CS325- Homework -2

Q1.

a. $T(n) = T(n-2) + n$
$T(n-1) = T(n-3) + n-1$
$T(n-2) = T(n-4) + n-2$
$T(n-3) = T(n-5)+n-3$
$T(n-4) = T(n-6) + n-4$
Substitution
Step 1 = $T(n) = T(n-2) + n$
$= (T(n-4)+n-2) + n$
Step-2 = $T(n-4) + n + n -2$
$= (T(n-6)+n-4) + n + n - 2$
Step-3 = $T(n-6) + n+n+n - 6$
$= (T(n-8)+n-6) + n + n + n -6$
Step -4 = $T(n-8) + n + n + n + n - 12$

$= T(n-2k) + k*n - k(k-1)$
Terminates when $n-2k = 0$
$K = n/2$
$T(0) + n^2/2 - (n - n/2)$

$= \Theta(n^2)$
If Terminated at $n - 2k = 1$
$K = (n-1)/2$
$T(1) + (n^2 - n)/2 - ((n-1)(n-2))/2$
$= \Theta(n^2)$

b. $T(n) = T(n-1) + 3$
$T(n-1) = T(n-2) + 3$
$T(n-2) = T(n-3) + 3$

Substitution

Step 1 -> T(n-1) + 3
Step 2 -> (T(n-2) + 3) + 3
Step 3 -> (T(n-3) +3) +3 +3
Step 4 -> (T(n-4) +3) + 3 +3 +3
T(n – k) + 3k
Terminates at n-k = 0
K = n
= T(0) + 3n
= Θ (n)

Terminates at n-k =1
K = n -1
= T(1) + 3(n -1)
= Θ (n)

c. T(n) = 2T(n/4) + n

Masters Theorem
A = 2
B = 4
F(n) = n
$n^{log\_4\_2} = n^{1/2}$
$f(n) > n^{1/2}$
Case 3
Regularity -> $n^{1/2} <= c.n$
C=3/4. For n>1
= Θ (n)

d. $T(n) = 4T(n/2) + n^2. \surd n$
Masters Theorem
A = 4
B = 2
$F(n) = n^{2.5}$
$n\ l^{og2\_4} = n^2$
$f(n) > n^2$, for n> 1

case 3

regularity-> $4. (n/2)^{2.5} <= c. n^{2.5}$

$0.707. n^{2.5} <= c. n^{2.5}$

True when c = 3/4

$= \Theta (n^{2.5})$

Q2.

a. 5*T(n/2) + O(n)
b. 2 * T(n-1) + 1
c. $9 * T(n/3) + O(n^2)$

Solve all three

a. 5 * T(n/2) + n

A = 5

B = 2

F(n) = n

$N^{\log2\_5} = n^{2.32}$

$n < n^{2.32}$

Case-1

$= \Theta (n^{2.32})$

b. 2 * T(n-1) + 1

Substitution

2*(2 T(n-2) +1) + 1

4*(2T(n-3) + 1) + 2 +1

8(2T(n-4) + 1) + 4 + 2 + 1

16(2T(n-5) + 1) + 8 +4 +2 + 1 = 32T(n-5) + 16+8+4+2+1

$= 2^k T(n – k) + 2^{k-1}$

Terminate at n – k = 0

K = n

2^n T(0) + 2^{n-1}

= Θ (2^n)

c. 9 * T(n/3) + O(n^2)
Masters Theorem
A = 9
B = 3
F(n) = n^2
N $^{\log 3\_9}$ = n^2
F(n) = n $^{\log b\_a}$
Case 2
= Θ ($n^{\log3\_9}$ log n)
= Θ (n^2 log n)

Algorithm C works better. When n >0
Θ (n^2 log n) < Θ ($n^{2.32}$) < Θ (2^n)

Q3.

