## Introduction to algorithms

- In this course, we will discuss problems, and algorithms for solving these problems.
- There are so many algorithms – why focus on the ones in the syllabus ?

L1.1

## Why study algorithms and performance?

- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
  - (e.g., by using big-$O$ –notation)
- In real life, many algorithms, though different from each other, fall into one of several ***paradigms*** (discussed shortly).
- These paradigms can be studied, and applied for new problems

## Why these algorithms (cont.)

1. **Main paradigms:**
   a) **Greedy algorithms**
   b) **Divide-and-Conquers**
   c) **Dynamic programming**
   d) **Brach-and-Bound (mostly in AI )**
   e) **Etc etc.**
2. **Other reasons:**
   a) **Relevance to many areas:**
      - **E.g., networking, internet, search engines…**
   b) **Coolness**

## The problem of sorting

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

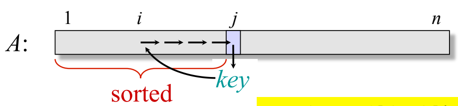
***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ .

**Example:**

***Input:*** 8 2 4 9 3 6

***Output:*** 2 3 4 6 8 9

## Insertion sort




**Invariants**: *A[ 1..j-1 ]* is sorted

*1 5 7 10 12 18* ***9*** *100 200*

*1 5 7 10 12 18 100 200*

Consider *A[j]=9*. Not in the correct place.
Need to make room for 9.
We shift all elements right, starting from 10.

