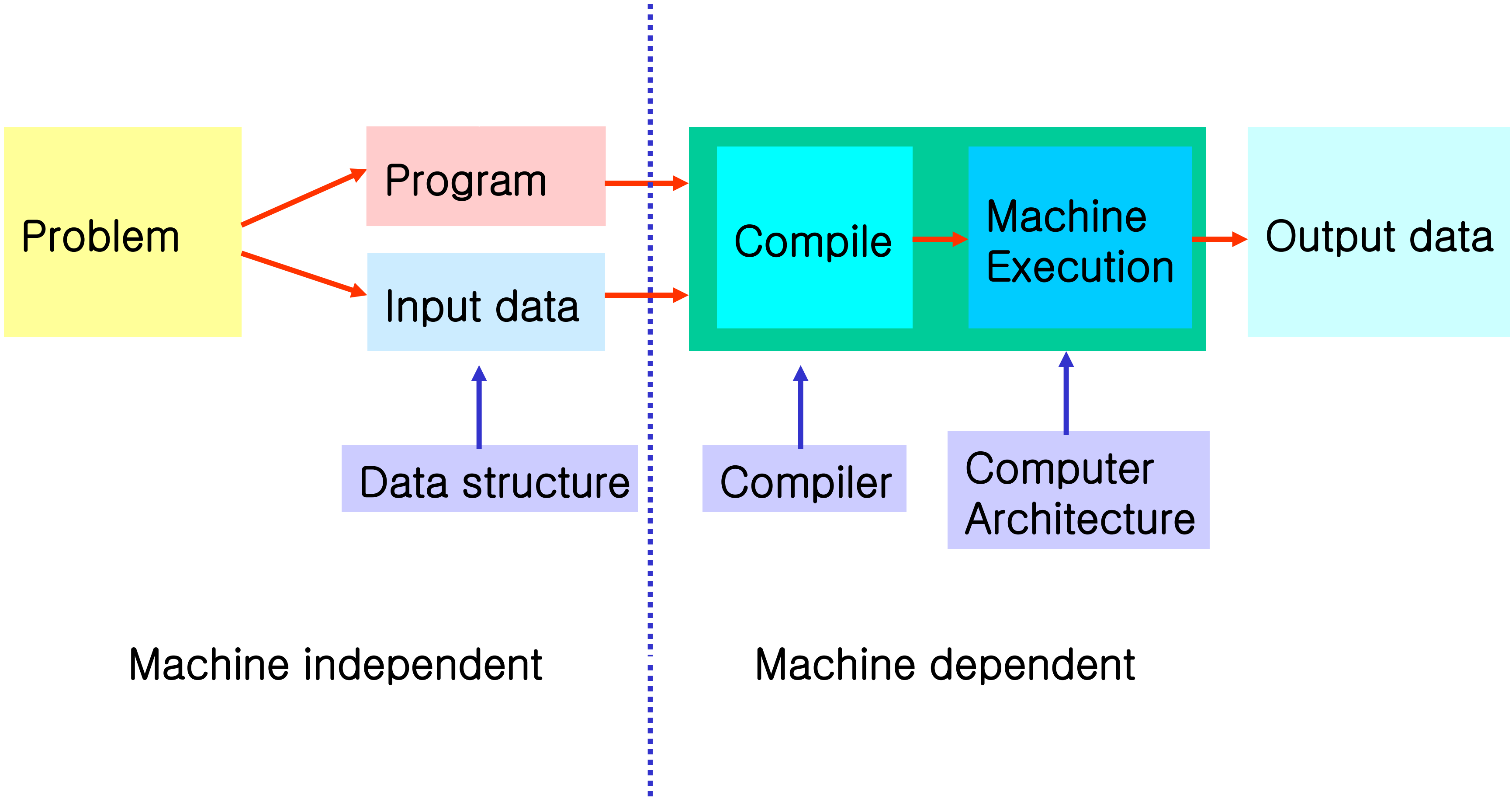
Data Structures and Algorithms

School of Electrical Engineering
Korea University

Analysis of algorithms (Chap. 1-2)

School of Electrical Engineering
Korea University

Overview


Problem
Program
Input data
Compile
Machine Execution
Output data
Data structure
Compiler
Computer Architecture
Machine independent
Machine dependent

Algorithm

- Algorithm definition
  - A finite sequence of instructions to solve a specific problem
  - Each instruction should be finished within a finite amount of time
  - Amount of resources such as time or space for the execution

Algorithm

- Algorithm description tools
  - English statements
  - Pseudo code
  - Programming language

Algorithm

- Algorithm Design Technique
  - Divide-and-Conquer
  - Heuristics
  - Dynamic programming
  - Backtracking
  - Branch and bound

Procedure of writing a program

1. Problem specification
2. Understanding the problem
3. Thinking about how to solve it
4. Writing code with input data
5. Repeat run & revise

실행시간을 짧게 하려면

- 고속 컴퓨터
- 다수의 컴퓨터
- 우수한 프로그램
- 효과적인 자료저장, 추출방식

등이 확보 되도록

Two (resource) issues

- Running time (execution time)
  seconds, minutes, hours,...
- Space(memory) requirement
  Kbytes, Mbytes, Gbytes,...

Better Programs

- Shorter estimated running time to solve the problem

- Way to access data

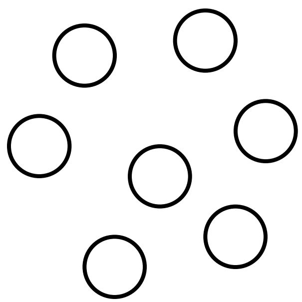
  O(N), O(log N)

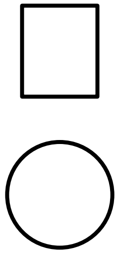
- Readability / easy debugging

How to get running times?

- Experiments
- Theoretical-mathematical analysis by estimation / counting
  - Topic of this chapter.

Analysis of Algorithms!





