CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 3.
Characterizing Running Times

3.1 O-notation, Ω-notation, and Θ-notation

Asymptotic Notations

- Asymptotic efficiency
  - Running time of an algorithm (or a function or a program) when the input size n is large.
  - How the running time of an algorithm increases with the size of the input in the limit.

- Asymptotic notation: O-notation
  - Characterizes an upper bound on the asymptotic behavior.
  - If running time of an algorithm as a function of input size is:
    - $T(n) = 7n^3 + 100n^2 - 20n + 6$
  - Then we can write that it is $O(n^3)$.
  - Also, it is true that the running time is $O(n^4)$ or $O(n^5)$ or $O(n^6)$.

Asymptotic Notations

- Asymptotic notation: Ω-notation
  - characterizes a lower bound on the asymptotic behavior.
  - If running time of an algorithm as a function of input size is:
    - $T(n) = 7n^3 + 100n^2 - 20n + 6$
  - Then we can write that it is $\Omega(n^3)$.
  - Also, it is true that the running time is $\Omega(n^2)$ or $\Omega(n)$.

- Asymptotic notation: Θ-notation
  - characterizes a tight bound on the asymptotic behavior.
  - It says that a function grows precisely at a certain rate.
  - If running time of an algorithm as a function of input size is:
    - $T(n) = 7n^3 + 100n^2 - 20n + 6$
  - Then we can write that it is $\Theta(n^3)$.

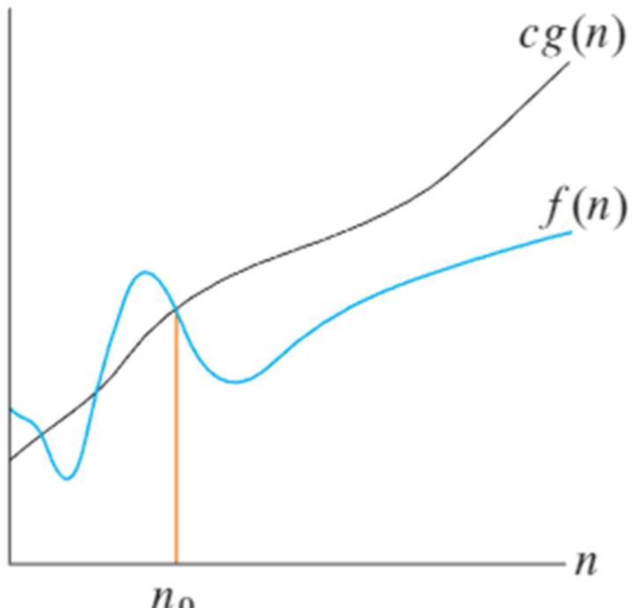
3.2 Asymptotic Notation: Formal Definitions

O-notation: asymptotic upper bound

- Formal definition of O-notation
  - g(n) must be asymptotically nonnegative
    - (nonnegative when n is sufficiently large)

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .^1$$

- $4n^2 + 100n + 500 = O(n^2)$
  - If we choose $n_0 = 10$ and $c = 19$,
  - $4n^2 + 100n + 500 \le cn^2$ for all $n \ge n_0$ holds



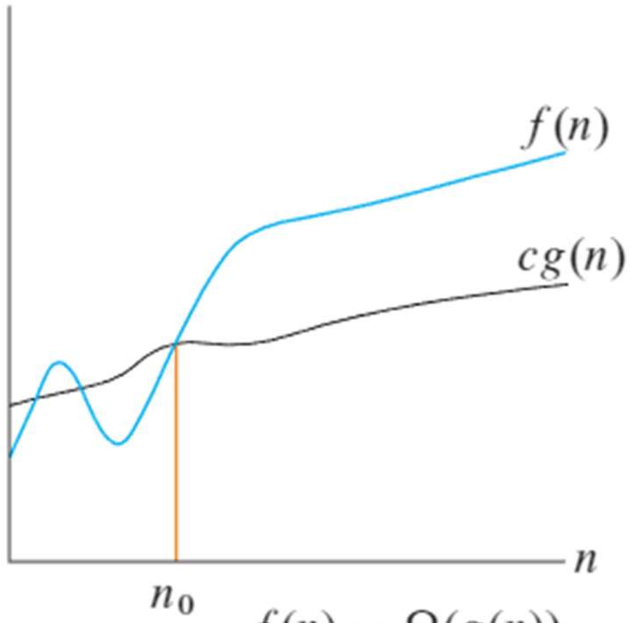
$f(n) = O(g(n))$

Ω-notation: asymptotic lower bound

- Formal definition of Ω-notation

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

- $4n^2 + 100n + 500 = \Omega(n^2)$
  - If we choose $n_0 = 1$ and $c = 4$,
  - $4n^2 + 100n + 500 \geq cn^2$ for all $n \geq n_0$ holds



