

# CS 311: Algorithm Design and Analysis

## Lecture 4

# Last Lecture we have

- Insertion Sort Algorithm
- Divide and Conquer (Binary Search, Merge Sort Algorithms)
- Master Theorem

# This Lecture we have

- Merge sort again !!!
- Recurrence
- Methods of solving recurrences
  - Substitution Method
  - Master Method
  - Recursion Tree

# Merge Sort

**MERGE-SORT**  $A[1 \dots n]$

1. If  $n = 1$ , done.
2. Recursively sort  $A[1 \dots \lceil n/2 \rceil]$  and  $A[\lceil n/2 \rceil + 1 \dots n]$ .
3. “*Merge*” the 2 sorted lists.

- How to express the cost of merge sort?

$$T(n) = 2T(n/2) + \Theta(n) \text{ for } n > 1, T(1) = 0 \Rightarrow \Theta(n \lg n)$$

# Merge

MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

# Merge Sort

1. **Divide**: Trivial.

2. **Conquer**: Recursively sort subarrays.

3. **Combine**: Linear-time merge.

$$T(n) = 2T(n/2) + \Theta(n)$$

*# subproblems*

*subproblem  
size*

*cost of dividing  
and combining*

# Recurrences

- The expression:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

is a *recurrence*.

- Recurrence: an equation that describes a function in terms of its value on smaller functions

# Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

# Solving Recurrences

- The substitution method (CLR 4.1)
  - The “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow ???$

# Solving Recurrences

- The substitution method (CLR 4.1)
  - The “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + 17 + n \rightarrow ???$

# Solving Recurrences

- The substitution method (CLR 4.1)
  - The “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \rightarrow \Theta(n \lg n)$

# Solving Recurrences

- Another option is what is called the “iteration method”
  - Expand the recurrence
  - Work some algebra to express as a summation
  - Evaluate the summation
- We will show several examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- $s(n) =$

$$c + s(n-1)$$

$$c + c + s(n-2)$$

$$2c + s(n-2)$$

$$2c + c + s(n-3)$$

$$3c + s(n-3)$$

...

$$kc + s(n-k) = ck + s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have
  - $s(n) = ck + s(n-k)$
- What if  $k = n$ ?
  - $s(n) = cn + s(0) = cn$

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have

- $s(n) = ck + s(n-k)$

- What if  $k = n$ ?

- $s(n) = cn + s(0) = cn$

- So
 
$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- Thus in general

- $s(n) = cn$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- $s(n)$

$$= n + s(n-1)$$

$$= n + n-1 + s(n-2)$$

$$= n + n-1 + n-2 + s(n-3)$$

$$= n + n-1 + n-2 + n-3 + s(n-4)$$

$$= \dots$$

$$= n + n-1 + n-2 + n-3 + \dots + n-(k-1) + s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- $s(n)$

$$= n + s(n-1)$$

$$= n + n-1 + s(n-2)$$

$$= n + n-1 + n-2 + s(n-3)$$

$$= n + n-1 + n-2 + n-3 + s(n-4)$$

$$= \dots$$

$$= n + n-1 + n-2 + n-3 + \dots + n-(k-1) + s(n-k)$$

$$= \sum_{i=n-k+1}^n i + s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have

$$\sum_{i=n-k+1}^n i + s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have

$$\sum_{i=n-k+1}^n i + s(n-k)$$

- What if  $k = n$ ?

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have

$$\sum_{i=n-k+1}^n i + s(n-k)$$

- What if  $k = n$ ?

$$\sum_{i=1}^n i + s(0) = \sum_{i=1}^n i + 0 = n \frac{n+1}{2}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for  $n \geq k$  we have

$$\sum_{i=n-k+1}^n i + s(n-k)$$

- What if  $k = n$ ?

$$\sum_{i=1}^n i + s(0) = \sum_{i=1}^n i + 0 = n \frac{n+1}{2}$$

- Thus in general

$$s(n) = n \frac{n+1}{2}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

- $T(n) =$   
 $2T(n/2) + c$   
 $2(2T(n/2/2) + c) + c$   
 $2^2T(n/2^2) + 2c + c$   
 $2^2(2T(n/2^2/2) + c) + 3c$   
 $2^3T(n/2^3) + 4c + 3c$   
 $2^3T(n/2^3) + 7c$   
 $2^3(2T(n/2^3/2) + c) + 7c$   
 $2^4T(n/2^4) + 15c$   
 $\dots$   
 $2^kT(n/2^k) + (2^k - 1)c$

# Changing variable example

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

*let  $m = \lg n$*

*Then*

$$S(2^m) = 2S(2^{m/2}) + m$$

*Using Master*

$$S(m) = O(m \lg m)$$

*Then*

$$T(n) = T(2^m) = S(m) = O(\lg n \lg \lg n)$$

*See page 86-87 in your book*

# Master theorem

- If  $T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + O(n^d)$  for some constants  $a > 0$ ,  $b > 1$ , and  $d \geq 0$ , then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