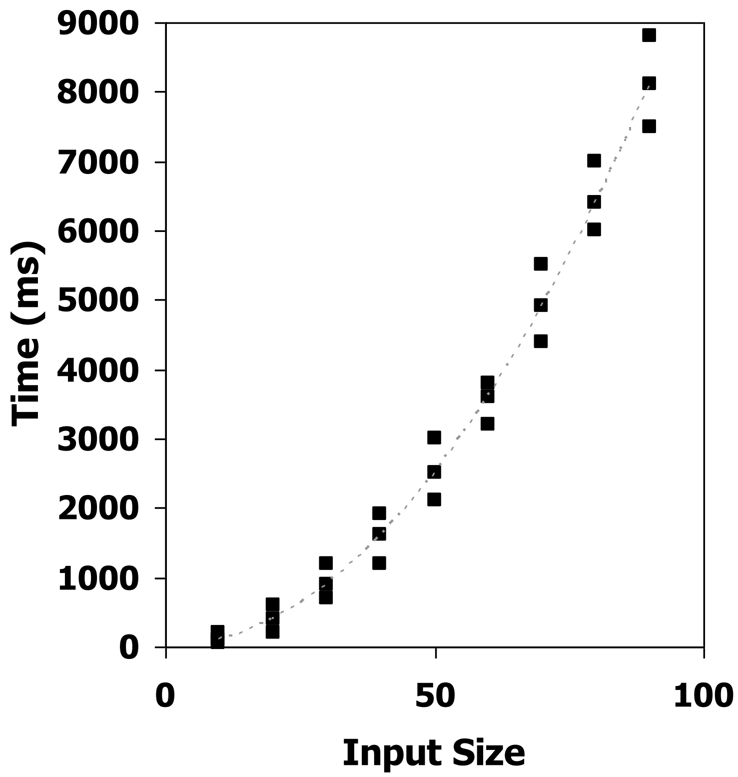
Input **Algorithm** **Output**

Running times

- Linear – N
- Logarithmic – log N
- Polynomial – $\sum b_k N^k = b_0 + b_1 N + b_2 N^2 + \ldots$
- Exponential – a^N
- (in between) – $N\sqrt{N}, \log^2 N$

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like system.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- Have to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on the inputs which are not considered in the experiment.
- For fair comparison of two algorithms, the same hardware and software environments should be used

Input-dependent exec. time

- Best-case exec. time: minimum
- Worst -case exec. time : maximum
- Avg.-case exec. time : in the middle

Pseudo-code

- High-level description of an algorithm
- More structured than English prose
- Less detailed than actual program
- Preferred notation for describing algorithms
- Hides program design details

Ex: find max of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

  currentMax ← ***A***[0]

  for ***i*** ← 1 **to** ***n*** – 1 **do**
    if *A*[*i*] > ***currentMax*** **then**
      currentMax ← ***A***[***i***]
  return ***currentMax***

Pseudo-code Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces

- Method declaration

  Algorithm *method* (*arg* [, *arg*…])

  Input …

  Output …

Pseudo-code Details

- Method call

  var.method **(*arg*** [, ***arg***…]**)**

- Return value

  return *expression*

- Expressions

  $\leftarrow$ Assignment (like = in Java)

  $=$ Equality testing (like == in Java)

  $\boldsymbol{n}^2$ Superscripts and other mathematical formatting allowed

Growth Rate of Running Time

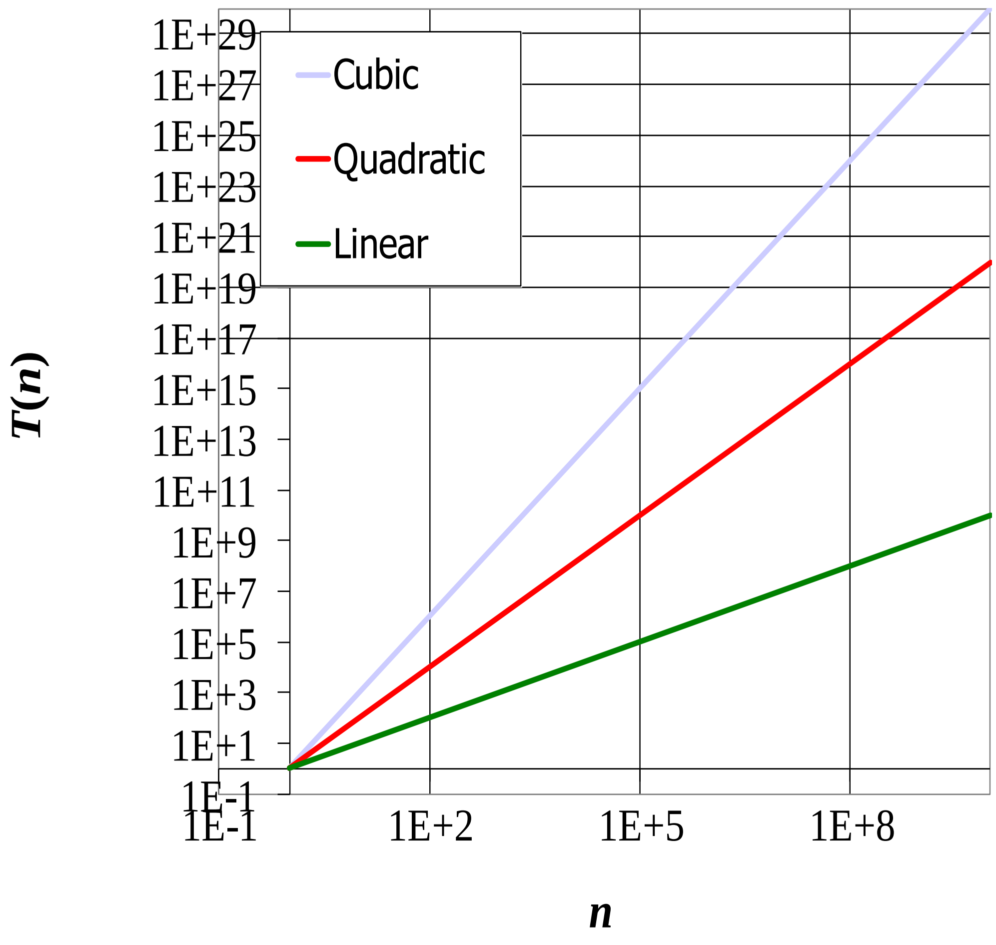
- Changing the hardware/ software environment
  - affects $T(n)$ by a constant factor, but
  - does not alter the growth rate of $T(n)$

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***

Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function

Growth Rates



Notations

- Focus on order of magnitude

  $$O(N), O(\log N), O(N \log N)$$

- *Big Oh, Theta, Omega*

  $$O \qquad \Theta \qquad \Omega$$

- Constant is not significant

  1.5 N -> O(N)

  120 N -> O(N)

- Lower ordered terms are ignored

  $16\ N^3 + 6\ N$ -> $O(N^3)$

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \le c * g(n) \quad \text{for } n \ge n_0$$

