The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 1, Semester 1 2021

Sample Solutions

Objectives

To improve your understanding of the time complexity of algorithms and recurrence relations. To develop problem-solving and design skills. To improve written communication skills; in particular the ability to present algorithms clearly, precisely and unambiguously.

Problems

1. **[3 marks]**

   Use repeated substitution (or telescoping) to determine the asymptotic upper bound for the running time of an algorithm defined by the recurrence relation below:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 4T(\frac{n}{2}) + d\,n & \forall n > 1 \end{cases}$$

   where: c and d are constants. Assume that n is a positive value and is a power of 2.

Solution:
Let $2^i = n$.

$$\begin{aligned} T(n) &= 4T(\tfrac{n}{2}) + dn \\ &= 4(4T(\tfrac{n}{4}) + \tfrac{n}{2}d) + dn \\ &= 16T(\tfrac{n}{4}) + 3dn \\ &= 16(4T(\tfrac{n}{8}) + \tfrac{n}{4}d) + 3dn \\ &= 64T(\tfrac{n}{8}) + 7dn \\ &= ... \\ &= 4^iT(\tfrac{n}{2^i}) + (2^n - 1)dn \\ &= n^2T(\tfrac{n}{n}) + (n-1)dn \\ &= (c+d)n^2 - dn \\ &\in \mathcal{O}(n^2) \end{aligned}$$

2. **[6 marks]**

   In this question, you must fill in the table below.

   Assuming that there is always at least one negative value and at least one positive value in the given data structure (either an array or a linked list for the purpose of this question), what is the worst-case run time when searching for the first negative value (the *key*) in the data structure given the constraints listed in the 'Order of elements' column?

Data structure	Order of elements	Time complexity	Brief justification
Array	Random order		
Array	Sorted in ascending order		
Array	Sorted in descending order		
Linked list	Random order		
Linked list	Sorted in ascending order		
Linked list	Sorted in descending order		

Solution:

(a) Array - Random order: $\mathcal{O}(n)$
Since elements are randomly ordered, we need to check for each index value (from lowest) in order to find the negative value with smallest index. Sequential search can be used. In the worst case, there is only one negative value and it is located at the end of the array, requiring n comparisons.

(b) Array - Sorted in ascending order: $\mathcal{O}(1)$
Since the array is sorted in ascending order and there is at least one negative value. It's guaranteed that the first element in the array is negative.

(c) Array - Sorted in descending order: $\mathcal{O}(logn)$
We may use a slightly modified binary search to locate the first negative value. When coming across a negative value, we keep search from its left sub-array ($hi \rightarrow mid-1$). When seeing a non-negative value, we continue with its right sub-array ($lo \rightarrow mid+1$). When the condition $lo <= hi$ is not satisfied, we return the value of lo, which is the index of first negative value.

(d) Linked list - Random order: $\mathcal{O}(n)$
Again we need sequential search. In the worst case, there is only one negative value and it is located at the end of the array.

(e) Linked list - Sorted in ascending order: $\mathcal{O}(1)$ Since elements are sorted in ascending order and there is at least one negative value. It's guaranteed that the first node in the linked list contains a negative value.

(f) Linked list - Sorted in descending order: $\mathcal{O}(n)$
With a linked list data structure, we couldn't take benefit from applying binary search as accessing a particular element requires $\mathcal{O}(n)$ time rather than $\mathcal{O}(1)$. We need to again use sequential search.

3. **[9 marks]**

Consider an integer array A that contains n distinct elements.

A student in COMP90038 has developed an algorithm CheckSum that can be used to determine if there are two distinct numbers in the array that add up to a key, which is passed as an argument to the function.

```
function CheckSum(A[0..n − 1], key)
    SelectionSort(A[0..n − 1])                    ▷ function is available to use
    i ← 0
    j ← n − 1
    while i < j do
        if A[i] + A[j] = key then
            return True
        else
            if A[i] + A[j] < key then
                i ← i + 1
            else
                j ← j − 1
    return False
```

(a) (2 marks) Another student in COMP90038 suggested that the algorithm is incorrect, however, you are convinced that the algorithm is in fact correct. Present a clear and concise argument (maximum of 100 words) that you could share with the other student explaining why the CheckSum algorithm is correct.

