ESO207A: Data Structures and Algorithms

End-semester exam

Max marks: 120
Time: 180 mins. 17-July-2017

1. *Answer all 7 questions. Questions 1 to 3 are from module 3 and questions 4 to 7 are from module 4. Each module is worth 60 marks. The question paper has 4 pages.*
2. **Answer all parts of a question together. Do not scatter them across the answer script.**
3. **You can refer only to your own handwritten class notes.** *You cannot refer to any other paper material or use any elctronic gadget during the exam. In particular please ensure that your mobile phone/smart watch/tablet/etc. is not with you.*

1. This question has 5 short answer questions. Please be brief and concrete in your answers.

   (a) What is the key difference between a problem that can be solved efficiently by the divide-and-conquer strategy and one that can be solved efficiently by dynamic programming and what is the consequence of this difference for dynamic programming solutions?

   Answer:
   Dynamic programming has overlapping sub-problems while Divide-and-conquer does not. This means if we adopt the naive approach and recursively solve all sub-problems in a dynamic programming solution the same sub-problem will be solved repeatedly and often this leads to exponential time complexity. So, either we have to solve the problem iteratively bottom up or cache the solutions to sub-problems (memoization) and avoid resolving them.

   (b) Most problems that we solved using dynamic programming were optimization problems. Give one example of a problem that can be solved efficiently by dynamic programming which is not an optimization problem and which takes exponential time if solved by a simple recursive algorithm. Briefly say why the simple recursive algorithm has exponential time complexity.

   Answer:
   Calculating the n^{th} Fibonacci number is an example of a non-optimization problem that has an efficient dynamic programming solution. Using a naive implementation of the recursive definition has exponential time complexity since we end up repeatedly solving the same sub-problem multiple times. The unfolded recursion is a binary tree where the height of the tree is $O(n)$ thus giving exponential complexity.

   (c) We solved the static optimal binary search tree problem by dynamic programming in class and as part of an assignment. What is the *dynamic* version of the optimal binary search tree problem and how can we use the static solution in a dynamic setting?

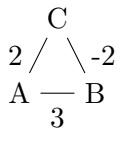
   Answer:
   The dynamic version of the problem allows inserts and deletes in the binary search tree. So, the tree is always changing and will not remain optimal. Also, the probability distribution

of the keys - both hits and misses - will change. One way to use the static algorithm is to reoptimize the tree periodically. Rebuilding the tree after every insert/delete will be very expensive. So, the tree can be rebuilt either after a certain number of inserts and deletes or when the expected value of the number of comparisons has changed enough (using a threshold).

(d) Supposing we use Dijkstra's greedy, single source shortest path algorithm on an undirected graph. What constraint must we have for the algorithm to work and why?

Answer:

Dijkstra's greedy algorithm will not work with negative weight edges. Being a greedy algorithm once a node is added to a path we cannot undo it. If we have negative weight edges then the greedy algorithm cannot work. Consider the following simple counter example with source A.



The shortest path to C from A is of weight 2 but if we allow negative edges we get a path with weight 1 via B undoing choice of A which is not allowed.

(e) Consider the graph in figure 1. If we start with node 10 in V_T as the starting node and use Prim's algorithm to construct the minimum spanning tree give the order in which nodes enter V_T. Also, give the minimum total weight.

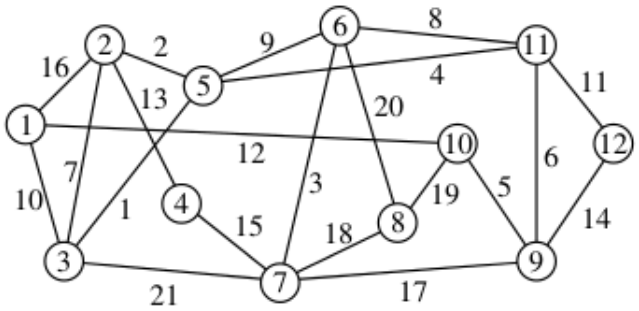


Figure 1: Graph for question 1(e).

Answer:

The order is 10, 9, 11, 5, 3, 2, 6, 7, 1, 12, 4, 8. Total weight 81.

[3×5=15]

2. (a) Recall the first coin problem where given a sequence of coins $c_1, \ldots, c_n$ of varying denominations (repetitions are possible) we have to pick coins such that the total value of the coins picked is a maximum with the constraint that if c_i is picked then c_{i-1} and c_{i+1}, when they exist, cannot be picked. One simple minded strategy is to add up the coins at odd positions and those at even positions and pick the sequence that has the higher total value. Give a counter example to show that this strategy will not work.

Answer:
Consider the sequence 1, 3, 1, 4, 5. The best pick is 3, 5 for a total of 8. All other compatible picks give a total of 7 or less.