(Ex) $2n + 10$ is $O(n)$

Proof: $2n + 10 \le cn$

$(c - 2)\, n \ge 10$

$n \ge 10/(c - 2)$

$\therefore \quad c = 3$ and $n_0 = 10$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - ✓ Drop lower-order terms
  - ✓ Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- (Ex)
  - We determine that algorithm $\boldsymbol{arrayMax}$ executes at most $\mathbf{7}\boldsymbol{n}-\mathbf{1}$ primitive operations
  - We say that algorithm $\boldsymbol{arrayMax}$ "runs in $\boldsymbol{O(n)}$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Theoretical Analysis-alternative

- Uses a high-level description of the algorithm instead of an implementation
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Computation Model

- Artificial machine with basic arithmetic operations such as +, −, *, /
- All with the same computing time – one unit (second?)
- How real ? -> asymptotic(for big N)

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important

(Ex)

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the max number of primitive operations executed by an algorithm, as a function of the input size

	# operations
Algorithm ***arrayMax***(A, n)	
$currentMax \leftarrow A[0]$	1
for $i \leftarrow 1$ **to** $n - 1$ **do**	n
if $A[i] > currentMax$ **then**	$(n - 1)$
$currentMax \leftarrow A[i]$	$(n - 1)$
return $currentMax$	1
Total	$3n$

예제: $\sum_{k=1}^{N} k^3$

```
int Sum (int N) {
    int i, PartialSum;

    PartialSum = 0;
    for (i=1; i<= N; i++)
            PartialSum += i * i * i;
    return PartialSum;
}
```

Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	**#operations**
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Simple Algorithms

- Finding a maximum(or min.)

$$T(N) = O(N)$$

- Sort

$$T(N) = O(N \log N)$$

- Binary Search

$$T(N) = O(\log N)$$

Binary Search

- Algorithm – very fundamental, very *important!*
  listed in Figure 2.9 on page 30

- Running time

$$T(N) = O(\log N)$$

Algorithm

- Algorithm efficiency in terms of
  - Time complexity
  - Space complexity
- Issues
  - How to estimate the time required for a program
  - How to reduce the running time of a program
  - The results of careless use of recursion

Time complexity

- Algorithm used
- Input size
- T(n) : function on input size n
  - $T_{avg}(N)$: Average running time
  - $T_{worst}(N)$: Worst running time
- $T_{avg}(N)$ often reflects typical behavior
- $T_{worst}(N)$ represents a guarantee for performance on any possible input

Definitions

- Establish a relative order among functions
- We compare relative rates of growth

1. $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c * f(N)$ when $N \geq n_0$ (Big-Oh)
   - The growth rate of $T(N)$ is less than or equal to that of $f(N)$

2. $T(n) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c * g(N)$ when $N \geq n_0$ (Omega)
   - The growth rate of $T(N)$ is greater than or equal to that of $g(N)$

Meaning

3. $T(N) = \Theta\ (h(N))$ if and only if $T(n) = O\ (h(N))$ and $T(n) = \Omega(h(N))$ (Theta)
   – The growth rate of $T(N)$ equals the growth rate of $h(N)$

4. $T(N) = o\ (p\ (N))$ if $T(n) = O\ (p(N))$ and $T(n) \neq \Theta\ (p(N))$ (Little-oh)
   – The growth rate of $T(N)$ is less than the growth rate of $p(N)$.

Algorithm analysis

- Definition 1 :

$T(n) = O(f(n))$ if there are positive constants c and n_0

such that $T(n) \leq c * f(n)$ for $n \geq n_0$

Says that eventually there is some point n past which $c * f(n)$ is always at least as large as $T(n)$, so that if constant factors are ignored, $f(n)$ is at least as big as $T(n)$.

Ex 1: $T(n) = (n+1)^2 = O(n^2)$

Ex 2: $T(n) = 3n^3 + 2n^2 = O(n^3)$

Algorithm analysis

Ex 1: $T(n) = (n+1)^2 = O(n^2)$

<proof>

Show $T(n) = (n+1)^2 \le c*n^2$ for some c and $n \ge n_0$

For c =4, $(n+1)^2 \le 4n^2$

$3n^2 - 2n - 1 \ge 0$

$n \ge 1 \rightarrow n_0 = 1$

$T(n) = (n+1)^2 \le 4n^2$ for c =4, $n \ge 1$

Algorithm analysis

Ex 2: $T(n) = 3n^3 + 2n^2 = O(n^3)$

<proof>

Show $T(n) = 3n^3 + 2n^2 \leq cn^3$ for some c and $n \geq n_0$

For c =4, $3n^3 + 2n^2 \leq 4n^3$

$n^3 - 2n^2 \geq 0$

$n \geq 2 \rightarrow n_0 = 2$

$T(n) = 3n^3 + 2n^2 \leq 4n^3$ for c =4, $n \geq 2$

Algorithm analysis

- Definition 2 :

  $T(n) = \Omega(f(n))$ if there are positive constants c and n_0
  such that $T(n) \geq c * f(n)$ for $n \geq n_0$

Ex 4: $T(n) = n^3 + 2n^2 = \Omega(n^3)$

<Proof>

Show $T(n) = n^3 + 2n^2 \geq cn^3$ for $c, n \geq n_0$

For $c = 1$, $n^3 + 2n^2 \geq n^3$

$2n^2 \geq 0 \rightarrow n \geq 1$

$T(n) = n^3 + 2n^2 \geq n^3$ for $c=1, n \geq 1$

Algorithm analysis

- Definition 3 :

  $T(n)$ is $\Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

- Definition 4 :

  $T(n)$ is $o(f(n))$ if $T(n) = O(f(n))$ and $T(n) \neq \Theta(f(n))$

Algorithm analysis

- Time complexity comparison
  - Relative growth rate for large n
  - L'Hospital's rule: $\lim_{n \to \infty} f(n) / g(n)$

