ESO207A: Data Structures and Algorithms
Quiz

Max marks: 50
Time: 60 mins. 1-July-2017

1. *Answer both questions. Keep answers brief and to the point.*
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 quiz. In particular please ensure that your mobile phone/smart watch is not with you.*

1. We are given a sequence S of n numbers. For example: $S = (2, 4, 3, 5, 1, 7, 6, 9, 8)$ where $n = 9$. We wish to find a longest increasing subsequence in S. In the example the subsequence $(2, 4, 5, 7, 8)$ is an increasing subsequence of length 5 and so is $(2, 3, 5, 7, 9)$. Actually, it turns out that for the example sequence S the longest increasing subsequence is of length 5.

   More formally, a *longest increasing sub-sequence* is the longest (that is maximum m) subsequence $L = (s_{i_1}, s_{i_2}, \ldots, s_{i_m})$ of the sequence $S = (s_1, \ldots, s_n)$ such that $i_1 < i_2 < \ldots < i_m$ and $s_{i_1} < s_{i_2} < \ldots < s_{i_{m-1}} < s_{i_m}$.

   Given the above answer the questions below. Always, explain the meanings of any symbols that you introduce or use.

   (a) Given that the maximum increasing subsequence length for the example set S is 5. Find all such sequences of S.

   Answer:
   There are 8 such sequences:
   2 3 5 6 8
   2 3 5 6 9
   2 3 5 7 8
   2 3 5 7 9
   2 4 5 6 8
   2 4 5 6 9
   2 4 5 7 8
   2 4 5 7 9

   (b) Write a recurrence which will allow us to use dynamic programming to find the length of the longest increasing subsequence for a sequence of n numbers. Remember to write the base cases.

   Answer:
   Let $l(i)$ be the length of the longest increasing sub-sequence (LIS) of the sequence $s_1, \ldots, s_i$. Then either s_i is part of the LIS or not. It is part of the sequence when $s_i > s_{i-1}$ otherwise it is not. This gives the following simple recurrence:

$$l(i) = \begin{cases} l(i-1), & s_{i-1} \geq s_i \\ l(i-1)+1, & s_{i-1} < s_i \end{cases}$$

The base cases are: $l(0) = 0$ and $l(1) = 1$. And the LIS is in $l(n)$.

The following code fragment shows a bottom-up implementation. This is for clarity and not part of the required answer.

```
l[0]=0; l[1]=1;
for (i=2;i<=n;i++) {
    if(s[i-1]>=s[i]) l[i]=l[i-1];
    else l[i]=l[i-1]+1;
}
printf("%d\n",l[n]);
```

(c) If we also want an actual longest increasing subsequence of S and not just its length briefly say what extra information must be stored and how it will be used to get the actual subsequence.

Answer:

The actual sequence values can be stored in a 1-dimensional array (say $a[]$) while backtracking over array $l[]$. Observe that if $l[i] > l[i-1]$ then $s[i]$ will be part of the sequence otherwise not. We start populating $a[]$ from $l[n]$ backwards to 1. $l[i] > l[i-1]$ implies $a[l[i]] = s[i]$.

The following code fragment shows the details. But it is for clarity and not part of the required answer.

```
for(i=n;i>0;i--)
        if (l[i]>l[i-1]) a[l[i]]=s[i];
```

[6,(9,3),7=25]

2. In a company there are n persons $p_1, \ldots, p_n$ and n jobs or tasks $t_1, \ldots, t_n$. We assume that each person can do any of the n tasks and each task is assigned to exactly one person. The cost to the company when p_i does task t_j is c_{ij}. Ideally, the company would like to incur minimum total cost by doing the best possible assignment of person to task for each person and each task.

   This problem is called the assignment problem.

   Given the above answer the questions below. Always, explain the meaning of any symbols that you introduce or use.

   (a) State the assignment problem formally as an optimization problem.

