CS300 : Introduction to Algorithms

Teaching staff

- Instructor : Prof. Choi, Sunghee (최성희)
  - Geometric Computing Lab. (gclab.kaist.ac.kr)
  - Office : E3-1 3404 (x3534)
  - Email : sunghee@kaist.edu
- TA : 김영훈 Yeonghun Kim neutrinoant@kaist.ac.kr
  성진영 Jinyoung Sung jysung710@kaist.ac.kr
  오진영 Jinyoung Oh sheogorath0213@kaist.ac.kr
  정승찬 Seungchan Jeong winner_sc@kaist.ac.kr
  최윤성 Yoonsung Choi giantsol2@kaist.ac.kr
  황지만 Jiman Hwang molehair@kaist.ac.kr
  김연수 Yeonsu Kim yeonsu.kim@kaist.ac.kr
  김태준 Taejun Kim xowns5979@kaist.ac.kr
  류형욱 Hyeonguk Ryu hyeonguk@kaist.ac.kr
  최재훈 Jaehoon Choi basedseal@kaist.ac.kr
  한조영 Joyoung Han hjyoung001@kaist.ac.kr

Course Information

- This course introduces basic concepts of design and analysis of algorithms.
- Textbooks
  - Algorithms by Dasgupta, Papadimitriou, and Vazirani [DPV]
  - Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein, MIT Press [CLRS]
- Website : klms.kaist.ac.kr
- Prerequisites : discrete mathematics (CS204), data structures (CS206)

Evaluation

- Paper Homework : 10%
- Programming Homework : 10%
- Midterm : 35%
- Final : 35%
- Participation (includes attendance, quiz) : 10%
- The instructor reserves the right to change this policy.
- Not “relative evaluation ” but “absolute evaluation” : encourage “collaboration” not “competition”.
- No tolerance on “cheating”! – if you are caught cheating, you will get F. No exception!

Online lecture and attendance

- Due to the huge class size, we could not offer real-time Zoom class.
- So, I will post each week’s video lecture and you may watch it anytime during the semester.
- The attendance will be checked by each week’s online quiz.
- You should submit the answer to the online quiz by the due date. No late answer will be considered.
- We will offer Zoom office hour.

Homework

- We’ll have 6 homeworks and 2 programming assignments.
- Copying is strongly forbidden. (If you copy, you will get F!)
- Try the homework on your own first!
- For the challenging problems, it might be useful to work together with other students.
- However, you should redo the solution from scratch by yourself, and write it up in your own words.
- You submit your homeworks to TA by the due day and time. (No delay accepted!)

How to study CS300

- Understanding the lectures is not enough.
- The best way is to solve exercises and problems in the textbook on your own without first consulting others' solutions.
- Also, try to explain the contents to your friends. Teaching is the best way to learn.
- Use Q&A board in klms to ask questions. Anyone can answer if the question is posted public. So, please post question as public so that we can use it for online discussion.

Contents

Design paradigms
- Divide-and-conquer
- Dynamic programming
- Greedy algorithms
- Randomized algorithms

Analysis techniques
- Recurrences
- Asymptotic analysis
- Probabilistic analysis

Graph algorithms
- Traversal, connectivity
- Minimum spanning trees
- Shortest paths

NP-completeness

Algorithm

- **Definition**: A well-defined computational procedure to solve a computational ***problem*** (to transform some input into a desired output).
- Statement of the ***problem*** specifies the desired ***input/output relationship***.
- Algorithm describes a specific computational procedure for achieving that input/output relationship.

Problem

- Example : sorting problem
  - Input : a sequence of *n* numbers $<a_1,a_2,\ldots,a_n>$.
  - Output : a permutation (reordering) $<a_1', a_2',\ldots,a_n'>$ of the input sequence s. t. $a_1' \leq a_2' \leq \ldots \leq a_n'$.
- ***Instance*** of a problem: a particular input of the problem
  - e.g.) $<8, 2, 4, 9, 3, 6>$ is an instance of the sorting problem.

Expressing algorithms

- We express algorithms in whatever way in the clearest and most concise.
- Often just use English.
- To make clear issues of control, use ***pseudocode*** similar to programming language.

Insertion sort

Insertion-Sort (A, *n*)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

► Insert A[*j*] into the sorted sequence A[1..*j*-1].

