CS1356 Introduction to Information Engineering

Homework 11

Let A[1..N] be an array of N elements. We define a Tournament of A as follows, in which the Swap(a,b) function exchanges the values of a and b.

```
Procedure Tournament(A[1..N])
    1. For i = 1 to N-1
    2.     If (A[i]>A[i+1]) then Swap(A[i], A[i+1])
```

1. Write a pseudo code for the Swap function

```
Procedure Swap(A , B)
    1. Temp = A
    2. A = B
    3. B = Temp
```

2. Prove the largest element of A will be placed in A[N] after a Tournament.

Step 1:

For N = 2 ,

if A[1] > A[2] , it would exchange the values of A[1] and A[2] –(1)

if A[1] < A[2] , it would do nothing. –(2)

for (1)& (2), the largest value of A would be placed in A[2]—Correct

Step 2:

We assume that N = m is correct .

Step 3:

For N = m+1 ,

By step2, we know that in the largest value of A[N-1] would be placed in A[N-1].(Because N = m would be correct).The procedure would compare A[N-1] and A[N] in the last. So if A[N-1] > A[N], it would exchange the values of A[N-1] and A[N] , the large largest value of A would be placed in A[N].—(a) If A[N-1] < A[N] , it would do nothing , but still the large largest value of A would be placed in A[N].—(b)

For (a)&(b) , it is proved N = m+1 also correct.

In step1~3 , we used **Inductive** to prove "*in which N = m+1 , the large largest value of A would be placed in A[N]*".

Q.E.D.

3. What is the number of comparisons made in one Tournament?

N-1

4. What is the time complexity of the Tournament function?
   In the big-theta notation.

∵This **Tournament** will do N-1 comparisons, f(n) = n-1 .

∴ θ (f(n)) = θ (n - 1) = n

The bubble sort algorithm uses the Tournament function to sort the elements of A, as shown below.

```
Procedure BubbleSort(A[1..N])
    1.  While (true)
    2.      Call Tournament(A[1..N])
    3.      If there is no swapping during the Tournament, stop
    4.  End while
```

5. Prove this algorithm is correct.
   (a) Prove this algorithm will stop.
   (b) Prove when the algorithm stops, elements in array A are sorted in the ascending order.
   (a) This algorithm will stop only when "No swapping during the Tournament" , so we need to prove that "No swapping during the Tournament". By Question 2 , we have proved that **"After a Tournament , the largest element of A with N element will be placed in A[N]"**, is also meant that **"after a Tournament , A[N-1] and A[N] wouldn't exchange next Tournament."**(Because A[N] must be larger than or equal to A[N-1]). So next time call Tournament again, we can guarantee that **"After a Tournament , the largest element of A with N-1 element will be placed in A[N-1]"**. According this, when we call Tournament N times, it would be sure that "**After a Tournament , the largest element of A with 1 element will be placed in A[1] and A[1] wouldn't exchange next Tournament."** Q.E.D.
   (b) By Question 5 (a) , we have proved that this algorithm will stop "**Because no swapping during the Tournament**." In other word, **A[N-1] must smaller than A[N] ,A[N-2] must smaller than A[N-1]…and so on**. So we can say array A are sorted in the ascending order.
6. Prove the time complexity.
   (a) Give an example for N=5, in which the data arrangement will cause the worst case?
   (b) In the worst case, how many times does the Tournament function need be called?
   (c) What is the time complexity of BubbleSort (in big-theta notation)?
   (a) 5 , 4 , 3 , 2 , 1
   (b) In the worst case N elements, we call N-1 times Tournament because each time we call **Tournament once only ensure one element will be placed in the correct position, but in the N-1's time we ensure the element in A[2] is the right one, it is also guaranteed the element in A[1] is correct.**
   (c) In the worst case , we should call Tournament n times .So $f(n) = (n-1)*(n-1) = n^2 - 2n + 1$

   $\therefore \theta(f(n)) = \theta(n^2 - 2n + 1) = n^2$

7. Can you use the property proven in 1. to speedup the BubbleSort algorithm?
   (a) Write a pseudo code that implements this idea.
   (b) Write a recursive version of pseudo code to implement this idea.

(c) Comparing (a) or (b) with the original algorithm, for the same data input, is the number of Tournaments changed?

(d) Comparing (a) or (b) with the original algorithm, for the same data input, is the number of Swaps changed?

(e) What is the time complexity of this new algorithm?

For speedup this algorithm, we can do is that "decrease the number of comparison." Because every time call Tournament , it ensures that "**one element will be placed in the correct position." (by 5)** So when once a Tournament is done, we can decrease the array size to decrease the comparison which the element placed in the correct position.