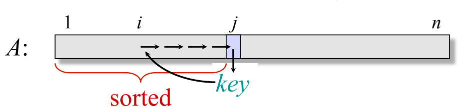
*1 5 7* ***9*** *10 12 18 100 200*



## Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)          //input: A[1 . . n]
    for j ← 2 to n             //outer loop
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key //inner loop
               do { A[i+1] ← A[i]
                    i ← i – 1}
           A[i+1] = key
```



## Example of insertion sort

8 **2** 4 9 3 6

L1.7

## Example of insertion sort

8 **2** 4 9 3 6

## Example of insertion sort

8 **2** 4 9 3 6

2 8 **4** 9 3 6

L1.9

## Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6

## Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6

## Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6

## 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

## 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

## 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

L1.15

## 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**

2 3 4 6 8 9 *done*

L1.17

## Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
  - Parameterize the running time by the size of the input *n*
  - Seek upper bounds on the running time *T(n)* for the input size *n*, because everybody likes a guarantee.

L1.18

## Kinds of 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-independent time

*What is insertion sort's worst-case time?*
- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

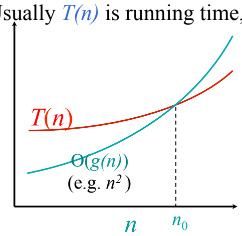
**BIG IDEA:**
- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \rightarrow \infty$ .

**"Asymptotic Analysis"**

## *O*-notation – cont.

we say that $T(n) = O(g(n))$ **iff**
there exists positive constants $c_1$, and $n_0$ such that
$0 \le T(n) \le c_1 g(n)$ for all $n \ge n_0$

Usually $T(n)$ is running time, and $n$ is size of input




- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

## *O*-notation – cont.

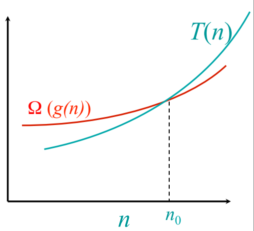
- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = O(n^3)$

## *Ω*-notation

***Math:***

We say that $T(n) = \Omega(\ g(n)\ )$ **iff** there exists positive constants $c_2$, and $n_0$ such that

$0 \le c_1\, g(n) \le T(n)$ for all $n \ge n_0$




***Engineering:***

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

L1.23

## Notation - cont

So if $T(n) = O(\ n^2\ )$ then we are also sure that
$T(n) = O(\ n^3\ )$ and that
$T(n) = O(\ n^{3.5}\ )$ and
$T(n) = O(\ 2^n\ )$

But it might or might not be true that $T(n) = O(\ n^{1.5}\ )$.

However, if $T(n) = \Omega(n^2\ )$ then it is **not** true that
$T(n) = O(\ n^{1.5}\ )$

L1.24

## Θ-notation

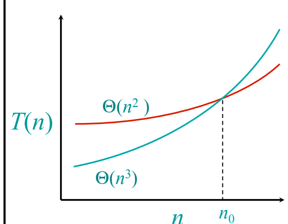
***Math:***
we say that $T(n) = \Theta(g(n))$ iff
there exist positive constants $c_1$, $c_2$, and $n_0$ such that
$0 \le c_1 g(n) \le T(n) \le c_2 g(n)$ for all $n \ge n_0$

In other words
$T(n) = \Theta(g(n))$ iff $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$

***Engineering:***

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

When $n$ gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.




## Insertion sort analysis

***Worst case:*** Input reverse sorted.

$T(n)=c+2c+3c+4c+\ldots+c(n-1) = cn(n-1)/2$