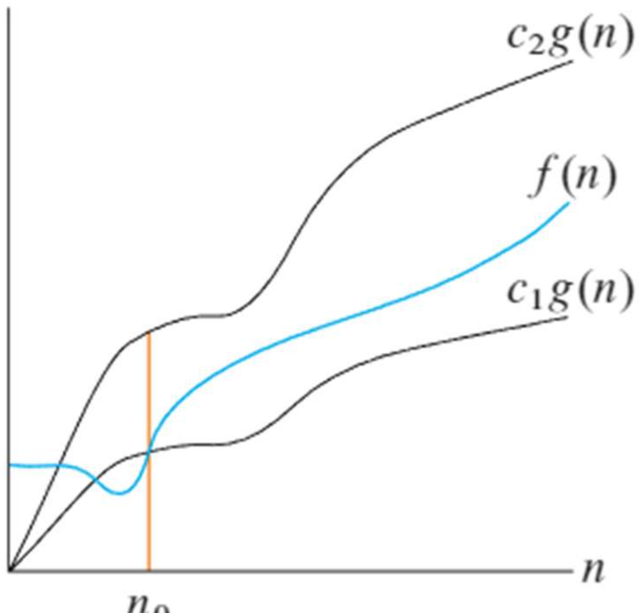
$f(n) = \Omega(g(n))$

Θ-notation: asymptotic tight bound

- Formal definition of Ω-notation

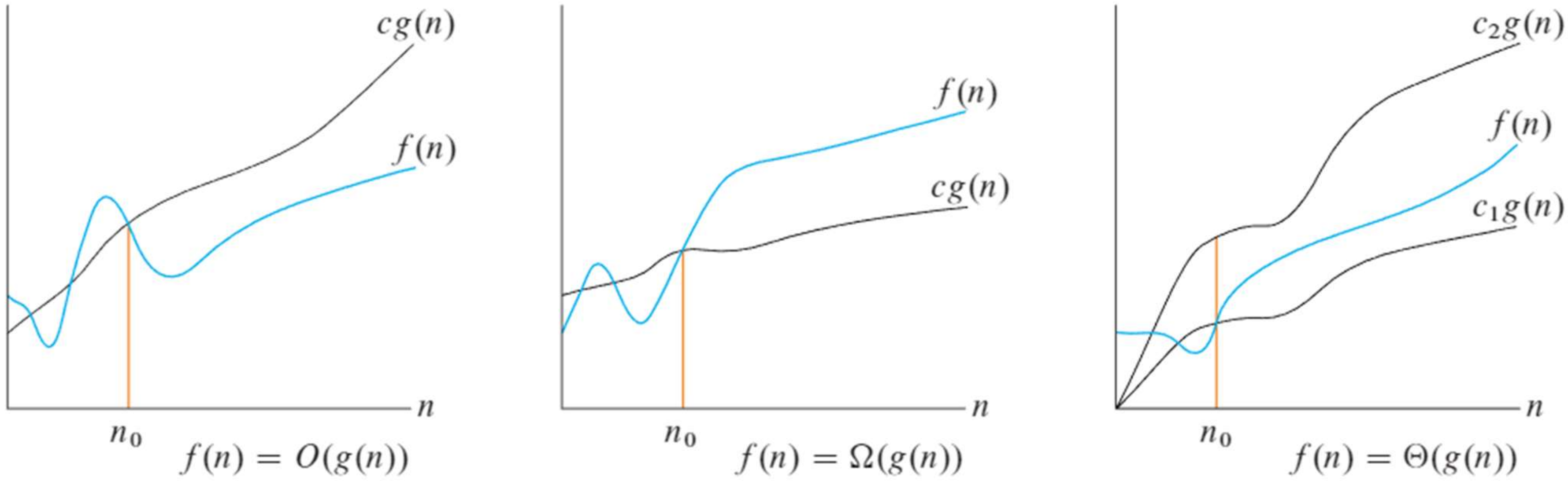
$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■



$f(n) = \Theta(g(n))$

Graphic examples of asymptotic notations



서강대학교 SOGANG UNIVERSITY

Asymptotic Notation and Running Times

- The worst case running time of insertion sort is $O(n^2)$, $\Omega(n^2)$, or $\Theta(n^2)$.
  - $\Theta(n^2)$ is the most precise and most preferred notation.
  - We should not say "running time of insertion sort is $\Theta(n^2)$", because its best case running time is $\Theta(n)$.

- The running time of merge sort is $\Theta(n \log n)$.
  - It is true for best, average, and worst case.

- People occasionally conflate O-notation with Θ-notation by mistakenly using O-notation to indicate an asymptotically tight bound.

- When we use an asymptotic notation, we typically use representative functions as g(n) in $\Theta\big(g(n)\big)$.
  - $1,\ \log n,\ n, n\log n, n^2, n^3, 2^n$

Asymptotic Notation in Equations and Inequalities

- When we say $4n^2 + 100n + 500 = O(n^2)$, we actually mean $4n^2 + 100n + 500 \in O(n^2)$.

- When we say $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$, we mean that $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ where $f(n) \in \Theta(n)$.

- We write $2n^2 + \Theta(n)$, our intention is that we do not care about the details of $\Theta(n)$ part.
  - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

o-Notation

- o-notation is used to denote an upper bound that is not asymptotically tight.