T(n) = 4.T[n/2] + cn

Dividing 4 parts of n/2 each

Masters Theorem:

A = 4

B = 2

F(n) = cn, Assume C = 1

$N^{\log\_2\_4}$ = n^2

F(n) <= n^2

So, Case 1

T(n) = Θ ($n^{\log\_2\_4}$)

T(n) = Θ (n^2)

Q4.

```
Assuming Array is sorted.
ternarysearch (Array, value, start, end)
{
Check if array is not empty

Calculate first 1/3rd
mid1 = start + (end-start) / 3

Calculate 2nd 1/3rd
mid2 = start + 2*(end-start) / 3

Check if the border element is the value
If (Array[mid1] == value)
Return 1
Else if (Array[mid2] == value)
Return 1

If not, traverse others
Else if (value < Array[mid1])
Search 1st third.
Return ternarysearch (Array, value, start, mid1-1)
Else if (value > Array[mid2]) {
Search 3rd third.
Return ternarysearch (Array, value, mid2+1, end);
Else
Middle third.
Return ternarysearch (Array, value, mid1, mid2);
}
```

Assume comparisons and calculations take constant time ‘c’

If the element is in first 1/3 rd, then we search only first 1/3rd

Calls a recursion of size (n/3)

T(n) = T(n/3) + C

Solving using Iteration

$T(n) = T(n/9) + c + c$

$= T(n/27) + c + c + c$

$= T(n/3^k) + kc$

Assume $n = 3^k$

$T(1) + c.\log_3 n$

$= \Theta(\log_3 n)$

Binary search is $\Theta(\log_2 n)$

Comparing Binary search Vs Ternary Search

Here, Ternary search looks more faster. But consider the number of comparisons doing at each step. We are doing 2 comparisons in every recursion step, which might increase the time.

And, In Binary search you can eliminate 50% values in one shot. In ternary best case, you can eliminate 66%, But if u consider worst case, you might be just removing 33% and again solving for 66% which is huge for large n.

Q5.

a. Merging two sorted arrays K1 and K2 with n and n elements results in array size of 2n which has O (n1 + n2) time. So again the merged array of size 2n, merge with K3 of size n then, the resultant array will be of size 3n, then merge to K4 and so on…

Merge(n,n)

Merge(2n,n)

Merge(3n,n)

….

Merge((k-1)n + n)

So, Overall Time complexity will be O(2n + 3n +4n + …. + (k-1)n

$= O(n(1+2+3+\ldots k)) = O(n(k(k+1)/2))$

= O (n.k^2)

b. Divide and conquer can be done in different ways. But I am using Heaps as temporary storage to sort my arrays.

1. Create a heap of size k.
2. Put all 1st elements of K Arrays in the Heap.
3. Create an output array of size k*n.
4. Get the minimum element from the heap and put in output array.
5. Replace that empty location with the next element from the same array.
6. And again get minimum, and put in array.
7. Repeat until the elements in that array are over.
8. Once, after all elements in that array are over make that field to NULL or Infinity.
9. Repeat Step 4 to 8 till full heap is having only NULLs or Infinity.

The Time complexity would be:

The step 3 to 8 runs for n*k times, because there are n*k elements.

Each Heapify (Inserting into Heap) takes log k time.

And O(1) for putting in array.

So, T(n) = O(N*k * log k) + O(1)

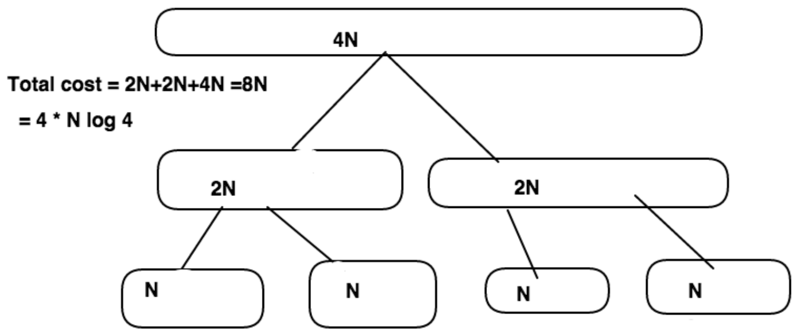
= O(kn log k)

Another Method:

Merge the lists pairwise, then merge the resulting lists pairwise, and so on, until there is just one list.

