



### Lab 3-2 A: Master Theorem Practice *Goal: Be able to solve recurrence relations with and without the Master Theorem*

### Solving Recurrences using the Master Theorem

The Master Theorem gives a general solution to recurrence relations that are of the form   
*T*(*n*) = *aT*(*n/b*) + *f* (*n*)where *f*(*n*)∈Θ(*nd*), *d ≥* 0- this is called the general divide and conquer recurrence

Master Theorem: If *a < bd* or logb (a) < d, *T*(*n*) ∈ Θ(*nd*)

If *a = bd* or logb (a) = d, *T*(*n*) ∈ Θ(*nd* log *n*)

If *a > bd* or logb (a) > d, *T*(*n*) ∈ Θ(*n*log *b a* )

Note: The same results hold with O instead of Θ

The key to understanding what is going on is the relationship between logb (a) and d.

Solve the following recurrences **using the Master Theorem** giving the Θ() complexity class for each.   
**Show the values for a, b, and d for each**

1. T(n) = 2T(n/2) + c a = \_\_\_2\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_0\_\_\_ Θ( n )
2. T(n) = T(n/2) + c a = \_\_\_1\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_0\_\_\_ Θ( log n )
3. T(n) = 4T(n/2) + c a = \_\_\_4\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_0\_\_\_ Θ( n2 )
4. T(n) = 2T(n/2) + n a = \_\_\_2\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_1\_\_\_ Θ( nlogn )
5. T(n) = T(n/2) + n a = \_\_\_1\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_1\_\_\_ Θ( n )
6. T(n) = 4T(n/2) + n a = \_\_\_4\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_1\_\_\_ Θ( n2 )
7. T(n) = 2T(n/2) + n2 a = \_\_\_2\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_2\_\_\_ Θ( n2 )
8. T(n) = T(n/2) + n2 a = \_\_\_1\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_2\_\_\_ Θ( n2 )
9. T(n) = 4T(n/2) + n2 a = \_\_\_4\_\_\_ b = \_\_\_2\_\_\_ d= \_\_\_2\_\_\_ Θ( n2logn )

Solve the following recurrences **using back substitution or the recursion tree method** giving the Θ() complexity class for each. You may need to consult the text or the Internet for necessary summation formulas. Show your work!

1. T(n) = T(n-1) + 2; T(0)=0

T(n - 2) + 2 + 2 T(n -3) + 2 + 2 + 2 T(n - k) + k2 n-k = 0 -> k = n T(0) + 2n Θ(n)

1. T(n) = T(n-1) + n ; T(0)=0

T(n - 2) + n + n T(n - 3) + n + n + n T(n - k) + kn n – k = 0 -> k = n T(0) + n2 Θ(n2)

1. T(n) = T(n-1) + 2n  T(0)=1

T(n - 2) + 2n – 1 + 2n T(n - 3) + 2n – 2 + 2n – 1 + 2n T(n - k) +

1. T(n) = 2T(n-1) + 1 ; T(0) = 0

2(2T(n - 2) + 1) + 1 2(2(2T(n - 3) + 1) + 1) + 1 2kT(n - k) + 2(k - 1) + 1 n – k = 0 -> k = n 2n\*0 + 2(n - 1) + 1 Θ(n)

### Lab 3-2 B: Divide and Conquer Problems *Goal: Review problems for Quick Sort and Quick Select*

Submit rigorous solutions to these problems

1. **Minimizing Weighing (warm up):** Suppose you are given n = 3k marbles that look identical, with one special marble that weighs more than the other marbles. You are also given a balancing scale that takes two items (or sets of items) and compares their weights. Design and analyze a divide and conquer algorithm to find the heavy marble using the balancing scale at most k times. Give the recurrence relation for the running time. Apply the Master Theorem to show that the running time is k.

Algorithm:

FindMarble(S[0, n - 1])

Weight = compare(S[0, n/2 - 1], S[n/2 + 1, n - 1] ) //return -1 if first half heavier, 0 if equal, and 1 if

second half heavier

If (weight == 0) then

Return S[n/2]

If (weight == -1) then

Return FindMarble(S[0, n/2])

Else do

Return FindMarble(S[n/2, n - 1])

Analysis:

Recurrence Relation:\_ T(n) = T(n / 2) + O(1)

Values of: a: 1 b: 2 d: 0

Asymptotic Running Time: log n

1. **Maximum Sum Contiguous Subsequence**: In computer science, the maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum. For example, for the sequence of values −2, 1, −3, 4, −1, 2, 1, −5, 4; the contiguous subarray with the largest sum is 4, −1, 2, 1, with sum 6.

