Design and Analysis of Algorithms

Lecture 5-6

Instructor: Dr. G P Gupta

Designing algorithms

- incremental approach
  - Insertion sort
- divide-and-conquer approach

Divide-and-conquer approach

- break the problem into several subproblems that are similar to the original problem but smaller in size,
- solve the subproblems recursively, and then
- combine these solutions to create a solution to the original problem.

Divide-and-conquer approach

- **Divide:**
  - the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer**
  - the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine:**
  - the solutions to the subproblems into the solution for the original problem.

Analyzing divide-and-conquer algorithms

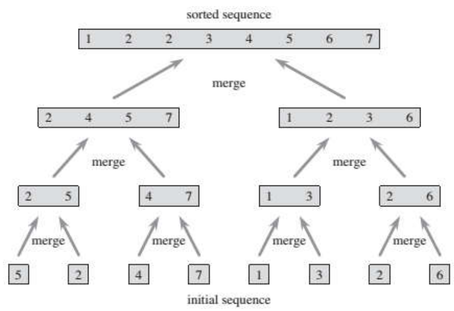
- When an algorithm ***contains a recursive call to itself***,
  - we can often describe its running time by a **recurrence equation or recurrence,**
  - which describes the overall running time on a problem of size ***n*** in terms of the running time on smaller inputs.
- A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

Merge sort algorithm cont..

- follows the divide-and-conquer paradigm
- **Divide:**
  - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:**
  - Sort the two subsequences recursively using merge sort.
- **Combine:**
  - Merge the two sorted subsequences to produce the sorted answer.

Merge sort algorithm cont..



L1.7

Merge sort algorithm cont..

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

L1.8

Merge sort algorithm cont..

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

L1.9

Example 3:Merge sort

MERGE-SORT $A[1 \,..\, n]$

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

Key subroutine: **MERGE**

L1.10

Merging two sorted arrays

20 12

13 11

7 9

2 1

L1.11

Merging two sorted arrays

20 12

13 11

7 9

2 1

1

L1.12

Merging two sorted arrays

20	12	20	12
13	11	13	11
7	9	7	9
2	1	2	
1			

L1.13

Merging two sorted arrays

20	12	20	12
13	11	13	11
7	9	7	9
2	1	2	
1		2	

L1.14

Merging two sorted arrays

20	12	20	12	20	12
13	11	13	11	13	11
7	9	7	9	7	9
2	1	2			
1		2			

L1.15

Merging two sorted arrays

20	12	20	12	20	12
13	11	13	11	13	11
7	9	7	9	7	9
2	1	2			
1		2		7	

L1.16

Merging two sorted arrays

20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11
7	9	7	9	7	9		9
2	1	2					
1		2		7			

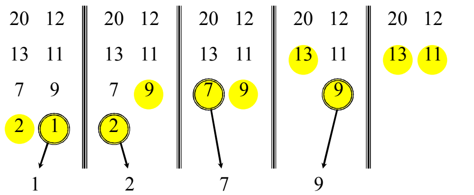
L1.17

Merging two sorted arrays

20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11
7	9	7	9	7	9		9
2	1	2					
1		2		7		9	

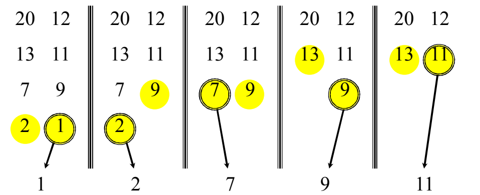
L1.18

Merging two sorted arrays



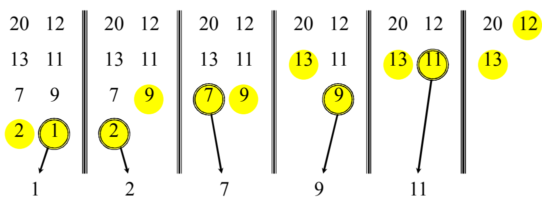
L1.19

Merging two sorted arrays



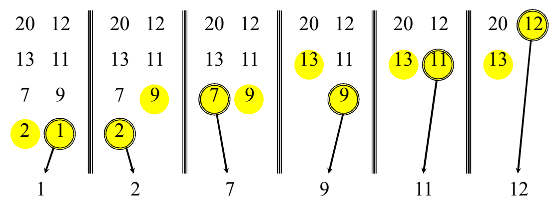
L1.20

Merging two sorted arrays



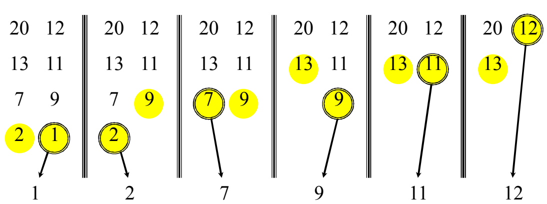
L1.21

Merging two sorted arrays



L1.22

Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

L1.23

Analyzing merge sort

$T(n)$	MERGE-SORT $A[1 \,.\,.\, n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$\Theta(n)$	***3. "Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

L1.24

Analyzing merge sort cont..