Solution:
If there is a pair of elements in the array that meet the condition, let's say $A[p]+A[q] = key$ where $0 \leq p < q \leq n-1$. Suppose we have i reached p prior to j reaching q, the value of i will no longer change as now $A[i] + A[j] > key$ so that we will keep decrement j. Similarly if j reached q first, we will increment the value of i as $A[i] + A[j] < key$. For either case, we will eventually have $i = p$ and $j = q$, such that the algorithm will correctly return True. On the other hand, if there is no such pair of values meets the condition, the algorithm clearly has no chance of returning a True. Each iteration in the loop will either increment the value of i or decrement the value of j, the loop will be terminated once $i = j$ and the algorithm will return False.

(b) (2 marks) What is time complexity of the CheckSum algorithm? Justify your answer.

Solution:
The algorithm has a complexity of $\Theta(n^2)$.
The SelectionSort algorithm itself has a time complexity of $\Theta(n^2)$.
Before entering the While loop, $j - i$ has a value of $n - 1$. Each iteration will decrease the value of $j - i$ by exactly one. As a result, the body of the While loop can be executed for at most $n - 1$ times. Hence the loop has a time complexity of $\mathcal{O}(n)$ (expressed using Big-Oh as we may have early return).
All other operations take constant time.

(c) (2 marks) In lecture 11, the MergeSort algorithm is introduced. MergeSort has a time complexity of $\Theta(n \log n)$. If we replace SelectionSort with MergeSort in the

CheckSum algorithm, will the time complexity of the CheckSum algorithm change? Justify your answer (maximum of 50 words).

Solution:
The complexity of this algorithm will become $\Theta(nlogn)$.
Apart from sorting, the body of this algorithm takes $\mathcal{O}(n)$ time, which is dominated by the complexity of SelectionSort and MergeSort.

(d) (3 marks) One of the tutors in COMP90038 suggested that the CheckSum algorithm can be modified to determine whether the array A contains three elements that sum up to zero.

Describe how you would change the algorithm listed above to meet this new requirement. Your description does not have to be in pseudocode. However, the expectation is that any text description will be less than 100 words.

What time efficiency does the modified algorithm achieve?

Solution:
We may enumerate each element $A[k]$ and use the same loop in CheckSum to see if we can find $A[i] + A[j] = -A[k]$, where $0 \leq i < j < k \leq n-1$. The array only need to be sorted once at the beginning of the algorithm. The overall complexity is $\mathcal{O}(n^2)$.
One possible version of modified algorithm is show below in pseudocode.

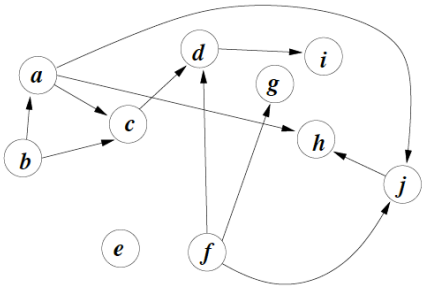
```
function CheckSumZero(A[0..n − 1])
    SelectionSort(A[0..n − 1])
    for k ← 2 to n − 1 do
        if CheckSum(A[0..k − 1], −A[k]) then
            return True
    return False
function CheckSum(A[0..n − 1], key)
    i ← 0
    j ← n − 1
    while i < j do
        if A[i] + A[j] = key then
            return True
        else
            if A[i] + A[j] < key then
                i ← i + 1
            else
                j ← j − 1
    return False
```

4. **[3 marks]**

Levitin suggests that a decrease-and-conquer approach can be used for the topological sorting problem. The idea is to repeatedly remove a source (a node with no incoming edges) and list the order in which such nodes were removed.

Describe an algorithm for this version of topological sort. Full marks will be given for an algorithm that runs in time $O(|V| + |E|)$.

Demonstrate your algorithm running using the following DAG.