The pairwise merging requires O(n) time for two lists. At each level we need total O(k * n) times. Number of levels (log k). Therefore total running time for the merging is O(kn log k).

Refer to Tree:



Q6.

a. Time Complexity of Insertion Sort: Θ(n^2), for n elements.
For n/k sublists and each with k elements: Θ(k^2*n/k) = Θ(kn).

b. Time Complexity of merge sort is Θ (n lg n)
Where is n is the sum at each level and lg n is the height.
Here In this case, we traverse the tree till we reach k elements, instead of 1elements, because the array is of length k and the sub array is sorted.
So, The height of tree, till we sort 1 element is Θ (lg n). The height of tree from k elements to 1 elements is again same as merge sort which is Θ (lg k).
Therefore height of tree from n to k is Θ (lg n) – Θ (lg k) = Θ (lg(n/k))
And at each level, sum is n.
So, t(n) = Θ (n lg(n/k)).

c. The total time complexity for this algorithm is merging+ sorting.
T(n) = Θ(kn) + Θ (n lg(n/k))
Standard merge sort : Θ (n lg n).
So, find k, such that Θ(kn) + Θ (n lg(n/k)) == Θ (n lg n).
if k grows faster than log n, then we will get something worse than Θ(n log n) because of the Θ(nk) term. So, k ≤ Θ(lg n).

Lets assume, k = Θ(lg n).

Θ(n lg n + n lg (n/lg n))
= Θ(n lg n + n (lg (n) - lg (lg n))

$= \Theta(n \lg n + n \lg (n) - n \lg (\lg n))$
$= \Theta(2n \lg n - n \lg (\lg n))$
$= \Theta(n \lg n)$.

So, $k <= \Theta(\lg n)$

d. Insertion sort will run faster than merge sort within a range of values.
$O(kn + n \lg(n/k)) <= O(n \lg n)$

It depends on size of n, if lgn is not very large then k should be close to lgn; otherwise, k should be less than lgn such that insertion sort is faster than merge sort for sorting k elements.
We need to select k such that the running time does not exceed $O(n \lg n)$
So, k should be between 1 and lg n.

Q7.

Finding Max and Min of a unsorted array using Recursion:

```
Pair MaxMin(array)
{
  if array_size = 1
    return element as both max and min
  else if arry_size = 2
 {   one comparison to determine max and min
    return that pair
  }
else    /* array_size  > 2 */
{     recur for max and min of left half
    recur for max and min of right half
    one comparison determines true max of the two candidates
    one comparison determines true min of the two candidates
    return the pair of max and min
}
}
```

Pseudo code:

```
Function MAXMIN (A, low, high)
 {
 if (high − low + 1 = 2) then
  { if (A[low] < A[high]) then
   {    max = A[high]; min = A[low].
        return((max, min)).
    } else
     {   max = A[low]; min = A[high].
        return((max, min)).
     }end if
  else
   {  mid = low+high/2
     (max_l , min_l ) = MAXMIN(A, low, mid).
     (max_r , min_r ) =MAXMIN(A, mid + 1, high).
  }end if
  Set max to the larger of max_l and max_r ;
 similarly, set min to the smaller of min_l and min_r .
  return((max, min)).
}
```

So, $T(n) = T(n) = 2T(n/2) + 2$

Because maxmin performs two recursive calls on partitions of two half of the total size of the lists and then makes two further comparisons to sort out the max/min for the entire list
Solve using Masters Theorem, Case 1
$T(n) = \Theta(n)$

For Iterative Method:
$T(n) = n$ for finding Max by comparing element with other elements.
Another n to find min by comparing with other elements of array.
$T(n) = \Theta(n) + \Theta(n) = \Theta(2n) = \Theta(n)$

Here, In this case, The time complexity of both Iterative and recursive approaches are same. But Number of comparisons will differ.
For Ex. Iterative has $1 + 2(n-2)$ and Recursive has $(3n/2) - 2$ comparisons.

Submitted By:

Lakshman Madhav Kollipara

For CS325_Homework 2