- **Divide:**
  - divide step just computes the middle of the subarray, which takes constant time. Thus, D(n)= $\Theta(1)$
- **Conquer:**
  - recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
- **Combine:**
  - uses MERGE procedure on an n-element subarray takes $\Theta(n)$ time , and so C(n)= $\Theta(n)$.

L1.25

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

L1.26

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

L1.27

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

L1.28

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

cn

$T(n/2)$ $T(n/2)$

L1.29

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

cn

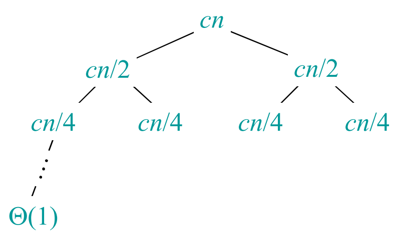
$cn/2$ $cn/2$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

L1.30

Recursion tree

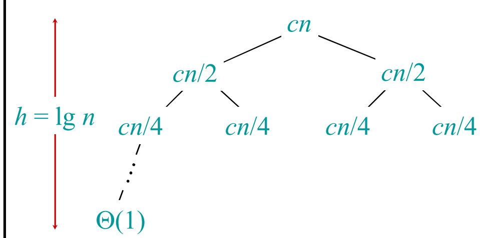
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.31

Recursion tree

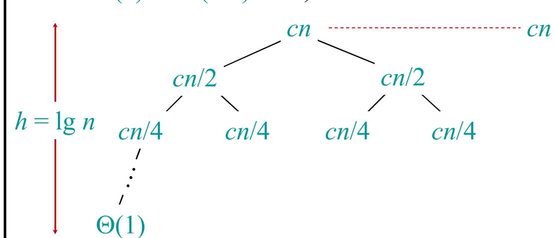
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.32

Recursion tree

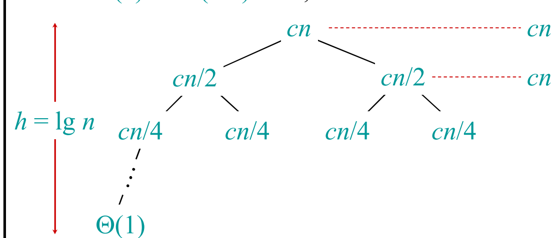
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.33

Recursion tree

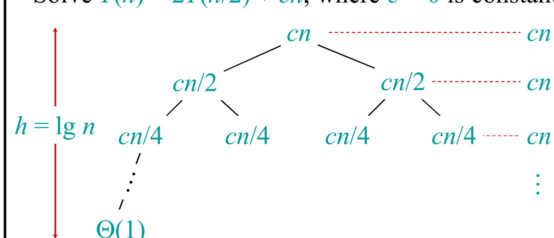
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.34

Recursion tree

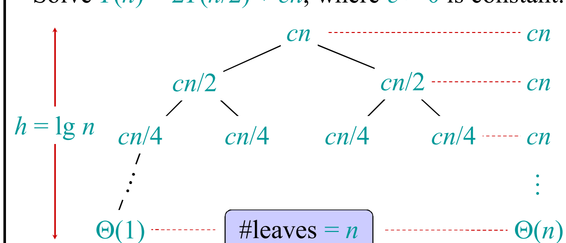
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.35

Recursion tree

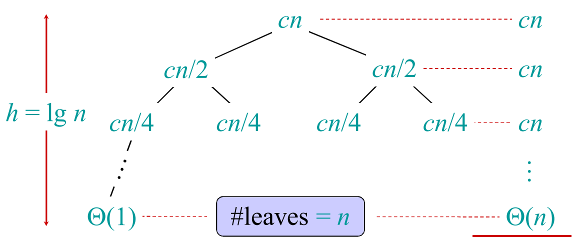
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.36

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total $= \Theta(n \lg n)$

L1.37

Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.

L1.38