(b) A string processing application that you are programming requires you to break-up a string into multiple parts. You are forced to use an existing library that has a string division function that divides a string into two parts at a given location. However, you find that the library function copies the string to be broken up before doing the division which takes time proportional to the length of the string. So, the order in which you divide the string changes the amount of time it takes to break-up the string.

For example, suppose the original string length was 25 and the breaks had to happen after 4 characters, 8 characters and 10 characters in the original string. Then if you choose the ordering 4, 8, 10 for division then the time taken is proportional to $25 + 21 + 17 = 63$. But if you choose the ordering 10, 8, 4 the time taken is proportional to $25 + 10 + 8 = 43$.

You have to devise a dynamic programming algorithm that given a string of length n and a sequence of m breaks finds the minimum time taken for the breaks. The breaks $n_1, \ldots, n_m$ are specified as the number of characters from the left end of the original string after which there should be a break.

i. First explain the basic idea of your algorithm then write the algorithm in pseudo code. (Please do not write C or any other programming language code but pseudo code. This means you should not write low level details.).

Answer:
We can write the following naive recurrence for the division problem.
First note that if $m = 0$ then there is no break so time is 0. Similarly, if there is just 1 break point the time taken is always l irrespective of where the break point is located. These are our base cases.
For the recursive case let b be the list of m break points in the original string of length l and assume the break is at break point b_i.
Assume we have the following functions:
$bleft(b, i)$: gives the list of break points to the left of the ith break point that is break points from 1 to $i - 1$.
$bright(b, i)$: gives the break points to the right of the ith break point that is from $i + 1$ to m. Note that $bright(,)$ will have to return suitably modified indices since it will now be in a string of size $l - b_i$.
Let function $f(l, b)$ give the minimum time required for breaking a string of length l at break points b. The we have:

$$f(l, b) = \begin{cases} 0, & \text{if } m \text{ is } 0 \\ l, & \text{if } m \text{ is } 1 \\ \min\limits_{i \in 1..m} \left(f(b_i, bleft(b, i) + f(l - b_i, bright(b, i)) + l, & m > 1 \end{cases}$$

The above recurrence breaks a string into two parts at every possible break point, adds the minimum time for each part and adds l to the result. And this happens recursively for each string. So, if we have a large number of breaks, say $O(l)$ breaks, we will end up

having a large number of overlapping computations. Actually, it looks at all possible permutations of break points at each stage to get the order giving the minimum time. We use memoization for a DP solution. We let MEMO be a $l \times l$ matrix where $MEMO[i][j]$ gives the minimum time taken for the substring between characaters i to j (both included). Changing the focus from breaks to characters in the string is the key to applying dynamic programming. We can define the new recurrence as follows where $f(i,j)$ gives the minimum time for a substring from i to j:

$$f(i,j) = \begin{cases} (j-i+1) + f(i,b) + f(b+1,j) & \text{if a break point } b \text{ exists between } i \text{ and } j \\ 0 & \text{if there is no break between } i,\ j \text{ - base case} \end{cases}$$

The answer we want is: $f(1,l)$. The algorithm looks as follows:

```
Initialize l × l array MEMO to ∞.
For all i, j such that j > i and there is no break between i and j set
MEMO[i][j] = 0.
This is a quick way to initialize large parts of the array.

function f(i,j) {
   if (f(i,j) ≠ ∞) return f(i,j)
   else {
     tmp = ∞
     for each b ∈ brks {
       if (b between i and j)
         tmp = min(tmp, j − i + 1 + f(i,b) + f(b+1,j))
       MEMO[i][j] = tmp //remember in MEMO
     }
   }
   return tmp
}
```

Note that each sub-problem is computed only once.

ii. What is the big-O time complexity of the algorithm above that you have written in pseudo code?

Answer:
The algorithm above fills up the MEMO array which is order $O(l^2)$ and it goes over each break. The number of breaks in the worst case can be $O(l)$ so complexity is $O(l^3)$.

[3,14,3=20]

3. (a) In a college with just one air conditioned room there are many requests to schedule activities in that room. Assume there are n requests numbered $1, 2, \ldots, n$. Each activity has a start time $s(i)$ and finish time $f(i)$. At any point in time the room can be allocated only to one activity. So, if some set of activities have overlapping time intervals then only one of them can get that room. The problem is to find a subset of maximum size that has non-overlapping activities so that the maximum number of activities can use the room during the day. Figure 2 shows a

set of 11 activities and a subset of size 4 at the bottom that is the maximum sized subset of non-overlapping activities that can be scheduled. This is called the interval scheduling problem.