    0 : $f(n) < g(n)$ for large n → $f(n) = O(g(n))$, $o(g(n))$

    c : $f(n) = c * g(n)$ for large n → $f(n) = \theta(g(n))$

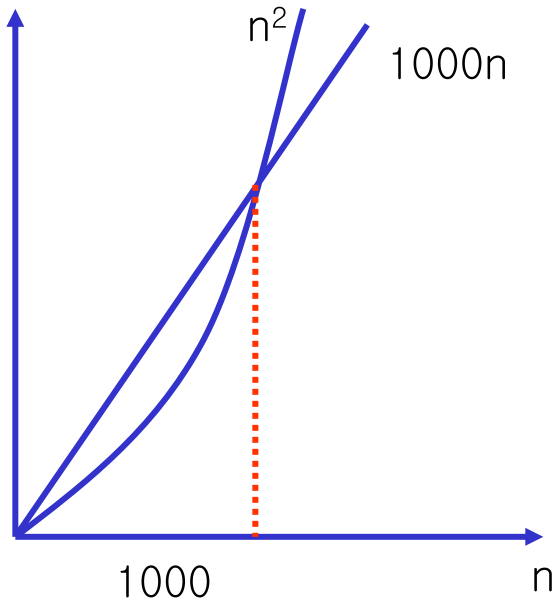
    ∞ : $f(n) > g(n)$ for large n → $f(n) = \Omega(g(n))$, $g(n) = o(f(n))$

    oscillate : no relation

Ex) $2^n > n^3 > n^2 \log n > n^2 > n \log n > n > \log^3 n > \log n > c$

Algorithm analysis

- Constant factor
  - $O(n^2)$ vs $O(1000n)$



$n \leq 1000 \rightarrow O(n^2)$

$n > 1000 \rightarrow O(1000n)$

For $n \rightarrow \infty$, $O(n^2) > O(n)$

Growth rates of typical functions

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates

Maximum subsequence sum algorithms

- Given (possibly negative) integers $\mathbf{A_1, A_2, \ldots, A_N}$, find the maximum value of $\sum_{k=i}^{j} \mathbf{A}_k$. (For convenience, the maximum subsequence sum is 0 if all the integers are negative.)

(Ex) −2, 11, −4, 13, −5, −2

The answer is 20 (subsequence 11, −4, 13)

Maximum subsequence sum algorithm 1

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

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i < N; i++ )
/* 3*/          for( j = i; j < N; j++ )
                {
/* 4*/              ThisSum = 0;
/* 5*/              for( k = i; k <= j; k++ )
/* 6*/                  ThisSum += A[ k ];

/* 7*/              if( ThisSum > MaxSum )
/* 8*/                  MaxSum = ThisSum;
                }
/* 9*/      return MaxSum;
        }
```

Maximum subsequence sum algorithm 2

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

/* 1*/      MaxSum = 0
/* 2*/      for( i = 0; i < N; i++ )
            {
/* 3*/          ThisSum = 0;
/* 4*/          for( j = i; j < N; j++ )
                {
/* 5*/              ThisSum += A[ j ];

/* 6*/              if( ThisSum > MaxSum )
/* 7*/                  MaxSum = ThisSum;
                }
            }
/* 8*/      return MaxSum;
        }
```

Maximum subsequence sum algorithm 3

```
        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;
```

Maximum subsequence sum algorithm 3

```
        MaxRightBorderSum = 0; RightBorderSum = 0;
        for( i = Center + 1; i <= Right; i++ )
        {
            RightBorderSum += A[ i ];
            if( RightBorderSum > MaxRightBorderSum )
                MaxRightBorderSum = RightBorderSum;
        }

        return Max3( MaxLeftSum, MaxRightSum,
                MaxLeftBorderSum + MaxRightBorderSum );
    }

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

Maximum subsequence sum algorithms 4