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow key$

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i+1] \leftarrow key$

8 2 4 9 3 6

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

  do $key \leftarrow A[j]$

    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

      do $A[i+1] \leftarrow A[i]$

        $i \leftarrow i-1$

    $A[i+1] \leftarrow key$

8 2 4 9 3 6

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow key$

8 2 4 9 3 6

2 8 4 9 3 6

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow key$

8 2 4 9 3 6

2 8 4 9 3 6

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

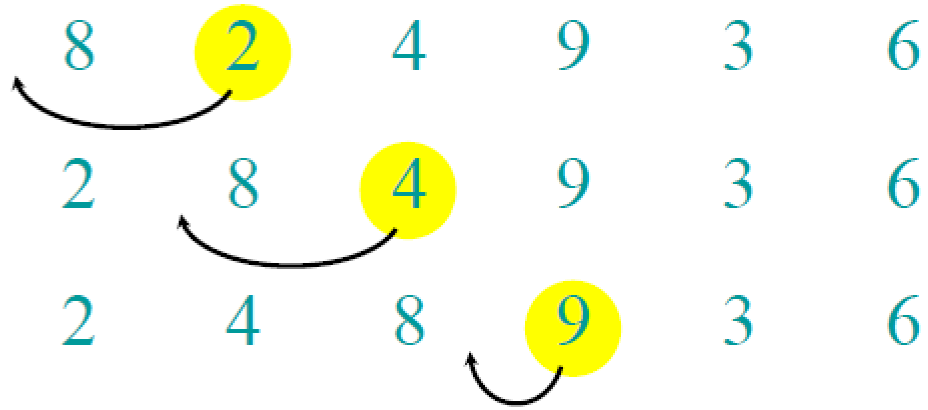
do $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow key$

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

Insertion sort (example)



Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow key$

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

  do $key \leftarrow A[j]$

    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

      do $A[i+1] \leftarrow A[i]$

        $i \leftarrow i - 1$

    $A[i+1] \leftarrow key$

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

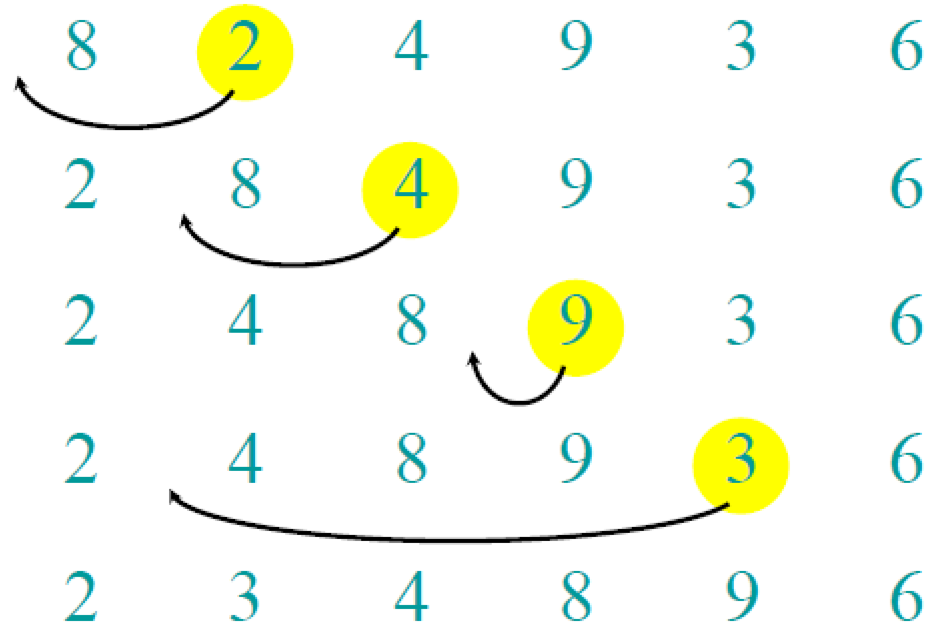
Insertion sort (example)

```
Insertion-Sort (A, 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
```

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

Insertion sort (example)

```
Insertion-Sort (A, 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
```



Insertion sort (example)

Insertion-Sort (A, n)

  for $j \leftarrow 2$ to n

   do $key \leftarrow A[j]$

    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > k$

     do $A[i+1] \leftarrow A[i]$

      $i \leftarrow i-1$

    $A[i+1] \leftarrow key$

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

Insertion sort (example)