Answer:

Let $f : \{p1, \ldots, p_n\} \rightarrow \{t_1, \ldots, t_n\}$ where f is a bijective (i.e. 1-1, onto) mapping between the domain and range of f - called an assignment. Also, let $\mathcal{A}$ be the set of all such possible mappings - that is $\mathcal{A}$ is the set of all possible assignments. Then the optimization problem is:

$$\operatorname*{argmin}_{f \in \mathcal{A}} \left(\sum_{i=1}^{n} c_{p_i, f(p_i)}\right)$$

The above equation says find the assignment that minimizes the total cost.

There are other ways to express the same thing.

(b) A naive algorithm will consider all possible ways in which persons can be assigned to jobs with one job assigned to exactly one person. and pick the one with minimum cost. How many pairings are possible and is it a feasible algorithm?

Answer:
There are $n!$ possible ways in which n persons can be assigned to n tasks. So, a naive brute force algorithm which evaluates every possible assignment and picks the minimum is clearly infeasible. Note that $n! > (n/2)^n$ which increases even faster than an exponential like 2^n.

(c) Devise a greedy algorithm for the assignment problem where the input is the matrix c_{ij} and the output is an assignment of a task to each person such that the total cost is a minimum. Write the pseudo-code and give its time complexity (big O).

Answer:
There are three possible greedy algorithms. In the first we go over the set of tasks one at a time and greedily select the person who does the task at least cost. The second is to go over the set of persons one by one and greedily assign the least cost task to that person. In the third we search array c for the smallest entry, store the association then delete the corresponding row and column from array c then repeat until assignment is done. The pseudo code for the first version is given below:

```
/*Input: c_ij: an n × n array with tasks as rows and persons as columns.*/
/*Output: array a[] of length n giving the index number of person assigned
 to task i.*/
for (i ← 1 to n) {
    a[i] ← argmin_{j∈1..n} c[i, j];//assigns column no. with minimum c[i, j].
    for (k ← 1 to n)
        c[k, a[i]] ← ∞;//person cannot be chosen again - ensures 1-1.
}
return a;
```

The time complexity is $O(n^2)$.
The pseudo code for the third version (which is definitely a better greedy algorithm) is also given below:

```
/*Input: c_ij: an n × n array with tasks as rows and persons as columns.*/
/*Output: array a[] of length n giving the index number of person assigned
 to task i.*/
for (k ← 1 to n) {
    Find smallest element in array c, say c[i, j];
    a[i] ← j;
    Replace i^th row and j^th column in array c by ∞;//ensure 1-1 assignment
}
return a;
```

The complexity of finding minimum in the c array is $O(n^2)$ and the for loop makes the overall complexity $O(n^3)$. This can be improved by using a heap for finding the minimum element. But there are $O(n)$ changes in every round where the row and column values are

replaced so we can do no better than $O(n^2)$.

Any of the above answers are acceptable.

(d) Using a simple example show that the greedy approach does not always yield an optimal solution.

Answer:
It is rather obvious that the greedy approach will not give an optimal assignment for the first two versions of the greedy algorithm. The following simple 2 task, 2 person cost matrix gives a non-optimal greedy assignment where for each task we find the lowest cost person.

	p_1	p_2
t_1	2	3
t_2	1	5

Greedy assignment: $(t_1, p_1), (t_2, p_2)$ with cost 7.
Optimal assignment: $(t_1, p_2), (t_2, p_1)$ with cost 4.

For the third version of the greedy algorithm consider the following c matrix.

	p_1	p_2	p_3
t_1	1	3	2
t_2	2	1	5
t_3	4	2	6

Greedy assignment: $(t_1, p_1), (t_2, p_2), (t_3, p_3)$ with cost 8.
Optimal assignment: $(t_1, p_3), (t_2, p_2), (t_3, p_1)$ with cost 7.

Any of the above answers is acceptable.

[5,5,(7,3),5=25]