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{, there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

- Example: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

- O-notation vs. o-notation
  - in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for some constant $c > 0$
  - in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for all constants $c > 0$
  - Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n gets large.

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

ω-Notation

- ω-notation is used to denote a lower bound that is not asymptotically tight.

$$\omega(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

- Example: $\frac{n^2}{2} = \omega(n^2)$, but $\frac{n^2}{2} \neq \omega(n^2)$.

- The relation $f(n) = \omega(g(n))$ implies that

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

Comparing Functions

- Transitivity

$$
\begin{aligned}
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\text{ imply } f(n) = \Theta(h(n)), \\
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\text{ imply } f(n) = O(h(n)), \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\text{ imply } f(n) = \Omega(h(n)), \\
f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) &\text{ imply } f(n) = o(h(n)), \\
f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &\text{ imply } f(n) = \omega(h(n)).
\end{aligned}
$$

- Reflexivity

$$
\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}
$$

- Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

- Transpose symmetry

$$
\begin{aligned}
f(n) = O(g(n)) &\text{ if and only if } g(n) = \Omega(f(n)), \\
f(n) = o(g(n)) &\text{ if and only if } g(n) = \omega(f(n)).
\end{aligned}
$$

Comparing Functions

- Smaller, larger functions
  - $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$
  - $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$

- Not all functions are asymptotically comparable
  - we cannot compare functions $f(n) = n$ and $g(n) = n^{1+\sin n}$ asymptotically
  - $g(n)$ oscillates between 0 and 2.

3.3 Standard Notations and Common Functions

Monotonicity

- A function $f(n)$ is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$.
- A function $f(n)$ is monotonically decreasing if $m \leq n$ implies $f(m) \geq f(n)$.
- A function $f(n)$ is strictly increasing if $m < n$ implies $f(m) < f(n)$.
- A function $f(n)$ is strictly decreasing if $m < n$ implies $f(m) > f(n)$.

Floors and Ceilings

- For any real number x,
  - $\lfloor x \rfloor$ is the greatest integer less than or equal to x.
  - $\lceil x \rceil$ is the least integer greater than or equal to x.
- For all integer n, $\lfloor n \rfloor = n = \lceil n \rceil$
- For all real x, $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- $-\lfloor x \rfloor = \lceil -x \rceil, -\lceil x \rceil = \lfloor -x \rfloor$
- For any real number $x \geq 0$ and integers $a, b > 0$, we have

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil, \quad \left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor, \quad \left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b}, \quad \left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a - (b-1)}{b}$$

- For any integer n and real number x, we have

$$\lfloor n + x \rfloor = n + \lfloor x \rfloor ,$$
$$\lceil n + x \rceil = n + \lceil x \rceil .$$

Modular Arithmetic

- For any integer a and any positive integer n, the value $a \bmod n$ is the remainder (or residue) of the quotient a/n.

- $a \bmod n = a - n\lfloor a/n \rfloor$
- $0 \leq a \bmod n < n$

- If $(a\ mod\ n) = (b \bmod n)$, we say that a is equivalent to b, modulo n.

Polynomials

- Given a nonnegative integer d, a polynomial in n of degree d is a function of $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

  - where the constants $a_0, a_1, \dots, a_d$ are the coefficients of the polynomial.
  - $a_d \neq 0$

- A polynomial is asymptotically positive if and only if $a_d > 0$.
- For an asymptotically positive polynomial $p(n)$ of degree d, $p(n) = \Theta(n^d)$.
- For any real constant $a \geq 0$, the function n^a is monotonically increasing, and for any real constant $a \leq 0$, the function n^a is monotonically decreasing.
- We say that function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

Exponentials

- For all real $a > 0$, m, and n, we have the following identities.

$$
\begin{aligned}
a^0 &= 1 , \\
a^1 &= a , \\
a^{-1} &= 1/a, \\
(a^m)^n &= a^{mn} , \\
(a^m)^n &= (a^n)^m , \\
a^m a^n &= a^{m+n} .
\end{aligned}
$$

- For convenience, we may assume $0^0 = 1$.
- Rates of growth of polynomials and exponentials

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

al constants $a > 1$ and b, we have

  - from which we can conclude that $n^b = o(a^n)$
  - any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Exponentials

- For all real x, $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$
  - where ! denotes the factorial function.

- For all real x, we have the inequality, $1 + x \leq e^x$
  - where equality holds only when $x = 0$.

