CS512 LECTURE NOTES - LECTURE 5

1 Examples of Recurrence Relations

We will try to use the Master Theorem in order to solve each one of the following recurrence relations. Notice that before we actually use the Master Theorem, we need to verify that all the conditions required in each case are satisfied.

1. $T(n) = 3T\left(\frac{n}{2}\right) + n^2$

   We first compare $f(n) = n^2$ to $n^{\log_b a} = n^{1.58\ldots}$.

   $$n^2 \in \Omega(n^{1.58\ldots})$$

   It appears that it is **Case 3**, but we must make sure that the regularity condition is satisfied:

   $$\begin{aligned} a\, f\left(\frac{n}{b}\right) &\leq c\, f(n) \quad 0 < c < 1 \\ 3\, \left(\frac{n}{2}\right)^2 &\leq c\, n^2 \\ \frac{3}{4} &\leq c \end{aligned}$$

   If we let $c = \frac{3}{4} < 1$ then clearly the regularity condition is satisfied and we have **Case 3**:

   $$T(n) \in \Theta(n^2)$$

2. $T(n) = T\left(\frac{n}{2}\right) + 2^n$

   We first compare $f(n) = 2^n$ to $n^{\log_b a} = n^0 = 1$.

   $$2^n \in \Omega(1)$$

   It appears that it is **Case 3**, but we must make sure that the regularity condition is satisfied:

$$\begin{aligned} a\, f\left(\frac{n}{b}\right) &\leq c\, f(n) \quad 0 < c < 1 \\ 2^{n/2} &\leq c\, 2^n \\ \frac{1}{2^{n/2}} &\leq c \end{aligned}$$

The greatest value of $\frac{1}{2^{n/2}}$ occurs when $n = 1$, so $c \geq \frac{1}{\sqrt{2}} = 0.7071\ldots$

If we let $c = 0.71 < 1$ then the regularity condition is satisfied and we have **Case 3**:

$$T(n) \in \Theta(2^n)$$

3. $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$

   We first compare $f(n) = n \lg n$ to $n^{\log_b a} = n^1 = n$.

   Can we claim that $n \lg n \in \Omega(n^{1+\epsilon})$?

   We know that $\lg n \in o(n^\epsilon)$, therefore $n \lg n \notin \Omega(n^{1+\epsilon})$

   We cannot use the Master Theorem in this case!

4. $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \lg n$

   The recurrence relations that can be solved using the Master Theorem assume that $f(n)$ represents the number of operations performed by an algorithm, and of course cannot be negative. So **Master Theorem does not apply in this case**.

2 THE SORTING PROBLEM

1. Input: List with n elements: $A[1], A[2], \ldots, A[n]$
2. Output: Sorted list. $B[1] \leq B[2] \leq \ldots \leq B[n]$

The sorted list is a permutation of the original list of elements. The elements can be anything, they can be student's records, could be employees, etc.

The analysis of sorting algorithms that we will obtain require that the following properties be satisfied:

- Every pair of elements must be comparable
- A compare function must be provided to compare pairs of elements. The function should determine if $a < b$, $a = b$, or $a > b$.
- The compare function must work in $O(1)$ time.
- The size of each element must be bounded by a constant.

First approach

```
Algorithm: SillySort
Input: A[1],A[2],...,A[n]
Output: sorted list A[1],A[2],...,A[n]

for each permutation B[1],B[2],...,B[n] of the
         original list
   if B[1]<=B[2]<=B[3]<=...<=B[n]
       return B[1],B[2],...,B[n]
```

The total number of permutations is $O(n!)$
Not a very useful algorithm, however notice that:

THE "IS SORTED" PROPERTY CAN BE VERIFIED IN $\Theta(n)$ time.

3 Sorting Algorithms Using Divide & Conquer

There are several algorithms that use the divide and conquer strategy:

1. Divide (carefully or easily) the list into sublists.
2. Recursively sort each one of the sublists
3. Combine the lists (carefully or easily)

Two sorting algorithms can easily be obtained using these strategies:

1. Quicksort: divide easily and combine carefully
2. Mergesort: divide carefully and combine easily

4 Quicksort

Quicksort uses the Divide and Conquer strategy:

- Divide carefully the original list into two sublists. Such that the elements of the first sublist are $<$ the elements of the second sublist.
- Recursively sort each list
- Combine the lists easily just concatenating them since they are already sorted.

The simplest strategy to divide the list into two (possibly empty) sublists in such a way that the elements of the first sublist are $<$ the elements of the second sublist is to select an element of the list as a "pivot" and partition the list according to this element. The simplest possible case is to use the first element of the list as a pivot.

PARTITION CAN BE DONE **IN PLACE** (without using more storage than that required to hold the array + a constant).

```
partition(a,first,last)
pivot=A[last]
i=first-1
for j=fist to last-1
   if (a[j] <= pivot)
      i++
      exchange A[i] with A[j]
i++
exchange A[i] with A[last]
return i
```

Notice that the last two instructions are necessary to put the pivot in the first list.

Partition can be used by the recursive quicksort algorithm:

```
quicksort(A,first,last)

if first<last
   q=partition(a,first,last)
   quicksort(a,first,q-1)
   quicksort(a,q+1,last)
```

The condition is given so that if the list is empty or has only one element, then it is assumed to be already sorted.

5 Example

5	2	9	10	1	8	4	12	**7**
5	2	1	4	**7**	8	10	12	9
5	2	1	**4**		8	10	12	**9**
2	1	**4**	5		8	**9**	12	10
2	**1**		5		8		12	**10**
1	2						**10**	12
1	2	4	5	7	8	9	10	12

6 Performance of Quicksort

The partition of the lists can be done in $\Theta(n)$ time.

Suppose that in a particular execution of partition there are d elements in the first sublist.

The operation of quicksort can be described by the following recurrence relation:

$$T(n) = T(n-q) + T(q) + \Theta(n)$$

Best Case

In this case the list is always partitioned in approximately equal halves, so we have that

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Using the Master Theorem ($a = 2$, $b = 2$, $d = 1$), $\log_2 2 = 1 = d$, therefore

$$T(n) \in \Theta(n \lg n)$$

What happens if it always partitions a 9-to-1 proportional split?