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$ [arithmetic series]

*Is insertion sort a fast sorting algorithm?*

- Moderately so, for small $n$.
- Not at all, for large $n$.

## Merge sort (divide-and-conquer algorithm)

MERGE-SORT $A[1 \mathinner{..} n]$

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

***Key subroutine:*** MERGE

## Merging two sorted arrays

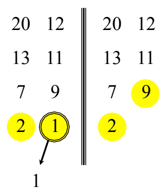
20 12
13 11
7 9
2 1

## Merging two sorted arrays

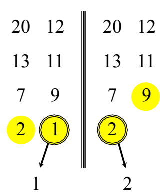
20 12
13 11
7 9
2 1

1

## Merging two sorted arrays

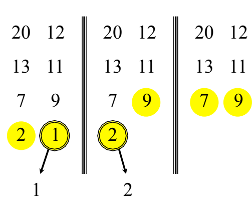



## Merging two sorted arrays

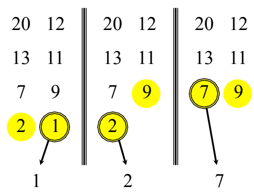



L1.32

## Merging two sorted arrays

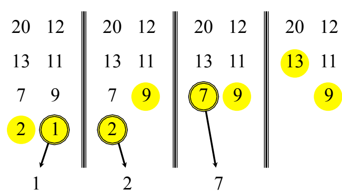


## Merging two sorted arrays



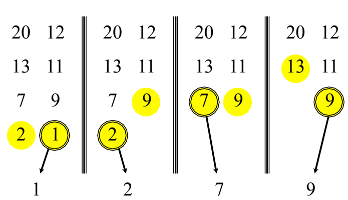

L1.34

## Merging two sorted arrays



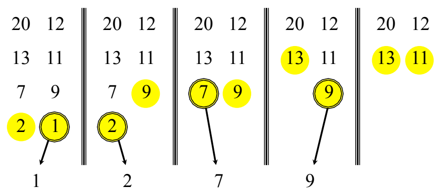

L1.35

## Merging two sorted arrays



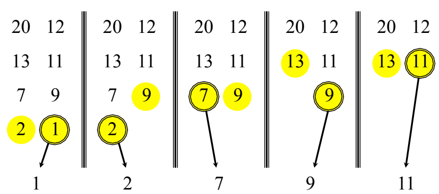

L1.36

## Merging two sorted arrays



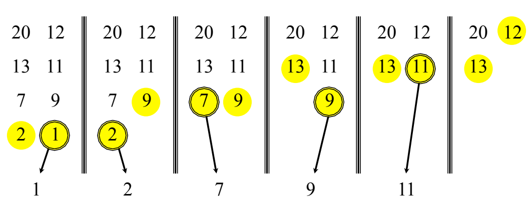

L1.37

## Merging two sorted arrays

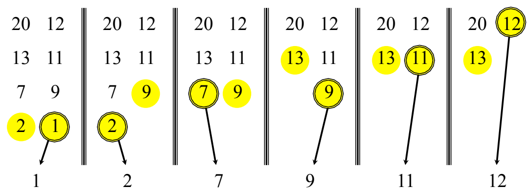



L1.38

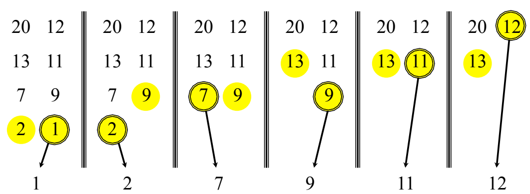
## Merging two sorted arrays



## Merging two sorted arrays




## Merging two sorted arrays




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

L1.41

## 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 |

***Abuse***

***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

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

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS provides several ways to find a good bound on $T(n)$.

## Recursion tree

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

## Recursion tree

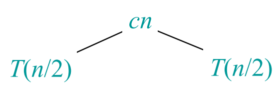
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

L1.45

## Recursion tree

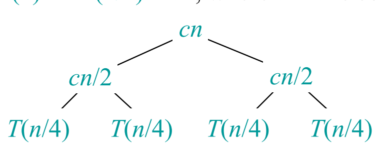
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.46

## Recursion tree

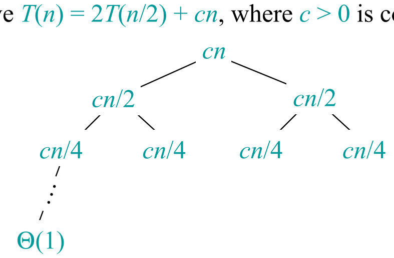
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.47

## Recursion tree

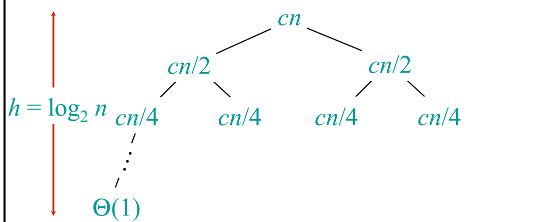
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.48

## Recursion tree

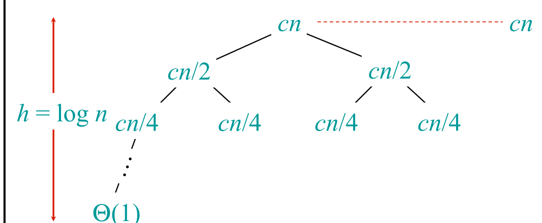
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.49

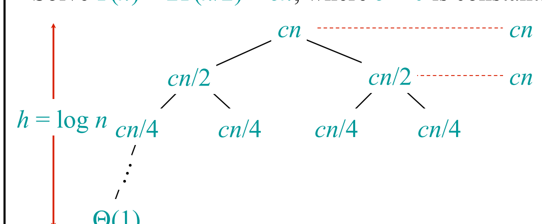
## Recursion tree

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




## Recursion tree

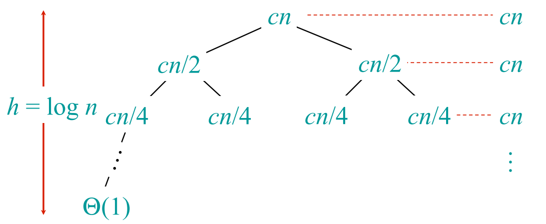
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.51

## Recursion tree

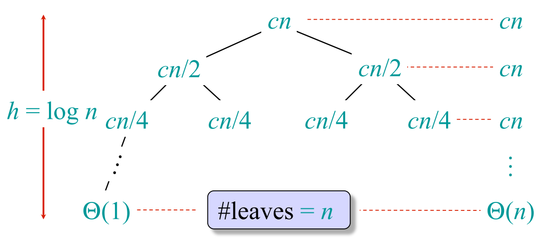
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.52

## Recursion tree

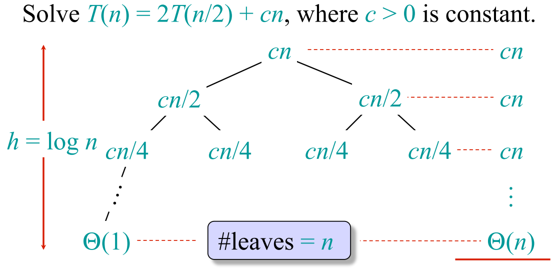
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.




L1.53

## Recursion tree

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




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

L1.54

## Conclusions

- $\Theta(n \log n)$ 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!

L1.55

## More examples (not in textbook)– iterative method (1)

