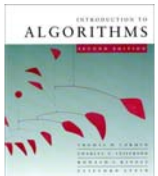
Introduction to Algorithms

6.046J/18.401J



Lecture 1

Prof. Piotr Indyk

Welcome to *Introduction to Algorithms*, Spring 2008

Handouts

1. Course Information
2. Calendar
3. Signup sheet (PLEASE return at the end of this lecture!)

Course information

1. Staff
2. Prerequisites
3. Lectures &Recitations
4. Problem sets
5. Describing algorithms
6. Grading policy
7. Collaboration policy
8. Textbook (CLRS)
9. Course web site
10. Extra help (HKN)

Design and analysis of algorithms

Theoretical study of how to solve computational problems efficiently

- "Computational problems": e.g., sorting data, finding shortest path, etc.
- "Solve": design an algorithm that does the job (correctly)
- "Theoretical": use the language of (algorithmic) mathematics to understand the performance of the algorithms.
- In particular, it enables us to define what "efficiently" means

Typical question: what is the fastest algorithm for a given problem ?

Performance vs. the rest of Course 6

Also important:

- modularity
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability
- …

Performance is often one of the key aspects:

- Searching the Web
- Analyzing the genome(s)
- …

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permuted input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

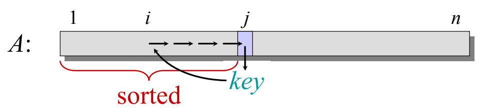
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)      ⊳ A[1 . . n]
  for j ← 2 to n
       do key ← A[ j]
          i ← j – 1
          while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
          A[i+1] = key
```



L1.7

Example of insertion sort

8 2 4 9 3 6

L1.8

Example of insertion sort

8 2 4 9 3 6

L1.9

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

L1.10

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

L1.11

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

L1.12

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

L1.13

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

L1.14

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

L1.15

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

L1.16

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

L1.17

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 *done*

L1.18

Running time of I-Sort ?

```
INSERTION-SORT (A, n) ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```

- Issues: the running time depends on
  - The input: an already sorted sequence is easy to sort
  - The processor: ZX80 vs Pentium

How can we develop metrics that do not depend on these factors ?

Input-independence

ANALYSES:

Worst-case: (usually)
- $T(n)$ = maximum time of algorithm on ***any*** input of size n.

Average-case: (sometimes)
- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)
- Cheat with a slow algorithm that works fast on *some* input.

Machine-independence

What is insertion sort's worst-case time?

BIG IDEA:
- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

"Asymptotic Analysis"

Θ-notation

Math:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ $\}$

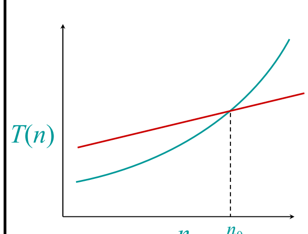
We write $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$

Engineering:
- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 – 5n + 6046 = \Theta(n^3)$

Also: $O()$, $\Omega()$, $o()$, $\omega()$, ...

Asymptotic analysis

When n gets large enough, a $\Theta(n)$ algorithm *always* beats a $\Theta(n^2)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.

Insertion sort analysis

Worst case:

$$T(n) = \Theta(\sum_{j=2}^{n} j) = \Theta(n^2)$$

Is insertion sort a fast sorting algorithm?
- Moderately so, for small n.
- Not at all, for large n.

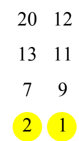
Merge sort

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

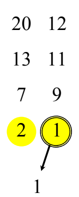
1. If $n = 1$, done.
2. Recursively sort $A[1 \,..\, \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \,..\, n]$.
3. "***Merge***" the two sorted lists into one.