1.  $a < b^d$   $T(n) \in \Theta(n^d)$
2.  $a = b^d$   $T(n) \in \Theta(n^d \lg n)$
3.  $a > b^d$   $T(n) \in \Theta(n^{\log_b a})$

# Master theorem examples

$$T(n) = 9T(n/3) + n$$

$$T(n) = T(2n/3) + 1$$

$$T(n) = 3T(n/4) + n \lg n$$

$$T(n) = 2T(n/2) + n \lg n$$

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

$$T(n) = 8T(n/3) + cn^2$$

$$T(n) = 7T(n/2) + cn^2$$

$$T(n) = 2T(n/4) + \sqrt{n}$$

$$T(n) = 9T(n/3) + 2^{\lg n}$$

# Recursion Tree

*See the example in page 110*

*Tree → summation*

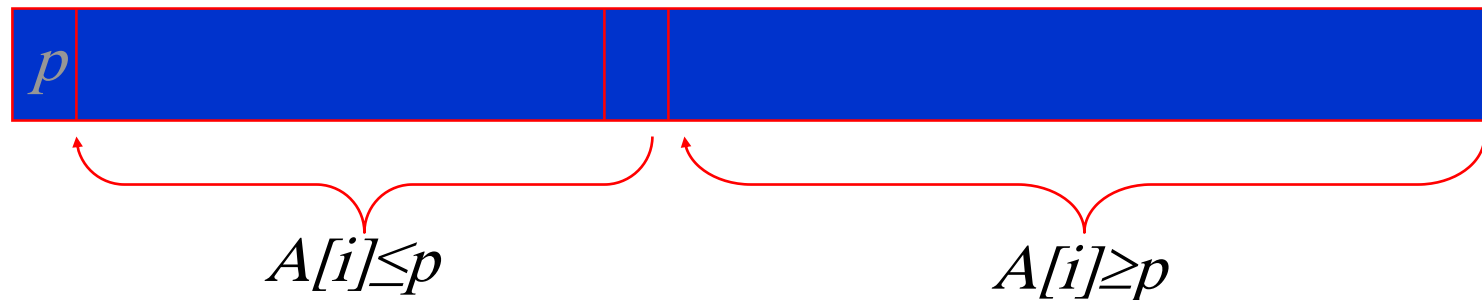
*Summation → The 1<sup>st</sup> item is greater than the last one*

*Summation → The 1<sup>st</sup> item is smaller than the last one*

*Summation → The items are equal*

# Quicksort by Hoare (1962)

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than or equal to the pivot
- Exchange the pivot with the last element in the first (i.e.,  $\leq$ ) sublist – the pivot is now in its final position
- Sort the two sublists recursively



- Apply quicksort to sort the list 7 2 9 10 5 4

# Quicksort Example

- Recursive implementation with the left most array entry selected as the pivot element.

0	15	12	3	21	25	3	9	8	18	28	5
1	9	12	3	5	8	3	15	25	18	28	21
2	8	3	3	5	9	12	15	21	18	25	28
3	5	3	3	8	9	12	15	18	21	25	28
4	3	3	5	8	9	12	15	18	21	25	28
5	3	3	5	8	9	12	15	18	21	25	28
6	3	3	5	8	9	12	15	18	21	25	28

# Quicksort Algorithm

- Input:
  - ⌚ Array  $E$  and indices  $first$ , and  $last$ , s.t. elements  $E[i]$  are defined for  $first \leq i \leq last$
- Output:
  - ⌚  $E[first], \dots, E[last]$  is a sorted rearrangement of the array
- ```
Void quickSort(Element[] E, int first, int last)
    if (first < last)
        Element pivotElement = E[first];
        int splitPoint = partition(E, pivotElement, first, last);
        quickSort (E, first, splitPoint -1 );
        quickSort (E, splitPoint +1, last );
    return;
```

# Quicksort Analysis

- Partition can be done in  $O(n)$  time, where  $n$  is the size of the array
- Let  $T(n)$  be the number of comparisons required by Quicksort
- If the pivot ends up at position  $k$ , then we have
  - $T(n) = T(n-k) + T(k-1) + n$
- To determine best-, worst-, and average-case complexity we need to determine the values of  $k$  that correspond to these cases.

# Best-Case Complexity

- ⌚ The best case is clearly when the pivot always partitions the array equally.
- ⌚ Intuitively, this would lead to a recursive depth of at most  $\lg n$  calls
- ⌚ We can actually prove this. In this case
  - $T(n) \approx T(n/2) + T(n/2) + n \Rightarrow \Theta(n \lg n)$

# Worst-Case and Average-Case Complexity

- The worst-case is when the pivot always ends up in the first or last element. That is, partitions the array as unequally as possible.
- In this case
  - $$\begin{aligned} T(n) &= T(n-1) + T(1-1) + n = T(n-1) + n \\ &= n + (n-1) + \dots + 1 \\ &= n(n+1)/2 \Rightarrow O(n^2) \end{aligned}$$
- Average case is rather complex, but is where the algorithm earns its name. The bottom line is:

$$A(n) \approx 1.386n \lg n \Rightarrow \Theta(n \lg n)$$

# The End