- When $|x| \leq 1$, we have the approximation
  - $1 + x \leq e^x \leq 1 + x + x^2$

- When $x \to 0$, the approximation of e^x by $1 + x$ is quite good
  - $e^x = 1 + x + \Theta(x^2)$

- For all x, $\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x$

Logarithms

- Notations for logarithms

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} , \\
\ln n &= \log_e n && \text{(natural logarithm)} , \\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} , \\
\lg \lg n &= \lg(\lg n) && \text{(composition)} .
\end{aligned}$$

- For any constant $b > 1$, the function $\log_b n$ is:
  - undefined if $n \leq 0$
  - strictly increasing if $n > 0$
  - negative if $0 < n < 1$
  - positive if $n > 1$
  - 0 if $n = 1$

Logarithms

- For all real $a > 0, b > 0, c > 0$, and n, we have

$$
\begin{aligned}
a &= b^{\log_b a} , \\
\log_c(ab) &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a , \\
\log_b a &= \frac{\log_c a}{\log_c b} , \\
\log_b(1/a) &= -\log_b a , \\
\log_b a &= \frac{1}{\log_a b} , \\
a^{\log_b c} &= c^{\log_b a} ,
\end{aligned}
$$

Logarithms

- When $|x| < 1$,
  - $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$.

- For $x > -1$
  - $\frac{x}{1+x} \leq \ln(1+x) \leq x$
  - equality holds only for $x = 0$.

- $f(n)$ is polylogarithmically bounded if $f(n) = O(\lg^k n)$ for some constant k.

- For all real constants $a > 0$ and b,
  - $\lg^b n = o(n^a)$
  - any positive polynomial function grows faster than any polylogarithmic function.

Factorials

- The notation ! (n factorial) is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0\,, \\ n \cdot (n-1)! & \text{if } n > 0\,. \end{cases}$$

- A weak upper bound on the factorial function is $n! \leq n^n$.

- Stirling's approximation gives us a tighter upper bound.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

- According to Stirling's appoxtimation, $\lg(n!) = \Theta(n \lg n)$

Functional Iteration

- We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value n.

- Formally, let $f(n)$ be a function over the reals. For nonnegative integers i, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases}$$

- If $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The Iterated Logarithm Function

- We use the notation $\lg^* n$ ("log star of n") to denote the iterated logarithm.

$$\lg^* n = \min \{ i \geq 0 : \lg^{(i)} n \leq 1 \}$$

  – logarithm function applied i times in succession, starting with argument n, until $\lg^{(i)} n$ becomes less than or equal to 1.

- The iterated logarithm is a very slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^* (2^{65536}) &= 5 . \end{aligned}$$

- Be sure to distinguish $\lg^{(i)} n$ from $\lg^i n$ (the logarithm of n raised to the ith power)!

Fibonacci Numbers

- We define the Fibonacci numbers F_i, for $i \geq 0$, as follows.

$$F_i = \begin{cases} 0 & \text{if } i = 0\,, \\ 1 & \text{if } i = 1\,, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2\,. \end{cases}$$

- The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Fibonacci numbers are related to the golden ratio ϕ and its conjugate $\hat{\phi}$, the two roots of the equation $x^2 = x + 1$.

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots, \end{aligned} \qquad \begin{aligned} \hat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots. \end{aligned} \qquad F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

- Since $|\hat{\phi}| < 1$, we have $\frac{|\hat{\phi}^i|}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}$, which implies $F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor$
  - ith Fibonacci number F_i is equal to $\frac{\phi^i}{\sqrt{5}}$ rounded to the nearest integer.
  - Thus, Fibonacci numbers grow exponentially.

Review: Efficient Algorithm Design

Efficient algorithm design: example 1

- <u>Sequential search</u> versus <u>binary search</u>
  - **Problem**: Determine whether x is in the sorted array S of n keys.
  - **Inputs**: positive integer n, sorted (non-decreasing order) arrays of keys S indexed from 1 to n, a key x.
  - **Outputs**: the location of x in S (0 if x is not in S).

  - Sequential search: $T(n) = O(n)$
  - Binary search: $T(n) = O(\log n)$

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

  - Why is the binary search more efficient?

Efficient algorithm design: example 2

- <u>The Fibonacci Sequence</u>
  - **Problem**: Determine the *n*th term in the Fibonacci sequence.
  - **Inputs**: a nonnegative integer *n.*
  - **Outputs**: the *n*th term of the Fibonacci sequence.

$$\boxed{f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \text{ for } n \geq 2}$$

<recursive: divide-and-conquer>

```
int fib (int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return fib(n-1) + fib(n-2);
}
```

