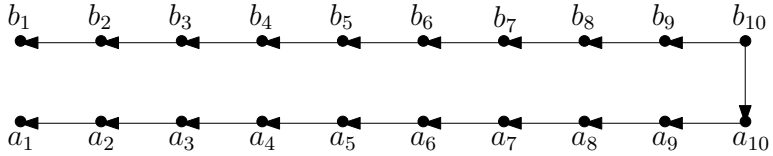
(a) The **Breadth First Search (BFS)** tree that results when running BFS on G **starting from node** a_1**.**



(b) The **Depth First Search (DFS)** tree that results when running DFS on G **starting from node** a_1**.**



(c) The **Breadth First Search (BFS)** tree that results when running BFS on G **starting from node** b_{10}**.**



(d) The **Depth First Search (DFS)** tree that results when running DFS on G **starting from node** b_{10}**.**