Solution:
Assume the graph is represented using adjacent list.
Create an array of size $|V|$. Go through the adjacent list and record the in-degree of each node in the array. ($\mathcal{O}(|V| + |E|)$)
Scan through the array, adding all nodes with zero in-degree into a queue. ($\Theta(|V|)$)
Repeat until the queue becomes empty: Take a node v from the queue, decrease the in-degree by one for all nodes in v's adjacent list. Once a node's in-degree becomes zero, add that node into the queue. ($\mathcal{O}(|V| + |E|)$)
The order of nodes added into the queue is a valid topologically sorted order.

5. **[9 marks]**

   (a) (5 marks) Design an algorithm which, given an array $D[0..n-1]$ ($n > 1$), constructs a graph with nodes $0, .., n-1$ such that node i has degree $D[i]$. Your algorithm should return the constructed graph using an adjacency list format. That is, your algorithm should return an array A whose elements are a list of nodes and corresponding edges consistent with the representation shown on the lecture slides: Lecture 7 - slide 13.

   You may assume that the function AddEdge(A, i, j), which adds node i into adjacency list $A[j]$ and node j into adjacency list $A[i]$, is available. Your algorithm should use this function when constructing the graph.

   It is important to note that it is possible that multiple graphs could be constructed satisfying the requirements listed above. However, your algorithm should just produce any one of the correct graphs. For example, given input array $D = [1, 2, 2, 1, 1, 3]$, it might produce this graph representation:

$$
\begin{aligned}
A[0] &= 2 \\
A[1] &= 2 \rightarrow 5 \\
A[2] &= 0 \rightarrow 1 \\
A[3] &= 5 \\
A[4] &= 5 \\
A[5] &= 1 \rightarrow 3 \rightarrow 4
\end{aligned}
$$

   Your algorithm must be written in unambiguous pseudocode. You should try to make your algorithm as efficient as you can.

   (b) (2 marks) Explain how your algorithm works (maximum of 100 words).

   (c) (2 marks) State the time complexity of your algorithm as a function of |V|=n and |E|=$sum(D[i])/2$.

Solution:
When we connect two nodes i and j with an edge, we will decrease both $D[i]$ and $D[j]$ by one. In the below answer, we call this updated $D[i]$ and $D[j]$ values as remaining degree.
It may not work to connect two random nodes (with degree greater than 0). For example, with the input $D = [1, 1, 2]$, we shouldn't connect the first two nodes with an edge as there will be no other nodes available to be connected with the last node. One possible idea is to repeatedly pick a node i with highest degree $D[i]$, connecting it with $D[i]$ number of nodes. These $D[i]$ nodes should have highest remaining degrees among all nodes (except node i of course).
A straightforward implementation would be, sort all nodes by their remaining degrees (while keeping track of their original indices), take the first node (say, has index i in the original array) and connect it with the next $D[i]$ nodes. This process is repeated until all nodes have remaining degree of zero. The time complexity of this approach is $\mathcal{O}(|V|^2 log|V|)$.

function ConstructGraph($D[0..n-1]$)
  Initialize($A[0..n-1]$) ▷ Initialize n empty linked lists
  $B[0..n-1] \leftarrow$ Sort($D[0..n-1]$) ▷ Use any $O(n \log n)$ sorting algorithm to get reversely sorted index so that $D[B[0]]$ is the largest number and $D[B[n-1]]$ is the smallest number
  for $i \leftarrow 0$ to $n-1$ **do** ▷ Iterate through each node
    if $D[B[i]] > 0$ **then** ▷ If the current node has remaining degree(s)
      for $j \leftarrow i+1$ to $i + D[B[i]]$ **do** ▷ Connect with the next $D[B[i]]$ nodes
        AddEdge($A, D[B[i]], D[B[j]]$)
        $D[B[j]] \leftarrow D[B[j]] - 1$
      $B[i+1..n-1] \leftarrow$ Sort($D[i+1..n-1]$) ▷ To maintain sorted property
  return $A[0..n-1]$