Key subroutine: **MERGE**

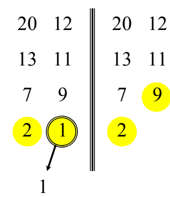
Merging two sorted arrays



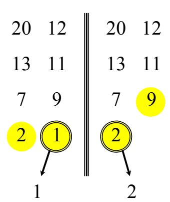
Merging two sorted arrays



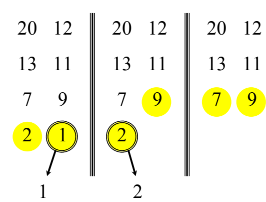
Merging two sorted arrays



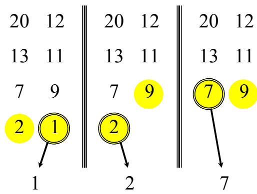
Merging two sorted arrays



Merging two sorted arrays

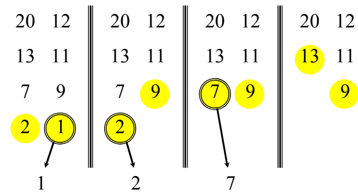


Merging two sorted arrays



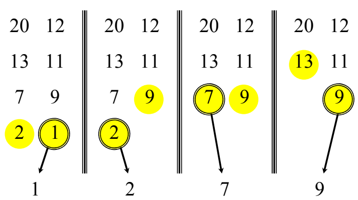
L1.31

Merging two sorted arrays



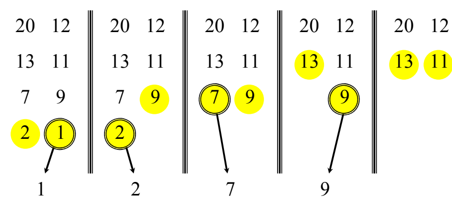
L1.32

Merging two sorted arrays



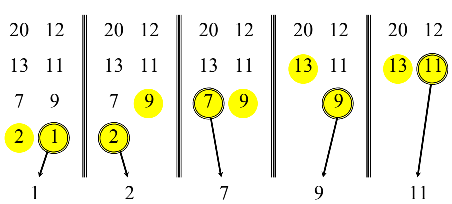
L1.33

Merging two sorted arrays



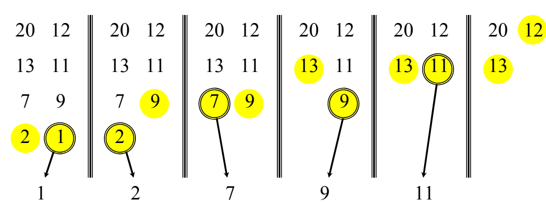
L1.34

Merging two sorted arrays



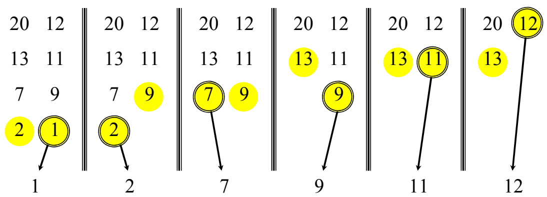
L1.35

Merging two sorted arrays

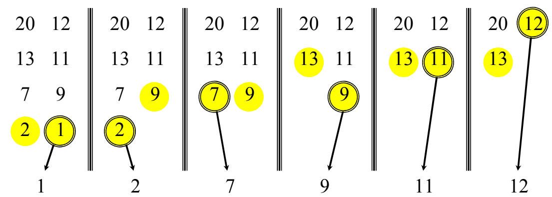


L1.36

Merging two sorted arrays



Merging two sorted arrays



Time to merge a total of n elements ?

$\Theta(n)$

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.

Recurrence for merge sort

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

- How to solve it ?
  - One approach on the next slide
  - More in CLRS and Lecture 2

Recursion tree

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

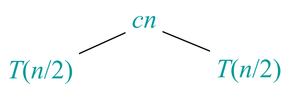
Recursion tree

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

$T(n)$

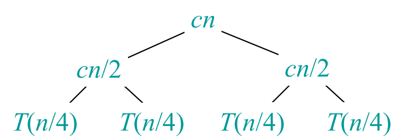
Recursion tree

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



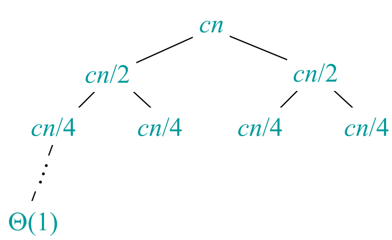
Recursion tree

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



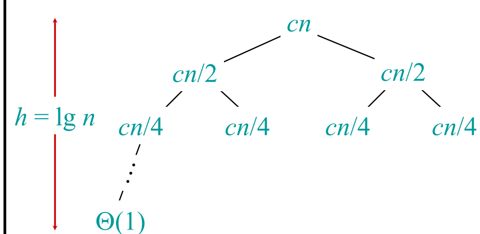
Recursion tree

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



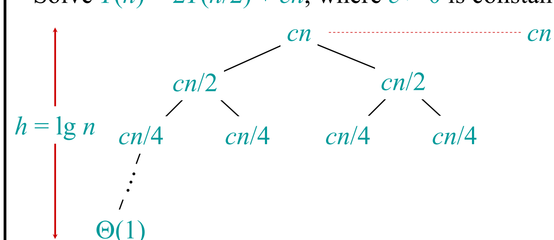
Recursion tree

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



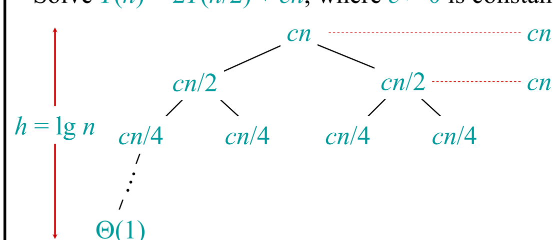
Recursion tree

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



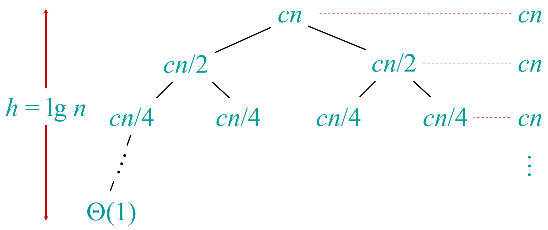
Recursion tree

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



Recursion tree

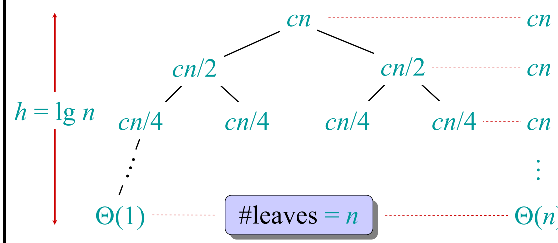
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.49

Recursion tree

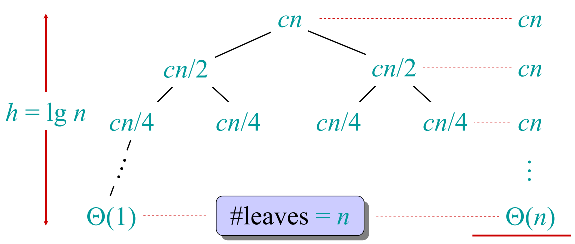
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



L1.50

Recursion tree

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



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

L1.51

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.
- Go test it out for yourself!

L1.52