```
        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;
        }
```

Maximum subsequence sum algorithms

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

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

Maximum subsequence sum algorithms

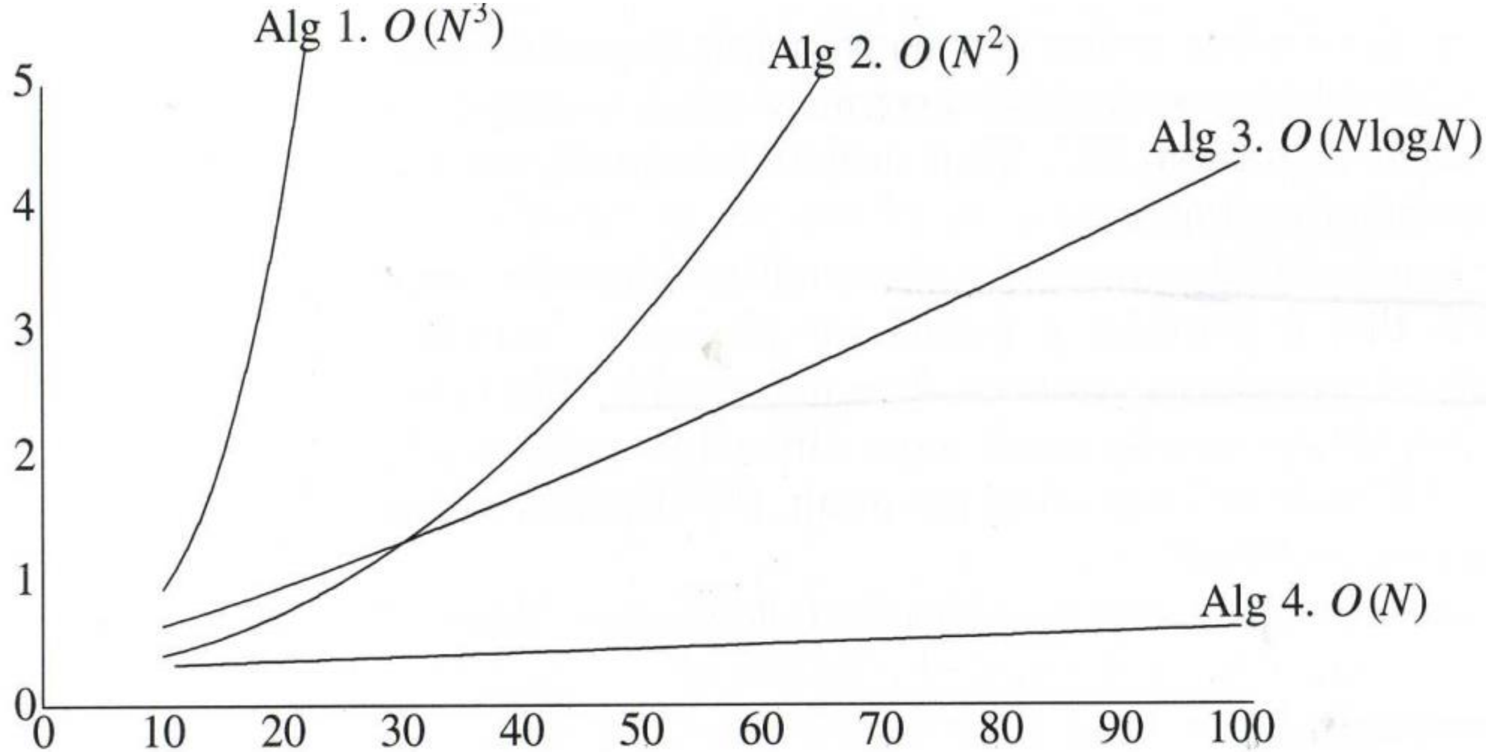


Figure 2.3 Plot (*N* vs. milliseconds) of various maximum subsequence sum algorithms

Maximum subsequence sum algorithms

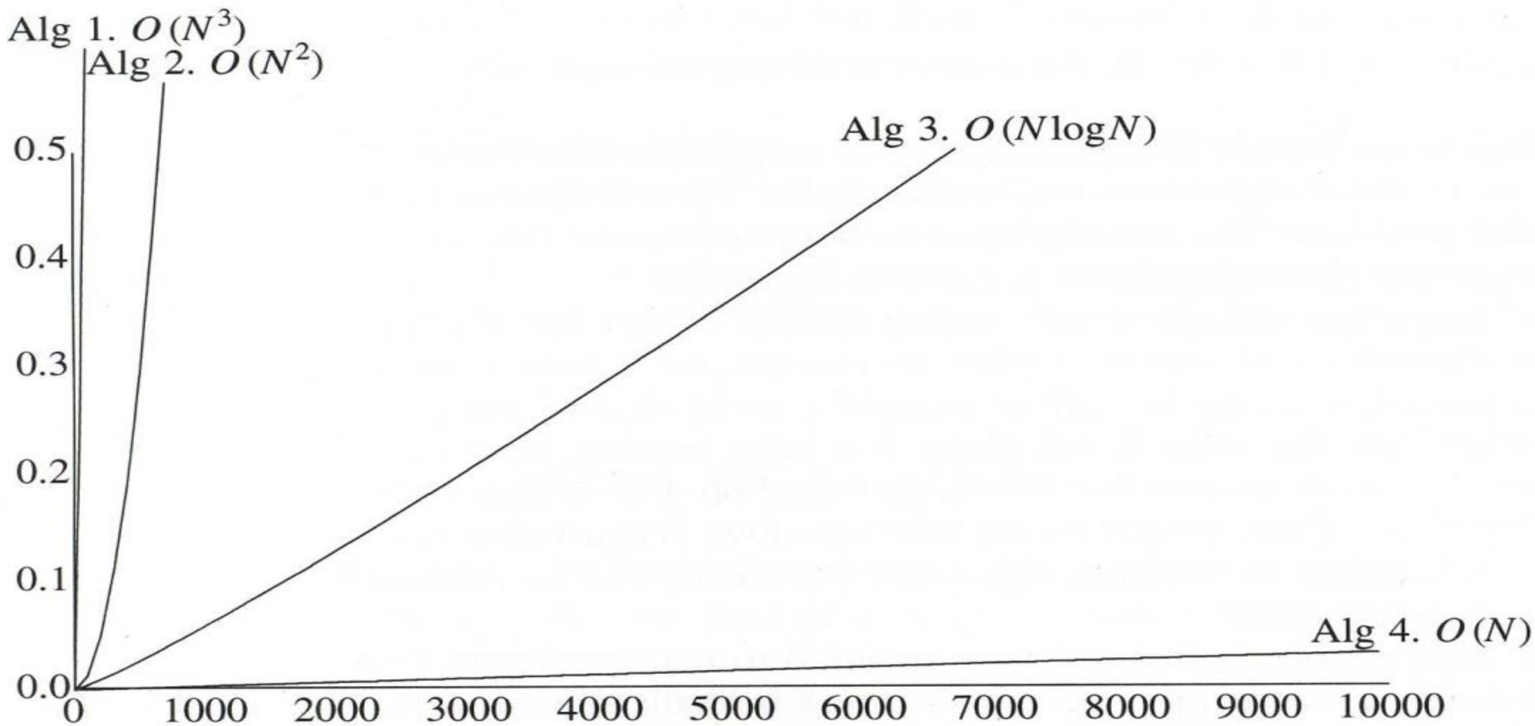


Figure 2.4 Plot (N vs. seconds) of various maximum subsequence sum algorithms

Algorithm analysis

Property 1: If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$,

$$T_1(n) + T_2(n) = O(\max(f(n), g(n))$$

<proof> By definition,

$T_1(n) \leq c_1 f(n)$ for c_1, $n > n_1$

$T_2(n) \leq c_2 f(n)$ for c_2, $n > n_2$

$T_1(n) + T_2(n)$

$\leq c_1 f(n) + c_2 f(n)$

$\leq c_1 \max(f(n), g(n)) + c_2 \max(f(n), g(n))$ for $n \geq \max(n_1, n_2)$

$\leq (c_1 + c_2) \max(f(n), g(n))$ for $n \geq \max(n_1, n_2)$

Algorithm analysis

Ex 7: $T(n) = T1(n) + T2(n) + T3(n) = O(n^2) + O(n^3) + O(n\log n)$

$T(n) =$

Ex 8: $T1(n) = n^4$ if n is even, n^2 if n is odd

$T2(n) = n^2$ if n is even, n^3 if n is odd

if n is even → $T(n) =$

if n is odd → $T(n) =$

Algorithm analysis

- Property 2: If $T(n) = O(f(n)+g(n))$ such that $g(n) \leq f(n)$ for all $n \geq n_0$

  $T(n) = O(f(n))$

  Ex) $T(n) = O(n^2 + n^3 + \log n)$

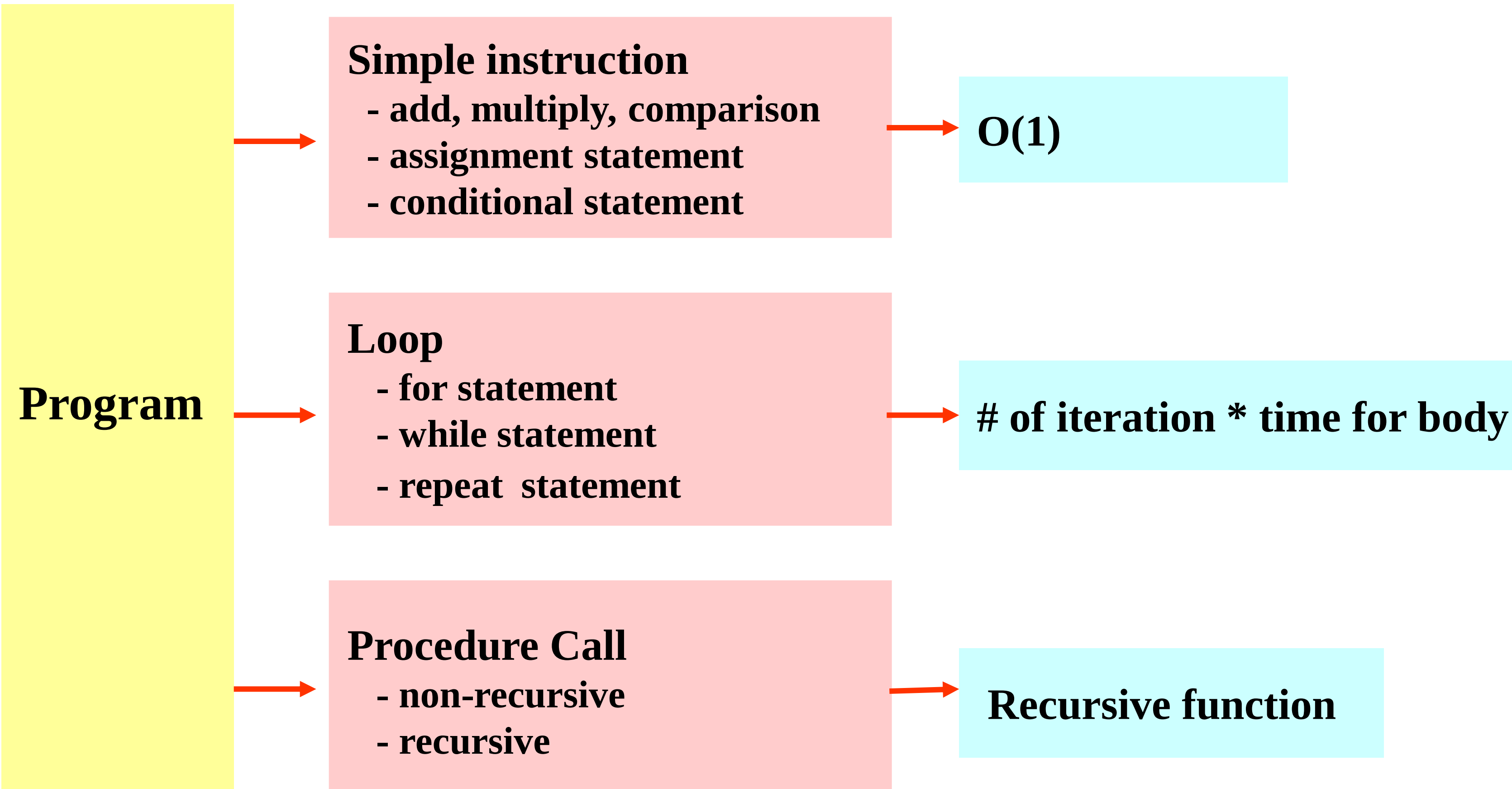
- Property 3: If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$,

  $T(n) = T_1(n) * T_2(n) = O(f(n) * g(n))$

- Property 4: $T(n) = O(c * f(n)) = O(f(n))$

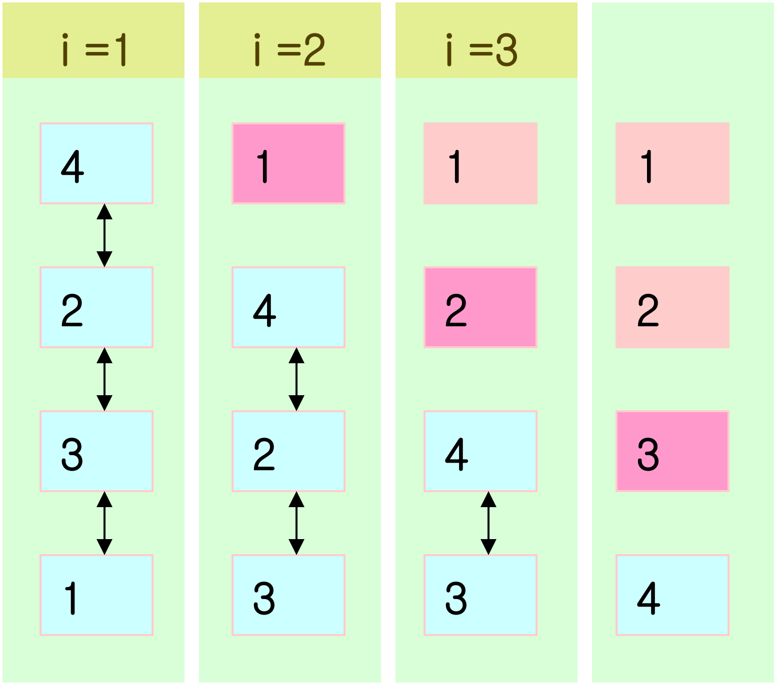
  Ex) $O(4/3n^3 + 1/2n^2 + 2) =$

Model of Computation



Bubble sort

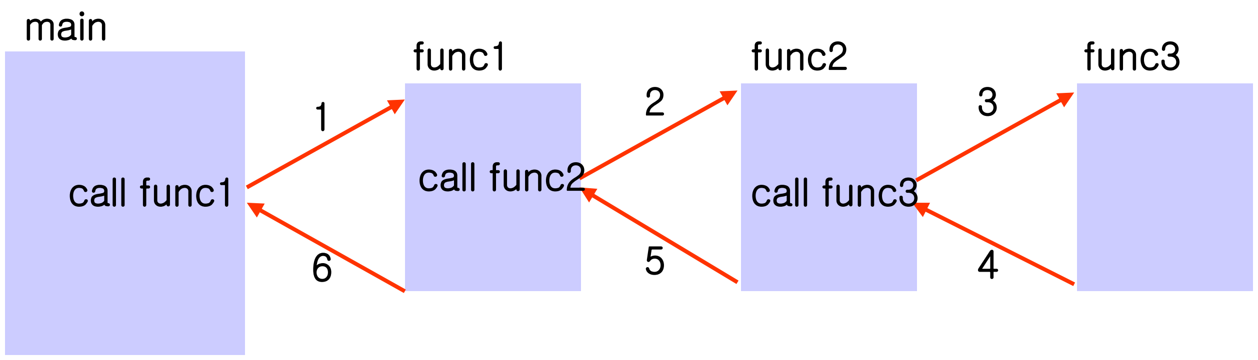
```
void Bubble (var A[1..n]) {
    int  i, j, temp;
    for (i=1; i ≤n-1; i++)
        for (j=n; j  ≥  i+1; j--)
            if A[j-1] > A[j] {
                temp = A[j-1];
                A[j-1] = A[j];
                A[j] = temp
            }
}
```



$$T(n) = \sum_{i=1}^{n-1} (n-i) * 1 = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$$

Function Call

- Non-recursive Call



◆ Recursive Call

$T(n) = f(T(k))$ for various value of k

Factorial