We may observe that after connecting the first node $B[0]$ with the next $D[B[0]]$ nodes, both $D[1..B[0]]$ and $D[B[0]..n-1]$ are still sorted. So instead of calling the Sort algorithm, we may simply apply (a slightly modified) Merge function to maintain the sorted property (in $\mathcal{O}(n)$ time). This will improve the overall complexity to $\mathcal{O}(|V|^2)$.
We can further improve the complexity by avoiding re-sort the array $B[]$. An additional array $P[0..n-1]$ can be created to store the **last index of each degree value**. For example, with the input $D = [4, 4, 2, 2, 1, 1, 0]$, we will have $B = [0, 1, 2, 3, 4, 5, 6]$ and $P = [6, 5, 3, -1, 2, -1, -1]$, indicating that the last 0 is at index 6, the last 1 is at index 5, the last 2 is at index 3 etc. When generating edges, we will take nodes having same remaining degrees by reversed order. For example, with the above example, we will connect the first node with the next three nodes as well as the second last node ($D' = [0, 3, 1, 1, 1, 0, 0]$). Now the complexity becomes $\mathcal{O}(|V|log|V| + |E|)$.
Consider that all node degrees must be between 0 and $n-1$, we may use the idea of counting sort to get the array B in linear time. An array (say, $T[]$) of n linked lists could be first created to categorise nodes with same degrees (i.e. $T[i]$ is a linked list storing all nodes with degree i). It requires $\mathcal{O}(n)$ time to scan through $D[]$ and place each node into $T[]$, and another $\mathcal{O}(n)$ time to iterate through $T[]$ and generate $B[]$. The overall complexity now becomes $\mathcal{O}(|V| + |E|)$.

(solution continued)

function SORT($A[0..n-1]$)
    $X[0..n-1] = [0, 1, 2, ..., n-1]$ ▷ X stores original index values of elements in A
    MERGESORT($A[0..n-1], X[0..n-1]$)
    return $X[0..n-1]$

function MERGESORT($A[0..n-1], X[0..n-1]$) ▷ A modified mergesort algorithm, which sorts elements in descending order, and keeps track of the index value of each element
    if $n > 1$ **then**
        for $i \leftarrow 0$ **to** $\lfloor n/2 \rfloor - 1$ **do** ▷ Copy left half of A to B
            $B[i] \leftarrow A[i]$
            $Y[i] \leftarrow X[i]$
        for $i \leftarrow 0$ **to** $\lceil n/2 \rceil - 1$ **do** ▷ Copy right half of A to C
            $C[i] \leftarrow A[\lfloor n/2 \rfloor + i]$
            $Z[i] \leftarrow X[\lfloor n/2 \rfloor + i]$
        MERGESORT($B[0..\lfloor n/2 \rfloor - 1], Y[0..\lfloor n/2 \rfloor - 1]$)
        MERGESORT($C[0..\lceil n/2 \rceil - 1], Z[0..\lceil n/2 \rceil - 1]$)
        MERGE($B[0..\lfloor n/2 \rfloor - 1], C[0..\lceil n/2 \rceil - 1], A[0..n-1], Y[0..\lfloor n/2 \rfloor - 1], Z[0..\lceil n/2 \rceil - 1], X[0..n-1]$)

function MERGE($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$, $Y[0..p-1]$, $Z[0..q-1]$, $X[0..p+q-1]$)
    $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
    while $i < p$ and $j < q$ **do**
        if $B[i] \geq C[j]$ **then**
            $A[k] \leftarrow B[i]$
            $X[k] \leftarrow Y[i]$
            $i \leftarrow i + 1$
        else
            $A[k] \leftarrow C[j]$
            $X[k] \leftarrow Z[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$ **then**
        for $m \leftarrow 0$ **to** $q - j - 1$ **do**
            $A[k + m] \leftarrow C[j + m]$
            $X[k + m] \leftarrow Z[j + m]$
    else
        for $m \leftarrow 0$ **to** $p - i - 1$ **do**
            $A[k + m] \leftarrow B[i + m]$
            $X[k + m] \leftarrow Y[i + m]$