- Divide-and-Conquer: $T(n) = O(2^n)$
- Dynamic Programming: $T(n) = O(n)$

<iterative: dynamic programming>

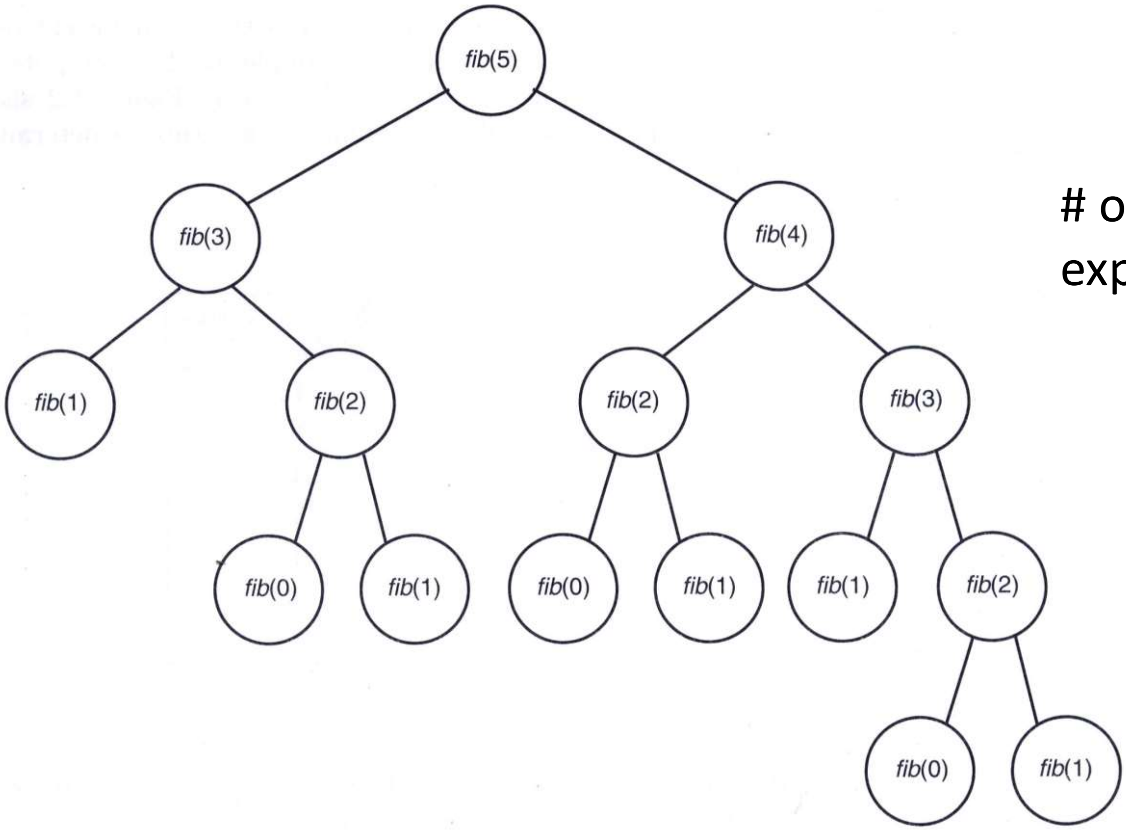
```
int fib(int n) {
  index i;
  int f[0 .. n];

  f[0] = 0;
  if (n > 0) {
    f[1] = 1;
    for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Time complexity: linear vs. exponential

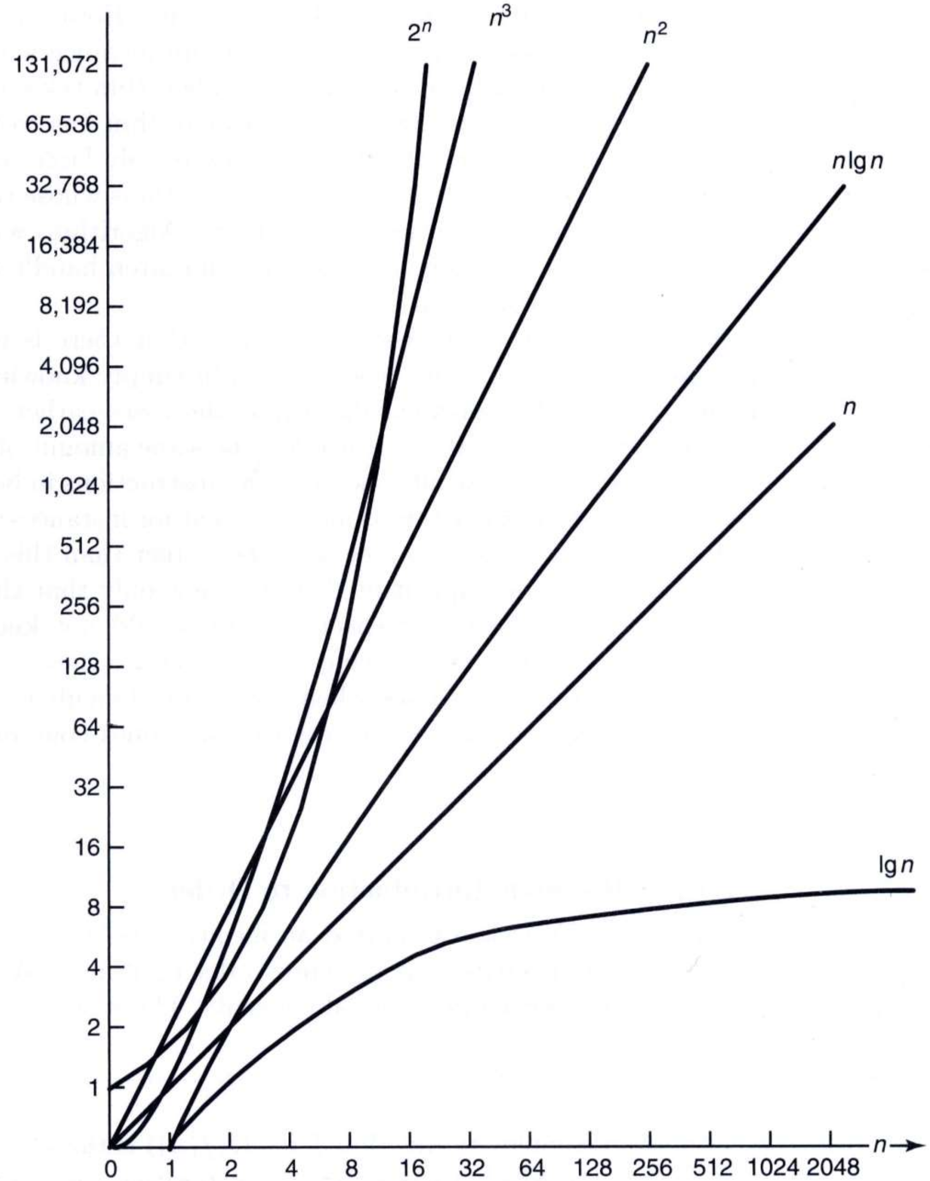
n	$n+1$	$2^{n/2}$	iterative Execution Time Using Algorithm 1.7	recursive (no data lookup) Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 $\mu s^\dagger$
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

Why is recursive algorithm less efficient?



# of recursion increases exponentially

Growth rates of some common complexity functions



서강대학교
SOGANG UNIVERSITY

Asymptotic Time Complexity of Programs