```
int fact ( int n) {
    if n <=1
        return 1
    else
        return n*fact(n-1)
}
```

$$T(n) = T(n-1) + c \quad \text{if } n > 1$$

$$= d \quad \text{if } n = 1$$

Factorial

$$T(n) = T(n-1) + c \quad \text{if } n > 1$$
$$= d \quad \text{if } n = 1$$

$T(n) = T(n-1) + c$

$= [T(n-2)+c]+c = T(n-2) + 2c$

$= [T(n-3)+c]+2c = T(n-3) + 3c$

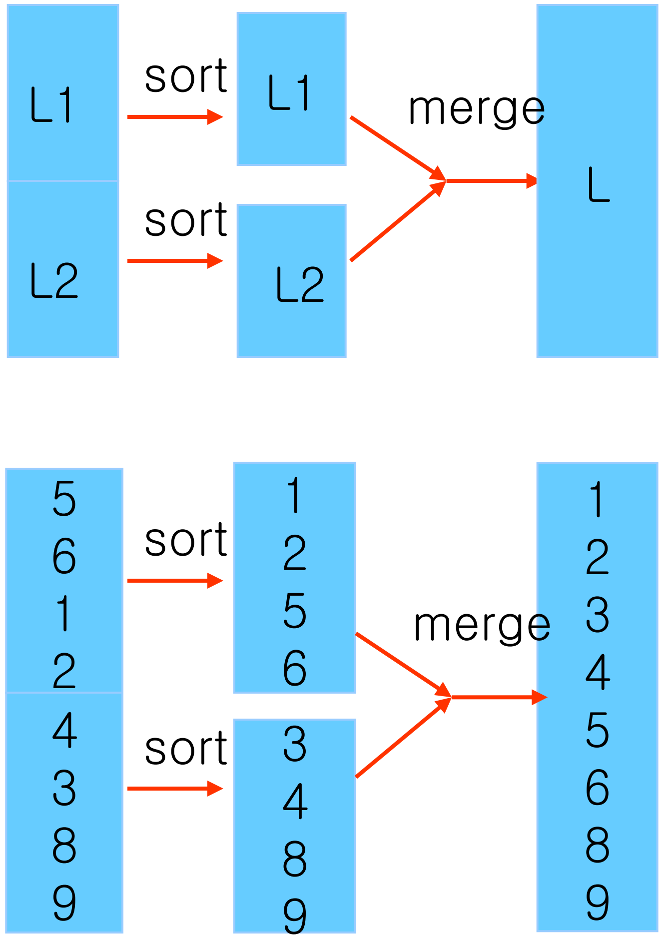
…..

$= T(1) + (n-1)c$

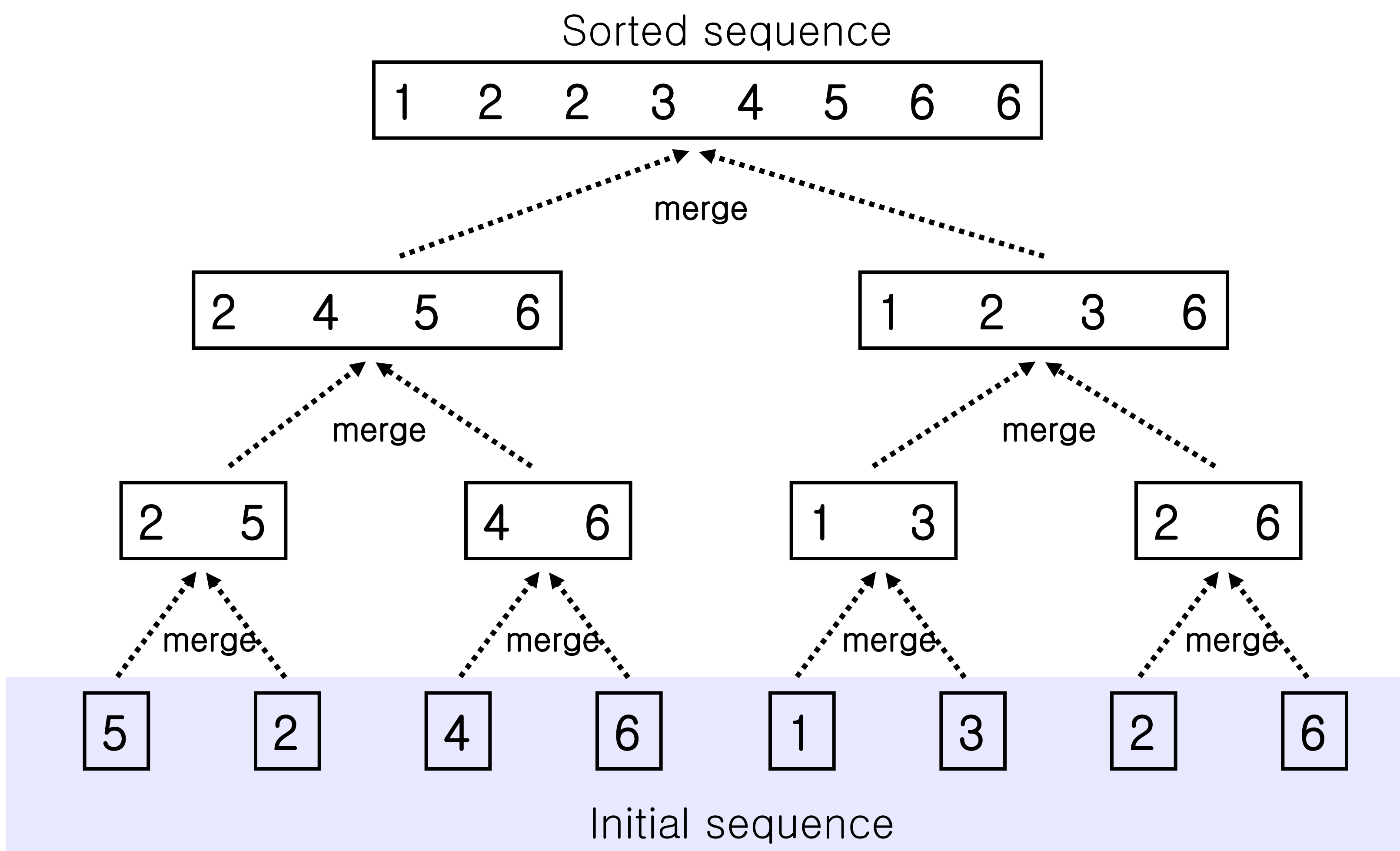
$= d + (n-1)\,c$

$= O(n)$

Mergesort


L1
L2
sort
sort
L1
L2
merge
L
5
6
1
2
4
3
8
9
sort
sort
1
2
5
6
3
4
8
9
merge
1
2
3
4
5
6
8
9

Example



Sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
Initial sequence

Merge-sort

