ESO207: Theoretical Assignment-1, Feb 2019

Q1. Given an array $P[0 : n-1]$ giving the daily price quotes for a stock for n consecutive days. The span of a stocks price on a given day i is the maximum number of consecutive days before i that the stocks price is less than or equal to its price on day i. This includes i-th day, so span will be at least 1. For example if $P = [50, 45, 35, 40, 60, 50, 55]$, then span is $S = [1, 1, 1, 2, 5, 1, 2]$.

(a) Give the "natural" algorithm (directly from the definition) to compute the array N. Determine the time and space complexity.

(b) Using a suitable data structure design a new algorithm for the problem which is more efficient. Determine its time and space complexity.

Solution of Q1

(a)

```
for i := 0 to n − 1 do
    p := i;
    count := 0;
    while P ≥ 0 AND P[p] ≤ P[i] do
        count := count + 1;
        p := p − 1;
    end
    S[i] := count;
end
return S;
```

For each i total number of While-loop iterations can be at most $i+1$. In each pass of the loop there is a constant amount of time required. So total time $T(n) \leq const. \sum_{i=0}^{n-1}(i-1) = const.n.(n+1)/2 = O(n^2)$.

(b) If we carefully look at this problem, we notice that if $A[i] > A[i+j]$, then no index less than i will contribute to $S[i+j]$. Thus when wew scan the values in the array P and reach i-th index, then we do not need to store any previous value which is less than $P[i]$. We will thus use a stack to solve this problem. See Algorithm 2.

Analysis: Suppose in the i-th iteration of the For-loop total number of PoP operations is k_i, then the cost of this iteration is $c_1 + c_2.k_i$. So the total cost of the algorithm is $O(c_1.n + c_2 \sum_i k_i)$. Observe that each item $(P[i, i])$ is pushed into the stack only once. So total number of Pop operations cannot be more than n. Hence $\sum_i k_i \leq n$. So the time complexity of the algorithm is $O(c_1.n + c_2.n) = O(n)$.

Q5. Find the time complexity of the following algorithms. Show the analysis.

```
Initialize an empty stack T;
for i := 0 to n − 1 do
    span := 1;
    if T ≠ empty then
        (a, j) := Top(T);
        while T ≠ empty AND a ≤ P[i] do
            Pop(T);
            span := i − j + 1;
            (a, j) := Top(T);
        end
    end
    S[i] := span;
    Push(T, (P[i], i));
end
```

Algorithm 1: Improved solution of Q1

(a)

```
Input: Array A and integers r, c, n
if r ≥ n OR c ≥ n then
    return 1;
else
    if r = n AND c = n then
        return 0;
    else
        return A[r, c] + min{Rec(A, r + 1, c), Rec(A, r, c + 1)};
    end
end
```

Algorithm 2: $Rec(A, r, c)$

Hint: Find a recurrence for the number of times $Rec(r + i, c + j)$ will be invoked.
(b) Suppose $Store$ is a global array containing -1 at all locations initially.

Solution of Q5

(a) In this algorithm any call $Rec(A, r, c)$ terminates in $O(1)$ time if $r \geq n$ or $c \geq n$. So the recurrence relation for the time complexity of this problem is

$$T(r, c) = \alpha + T(r + 1, c) + T(r, c + 1)$$

```
Input: Array A containing positive entries at all locations and integers r, c, n
if (r > n OR c > n) then
|   return 1;
else
|   if (r = n AND c = n) then
|   |   return 0;
|   else
|   |   if Store[r, c] = −1 then
|   |   |   Store[r,c] := A[r, c] + min{Magic(A, r + 1, c), Magic(A, r, c + 1)} ;
|   |   |   return Store[i, j];
|   |   end
|   end
end
```

Algorithm 3: $Magic(A, r, c)$

where $T(r, c) \le \beta$ if $r \ge n$ or $c \ge n$ where β is a constant.

To determine $T(r, c)$ for general r and c we will expand this recurrence relation. So

$T(n-1, n-1) = \alpha + T(n, n-1) + T(n-1, n) = \alpha + 2\beta$.

$T(n-2, n-1) = \alpha + T(n-1, n-1) + T(n-2, n) = 2\alpha + 3\beta$. Similarly $(n-1, n-2) = 2\alpha + 3\beta$.

$T(n-3, n-1) = \alpha + T(n-2, n-1) + T(n-3, n) = 3\alpha + 4\beta$ and $T(n-1, n-3) = 3\alpha + 4\beta$

So in general the pattern for $T(n-i, n-1) = i.\alpha + (i+1)\beta$. To verify this use induction $T(n-i, n-1) = \alpha + T(n-i+1, n-1) + T(n-i, n) = \alpha + \beta + T(n-i+1, n-1) = i.\alpha + (i+1)\beta$. Hence the claim is verified.

Now consider the remaining arguments. $T(n-2, n-2) = \alpha + T(n-1, n-2) + T(n-2, n-1) = 5\alpha + 6\beta$.

$T(n-3, n-2) == \alpha + T(n-2, n-2) + T(n-3, n-1) = 9\alpha + 10\beta$. Similarly we see that $T(n-2, n-3) = 9\alpha + 10\beta$.

Note that $\binom{4}{2} = 6$ and $\binom{5}{2} = 10$. So we see a pattern $T(n-i, n-j) = \binom{i+j}{i}.(\alpha + \beta) - \alpha$ for all $n - i < n - 1$ and $n - j < n - 1$. Again we can verify it by using induction. $T(n-i, n-j) = \alpha + (\alpha + \beta)(\binom{i+j-1}{i-1} + \binom{i+j}{j-1}) - 2\alpha = (\alpha + \beta).\binom{i+j}{i} - \alpha$. Observe that this expression also works for $T(n-i, n-j)$ even for $i = 1$ or $j = 1$.

So the time complexity for $T(n-i, n-j)$ is $O(\binom{i+j}{i})$.

(b) In this case when we make a call $magic(n-i, n-j)$ nested calls make the call $magic(n-i', n-j')$ for all $i' \ge i$ and $j' \ge j$ as it did in *Rec* case. So the total number of different calls are $i.j$. But the impact of *Store* is that no call $magic(n-i', n-j')$

is ever executed more than once.

In this case we will estimate a bound for the time by counting the total number of times *Store* value is checked and the total number of times some *store* value is assigned.

So we see that $magic(i + j)$ is invoked once by $magic(i - 1, j)$ and once by $magic(i, j - 1)$. But only the first invocation is carried out. So we make two checks of *Store* for each $(n - i', n - j')$ and we compute $Store(n - i, n - j')$ only once. So the total time complexity is $O(i.j)$.