```
x = x + 1;
for (i = 1; i <= n; i++)
  y = y + 2;
for (i = n; i >=1; i--)
  for (j = n; j >= 1; j--)
     z = z + 1;
```

Time complexity: $c_0 + c_1 n + c_2 n^2 = O(n^2)$

```
c = 0; // n > 0
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    for (k = 1; k <= n; k = k*2)
      c += 2;
```

Time complexity: $c(\lceil \log_2 n \rceil + 1) * n * n = O(n^2 \log n)$

```
i = 1; j = 1; m = 0; // n > 0
while (j <= n) {
  i++;
  j = j + i;
  m = m + 2;
}
```

Time complexity: $??? = O(\sqrt{n})$

서강대학교 SOGANG UNIVERSITY

Execution times for algorithms with the given time complexities

n	logarithmic $f(n) = \lg n$	linear $f(n) = n$	n-log-n $f(n) = n \lg n$	quadratic $f(n) = n^2$	cubic $f(n) = n^3$	exponential $f(n) = 2^n$ $< n!$ (factorial)
10	0.003 μs^*	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 $\text{ms}^\dagger$
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years	
10^7	0.023 μs	0.01 s	2.660 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.660 s	115.70 days	3.17×10^7 years	
10^9	0.030 μs	1.00 s	29.900 s	31.70 years		

constant

Binary search

Finding the maximum

Merging two sorted lists

Merge sort

Finding the closest pair of points

Bubble sort

Finding all-pairs shortest path

Many intractable combinatorial problems

Polynomial ↔ Exponential

Algorithms and their time complexities

Notation	Name	Example
$\mathcal{O}(1)$	constant	Determining if a number is even or odd
$\mathcal{O}(\log^* n)$	iterated logarithmic	The `find` algorithm of Hopcroft and Ullman on a disjoint set
$\mathcal{O}(\log n)$	logarithmic	Finding an item in a sorted list with the binary search algorithm
$\mathcal{O}((\log n)^c)$	polylogarithmic	Deciding if n is prime with the AKS primality test
$\mathcal{O}(n^c), 0 < c < 1$	fractional power	searching in a kd-tree
$\mathcal{O}(n)$	linear	Finding an item in an unsorted list
$\mathcal{O}(n \log n)$	linearithmic, loglinear, or quasilinear	Sorting a list with heapsort, computing a FFT
$\mathcal{O}(n^2)$	quadratic	Sorting a list with insertion sort, computing a DFT
$\mathcal{O}(n^c), c > 1$	polynomial, sometimes called algebraic	Finding the shortest path on a weighted digraph with the Floyd-Warshall algorithm
$\mathcal{O}(c^n)$	exponential, sometimes called geometric	Finding the (exact) solution to the traveling salesman problem (under the assumption that P ≠ NP)
$\mathcal{O}(n!)$	factorial, sometimes called combinatorial	Determining if two logical statements are equivalent [1], traveling salesman problem, or any other NP Complete problem via brute-force search
$\mathcal{O}(2^{c^n})$	double exponential	Finding a complete set of associative-commutative unifiers [2]

Worst-case vs. average-case time complexity

S_n: the set of all inputs of size n

$c(I)$: the cost of the algorithm on input I

$p(I)$: the probability that input I occurs

- Worst-case complexity

$$T_W(n) = \max\{c(I) \mid I \in S_n\}$$

- Average-case complexity

$$T_A(n) = \Sigma_{I \in S_n} p(I) \cdot c(I)$$

- What is the worst-case and average-case complexity of sequential search?

Reviews - Summation

- Sums of powers

  - $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  - $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$
  - $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^{n} i\right]^2$
  - $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
  - $\sum_{i=0}^{n} i^s = \frac{(n+1)^{s+1}}{s+1} + \sum_{k=1}^{s} \frac{B_k}{s-k+1}\binom{s}{k}(n+1)^{s-k+1}$

    where B_k is the kth Bernoulli number.
  - $\sum_{i=1}^{\infty} i^{-s} = \prod_{p \text{ prime}} \frac{1}{1-p^{-s}} = \zeta(s)$

    where $\zeta(s)$ is the Reimann zeta function.

- Growth rates

  - $\sum_{i=1}^{n} i^c \in \Theta(n^{c+1})$ for real c greater than -1
  - $\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$
  - $\sum_{i=1}^{n} c^i \in \Theta(c^n)$ for real c greater than 1
  - $\sum_{i=1}^{n} \log(i)^c \in \Theta(n \cdot \log(n)^c)$ for nonnegative real c
  - $\sum_{i=1}^{n} \log(i)^c \cdot i^d \in \Theta(n^{d+1} \cdot \log(n)^c)$ for nonnegative real c, d
  - $\sum_{i=1}^{n} \log(i)^c \cdot i^d \cdot b^i \in \Theta(n^d \cdot \log(n)^c \cdot b^n)$ for nonnegative real $b > 1$, c, d

Comparing orders of growth

- How do you compare orders of growth of two functions?
  - One possible way is to compute the limit of the ratio of two functions in question.

$$\boxed{x = \lim_{n\to\infty} \frac{f_1(n)}{f_2(n)}}$$

- If $x = 0$, f_1 has a smaller order of growth than f_2.
- If $x = c$, f_1 has the same order of growth as f_2.
- If $x = \infty$, f_1 has a larger order of growth than f_2.

  - Ex. 1: $\log_2 n$ vs $\sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = ?$$

  - Ex. 2: $n!$ vs 2^n

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}(\frac{n}{e})^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = ?$$

$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ for large value of n: **Stirling's formula**

Reviews – run time analysis

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[i][j] = b[i][j] + c[i][j];
```