```
void Merge-sort( L, n) {
    if n <=1
        return L
    else {
        Merge-sort (L1, n/2);
        Merge-sort (L2, n/2);
        return merge (L1, L2, n/2);
}
```

$$T(n) = 2T(n/2) + c_1 n \quad \text{if } n > 1$$

$$= c_2 \quad \text{if } n = 1$$

Merge-sort

$T(n) = 2T(n/2) + c_1 n$ if $n > 1$

$= c_2$ if $n = 1$

$T(n) = 2T(n/2) + c_1 n$

$= 2[2T(n/2^2) + c_1 n/2] + c_1 n = 2^2 T(n/2^2) + 2c_1 n$

$= 2^2[2T(n/2^3) + c_1 n/2^2] + 2c_1 n = 2^3 T(n/2^3) + 3c_1 n$

…

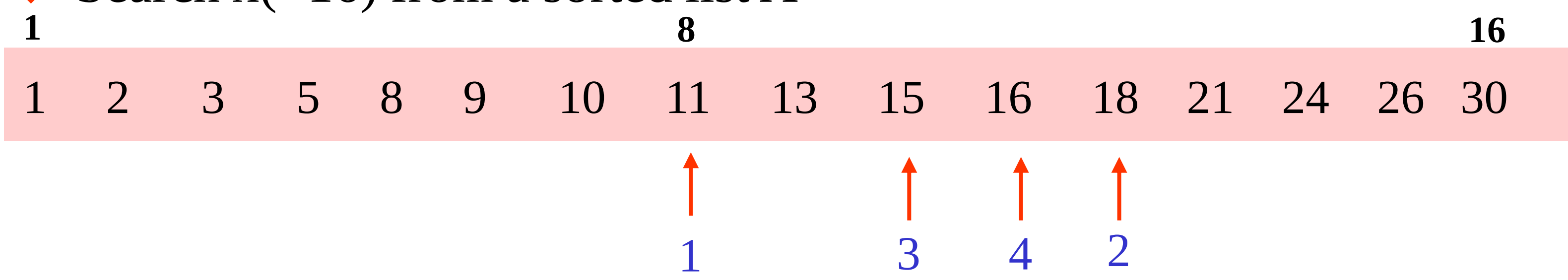
$= 2^r T(n/2^r) + r c_1 n \rightarrow n/2r = 1,\ n = 2r,\ r = \log n$

$= n c_2 + c_1 n\log n$

$= O(n\log n)$

Binary Search

- Search x(=16) from a sorted list A




```
int binary-search (A, low, high, x) {
    mid = (low+high)/2;
    if (A[mid]=x) then
        return mid
    else if (A[mid]>x) then
        binary-search(A,low,mid-1,x)
    else binary-search(A,mid+1,high,x)
    }
```

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

Binary Search

$$T(n) = T(n/2) + 1 \quad \text{if } n > 1$$
$$= 1 \quad \text{if } n = 1$$

$T(n) = T(n/2) + 1$

$= [T(n/2^2) + 1] + 1 = T(n/2^2) + 2$

$= [T(n/2^3) + 1] + 1 = T(n/2^3) + 3$

…

$= T(n/2^r) + r \rightarrow n/2^r = 1, \; n = 2^r, \; r = \log n$

$= T(1) + \log n = 1 + \log n$

$= O(\log n)$

GCD(Greatest Common divisor)

```
int GCD (M, N) {
    while (N!=0){
        rem = M % N;
        M   = N;
        N   = rem;
    }
    return M
}
```

M	N	rem
36	15	6
15	6	3
6	3	0
3	0	

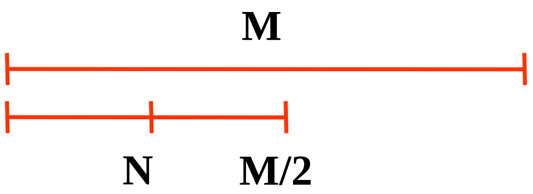
GCD(Greatest Common Divisor)

(Theorem 2.1) If M >N, M mod N < M/2

<Proof>

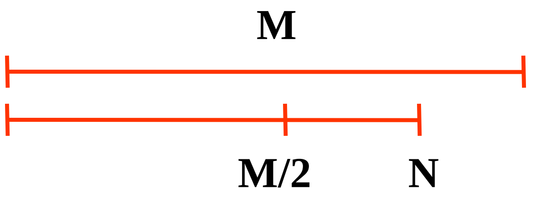
Case 1 : $N \le M/2$

$M \bmod N < M/2$



Case 2 : $N > M/2$

$M \bmod N \le M-N < M/2$



$$T(n) = T(n/2) + 1 \quad \text{if } n > 1$$
$$= 1 \quad \text{if } n = 1$$
$$= O(\log n)$$

Fibonacci numbers

```
long Fib (int n) {
     if (n <= 1)
          return 1
     else
          return Fib(n-1) + Fib(n-2)
}
```

$F_{i+1} = F_i + F_{i-1}\ ,\ F_0 = F_1 = 1$

T(n) = T(n-1) + T(n-2) +2

T(n) = T(n-1) + T(n-2) → T(n) < $(5/3)^k$ = O($(5/3)^k$)

<Proof> By induction

Base step : for n = 1, T(1) = 1 < $(5/3)^1$

Induction step : Suppose it holds for n ≤ k.

Then, we want to show it holds for n = k+1

(See page 6 of text.)