(a)

```
Procedure BubbleSort(A[1..N])
    1. For j = N to 1
    2.     Call Tournament(A[1..j])
    3.     If there is no swapping during the Tournament, stop
    4. End while
```

(b)

```
Procedure BubbleSort(A[1..N])
    If(N = 1)
        Then (Report that the sorting has been done.)
    Else
        Call Tournament(A[1..N])
        Call BubbleSort(A[1...N-1])
    End If
```

(c) Compare (a) with the original algorithm for the same data input, we don't change the number of Tournaments. ((b) is , too.)

(d) Compare (a) with the original algorithm for the same data input, the number of swaps is the same ((b) is, too.)

*** *Those two algorithms only reduce the number of comparisons.*

(e) ∵This algorithm will do N times **Tournament**, and each time its comparison

decrease 1, so $f(n) = n + (n-1) + (n-2) + \cdots + 1 = n*(n+1)/2 = (n^2+n)/2$

∴ θ (f(n)) = θ ($(n^2+n)/2$) = n^2

The BinarySearch algorithm that finds the *TargetEntry* of a **sorted List** of N elements is sketched as follows. (From textbook 5.5)

```
procedure Search (List, TargetValue)
if (List empty)
  then
     (Report that the search failed.)
  else
     [Select the "middle" entry in List to be the TestEntry;
      Execute the block of instructions below that is
          associated with the appropriate case.
             case 1: TargetValue = TestEntry
                      (Report that the search succeeded.)
             case 2: TargetValue < TestEntry
                      (Apply the procedure Search to see if TargetValue
                            is in the portion of the List preceding TestEntry,
                            and report the result of that search.)
             case 3: TargetValue > TestEntry
                     (Apply the procedure Search to see if TargetValue
                           is in the portion of List following TestEntry,
                           and report the result of that search.)
     ] end if
```

8. Prove the correctness of this algorithm
   (a) The algorithm will stop
   (b) If the TargetValue is in the List, then the algorithm will find its index. If the TargetValue is not in the List, the algorithm will report failed.

   Note that: This list is a sorted list, so the latter element is larger than the former everywhere in list.

   (a) This algorithm will stop when "**List empty(The search failed)**" or "**the TargetValue is found in List(The search succeeded)**", so we need to prove these two conditions. Every time when the procedure *Search* is called, we have three cases: "TargetValue is smaller than Middle", "TargetValue is equal to Middle", and "TargetValue is larger than Middle". When the TargetValue is equal to Middle, this procedure will report the search succeeded and the program will stop because **the TargetValue is found in List(The search succeeded)**. If the other two cases happen, we would do procedure *Search* again, but the input size is less than half of the original list. So each time when we call procedure *Search*, the input size would be halved. When the input size equals to 0, the algorithm will stop too.

   (b) (Induction)
   For N=1, the only element is also the 'Middle'. If TargetValue equals to it, the algorithm will correctly report the answer. If TargetValue does not equal to it, the algorithm will call itself again, and returns 'not found', since the list is empty.

   Assume for N<=k, the algorithm can correctly report the search results.
   For N=k+1, if TargetValue is in the list and equals to Middle, the algorithm will report found. If TargetValue is smaller than Middle, the algorithm will search the list from the first element to Middle. Since the list is sorted, elements in the back-half won't contain TargetValue. Therefore, the result of searching the

front-half will be the result of searching entire list. Since we had assumed that for N<=k, the algorithm can correctly report the answer, the algorithm will report the correct answer for N=k+1.

9. What is the time complexity of this algorithm in the big-theta notation? Justify your answer.

(a) Why does it choose 'middle' entry as the TestEntry?

Supposed that the input size in List is N, each time we call procedure *Search* the number of comparison is 1. But next time we call, the input size will be halved. But this will stop when input size = 0.

Supposed we call the procedure *Search* **k** times , so $\mathbf{2^k = N}$--(2*2*2..*2(k times) = N)

→ $\mathbf{k = \log_2 N = \log N}$

So f(n) = 1(N=n) + 1(N=n/2) + 1(N=n/4) +⋯+1 (N = 0) = 1 * logN = logN

$\mathbf{\therefore \theta (f(n)) = \theta (\log N) = \log N}$

(a) Every time we choose Middle as the TestEntry , if "TargetValue is larger than Middle" or "TargetValue is smaller than Middle", it guarantees that **"at least there is half elements in list is larger or smaller than TargetValue".(Because list is sorted , if the Middle is smaller than TargetValue , the element which position in front of Middle must smaller than TargetValue. And so on.)** We can remove these elements because it is impossible that **the TargetValue will be in the partition.** So choosing the middle will guarantee every time we do *Search*, the half of input size can be removed.