```
NoNeed(n){
    • If (n<1) return ;
    • Print('*')
    • NoNeed(n-1)
}
```

Recursion formula: $T(n)=c+T(n-1)$, where $T(1)=c$. We can solve it using the **iteration method:**

$T(n)= c+T(n-1)=$
$c+\{c+T(n-2)\} = 2c+T(n-2) =$
$2c+\{ c+T(n-3) \} = 3c+T(n-3) = \ldots =$ (pick $k<n$)
$kc+T(n-k) =$ (setting $k = n-1$) …
$(n-1)c+T(1)=nc$

L1.56

## More examples (2)

```
NoNeed(n){
    if (n<1) return ;
    for( i=1 ; i<n ; i++)   print(*)
    NoNeed(n-1)
}
```

Recursion formula: $T(n)=cn+T(n-1)$, where $T(1)=c$. We can solve it using the **iteration method:**

$T(n)= cn+T(n-1)=$
$cn+\{c(n-1)+T(n-2)\} =$
$c[n+(n-1)]+\{c(n-2)+T(n-3)\}$
$=c[n+n-1+n-2+n-3]+T(n-3) = \ldots =$ (pick $k<n$)
$=c[n+n-1+n-2+ n-3+\ldots+n-k]+T(n-k-1) =$
(setting $k = n-1$) …
$c[ n+ n-1+n-2 + n-3+\ldots+1]+T(1)=$
$c[ 1+2+3+\ldots +n]+T(1)= cn(n+1)/2 = \Theta(n^2)$.

L1.57

# More examples (3)

- Read(n); $k=1$ ;
- while( $k \le n$ ) $k=2k$ ;

•We know that each iteration takes $O(1)$ times. Need to find the number of iterations.

- •After the first iteration $k=2=2^1$
- •After the 2nd iteration $k=4=2^2$
- •After the 3rd iteration $k=8=2^3$
- •....
- •After the $j$' th iteration $k=2^j$

**Recall:** $\log(ab)=\log(a)+\log(b)$
$\log(a^b) = b \log a$
$\log_a(x) = \log_b(x) / \log_b a$

•Assume $j$ iterations occurs until the loop exits. After the last one we have that $k=2^j < 2n$.

•Taking $\log_2$ from both sides, we have that

$\log_2 k = \log_2(2^j) < \log_2(2n)$ or..

$j \log_2 2 < \log_2(2) + \log_2(n)$ or..

$j < \log_2 n + 1$ or $j = O(\log_2 n)$. $T(n)=O(\log n)$

•Homework: Prove $T(n)=\Theta(\log n)$

L1.58

# More examples (a bit tricky)

```
read(n)
for(i=1 ; i < n ; i++)
  for( j=i ; j <n ; j += i )
     print("*") ;
```

•Naïve analysis:

- •The outer loop (on $i$) runs exactly $n-1$ times
- •The inner loop (on $j$) runs $O(n)$ times.
- •Together $O(n^2)$ times.

•More "sensitive" analysis:

- •For $i=1$ we run through $j=1,2,3,4...n$, total $n$ times.
- •For $i=2$ we run through $j=2,4,6,8,10...n$, total $n/2$ times.
- •For $i=3$ we run through $j=3,6,9,12...n$, total $n/3$ times .
- •For $i=4$ we run through $j=4,8,12,16...n$, total $n/4$ times.
- •For $i=n$ we run through $j=n$, total $n/n=1$ time.

•Summing up: $T(n)=n+n/2+n/3+n/4+...n/n = n(1+1/2+1/3+1/4+...1/n) \approx n \ln n$

Harmonic sum

# More examples: Geometric sum

```
read(n) ; a=0.31415926
while( n>1) {
    For( j=1; j<n ; j++ ) print("*")
    n=a*n ; }
```

•The **first** time the outer loop is called, the "print" is called $n$ times.
•The **2nd** time the outer loop is called, the "print" is called $an$ times.
•The **3rd** time the outer loop is called, the "print" is called $a^2 n$ times…
•The **k' th** time the outer loop is called, the "print" is called $a^k n$ times

•Let $t$ be the number of iterations of the outer loop. Then the total time

$$= n + an + a^2n + a^3n + \ldots a^t n = n(1 + a + a^2 + a^3 + \ldots a^t) < n(1 + a + a^2 + a^3 + \ldots a^t + \ldots) = n / (1-a) = O(n).$$

•Same analysis holds for any $a<1$

**Recall:** $1+a+a^2+\ldots+a^t = (1-a^{t+1})/(1-a)$.
**If** $a<1$ **then** $1+a+a^2+\ldots+a^t+\ldots = 1/(1-a)$

## Properties of big-O

- **Claim:** if $T_1(n)=O(g_1(n))$ and $T_2(n)=O(g_2(n))$ then $T_1(n)+T_2(n)=O(g_1(n) + g_2(n))$
- **Example**: $T_1(n)=O(n^2)$, $T_2(n)=O(n \log n)$ then $T_1(n)+T_2(n)=O(n^2 + n \log n) =O(n^2)$
- **Proof:** We know that there are constants $n_1, n_2, c_1, c_2$ **s.t.**
  - for every $n>n_1$ $T_1(n) < c_1 g_1(n)$. (definition of big-$O$ )
  - for every $n>n_2$ $T_2(n) < c_2 g_2(n)$. (definition of big-$O$ )
  - Now set $n'=\max\{n_1, n_2\}$, and $c'=c_1+c_2$, then
    - for every $n>n'$ we have that
    - $T_1(n)+T_2(n) < c_1 g_1(n) + c_2 g_2(n) \leq$
      $c' g_1(n) + c' g_2(n) =$
      $c' ( g_1(n) + g_2(n) )$

## More properties of big-O

- **Claim:** if $T_1(n)=O(g_1(n))$ and $T_2(n)=O(g_2(n))$ then $T_1(n)\, T_2(n)=O( g_1(n)\, g_2(n) )$
- **Example:** $T_1(n)=O(n^2)$, $T_2(n)=O(n \log n)$ then

  $T_1(n)\, T_2(n)=O(n^3 \log n )$

- Similar properties hold for $\Theta$, $\Omega$