Algorithm:

MaxSumArray(arr[], left, right)

If left == right then

Return arr[1]

Mid = left + right / 2

Return max(MaxSumArray(arr[], left, mid), MaxSumArray(arr[], mid + 1, right),

MaxCross(arr[], left, mid, right))

MaxCross(arr[], left, mid, right)

Sum = 0;

Lsum = arr[left]

For i from left + 1 to mid do

Sum += arr[i]

If sum > lsum

Lsum = sum;

Sum = 0

Rsum = arr[mid + 1]

For i from mid + 2 to right do

Sum += arr[i]

If sum > rsum then

Rsum = sum

Return max(lsum, rsum, lsum + rsum)

Analysis:

Recurrence Relation: T(n) = 2T(n / 2) + O(n)

Values of: a: 2 b: 2 d: 1

Asymptotic Running Time: n log n

1. **k-way merge revisited**: In the lab on Heap Sort, you found a way using a heap to merge k sorted lists each with n/k items each into a single sorted list of n items of O(n log k) complexity. In this lab, you goal is to find a divide and conquer algorithm that is also more efficient than the brute force approach. The brute force approach is conceptually to:

* merge two of the lists into a list – call it result2
* merge a third list with result – call it result3
* …
* until you get result n (obviously you would most likely need to be careful about storage constrains but ignore that here

**a. Show the time complexity of this brute force algorithm** Θ**(kn)**

T(n) = kO(n), since you are merging lists, each merge takes at most n comparisons. Since we are merging k lists, we get k times n comparisons.

**b. Give pseudo code for a divide and conquer algorithm** that solves the problem and show it is more efficient. For simplicity of the analysis of complexity you should assume that both n and k are powers of 2. Thus let n = 2s and k= 2t. Note that this also implies that n/k is 2s-t.

Combine two lists at a time, with the same length, until only one list remains, using:

Merge(A[], left, right, mid) left = first list, right = second list

I = 0

J = 0

K = 0

While (I < left.length and j < right.length)

If left[i] <= right[j] then

A[k++] = left[i++];

Else

A[k++] = left[j++];

While (I < left.length)

A[k++] = left[i++];

While (J < right.length)

A[k++] = right[j++];

Return A

**Finding local min in nxn grid:** Given an nxn grid, A of distinct numbers. A number is a local minimum if it is smaller than all its neighbors. That is A[i,j] is a local min if A[i,j] < min {A[i-1,j], A[i+1,j], A[i,j-1], A[i,j+1]} if 1 < i,j < n, For side and corner points there are only 3 and 2 numbers to compare to A[i,j] respectively. Use the divide and conquer paradigm to find a local minimum with only O(n) comparisons. Note there are numbers in the array so you cannot check all the numbers.

* Finding a solution for this problem can be quite difficult. In many problems it is a good idea to develop a simpler version of the problem and see if solving that version gives some insight in how to attack the more difficult version.
* A simpler version of this problem is the one dimensional **(1-D)** version: Given an array of distinct integers find a local minimum. At first glance this is trivial, just do a linear search for a global minimum and this is certainly guaranteed to be a local minimum.
* On the other hand, revisiting the original problem, it wants a O(n) solution with n2 numbers. This means that the **1-D** version with n numbers should be faster than O(n). In addition, the problem is framed as being solvable using divide and conquer.

1. **Find a local min in a 1 dimensional array of integers:** Use divide and conquer to define an algorithm that solves this problem in O(log n) number of comparisons.

FindLocalMin(A[], low, high)

If high > low

Return false

Mid = (low + high) / 2

If (mid == 0 and A[mid] < A[mid + 1])

Return true

Else if(mid == 0)

Return false

If (mid == A.length -1 and A[mid] < A[mid - 1])

Return true

Else if(mid == A.length - 1)

Return false

If (A[mid] < A[mid + 1] and A[mid] < A[mid - 1])

Return True

If (A[mid] > A[mid + 1])

Return FindLocalMin(A[], mid + 1, high)

Return FindLocalMin(A[], low, mid)