$O(n^2)$

```
for (i = 1; i <= n; i++)
  if (i % 2 == 0) a[i] = 1;
  else a[i] = -1;
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    b[i][j] = i + j;
```

$O(n^2)$

```
for (i = 1; i <= n; i++) {
  if (i % 2) {
    for (j = 1; j <= n; j++)
        a[i][j] = i + j;
  }
  else {
    for (j = 1; j <= n; j++) {
      a[i][j] = 0;
      for (k = 1; k <= n; k++)
        a[i][j] += k;
    }
  }
}
```

$O(n^3)$

```
x = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= i; j++)
    x += i + j;
```

$O(n^2)$

```
x = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= i; j++)
    for (k = 1; k <= j; k++)
      x += i + j + k;
```

$O(n^3)$

```
x = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= i*i; j++)
    if (j % i == 0)
      for (k = 1; k <= j; k++)
        x++;
```

$O(n^4)$

What is the worst-case time complexity of each loop?

```
// n = 2^k for some positive
//               integer k
for (i = 1; i < n; i++) {
  j = n;
  while (j >= 1) {
    // some O(1) computation
    j = j/2;
  }
}
```

$O(n \lg n)$

```
// float x[n][n+1];
for (i = 0; i <= n-2; i++)
  for (j = i+1; j <= n-1; j++)
    for (k = i; k <= n; k++)
      x[j][k] = x[j][k] - x[i][k]*x[j][i]/x[i][i];
```

$O(n^3)$

```
// n = 2^k for some positive
//               integer k
i = n;
while (i >= 1) {
  j = i;
  while (j <= n) {
    // some O(1) computation
    j = 2*j;
  }
  i = i/2;
}
```

$O(\lg^2 n)$

Magic square

```
// n: odd integer
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    s[i][j] = 0;
s[0][(n-1)/2] = 1;
j = (n-1)/2;
for (key = 2; key <= n*n; key++) {
  k = (i) ? (i-1) : (n-1);
  l = (j) ? (j-1) : (n-1);
  if (s[k][l]) i = (i+1)%n;
  else {
    i = k; j = l;
  }
  s[i][j] = key;
}
```

$O(n^2)$

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

What is the worst-case time complexity of each loop?

Algorithm design example

- Maximum Subsequence Sum (MSS) problem

  Given N (possibly negative) integers $A_0, A_1, \cdots, A_{N-1}$, find the maximum value of $\sum_{k=i}^{j} A_k$ for $0 \leq i \leq j \leq N-1$. (For convenience, the maximum subsequence sum is 0 if all the integers are negative.)