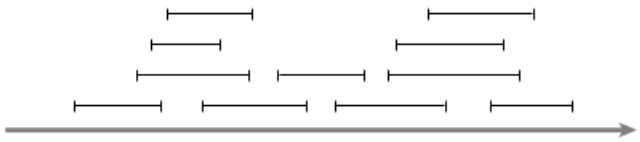


Figure 2: An example of interval scheduling for 11 intervals

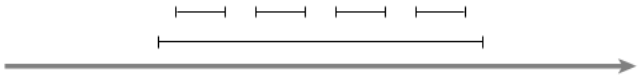
As the resident algorithmicist you decide to use the greedy strategy to find this subset. You come up with the following four different possibilities to be greedy. Let S be the set of activities to be scheduled and A be the subset being constructed which is empty at the start.

1. Repeat the following until S is empty.
   `Choose activity` $i \in S$ `such that` $s(i) < s(j)$, $\forall j \in S, j \neq i$.
   `Add` i `to` A `and remove all activities from` S `that overlap with` i.
   Earliest starting first strategy.
2. Repeat the following until S is empty.
   `Choose activity` $i \in S$ `such that` $f(i) - s(i)$ `is minimum`, $\forall i \in S$.
   `Add` i `to` A `and remove all activities from` S `that overlap with` i.
   Shortest interval first strategy.
3. Repeat the following until S is empty.
   `Choose activity` $i \in S$ `such that it has the smallest number of overlaps with other activities in` S.
   `Add` i `to` A `and remove all activities from` S `that overlap with` i.
   Least number of overlap conflicts first strategy.
4. Repeat the following until S is empty.
   `Choose activity` $i \in S$ `such that` $f(i) < f(j)$, $\forall j \in S, j \neq i$.
   `Add` i `to` A `and remove all activities from` S `that overlap with` i.
   Earliest finishing time first strategy.

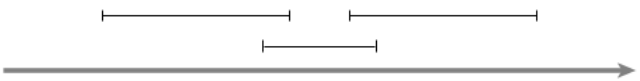
Answer the following with respect to the description above.

i. Find counter examples to argue that the first three greedy strategies 1 to 3 may find subsets that are not optimal. Show these counter examples as interval diagrams as in figure 2. (The first two are easy the third is harder.)

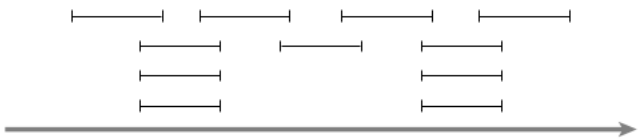
Answer:



Above: Counter example for strategy 1.



Above: Counter example for strategy 2.



Above: Counter example for strategy 3.
The trick here is to start with a solution and add activities that overlap so that parts of right solution are removed leading to a sub-optimal solution.

ii. Argue that the last strategy (earliest finish time first (EFTF)) will indeed always find the optimal subset.
(Hint: Compare with a possible optimal schedule O and argue they must have the same size.)

Answer:
Let A be the schedule returned by the EFTF algorithm containing intervals $i_1, \ldots, i_k$ in order. Let O be an optimal schedule that contains the intervals $j_1, \ldots, j_m$ in order. We will argue that $k = m$ thereby showing that A is as large as O. In general they need not be identical since many optimal schedules are possible. A is clearly a feasible schedule since the algorithm ensures that no overlapping intervals are in A. O is a feasible schedule by assumption.
Claim 1: For all $1 \leq r \leq k$, $f(i_r) \leq f(j_r)$.

Proof. This is clearly true for $r = 1$ since $f(i_1)$ is the earliest finish time. Assume it is true for $r-1$, $f(i_{r-1} \leq f(j_{r-1}))$. Now for r if $f(j_r)$ is earlier than $f(i_r)$ then the EFTF algorithm since it always chooses the earliest finishing activity always had the option of choosing j_r rather than i_r so $i_r \leq j_r$ for all r. Note that j_r must be present in S when the EFTF algorithm chooses i_r because $f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$. □

Claim 2: $k = m$

Proof. Let $m > k$. The EFTF algorithm stops when S is empty. Now $f(i_k) \leq f(j_k)$ (by claim 1). Since $m > k$ then there is an interval j_{k+1} such that $f(i_k) \leq s(j_{k+1})$ so there exists an interval that does not overlap with any interval in A so this interval must be in S so S cannot be empty a contradiction. So, $m = k$. □

(b) Given an arbitrary connected graph $G = (V, E)$ with edge weights in $\mathbb{R}^+$ will the minimum cost edge in G (assume there is only one such edge in G) always be present in every MST of G? If your answer is YES then give a short justification, if it is NO then give a counter example.

Answer:
Yes - the minimum weight edge will always be in the MST.
Let the minimum weight edge be e and not be in the MST. Add e to the MST. This will produce a cycle. Since e is the unique edge with the lowest weight all other edges will have weights larger than e. Removing one of these other edges gives us an a tree with a weight less than the earlier MST which is a contradiction. So, e must have been in the MST.

[(2,2,5),12,4=25]