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

do $key \leftarrow A[j]$

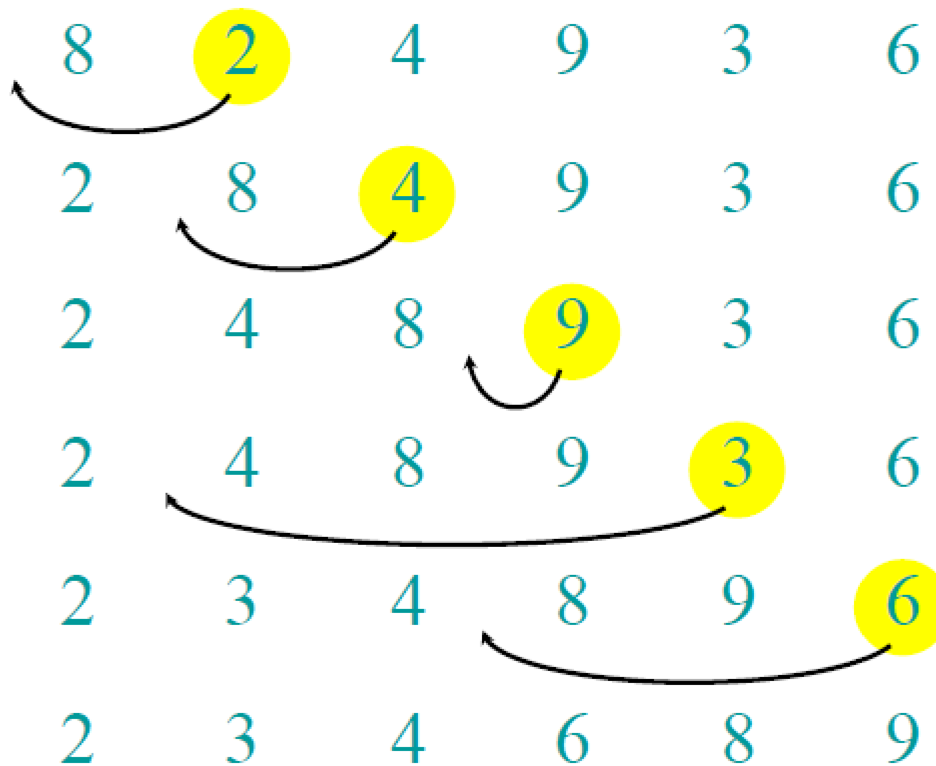
$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

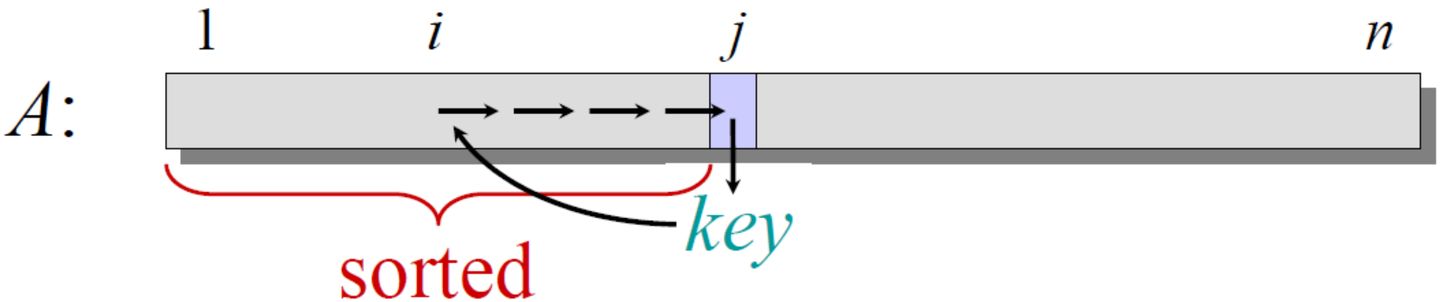
$i \leftarrow i-1$

$A[i+1] \leftarrow key$



Insertion sort (pseudocode)

```
Insertion-Sort (A, n)
    for j ← 2 to n
        do key ← A[j]
    ► Insert A[j] into the sorted sequence A[1..j-1].
            i ← j - 1
            while i > 0 and A[i] > key
                do A[i+1] ← A[i]
                    i ← i-1
            A[i+1] ← key
```

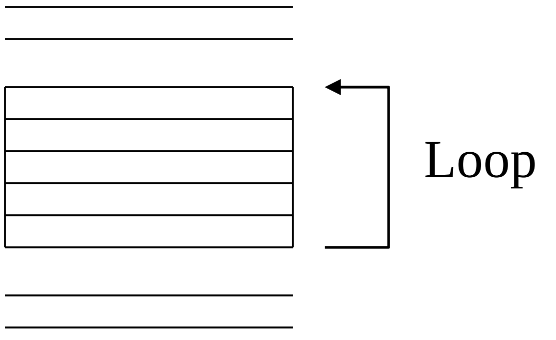


Correctness

- An algorithm is said to be ***correct*** if, for every input instance, it ***halts*** with the ***correct output***.
- We say that a correct algorithm ***solves*** the given computational problem.

Loop invariants

- Loop invariants
  - Program structure



- Definition: (Loop invariant)
  - Loop invariants are conditions and relationships that are satisfied by the variables and data structures at the end of each iteration of the loop.

Loop invariants

- Often use loop invariants to help us understand why an algorithm is correct.
- Must show three things about a loop invariants (similar to mathematical induction) :
  - ***Initialization*** : It is true prior to the first iteration of the loop.

    (a base case of the induction)
  - ***Maintenance*** : If it is true before an iteration of the loop, it remains true before the next iteration.

    (inductive step)
  - ***Termination*** : When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness of insertion sort

- Loop invariant : At the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.
- Initialization : when j=2, A[1..j-1] consists of the single element A[1]. Trivially sorted.
- Maintenance : need a loop invariant for the inner while loop. Informally, it works by moving A[j-1], A[j-2], A[j-3], and so on, by one position to the right until the proper position for A[j] is found.
- Termination : ends when j = n+1, A[1..n] consists of the elements originally in A[1..n] but in sorted order.

Running time of insertion sort

- Depends on input (ex) already sorted/ reverse sorted)
- Depends on input size
  - Parameterize in input size
- Want upper bounds, in general.
  - Gives a guarantee to users

Kinds of analysis

- 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.

Analysis

- Usually, interested in the worst-case running time because
  - It gives an upper bound
  - For some algorithms, the worst case occurs often.
  - Average case is often as bad as the worst case.

Analysis

Insertion-Sort (A, n)

for $j \leftarrow 2$ to n

  do $key \leftarrow A[j]$

    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

      do $A[i+1] \leftarrow A[i]$

        $i \leftarrow i-1$

    $A[i+1] \leftarrow key$

Machine-independent time

- What is insertion sort's worst-case time?
  - Depends on the speed of computer

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

Asymptotic Analysis

- Look only at the leading term of the formula for running time.
- Example : for insertion sort, the worst-case running time is $an^2 + bn + c$.
- It *grows like* n^2.
- We say that the running time is $\Theta(n^2)$.

- Usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Θ-notation

- $\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that

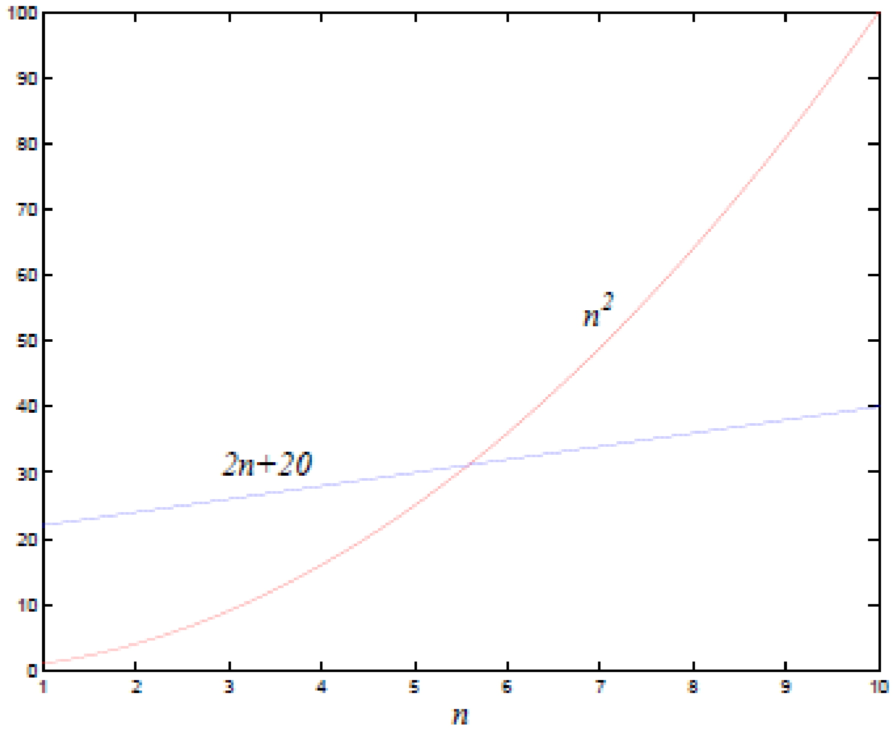
  $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$ $\}$

  $g(n)$ is an ***asymptotic tight bound*** for $f(n)$

  – Drop low-order terms and ignore leading constants

Asymptotic performance

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



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

Insertion sort analysis

- Worst case : input reverse sorted

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

- Average case : all permutations equally likely.

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

Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$$

Naïve recursive algorithm

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n − 1) + fib1(n − 2)
```

Analysis

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n − 1) + fib1(n − 2)
```

- Let T(n) denote *computer steps* needed to compute fib1(n).
- $T(n) \leq 2$ for $n \leq 1$
- $T(n) = T(n\text{-}1) + T(n\text{-}2) + 3$ for $n > 1$
- $T(n) \geq F_n$
- $F_n \approx 2^{0.694n}$
- T(n) is *exponential in n* ! – very slow except for very small *n*
- Can we do *better*?

Better algorithm

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i − 1] + f[i − 2]
return f[n]
```

A more careful analysis

- We counted the number of *basic computer steps* executed by each algorithm assuming that each step takes a constant amount of time.
- *Basic* computer steps : branching, loading, storing, comparisons, simple arithmetic, and so on
- This is a very useful simplification.
- We may assume adding two small numbers (like 32-bit numbers) takes a constant time.
- n-th Fibonacci number is not small, though. (about $0.694n$ bits long)
- How much time does it take to add two n-bit numbers?

Better algorithm

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

Why Algorithm Analysis?

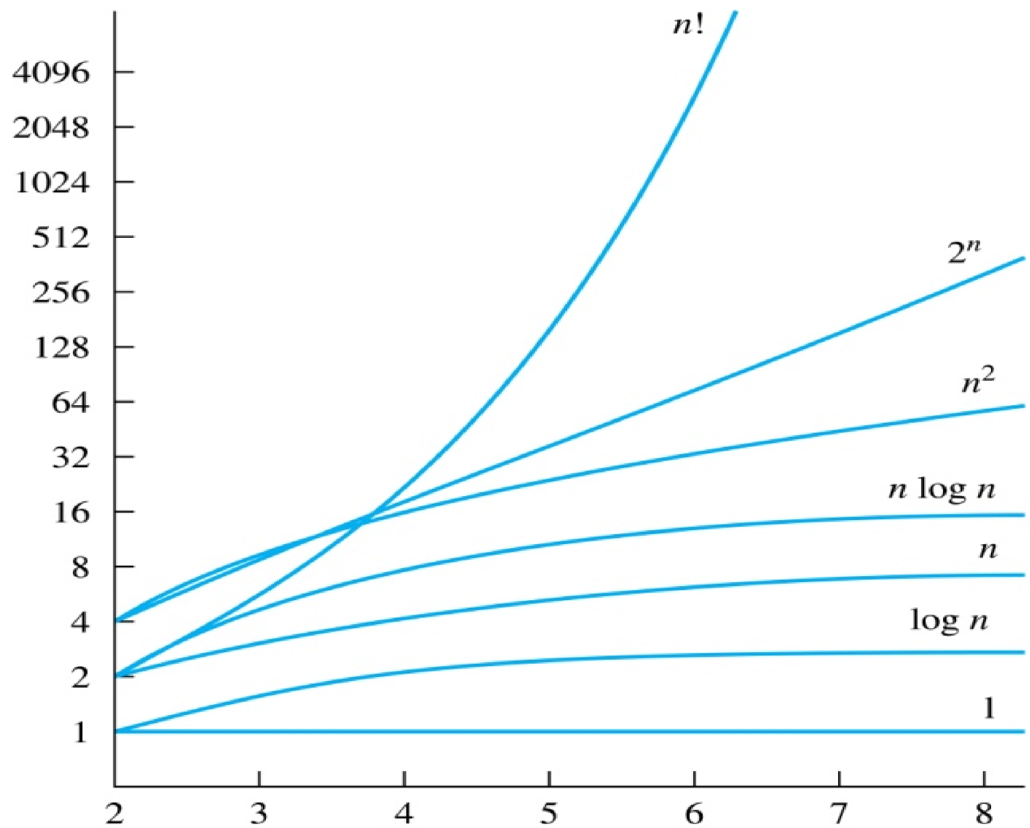
- Composability : If an algorithm is proven correct and has a guarantee on its running time, we can reuse it as a subroutine.
- Scaling : Asymptotic running time tells us how the running time scales with the problem size.
- Algorithm Design : Analysis often leads to insights that lead to better algorithms.
- Understanding : Analysis can teach us what parts of algorithms are important for what kind of input, so we can more easily solve related problems.
- Complexity theory : "How hard is problem X?"

How important is time complexity ?

Algorithm	1	2	3	4	
Time function (microsec.)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	2^n
Input size (n)			*Solution time*		
10	.00033 sec.	.0015 sec.	.0013 sec.	.0034 sec.	.001 sec.
100	.003 sec.	.03 sec.	.13 sec.	3.4 sec.	$4 \cdot 10^{16}$ yr.
1,000	.033 sec.	.45 sec.	13 sec.	.94 hr.	
10,000	.33 sec.	6.1 sec.	22 min.	39 days	
100,000	3.3 sec.	1.3 min.	1.5 days	108 yr.	
Time allowed			*Maximum solvable input size (approx.)*		
1 second	30,000	2,000	280	67	20
1 minute	1,800,000	82,000	2,200	260	26

Asymptotic growth

- ***constant*** 17
- ***logarithmic*** $\log(n)$
- ***square root*** $\sqrt{n}$
- ***linear*** n
- ***quasilinear*** $n \cdot \log(n)$
- ***quadratic*** n^2
- ***cubic*** n^3
- ***polynomial*** $c \cdot n^c$
- ***superpolynomial*** $n^{\log(n)}$
- ***exponential*** 2^n
- ***doubly exponential*** 2^{2^n}
- ***tetration*** $2^{2^{\cdots^{2^n}}}$



Notations

Theta	$f(n) = \theta(g(n))$	$f(n) \approx c\, g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) < c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) > c\, g(n)$

Θ-notation

- $\Theta(g(n)) = \{ f(n)$: 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 $n \ge n_0$ $\}$

$g(n)$ is an ***asymptotic tight bound*** for $f(n)$

Ex) $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, $n_0 = 8$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c, \;\; c \in \mathbf{R}^+ \Rightarrow f \in \theta(g)$$

O-notation

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

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Example : $2n^2 = O(n^3)$, with c=1 and $n_0 = 2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \in \mathbf{R}^* \Rightarrow f \in O(g)$$

$\mathbf{R}^*$= the set of non-negative real numbers

Ω-notation

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

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$

Example : $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$

$$\left[\begin{array}{l} \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \quad \text{or} \\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \end{array} \right] \Rightarrow f \in \Omega(g)$$

o-notation

- $o(g(n)) = \{ f(n)$: for ***any*** positive constant $c > 0$, there exists a positive constant n_0 such that $0 \leq f(n) < c\, g(n)$ for all $n \geq n_0 \}$

$f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$$

$n - 5, n, n^2, 10^{10}n^2 + 10^5 n + 10^9, n^2 - 9 \in o(n^3)$

ω-notation

- $\omega(g(n)) = \{ f(n)$: for any positive constant $c > 0$, there exists a positive constant n_0 such that $0 \leq c\, g(n) < f(n)$ for all $n \geq n_0 \}$

$f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \;\Rightarrow\; f(n) = \omega(g(n))$$

$$n^2, 10^{10} n^2 + 10^5 n + 10^9, n^3 - 9 \in \omega(n)$$

Note: $g(n) = o(f(n)) \Leftrightarrow f(n) = \omega(g(n))$

Theorem

$f(n) = \Theta(g(n))$ if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Properties of O, Ω, θ

Let f, g, h: $\mathbf{N} \to \mathbf{R}^*$. Then,

- P1: (Transitivity)

  $f \in O(g)$ and $g \in O(h) \Rightarrow f \in O(h)$

  How about Ω, θ, o, ω ?

- P2: $f \in O(g) \Leftrightarrow g \in \Omega(f)$

  $f \in o(g) \Leftrightarrow g \in \omega(f)$

  Duality

- P3: $f \in \theta(g) \Rightarrow g \in \theta(f)$
- P4: θ is an equivalence relation
- P5: $O(f+g) = O(\max\{f, g\})$

  How about Ω, θ, o, ω ?

[Proof] Exercise

Theorem: $\log n$ is in $o(n^\alpha)$ for any $\alpha > 0$. n^k is in $o(2^n)$ for any $k > 0$.