- Example: (-2, 11, -4, 13, -5 -2) → MSS = 20

Maximum Subsequence Sum – Algorithm 1

- Strategy
  - Enumerate all possibilities one at a time.
  - No efficiency is considered, resulting in a lot of unnecessary computation!

```
int
MaxSubsequenceSum( const int A[ ], int N )
{
    int ThisSum, MaxSum, i, j, k;

    MaxSum = 0;
    for( i = 0; i < N; i++ )
        for( j = i; j < N; j++ )
        {
            ThisSum = 0;
            for( k = i; k <= j; k++ )
                ThisSum += A[ k ];

            if( ThisSum > MaxSum )
                MaxSum = ThisSum;
        }
    return MaxSum;
}
```

Is this for-loop OK for you?

$$\boxed{O(N^3)}$$

$$\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j} 1 = \frac{N^3+3N^2+2N}{6}$$

$$\sum_{j=i}^{N-1}(j-i+1) = \frac{(N-i+1)(N-i)}{2}$$

$$\sum_{k=i}^{j} 1 = j-i+1$$

Maximum Subsequence Sum – Algorithm 2

- Strategy
  - Get rid of the inefficiency in the innermost for-loop.
  - Notice that $\displaystyle\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k.$

```
int
MaxSubSequenceSum( const int A[ ], int N )
{
    int ThisSum, MaxSum, i, j;

    MaxSum = 0;
    for( i = 0; i < N; i++ )
    {
        ThisSum = 0;
        for( j = i; j < N; j++ )
        {
            ThisSum += A[ j ];

            if( ThisSum > MaxSum )
                MaxSum = ThisSum;
        }
    }
    return MaxSum;
}
```

$$\sum_{i=0}^{N-1}\sum_{j=i}^{N-1} 1 = ???$$

$\boxed{O(N^2)}$

Maximum Subsequence Sum – Algorithm 3

- Strategy
  - Use the **Divide-and-Conquer** strategy.
  - The maximum subsequence sum can be in one of three places.

$\boxed{O(N \log N)}$ ← why?

```
        static int
        MaxSubSum( const int A[ ], int Left, int Right )
        {
            int MaxLeftSum, MaxRightSum;
            int MaxLeftBorderSum, MaxRightBorderSum;
            int LeftBorderSum, RightBorderSum;
            int Center, i;

/* 1*/      if( Left == Right )  /* Base Case */
/* 2*/          if( A[ Left ] > 0 )
/* 3*/              return A[ Left ];
                else
/* 4*/              return 0;

/* 5*/      Center = ( Left + Right ) / 2;
/* 6*/      MaxLeftSum = MaxSubSum( A, Left, Center );
/* 7*/      MaxRightSum = MaxSubSum( A, Center + 1, Right );

/* 8*/      MaxLeftBorderSum = 0; LeftBorderSum = 0;
/* 9*/      for( i = Center; i >= Left; i-- )
            {
/*10*/          LeftBorderSum += A[ i ];
/*11*/          if( LeftBorderSum > MaxLeftBorderSum )
/*12*/              MaxLeftBorderSum = LeftBorderSum;
            }

/*13*/      MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/      for( i = Center + 1; i <= Right; i++ )
            {
/*15*/          RightBorderSum += A[ i ];
/*16*/          if( RightBorderSum > MaxRightBorderSum )
/*17*/              MaxRightBorderSum = RightBorderSum;
            }

/*18*/      return Max3( MaxLeftSum, MaxRightSum,
/*19*/                   MaxLeftBorderSum + MaxRightBorderSum );
        }

        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            return MaxSubSum( A, 0, N - 1 );
        }
```

Maximum Subsequence Sum – Algorithm 4

- Strategy
  - Use the **Dynamic Programming** strategy.
  - Idea

    $B[i]$: the sum of a maximum subsequence that ends at index i

    $$\longrightarrow B[i] = \max\{\, B[i-1] + A[i],\ A[i]\,\}$$

```
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, j;

/* 1*/      ThisSum = MaxSum = 0;
/* 2*/      for( j = 0; j < N; j++ )
            {
/* 3*/          ThisSum += A[ j ];

/* 4*/          if( ThisSum > MaxSum )
/* 5*/              MaxSum = ThisSum;
/* 6*/          else if( ThisSum < 0 )
/* 7*/              ThisSum = 0;
            }
/* 8*/      return MaxSum;
        }
```

$$\boxed{O(N)}$$

Why is complexity important?

Figure 2.2 Running times of several algorithms for maximum subsequence sum (in seconds)

Algorithm		1	2	3	4
Time		$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
Input Size	$N = 10$	0.00103	0.00045	0.00066	0.00034
	$N = 100$	0.47015	0.01112	0.00486	0.00063
	$N = 1{,}000$	448.77	1.1233	0.05843	0.00333
	$N = 10{,}000$	NA	111.13	0.68631	0.03042
	$N = 100{,}000$	NA	NA	8.0113	0.29832

Maximum Sum Subrectangle in a 2D array

- Problem
  - Given an m x n array of integers, find a subrectangle with the largest sum. (In this problem, we assume that a subrectangle is any contiguous sub-array of size 1x1 or greater located within the whole array.)

- Note
  - What is the input size of this problem?
  - How many subrectangles are there in an m x n array?
  - For the case of m = n,
    - Design an $O(n^4)$ algorithm.
    - Design an $O(n^3)$ algorithm.

0	**-2**	**-7**	**0**
9	**2**	**-6**	**2**
-4	**1**	**-4**	**1**
-1	**8**	**0**	**-2**

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr