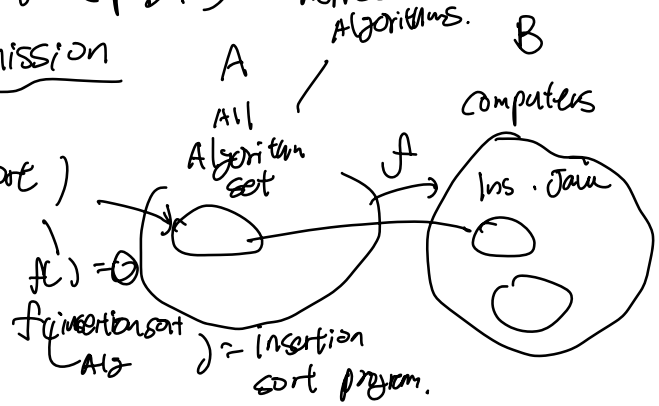
HW#1 - due 9/25/sun midnight

by perspace@korea.ac.kr (PDF)

no late submission

include non deterministic Algorithms.



2. (Insertion sort, merge sort)

$f: A \to B$

① Is f computable?

② Can AI compute f?

③ How hard is it to compute f?

2. resources (alg) → time ↗ → more computation

space → more computation. with some exception

true | Insertion sort $- n^2$
merge sort $- n \lg n$

① what if we are given rand(1, n)

rand(1, 10) $\frac{1}{10}$ 1, 2 .. 10 $\frac{1}{10}$

rand(1, 100) $\frac{1}{100}$

Can randomness be a resources.

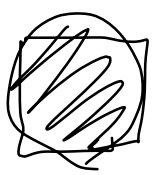
Yes or No with reasons

② interaction

person

P ⟷ V

Communicate



two computer.

whether can we possible to say interaction is a resource.

midterm : 10/20 (Thu) ~ 10/26(Wed) not using ~~Blackboard~~
10/24 → review
10/25 → exam
9/5(Mon) ~ 10/17 (Mon)

hyperspace@korea.ac.kr

Algorithms [COSE 214]
-Monday (2-3:15pm) / Tuesday [2:00-3:15pm]
-정보통신관 room 202 [college of informatics building]

textbook → introduction to algorithms.

Textbook: Introduction to Algorithms, 2nd edition (T.H.Cormen at al. MIT Press)
알고리즘의 능력과 한계, 박성빈, 커뮤니케이션북스, 8월, 2020 – chapters 1,3,5,6,11 (not required)

Final
10/18(Tues) ~ 12/8(Tue

Grading policy
-Midterm exam: 45 points / Final exam: 50 points
-Attendance: 5 points [using the record on the Blackboard system]

in class exams
exams on Tuesday.
day before → review
start of the class

Note
- In this class, programming assignments will NOT be assigned.

① programming language
② Focus on Algorithms only

what will we learn
① design techniques
② analysis → chapter 2
time (runtime)

Topics, briefly

-Design techniques
  (1) incremental approach – insertion sort (chapter 2)
  (2) divide and conquer – merge sort (chapter 2), quick sort (chapter 7)
  (3) using a data structure – heap sort (chapter 6)
  (4) dynamic programming – chapter 15
  (5) greedy algorithms – chapter 16

-Analysis techniques
  (1) worst case complexity
  (2) average case complexity – randomized quick sort
      (chapter 5, chapter 7)

-Graph algorithms
-Datat structures
-P vs NP question
-Quantum algorithms (Deutsch algorithm)

1. ch22 Graph search Algorithms
2. 22.2 DFS – Depth first search
   - a stack (ch10) — ideas behind stack
   - 22.3 BFS – breadth first search
   - a queue

each label called an instance of problem

a shortest path

shortest paths to

a_1 a_2 a_3 a_4
1, 10, 6, 5
e.g. begin

Chapter 1

-(Computational) problem → an infinite set

(1) Clear specification about input / output
(2) An instance of a problem
(3) Sorting problem
input: a finite sequence of n integers (comparable)
output: a finite sequence of n integers given as input arranged according to a certain criterion, say a non-decreasing order
an instance of the sorting problem -1, 2, 2, -10, 5
Given a computational problem , there are always infinitely many instances of the problem

sorting problem
input : a finite sequence of elements $a_1, a_2 .. a_n$ (comparable)
output : a permutation → rearrange of $a_1, \dots a_n$ subject to a condition $0 \le 0 \le 0 \le 0$

n!

-An algorithm is a precise specification of what needs to be done to compute an output for a problem under consideration
(1) finiteness – the number of steps
(2) correctness – for each instance, it should compute a correct output

① for all $x \in I$ / $\forall x \in I$ ② finiteness ③ correctly

ch17, 17.4

Randomized Alg

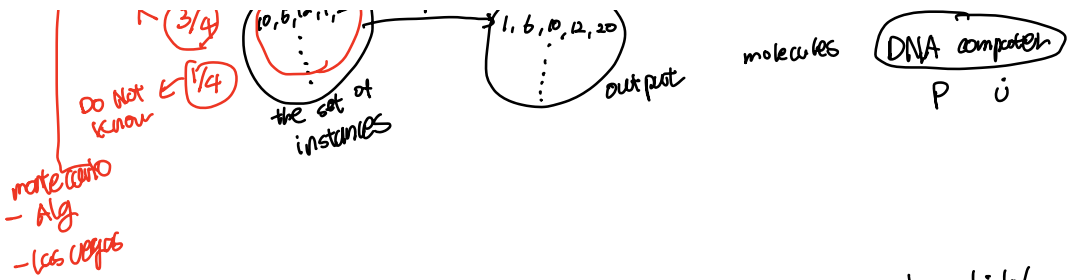
- an algorithm A — a problem

work correct

map

we can see the problem as two sets

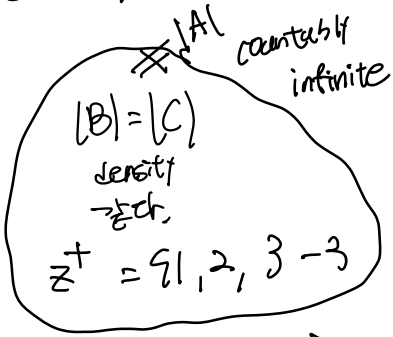
procedure of which each of these instance computed in finite numbers of steps.



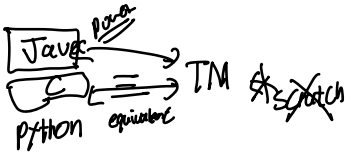
① the set of problems A uncountable
② the set of Algorithms : B $a_1, a_2, a_3 \cdots$ Z^+
③ the set of programs : C p_1, p_2, p_3
Cin Java, C, Python ...)
Turing - complete pc

|A|
|B|
|C|

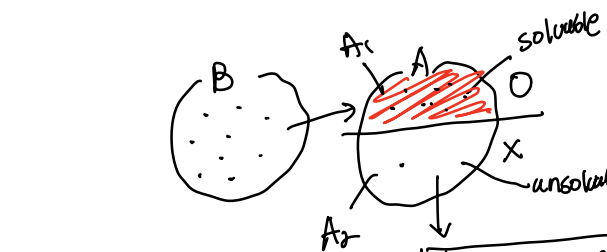
cardinality density



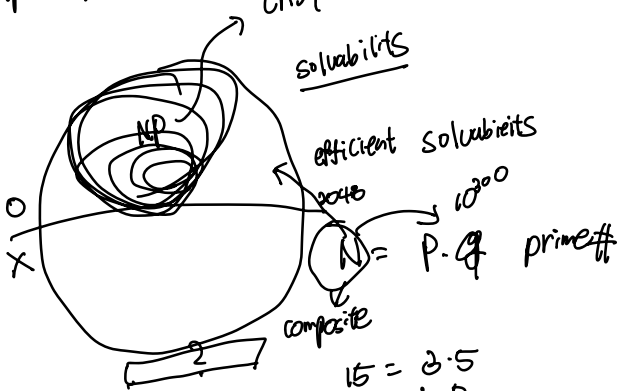
$D = \{1, 2, 3\}$
$|D| = 3$



구현하고 싶은 모든 프로그램은
TM으로 만들 수 있다 (Church)



$|B| = |A_1|$
2.



15 = 3·5
21 = 3·7

problem solved by algorithms

Chapter 1

-The set of all computational problems is uncountable
-The set of all algorithms is countably infinite

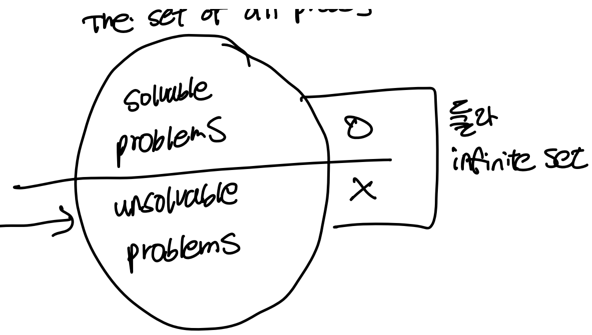
Therefore there does not exist a 1:1 correspondence between these two sets.
In other words, there are algorithmically unsolvable problems.

Halting Problem
-input: a computer program P, an input x for P
-output: true if P(x) halts
false otherwise

Claim: Halting Problem is not solvable. In other words, there does not exist an algorithm that solves the Halting Problem

CH1

Halting problem
↓
unsolvable problem



컴파일러 프로그램
1개의 인풋을 가지는

Arbitrary input for x

input : P , x

output : { true if P(x) halts. / false otherwise

P( ) — 개념같음.

P
P(,)
P(, ,)

귀류법 reduction

preposition : Halting problem is ~~Not~~ solvable
 ″ Not decidable

증명 by contradiction

Let's assume this problem is solvable

1. Assume that halting problem is solvable

2. ∃ (there exists) an algorithm → HALT 라고하자

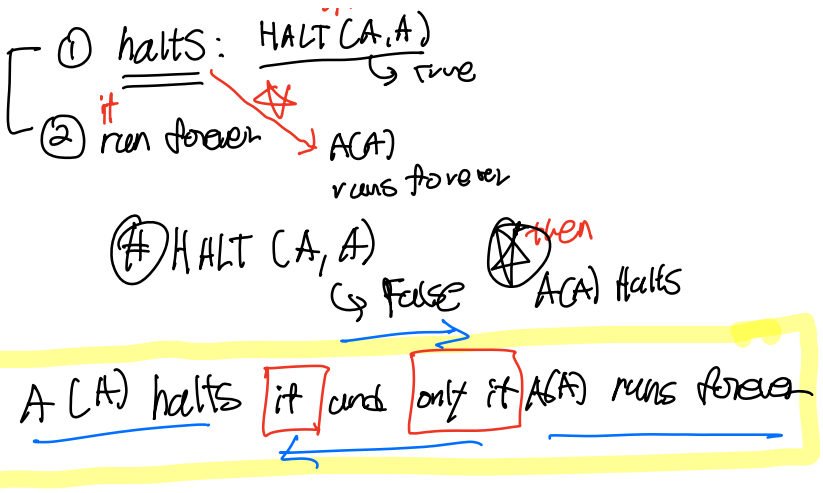
HALT (P, x)
if P(x) halts then return T
else return F

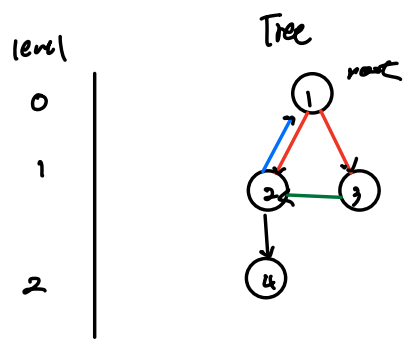
3. we create a program A (☆ 프로그램)

A (P)
if HALT (P, P) then run forever

4. Halt(A, A) = True

A(A) = ?


① halts: HALT(A,A)
true
② it runs forever
A(A) runs forever
#HALT(A,A)
false
then
A(A) halts
A(A) halts it and only it A(A) runs forever


level
0
1
2
Tree
root
1
2
3
4

Chapter 1

Claim: Halting Problem is not solvable. In other words, there does not exist an algorithm that solves the Halting Problem

Proof: Assume that the opposite is true – i.e., assume that there exists an algorithm that solves the Halting Problem. Let's call this algorithm HALT.

```
HALT(P, x)
if P(x) halts then return true
        else return false
```

Using this algorithm, we can write a program A whose behavior is as follows:

```
A(P)
if HALT(P, P) then run forever
```

A(P)
if HALT(P, P) then run forever

Now, let's analyze the behavior of A(A)

(1) if HALT(A,A) returns true then A(A) does not halt

=> impossible because HALT(A,A) returns true if A(A) halts

(2) if HALT(A,A) returns false then A(A) halts

=> impossible because HALT(A,A) returns false if A(A) does not halt

Therefore the claim is true – that is, there does not exist an algorithm that solves the Halting Problem

chapter 2: sorting problem

input : integers (distinct)

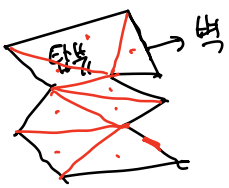
- ascending order (1, 2, 7, 10) 동일하지 않은 서로 다른 수들
- non decreasing → (1, 2, 2, 7, 10)

an incremental Approach

ex)

cf.

Computational Geometry → 이 책들에서 이런 예시가 사용된다.

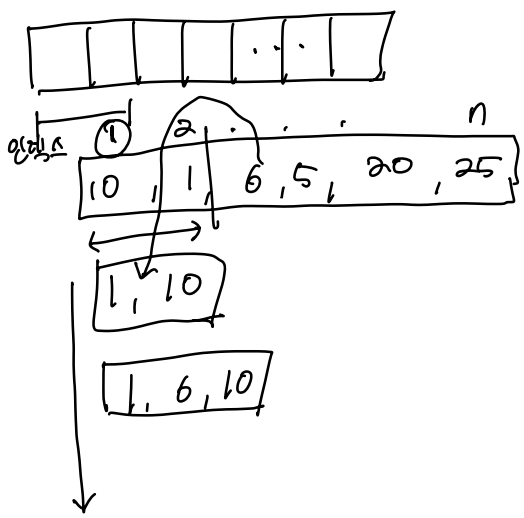


가장 적은 수의 카메라가 모든 곳을 커버하게!

△ 사용

왕상 교재 참조

Insertion Sort

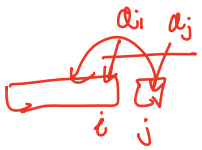


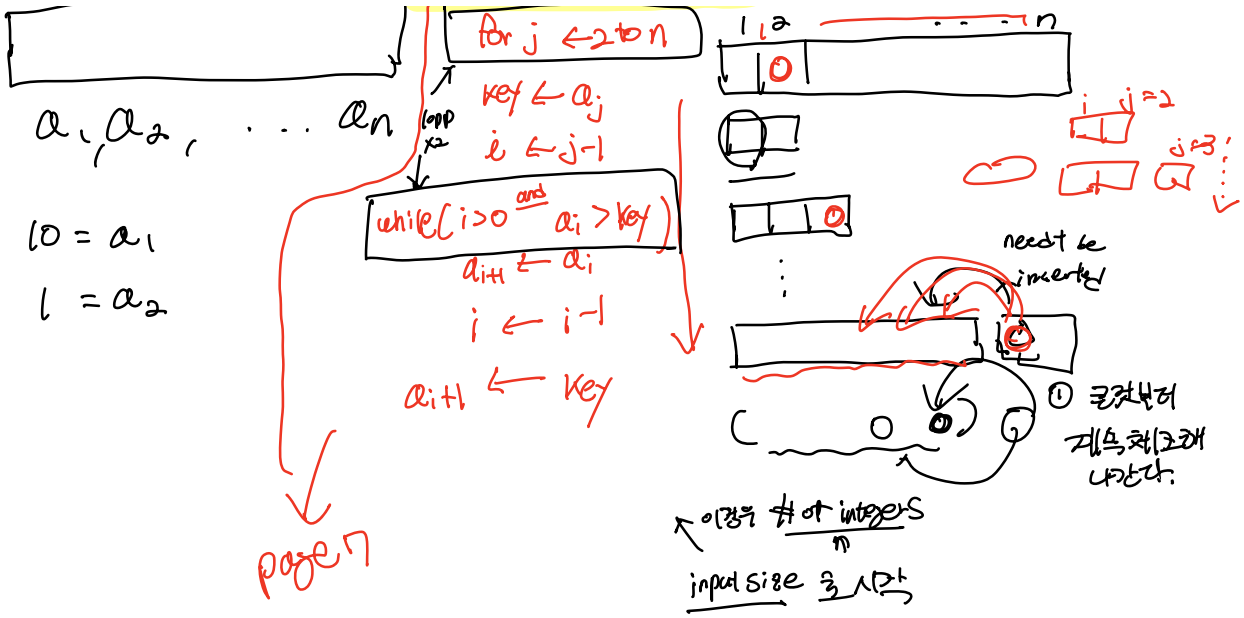
A

in-place

amount of main memory kept constant

Insertion Sort (A) n-1 elements

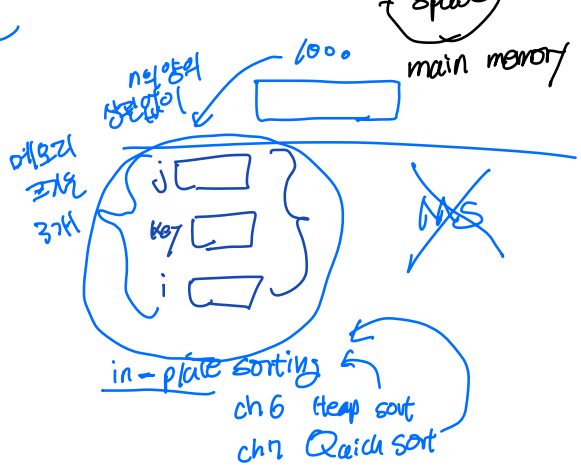




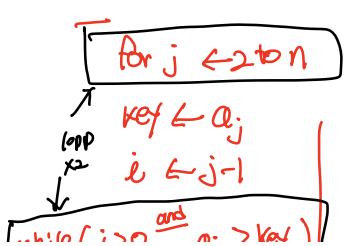
~ ~ ~ analysis of algorithms ~ ~ ~ ~ ~

- to predict resources
  - time
  - space

space



time



n-1
↓
n-1개의 프로세스

special 이런아님

ex) {1, 2, 3, 4}

4!개의 크기별 서

1, 2, 3, 4
best

$a_{i+1} \leftarrow a_i$

$i \leftarrow i-1$

$a_{i+1} \leftarrow key$

각각 인풋마다 concrete한 숫자 따라가보면 가질수 없다.

⋮

4, 3, 2, 1 — worst — $\frac{n(n-1)}{2}$

① | 1 | 2 | 3 | 4 |

| 1 | 2 |

| 1 | 2 | 3 |

3번

최대값: $1+2+3+\cdots n-1 = \frac{n(n-1)}{2}$

최소값: $n-1$

② | 4 | 3 | 2 | 1 |

1 | 3 | 4 |

2 | 2 | 3 | 4 |

3 | 1 | 2 | 3 | 4 |

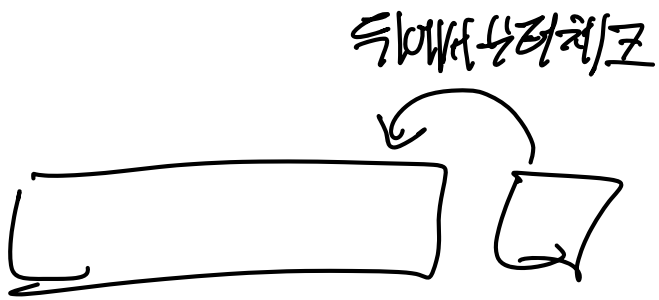
6번

input distinction → ch5 → ch7

인풋은 어떻게 주어지는가.

randomization 이랑 관련 있음.

Incremental approach – insertion sort algorithm



Insertion Sort(a_1, a_2, a_3, ⋯, a_n)

```
1. for j ← 2 to n
2.     key ← a_j
3.     i ← j - 1
4.     while (i > 0 and a_i > key)
5.         a_{i+1} ← a_i
6.         i ← i - 1
7.     a_{i+1} ← key
```

Line 1 to line 7: iterated n-1 times

for each iteration, line 2 is executed once, line 3 is executed once, line 4 to 6 are repeated, line 7 is executed once

Insertion Sort(a_1, a_2, a_3, $\cdots$, a_n)

```
1. for j ← 2 to n
2.     key ← a_j
3.     i ← j - 1
4.     while (i > 0 and a_i > key)
5.         a_{i+1} ← a_i
6.         i ← i - 1
7.     a_{i+1} ← key
```

Sort 5,-1,1,3 using Insertion sort
a1=5,a2=-1,a3=1,a4=3

1st iteration
j=2, key=-1, i=1
check conditions in line 4
since they are true line 5 and line 6 are executed
a2=5,i=0
check conditions in line 4 again, but i=0
execute line 7 and a1=-1

At this stage, part of 5,-1,1,3 is sorted; i.e.,
a1=-1,a2=5

Similarly, 2nd iteration, 3rd iteration are performed

Line 1 to line 7: iterated n-1 times
for each iteration, line 2 is executed once, line 3 is executed once,
line 4 to 6 are repeated, line 7 is executed once

Insertion Sort(a_1, a_2, a_3, $\cdots$, a_n)

```
1. for j ← 2 to n
2.     key ← a_j
3.     i ← j - 1
4.     while (i > 0 and a_i > key)
5.         a_{i+1} ← a_i
6.         i ← i - 1
7.     a_{i+1} ← key
```

Sort 5,-1,1,3 using Insertion sort
a1=5,a2=-1,a3=1,a4=3

1st iteration
At this stage, part of 5,-1,1,3 is sorted; i.e.,
a1=-1,a2=5

2nd iteration
j=3,key=1,i=2
conditions in line 4 - true
a3=5,i=1
conditions in line 4 – false
execute line 7 and a2=1
At this stage, a little bit more is solved; i.e.,
a1=-1,a2=1,a3=5

After 3rd iteration
a1=-1, a2=1, a3=3, a4=5

Chapter 2

Analysis of an algorithm – prediction of resources such as the number of steps taken (time complexity), the amount of memory space (space complexity), the network bandwidth, etc that an algorithm needs during execution

Insertion Sort(a_1, a_2, a_3, $\cdots$, a_n)

1. for j ← 2 to n
2. key ← a_j
3. i ← j - 1
4. while (i > 0 and a_i > key)
5. a_{i+1} ← a_i
6. i ← i - 1
7. a_{i+1} ← key

The number of steps taken by the insertion sort is influenced by the number of iterations in lines 4, 5, 6.

To illustrate the idea, consider two cases
- a1=1,a2=2,a3=3,a4=4

=>conditions of line 4 – false for each iterations
none of line 5, 6 are executed

-a1=4,a2=3,a3=2,a4=1
- Conditions of line 4 – true for each iteration
line 5, 6 are executed once, twice, and three times for each iteration

Insertion Sort(a_1, a_2, a_3, ⋯, a_n)

```
1. for j ← 2 to n
2.     key ← a_j
3.     i ← j - 1
4.     while (i > 0 and a_i > key)
5.         a_{i+1} ← a_i
6.         i ← i - 1
7.     a_{i+1} ← key
```

Time complexity

(1) a1=1,a2=2,a3=3,a4=4

=>conditions of line 4 – false for each iterations
none of line 5, 6 are executed

Best case: the number of steps is proportional
To the number of input data (= input size)

(1) a1=4,a2=3,a3=2,a4=1

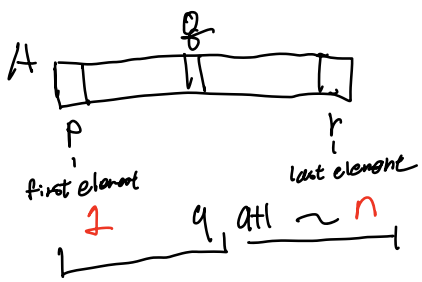
- Conditions of line 4 – true for each iteration
  line 5, 6 are executed once, twice, and three times for each iteration

Worst case: the number of steps is proportional to 1+2+3+…+(n-1).

Divide & Conquer

3 steps → 1. divide
2. Conquer → partial Result
3. Combine

Merge Sort (A)



MergeSort (A, p, r)

if p < r

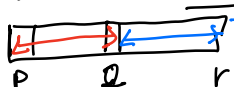
then $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$ → floor operation → $\lfloor 4, 2 \rfloor = 4$

MergeSort (A, p, q) → C

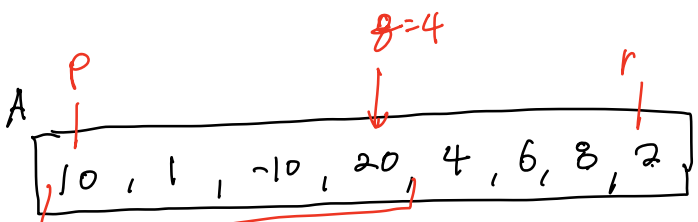
MergeSort (A, q+1, r) → C

combine

n → Merge (A, p, q, r) → combine



예

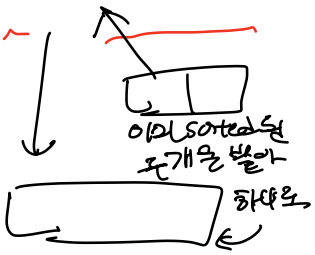


$q \leftarrow \lfloor \frac{1+8}{2} \rfloor = \lfloor 4.5 \rfloor \rightarrow 4$

MS (A, 1, 4)

MS (A, 5, 8)

M (A, 1, 4, 8)

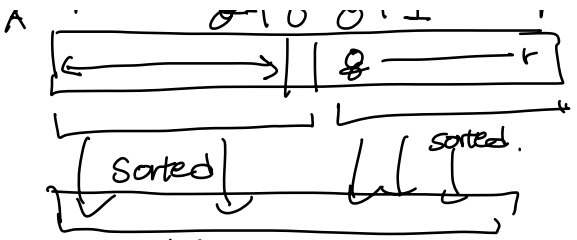


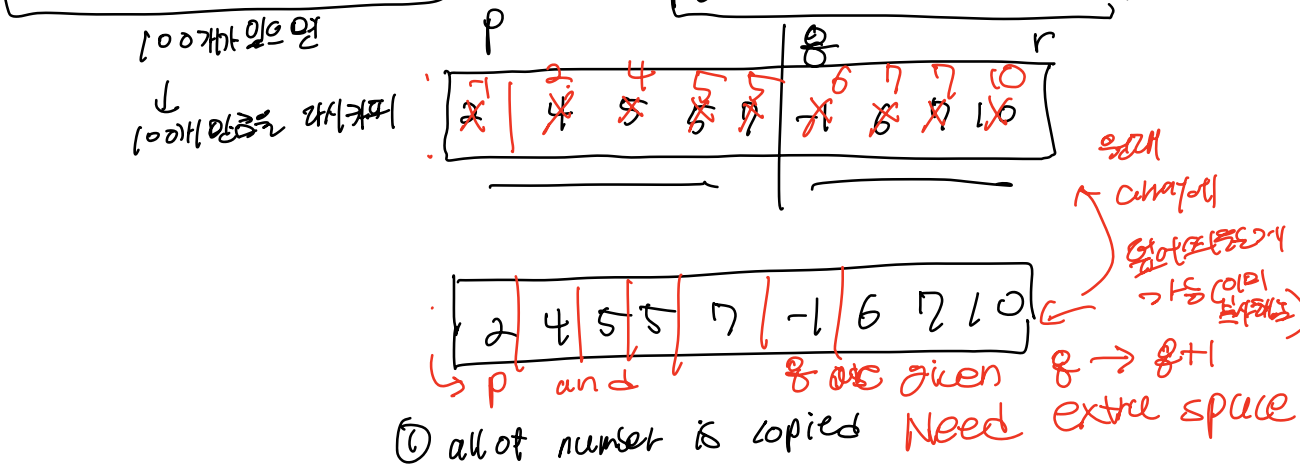
Non decreasing order (같은 수 여러번 나올 수 있다.)

p ~ q ~ q+1 ~ r

Merge (A, p, q, r)

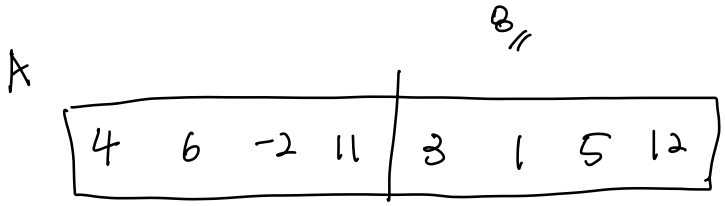
need an extra space





① all of number is copied Need extra space

Number of steps proportion to n

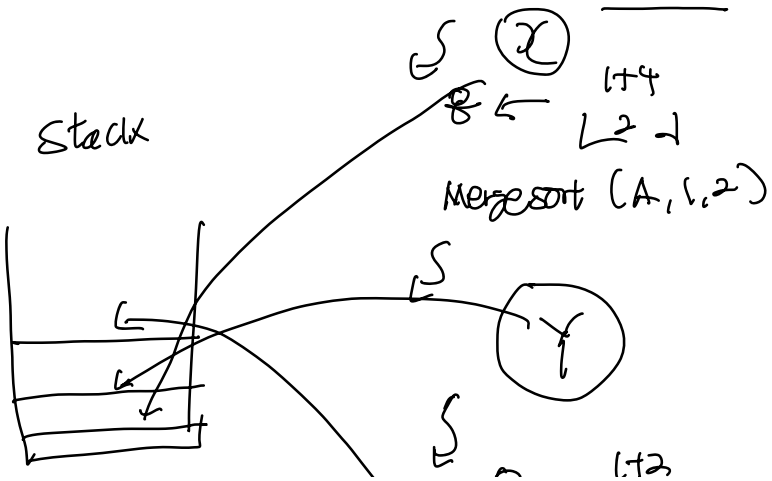


invoke

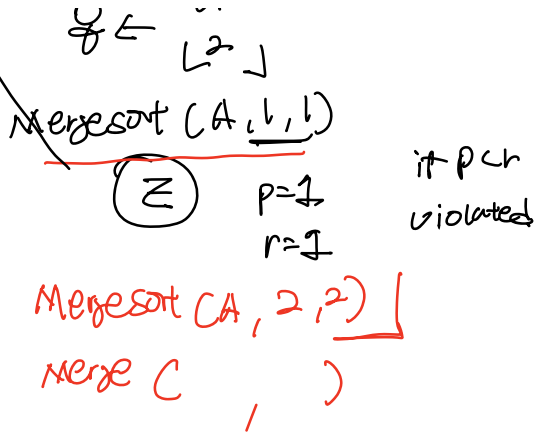
MergeSort (A, 1, 8)

$q \leftarrow \lfloor \frac{1+8}{2} \rfloor = 4$

MergeSort (A, 1, 4)



we use stack data structure



Chapter 2

Divide and conquer – Merge sort algorithm

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Chapter 2

Divide and conquer – Merge sort algorithm

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋                    divide
3          MERGE-SORT(A, p, q)                conquer
4          MERGE-SORT(A, q + 1, r)            conquer
5          MERGE(A, p, q, r)                  combine
```

Run time – linear time

```
MERGE(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
4  for i ← 1 to n1
5       do L[i] ← A[p + i − 1]                 ┐ copy
6  for j ← 1 to n2                             │
7       do R[j] ← A[q + j]                     ┘
8  L[n1 + 1] ← ∞   → ∞ largest possible number in program language = a sentinel
9  R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 for k ← p to r
13      do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

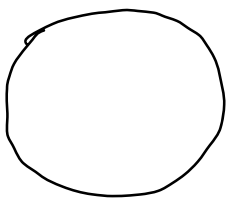
Runtime (= time complexity)

~ # of steps

a recurrance (equation)

ex) $a_n = 4a_{n-1} + 1$

input size

$T(n) =$ 

merge sort (A, P, r)

if p < r

$q \leftarrow \frac{p+r}{2}$

$T(\frac{n}{2})$ ← merge sort (A, P, q)

$T(\frac{n}{2})$ ← merge sort (A, q+1, r)

n ← merge (A, P, q, r)

$$T(n) = C_1 + 2T(\tfrac{n}{2}) + (n)$$

constant, linear function

approximation

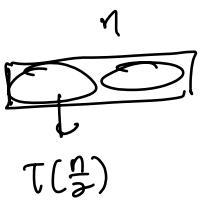
$$T(n) = \begin{cases} 2T(\frac{n}{2}) + C \cdot n, & n \geq 2 \\ C, & n = 1 \end{cases}$$

$n \geq 2$

$n = 1$

ms (A, 1, 1)

ms (A, 2, 2)



closed-form set

Ch4 top) = $n \lg n$ 7.4

① substitution method.

② recursion tree

③ master method.

top-level cost

Recursion tree method

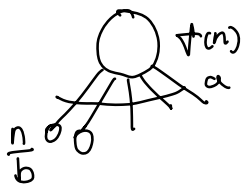
$T(n) = 2T(\frac{n}{2}) + cn$, $n \geq 2$

combine. $D + C$

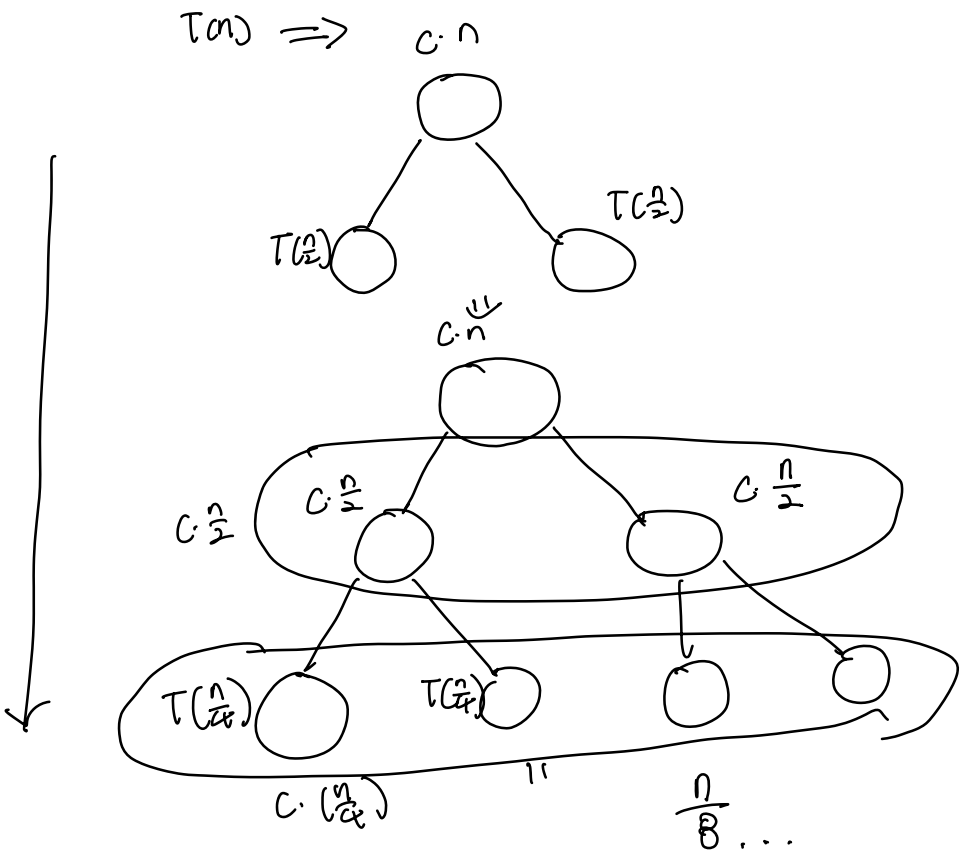
$T(n) = C_1 + 2T(\frac{n}{2}) + n$

$T(n) = a\,T(\frac{n}{b}) + f(n)$ $b > 1$

- Divide
- conque
- combine



Ex $T(n) = 9T(\frac{n}{4}) + c \cdot n$



input size

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \cdots$

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$

$\lg n$ (log base 2)

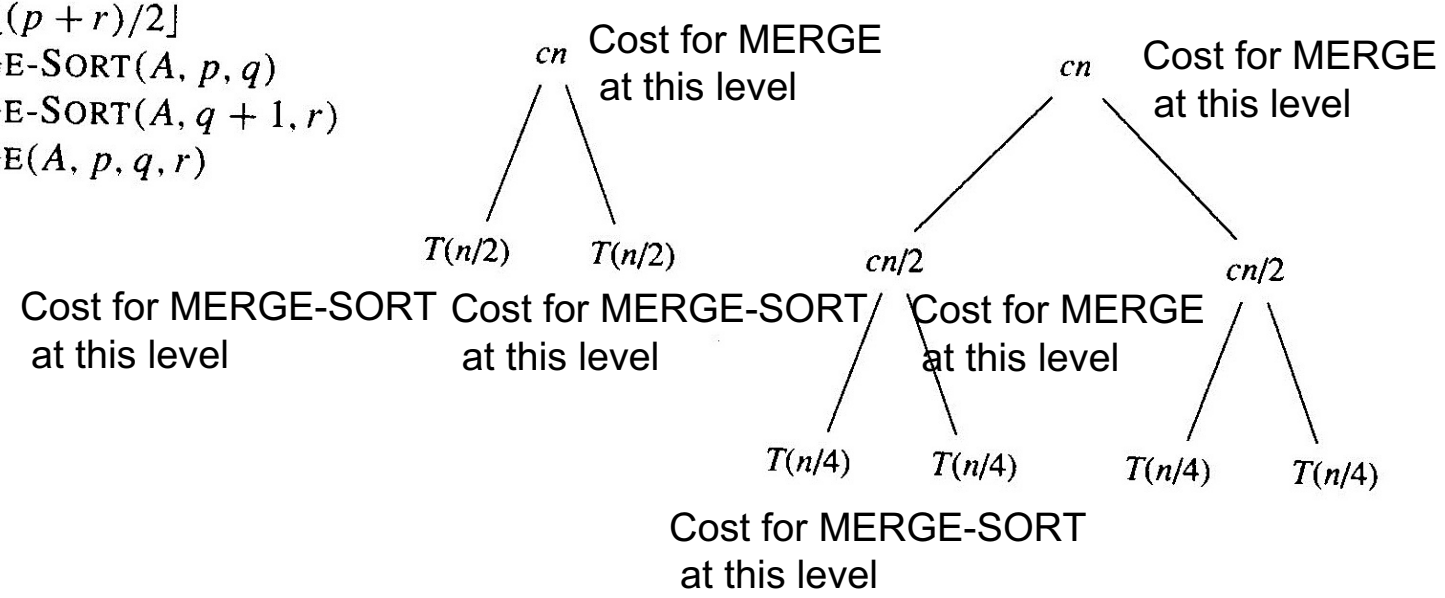
Ex)

$$T(n) = T\left(\frac{1}{2}n\right) + T\left(\frac{2}{3}n\right) + n$$

Chapter 2

Divide and conquer – Merge sort algorithm

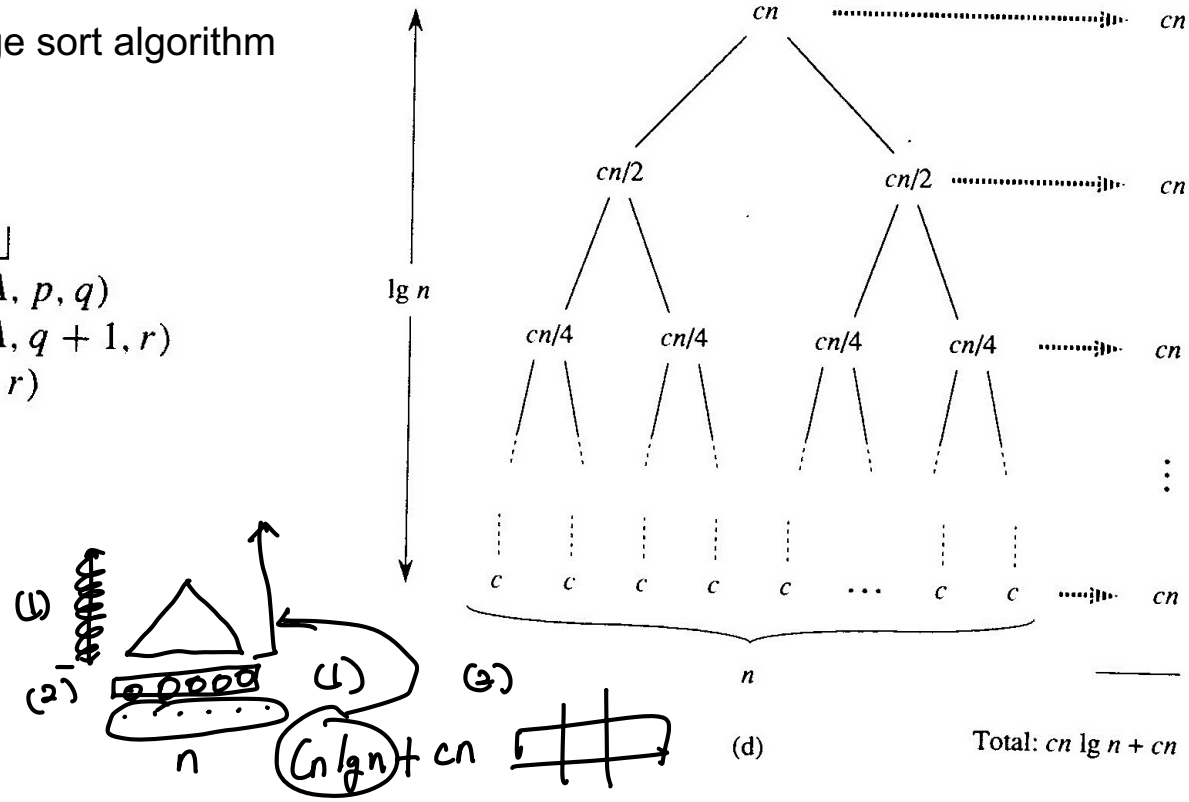
```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

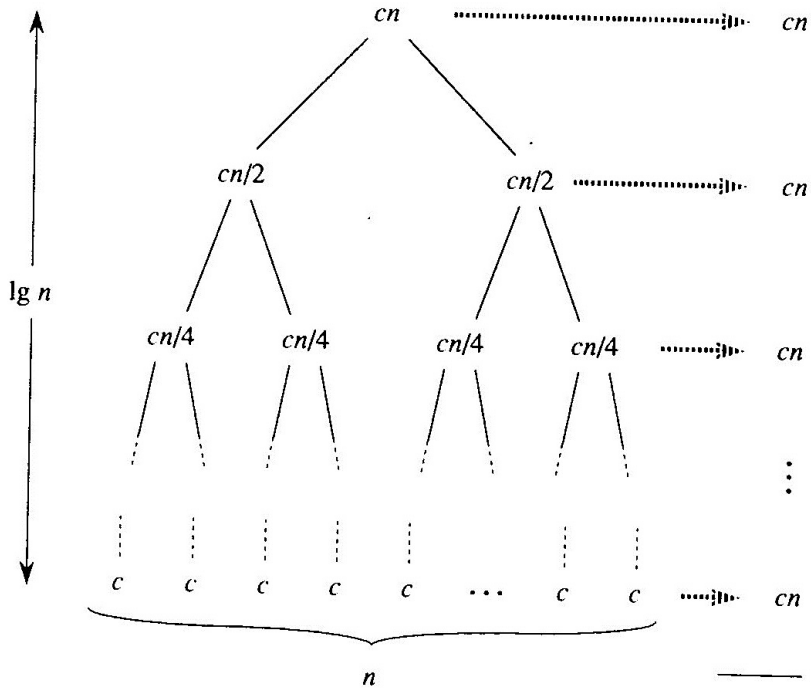


Chapter 2

Divide and conquer – Merge sort algorithm

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```





(d) Total: $cn \lg n + cn$

how many leaves ?

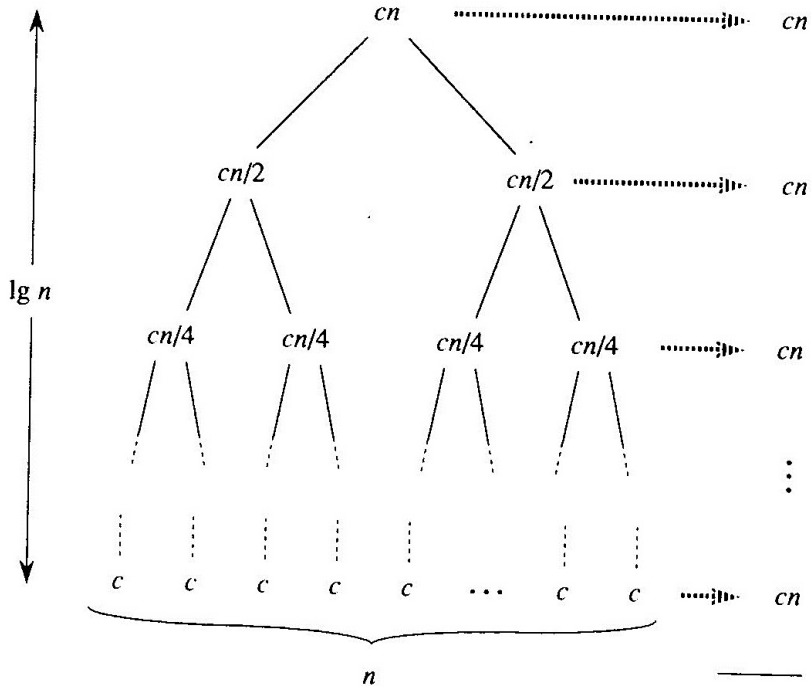
1st level 1

2nd level 2

3rd level 2^2

⋮

$\log n - 1^{th}$ level $2^{\log_2 n} = n$



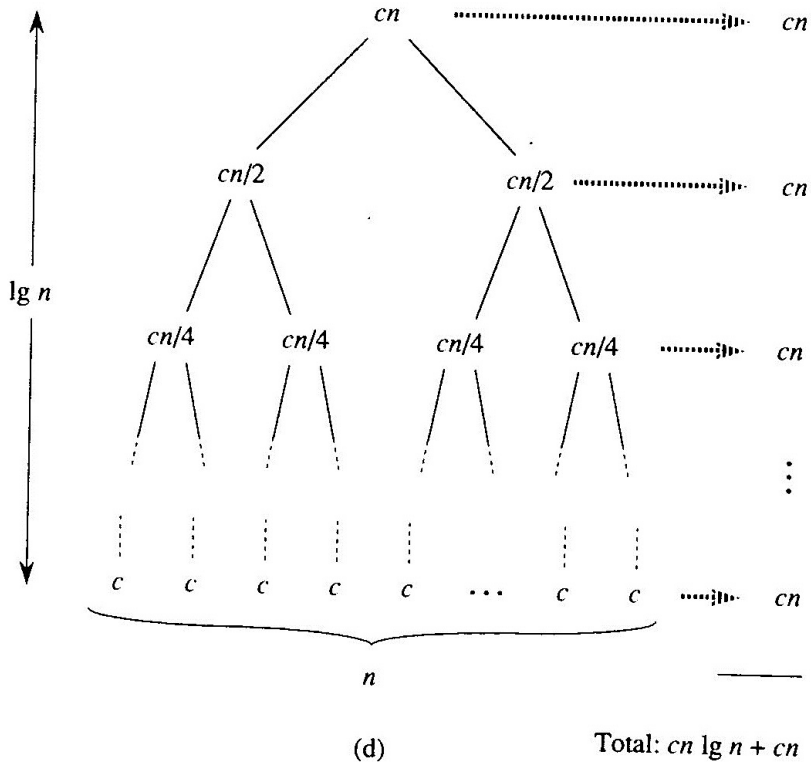
(d) Total: $cn \lg n + cn$

$n \quad \frac{n}{2} \quad \frac{n}{4} \quad \frac{n}{8} \quad \cdots \quad \frac{n}{2^k}$

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$



A recursion tree

A recurrence (equation)

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

chapter 3

Asymptotic notations

O Big Oh

o little oh

Ω Big Omega

ω little ω

Θ theta

$n \to \infty$

n : input size

---

the cost of space / runtime of an Alg

Growth rates of functions

$f(n) = n^2$

$g(n) = n^3$

$f(n) \Leftarrow$ same $\Rightarrow g(n)$

$f'(n) = 2n \quad h'(n) = 8n$

$h(n) = 4n^2$

$n \to \infty$

- largest order

$\log n \quad 10 \cdot \log n$

same class of growth rate

Given a function, $f(n) \geq 0$, $n \to \{0, 1, 2 \cdots\}$

$O(f(n)) = \{$ an infinite set

grow slower than $f(n)$

or

grow at the same rate as $f(n)$ $\}$

coefficients do not matter

$n^2 = 4n^2$

Ex) $f(n) = 2n^2 + 1$

$O(f(n)) = \{ 10n^2,\ n^2+15,\ 2n+7,\ \log^n, \cdots \}$

9/19

$\lim_{n \to \infty}$

Ch3 Asymptotic Notations

↳ Growth rates of function (= time complexity of an Algorithms.)

runtime

domain - natural numbers

시간 이라서 +

Algorithms

$A : N \to R^+$

input size $\{0, 1, 2 \cdots\}$

needs

space complexity

runtime of A알고리즘

Ex) $f(n) = n^2 + 1$

$f(n) = n^3$

runtime of A알고리즘

problem

$\lim_{n \to \infty}$

즉 A알고리즘은 알고리즘으로 같은 문제를 푼다.

Symtutic Notation

Given a function $f(n)$

$O(f(n)) = \{$ $\}$

$o(f(n)) = \{$ $\}$

$\Omega(f(n)) = \{$ $\}$

$\omega(f(n)) = \{$ $\}$

$\Theta(f(n)) = \{$ 3 $\}$ $\quad \Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$

① $f(n)$: a polynomial function ← n^3+2n^2+1

② ∴ Not " ← $\log_2 n$, $\log \log_2 n$

2) $f(n)$: a polynomial function

$f(n) = n^4 + n^2 + 1$

Any polynomial function only largest order matter

$f(n) = 100n^2 + n + k$

$O(f(n)) = \{$ 3 $\}$ ↑

① all functions with the same order

Ex) $f(n) = 3n^2 + n + 1$

$O(f(n)) = \{ 100n^2,\ n^2+n+1,\ 100n^2+n+1 \cdots \}$

② all functions with the smaller order → $O(f(n))$ (little oh)

$Of(n) = \{$ 1, $n+1$ $\}$

$f(n)$ ✗ we do not deal with coefficients.

2) Not a polynomial f

③ all f that grow at the same rate

~~coefficient~~ $f(n) = \log n$

$Of(n) = \{ 10 \lg n,\ 100 \lg_{10} n \}$

④ all of that grow slower than f(n) → little oh $o(f(n))$

$f(n) = \log\log\log n$

$O(f(n)) = \{ g(n), \ldots \}$

$o(f(n)) = \{ h(n), \ldots \}$

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

$\lim_{n\to\infty} \frac{h(n)}{f(n)} = 0$ ← smaller

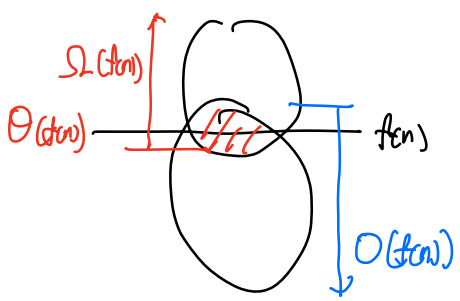
↓

경계보다 더 작은 애들

Ex $n^2 + 5\lg n$

Sara Because

Given a function f(n)



$f(n) = n^2 + n + 1$

$\frac{2n^2+1}{2} = O(f(n))$ → allowed

적어도 set 안에

abusive usages of function

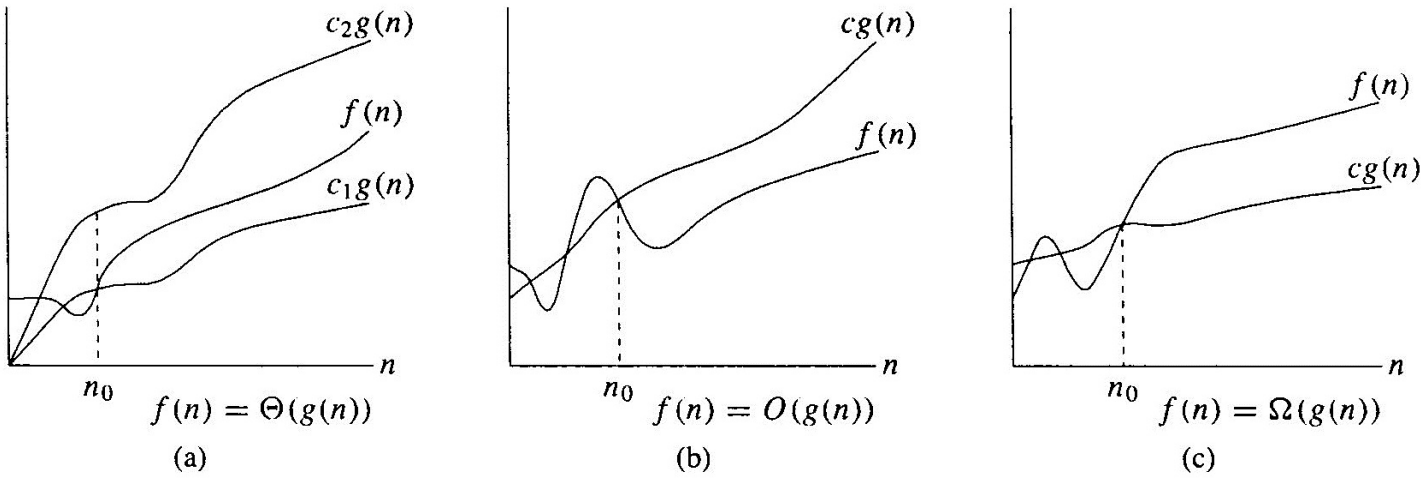
P.21

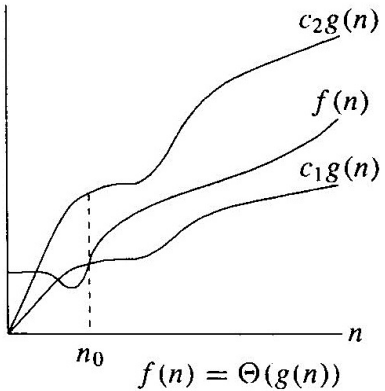
Def a function f
is polynomially bounded
if $f = O(n^k)$

Ex) $f(n) = \log_2 n$ – 보다큰 O에 다음어가나가

$f(n) = O(n^2)$
$O(n^3)$ – 이런식의 표현도 포함된다.
$O(n^1)$

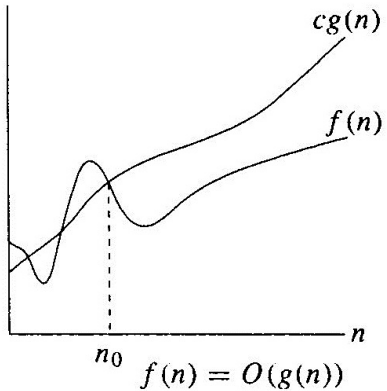
Chapter 3 Growth of functions





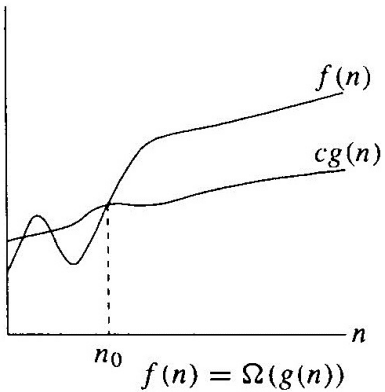
$f(n) = \Theta(g(n))$

(a)



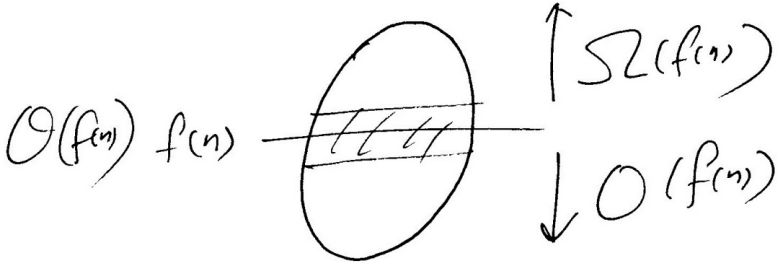
$f(n) = O(g(n))$

(b)



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

(c)



Ex $f(n) = n^2 + 1$

$\Theta(n) = \{ 2n^2,\ 3n^2,\ \frac{1}{2}n^2 + 5,\ \cdots \}$

$O(f(n)) = \{ n^2,\ n+10,\ \frac{1}{2}n + 20,\ \cdots \}$

$\Omega(f(n)) = \{ \frac{1}{3}n^2,\ n^3 + 5,\ n^K + 2,\ \cdots \}$

Polynomials

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

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

where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an

real constant $a \leq 0$, the function n^a is monotonically decreasing. We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

↳ Not necessarily polynomial function

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

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

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

Ch 5 probabilistic analysis → Ch 7이랑 별건 (7.4.1)

[Ins sort
  Ms

C_i : interviews
C_h : hiring

hiring problem
- a company
- want to hire an assistant

Assumptions ① n : candidates ($n \geq 1$)

② distinct competitiveness

→ ③ all $n!$ permutations are equally likely.

1
2
3
⋮
n
ranks

hiring policy

a room

one by one

cores in possible

$3!$

a_1	a_2	a_3
a_3	a_2	a_1
⋮		

$3! = 6$

1. Always, 1st person interviewed - hire

2. fire / hire
   C_h

n candidates $C_i << C_h$

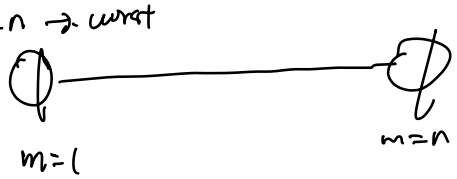
total cost = $n \cdot C_i + m C_h$

m : # of persons

# of people hired.

total cost = $n \cdot C_i + m \cdot C_h$

$m = 1 \rightarrow$ best
$m = n \rightarrow$ worst



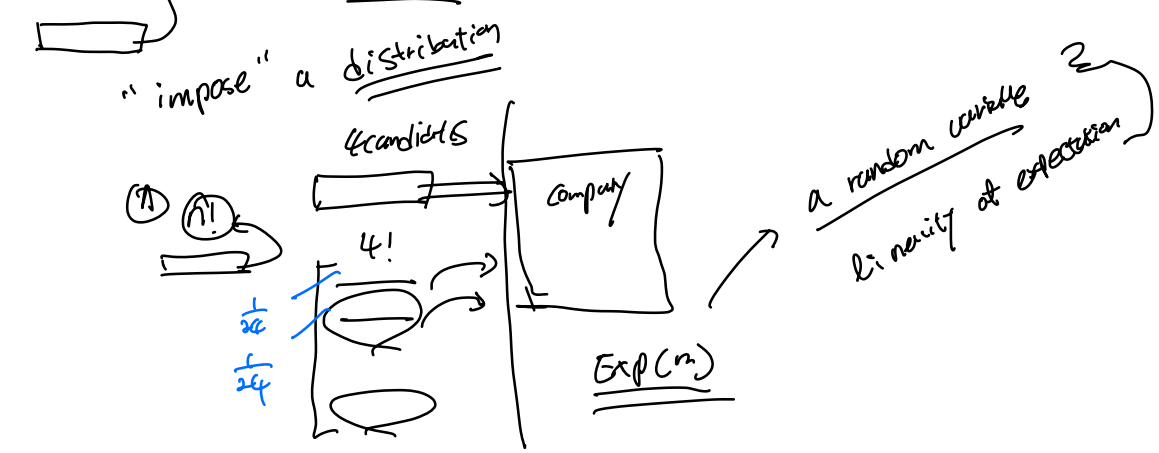
$n \cdot C_i + m \cdot C_h$

?

① $n!$

Exp (m) ?

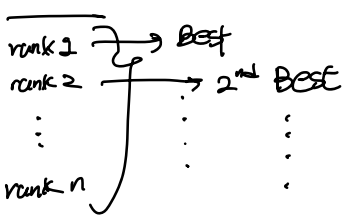
"impose" a distribution



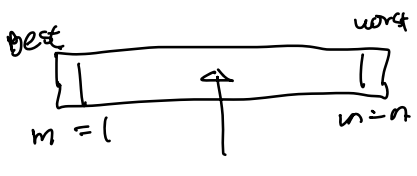
m : # of times that an assistant is hired

Assumptions

- distinct competitiveness



Expected # of times that an assistant is hired



total cost = $n C_i + m C_h$ — Exp(m) ?

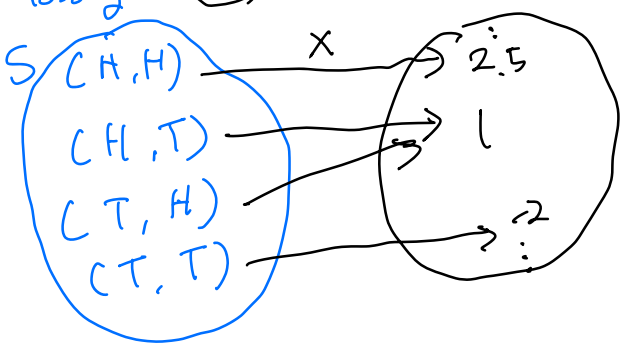
?

- a random variable → X → E(X)
- an indicator random variable I_A → an Event A — 5.1
- linearity of expectation.

Random variable

a function

Ex tossing 2 fair coins $\frac{1}{2}$



$E(X) = 2.5 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + (-2) \cdot \frac{1}{4}$

$= \frac{2.5}{4}$

5.1

- an indicator random variable

I_A → an Event A

i.o

indicator Random variable

a sample space → S domain → 0, 1

$E(I_A) = Pr\{A \text{ occurs}\}$ → 5.1 proof

I_A { 1 if A occurs; 0 otherwise. } INDICATE

- linearity of Expectation

n r. variables

$X_1, X_2 \cdots X_n$

$E(X_1 + X_2 + X_3 \cdots + X_n) = E(X_1) + E(X_2) \cdots + E(X_n)$

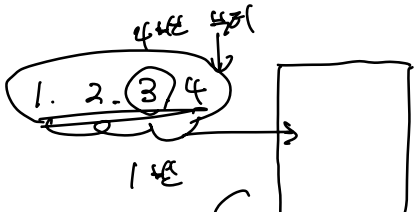
$n \cdot C_i + m \cdot C_h$

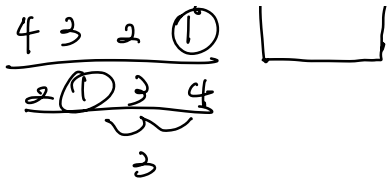
$C_i << C_h$

Assumptions

① distinct $^{n}\smile$ ranks

Rank 1 2 3 4





X : total # of times that an assistant is hired.

$E(X)$ $\quad X = \sum_{i=1}^{n} X_i$

indicator Rand vari

$X_1, X_2, \ldots, X_n$

1 if 1st

X_i : 1. if i^{th} person is hired

0 otherwise

$$E(X) = E\left(\sum_{i=1}^{n} X_i\right)$$

$$= E(X_1) + E(X_2) + \cdots + E(X_n)$$

$$= Pr\{1^{st} \text{ person is hired}\} + Pr\{2^{nd} \text{ person is hired}\} + Pr\{3^{rd} \cdots\}$$

$$= 1 + \cdots$$

Assumptions

Any permutation is equally likely

random permutation

5! = 120

1, 2, 3, 4, 5 → 1/120

2, 1, 3, 4, 5 → 1/120

⋮

5, 4, 3, 2, 1

$P\{2^{nd} \text{ person hired}\}$ 5 4 → 1/2

Rand → 1 2 3

0 1 3 2

X 1 3 2

0 2 1 3

X 2 3 1

3/6

Company

0 3 12

x 3 21

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots \frac{1}{n} \approx O(\log n)$

Chapter 5 Probabilistic analysis

The hiring problem – not a computational problem, but this problem can be used to introduce the idea about randomized algorithms

Company

Hiring policy – the first candidate is always hired. From then on, there are 2 possibilities – hire and fire, OR skip

n candidates – it is assumed that each candidate has a different competitiveness

We want to analyze the cost
-Best situation
-Worst situation
-On average?

Hiring policy – the first candidate is always hired. From then on, there are 2 possibilities – hire and fire, OR skip

n candidates – it is assumed that each candidate has a different competitiveness

We want to analyze the cost that the company has to pay

HOW?

Assume that
-Cost for interviewing: C_i
-Cost for hiring: C_h
-Cost for firing: 0

(for the company)
Best situation: n * C_i + 1 * C_h (the first candidate is the most competitive person)
Worst situation: n * C_i + n * C_h

“on average” how many candidates need to be hired?

Hiring policy – the first candidate is always hired. From then on, there are 2 possibilities – hire and fire, OR skip

n candidates – it is assumed that each candidate has a different competitiveness

We want to analyze the cost that the company has to pay

“on average” how many candidates need to be hired?

-We need the information about “distribution” – there are n! possibilities

(1) Are all of these possibilities equally likely?
(2) In general, this information is not available in advance

Therefore, we can “impose” a certain distribution – in other words, we ASSUME that all of these possibilities are equally likely?

This is implemented by using a “random number generator”

We define probability in terms of a ***sample space*** S, which is a set whose elements are called ***elementary events***. Each elementary event can be viewed as a possible outcome of an experiment. For the experiment of flipping two distinguishable coins, we can view the sample space as consisting of the set of all possible 2-strings over {H, T}:

$$S = \{\text{HH}, \text{HT}, \text{TH}, \text{TT}\} .$$

An ***event*** is a subset[1] of the sample space S. For example, in the experiment of flipping two coins, the event of obtaining one head and one tail is {HT, TH}. The

since elementary events, specifically those in A, are mutually exclusive. If S is finite and every elementary event $s \in S$ has probability

$$\Pr\{s\} = 1/|S| ,$$

then we have the ***uniform probability distribution*** on S. In such a case the experiment is often described as "picking an element of S at random."

A (discrete) random variable is a function from a sample space to the set of real numbers

The simplest and most useful summary of the distribution of a random variable is the "average" of the values it takes on. The ***expected value*** (or, synonymously, ***expectation*** or ***mean***) of a discrete random variable X is

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\} , \tag{C.19}$$

which is well defined if the sum is finite or converges absolutely. Sometimes the expectation of X is denoted by μ_X or, when the random variable is apparent from context, simply by μ.

Consider a game in which you flip two fair coins. You earn \$3 for each head but lose \$2 for each tail. The expected value of the random variable X representing your earnings is

$$\begin{aligned} \mathrm{E}[X] &= 6 \cdot \Pr\{2 \text{ H's}\} + 1 \cdot \Pr\{1 \text{ H}, 1 \text{ T}\} - 4 \cdot \Pr\{2 \text{ T's}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1 . \end{aligned}$$

The expectation of the sum of two random variables is the sum of their expectations, that is,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] , \tag{C.20}$$

whenever $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ are defined. We call this property ***linearity of expectation***, and it holds even if X and Y are not independent. It also extends to finite and

for converting between probabilities and expectations. Suppose we are given a sample space S and an event A. Then the ***indicator random variable*** $\mathrm{I}\{A\}$ associated with event A is defined as

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs}, \\ 0 & \text{if } A \text{ does not occur}. \end{cases} \tag{5.1}$$

Lemma 5.1
Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\}, \end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

How can we analyze the average cost for the hiring problem?

Random variable X – total number of times that a candidate is hired

Indicator random variable X_i

if i_th candidate is hired, X_i is 1
otherwise, X_i is 0

$X = X_1 + X_2 + X_3 + \ldots X_n$

Our goal is to compute the expected value of X, i.e., $E(X) = E(X_1 + X_2 + X_3 + \ldots X_n)$
By the linearity of expectation, $E(X) = E(X_1) + E(X_2) + \ldots + E(X_n)$
Using lemma 5.1 $E(X) = 1 + 1/2 + 1/3 + \ldots 1/n = O(\lg n)$

Ch 6 HeapSort

using DS

1. incremental Apr
   insertion sort X
2. D & C
   mergesort X
3. using DS
   HeapSort

Heap ① DS → a heap → 6.1 → ① Max heap → ② min
② memory Region
eg) malloc

(Binary) Heap → 3 ary heap, 4 ary heap, ⋮, d-ary heap

- Prim's Alg ch23
- Dijkstra's Alg ch.3

Goal: Define Max Heap A tree

an undirected graph $G(V, E)$

a directed graph $G = (V, E)$

↳ $E \subseteq V \times V$

infinite graph

V: a finite set ($\neq \emptyset$) nodes

$V \times V$: the set of Cadesian product all ordered pairs of element in V

Ex $V = \{a, b\}$

$V \times V = \{(a,a), (a,b), (b,a), (b,b)\}$

E is a relation on V

1) An undirected graph $G = (V, E)$

E is a subset of the set of all unordered pairs of V

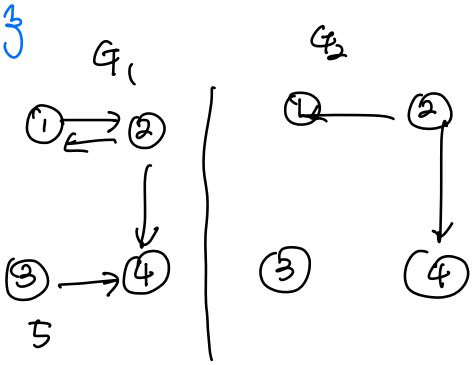
$V \times V$ < $\{(a,a), (a,b), (b,b)\}$ / $\{(a,a), (b,a), (b,b)\}$

Ex $V = \{1, 2, 3, 4\}$

Edges < $E_1 = \{(1,2), (2,1), (3,4), (2,4)\}$ / $E_2 = \{(2,1), (2,4)\}$

G_1 (V, E_1)

G_2 (V, E_2)



Max Heap 6.1

↓

(An) undirected graph $G = (V, E)$

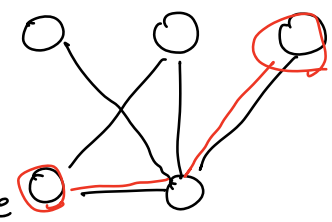
Def fine tree a binarily tree

① an undirected graph ↙

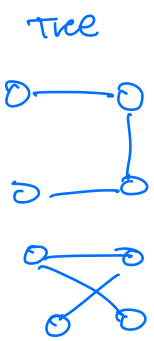
② connected — between any two nodes there must be always a way

③ acyclic = not cycle

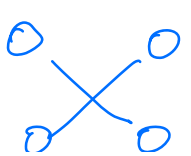
a sequence of edges starting node = ending node



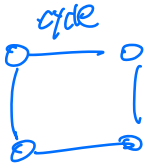
Tree



not connected



cycle

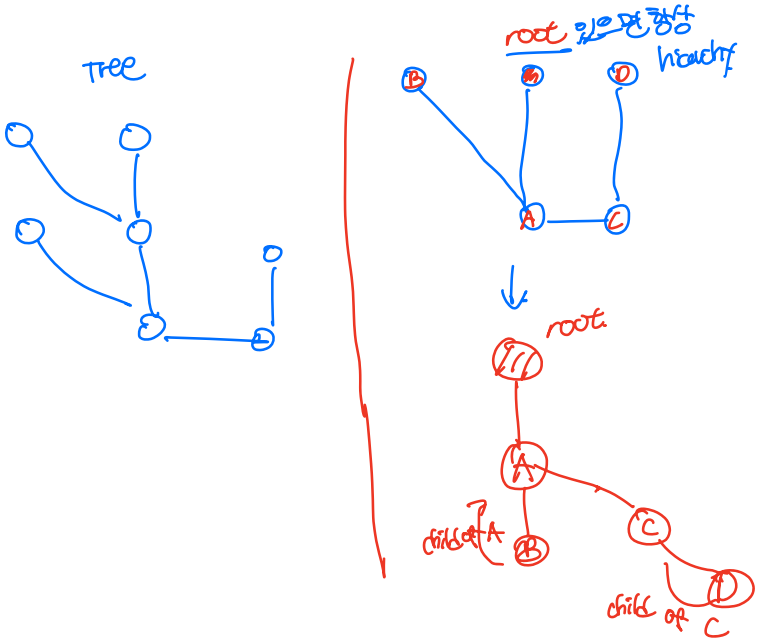


↓

connected Tree

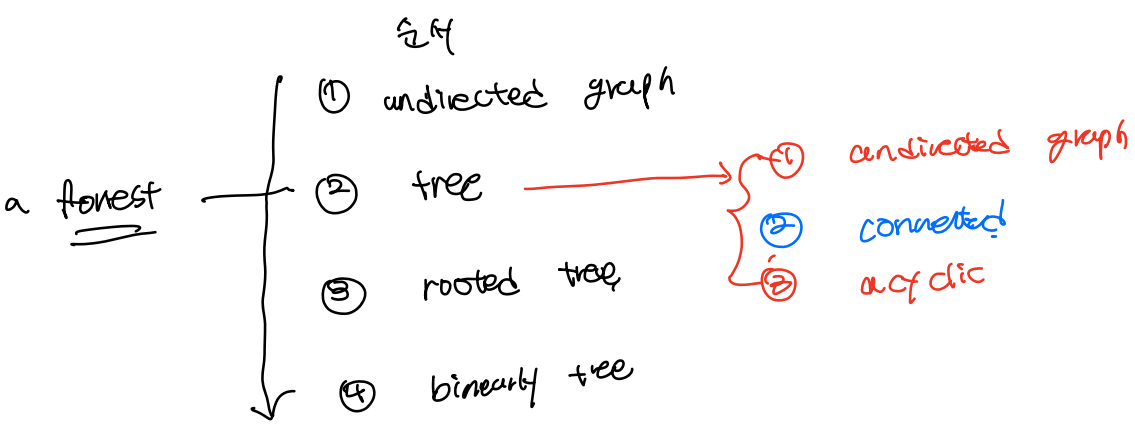
a designated node

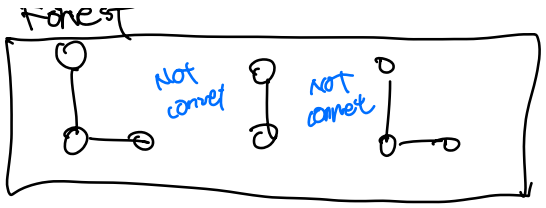
root



A binary tree

- a rooted tree
- any node can have at most 2 children

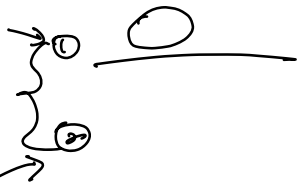




define

Max Heap

"shape"



max-heap property

complete binary tree

almost

- full binary tree

each level is completely filled

every complete binary → almost complete binary

(exception

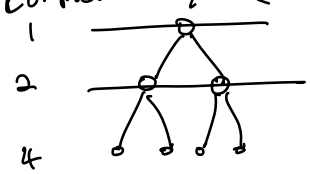
① undirected graph

② tree

③ rooted tree

④ binary tree

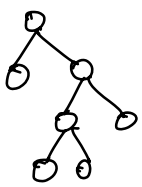
complete binary tree



full binary tree

each node

- zero children
- 2 children



Chapter 6 Heapsort

It is possible to design an algorithm using a data structure.

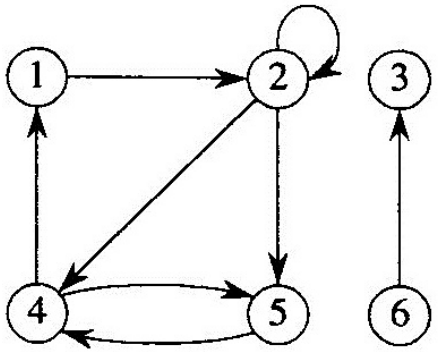
A (binary) heap is a special kind of a tree.

Let A be a non-empty set.

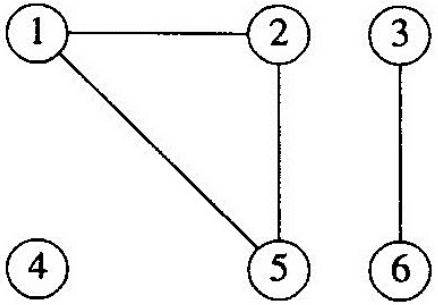
A relation on A is a subset of A X A.

A directed graph is a structure (V, E) in which V is a non-empty finite set and E is a relation on V.

An undirected graph is a structure (V, E) in which V is a non-empty finite set And E is a set of unordered pairs of V

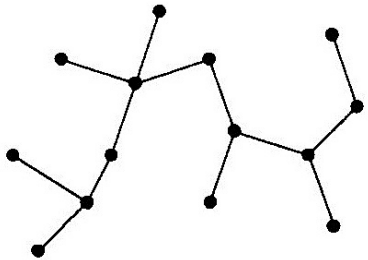


(a)

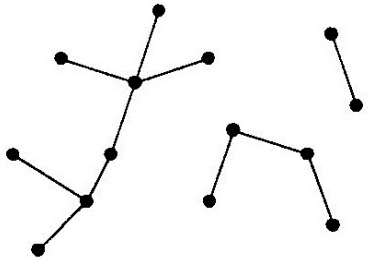


(b)

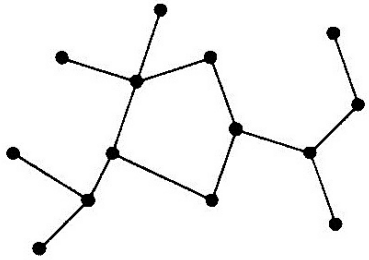
A tree is an undirected graph that is acyclic and connected.
A forest is an undirected graph that is acyclic.
A rooted tree is a tree in which a node is designated as the root.



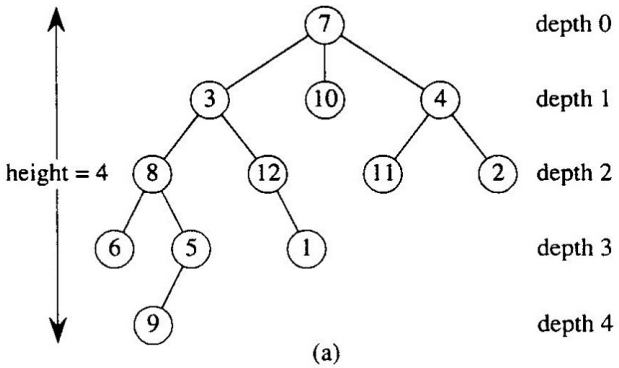
Not a rooted tree



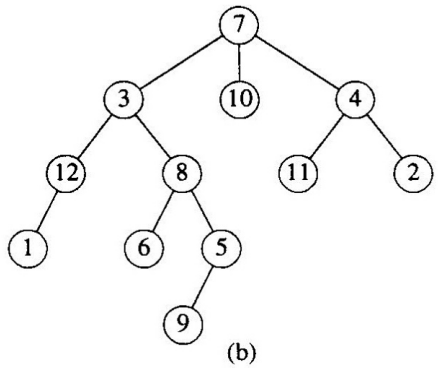
forest



Undirected graph



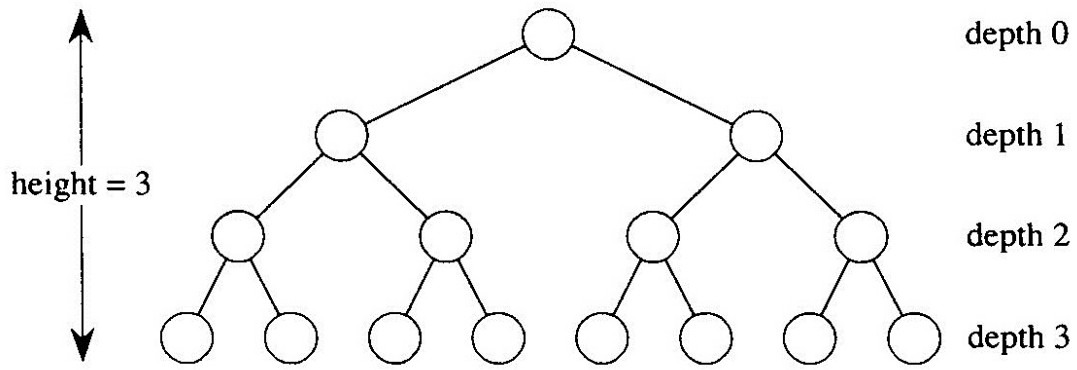
Rooted tree



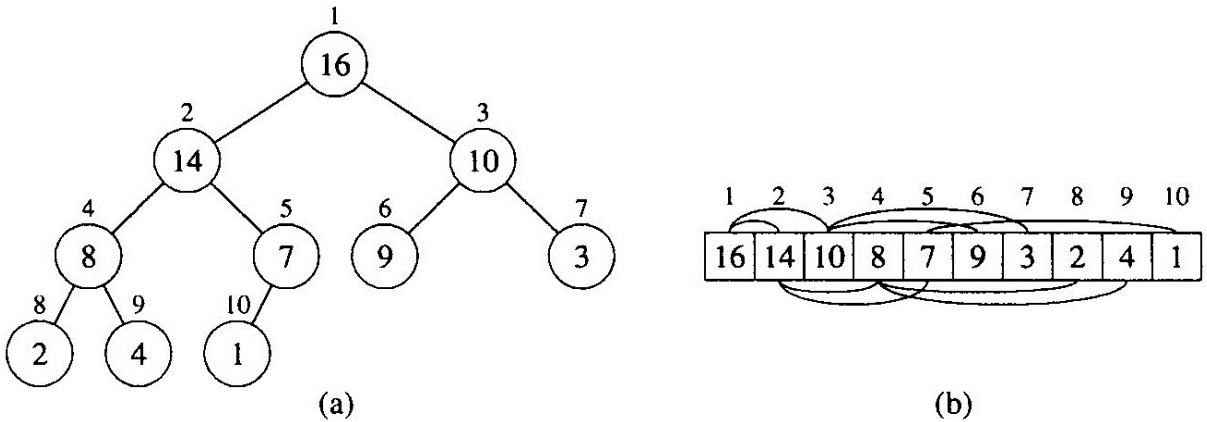
(a) And (b) are different rooted trees

A binary tree is a rooted tree in which each node has AT MOST two children.

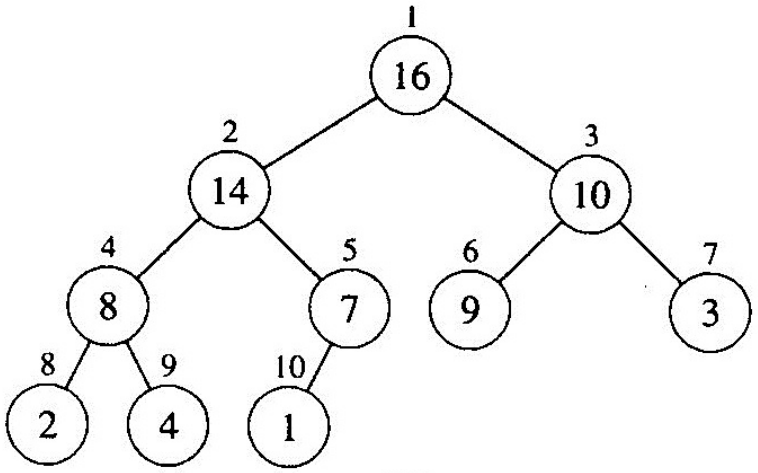
A complete binary tree is a binary tree in which every level is completed filled.



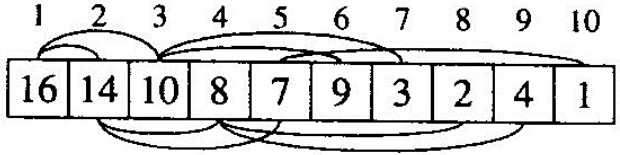
An almost complete binary tree is a complete binary tree with one except at the last level
-(1) it does not have to be completely filled.
-(2) nodes should be filled from the leftmost part toward right direction
Note that a complete binary tree is an almost complete binary tree



A max heap is an almost complete binary tree with “max-heap property”



(a)



(b)

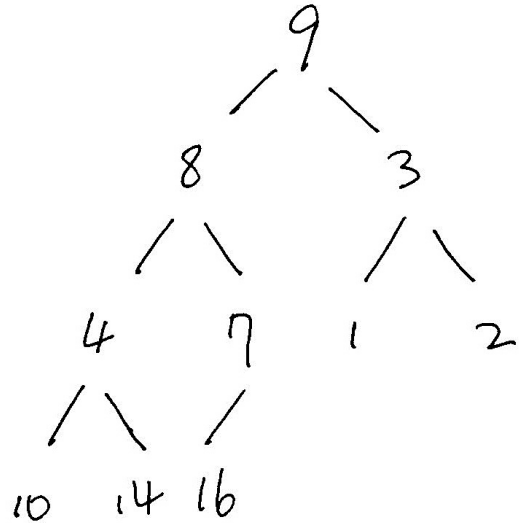
PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Index of node that contains 10: 3

Index of the parent of this node: 1
Index of the left child of this node: 6
Index of the right child of this node: 7



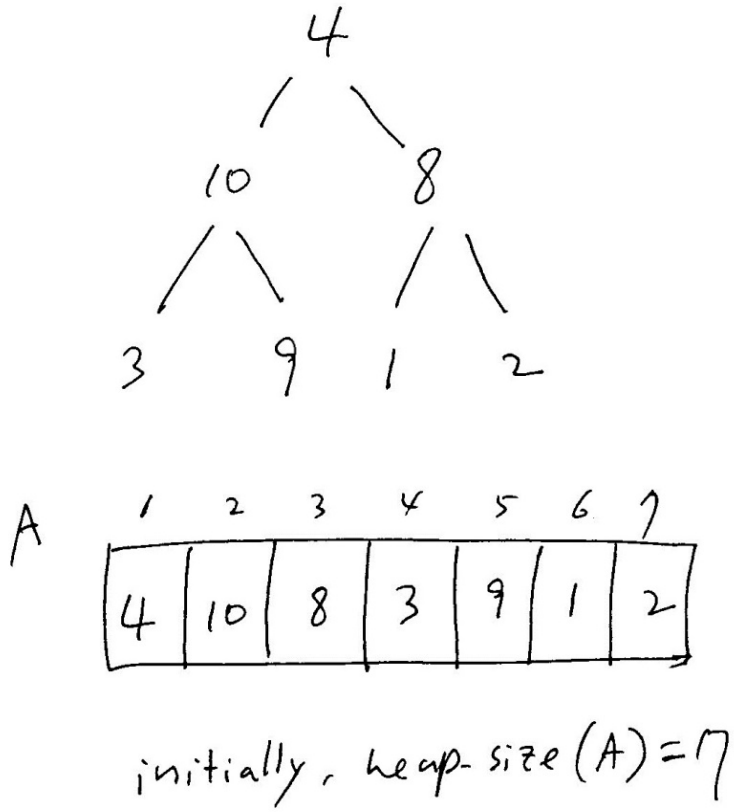
A

9	8	3	4	7	1	2	10	14	16

heap-size (A) = 7

Before the execution of MAX-HEAPIFY
-left subtree is a max-heap
-right subtree is a max-heap

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

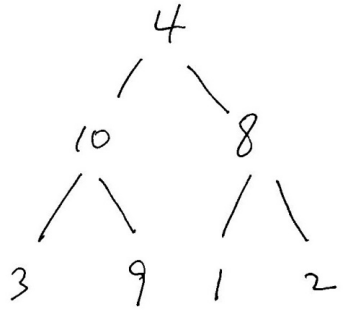


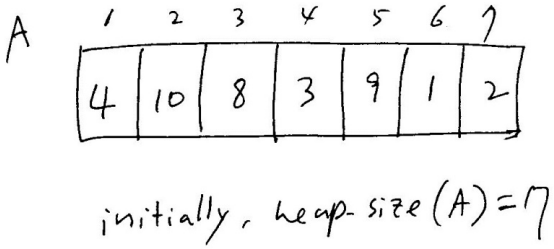

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY(A,1)

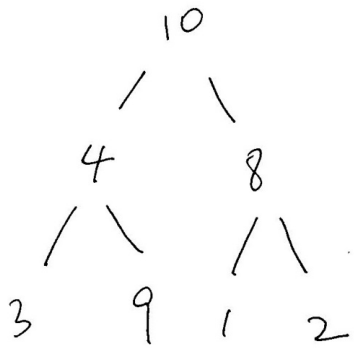
Because LEFT(1) = 2, RIGHT(1) = 3 and
2 ≤ 7, A[2] > A[1], largest = 2 In addition,
3 ≤7, but A[3] is not greater than A[2]
in line 8, 2 ≠ 1, therefore in line 9, swapping is done
and in line 10, MAX-HEAPIFY(A, 2)

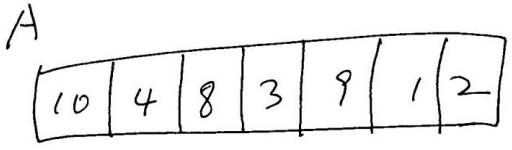
Before MAX-HEAPIFY(A,1)





After MAX-HEAPIFY(A,1)



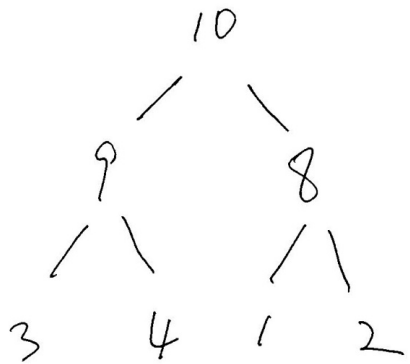


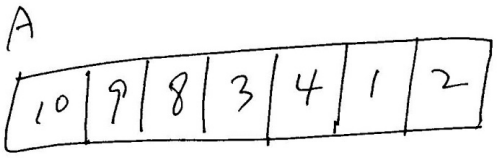

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY(A,2)

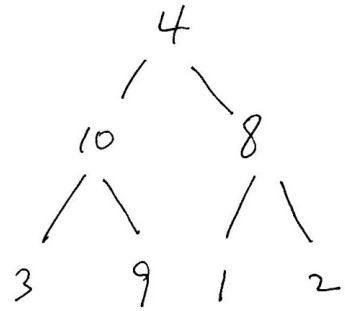
Because LEFT(2) = 4, RIGHT(2) = 5 and $4 \leq 7$, but A[4] is not greater than A[2], largest = 2.
in line [6], $5 \leq 7$ and A[5] > A[2], therefore largest=5
in line 8, $5 \neq 2$, therefore in line 9, swapping is done and in line 10, MAX-HEAPIFY(A, 5)

After MAX-HEAPIFY(A,2)






```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```



The input before MAX-HEAPIFY executes

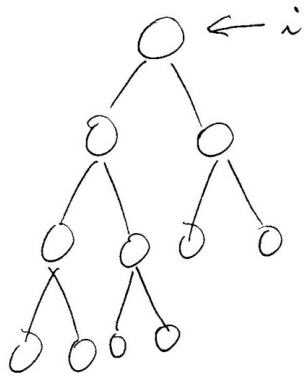
MAX-HEAPIFY(A,5)

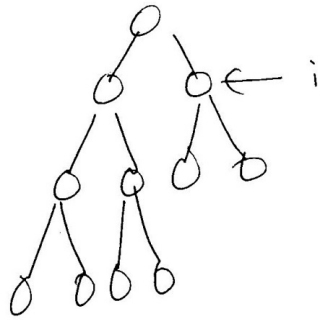
Because LEFT(5) = 10, RIGHT(2) = 11 and
10 is greater than 7, largest = 5.
in line [6], 11 is greater than 7 line 7 is not executed

in line 8, largest is the same as i
this terminates.

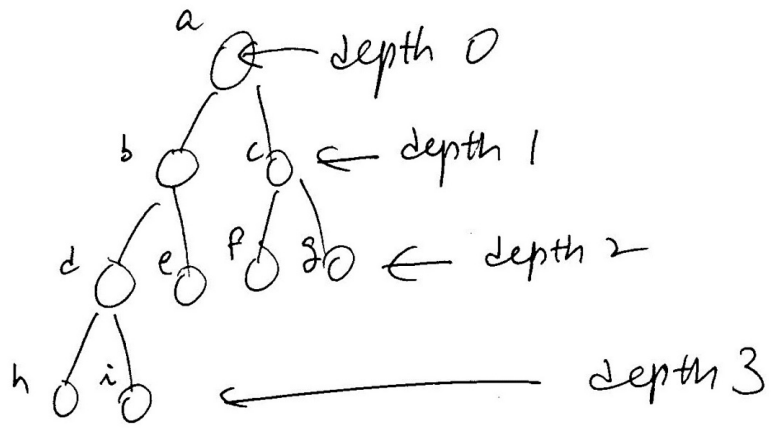
Time complexity of MAX-HEAPIFY – depends on the position of i

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```





The height of a node : 0 at the leaf, grows upwards
The height of a tree: the height of the root node
The depth of a node: 0 at the root, grows downwards



height 0 : h, i, e, f, g

height 1 : d, c

✡ height 2 : b

✡ height 3 : a

the height of this tree = the height of the root = 3

The height of an n-element heap is $O(\lg n)$

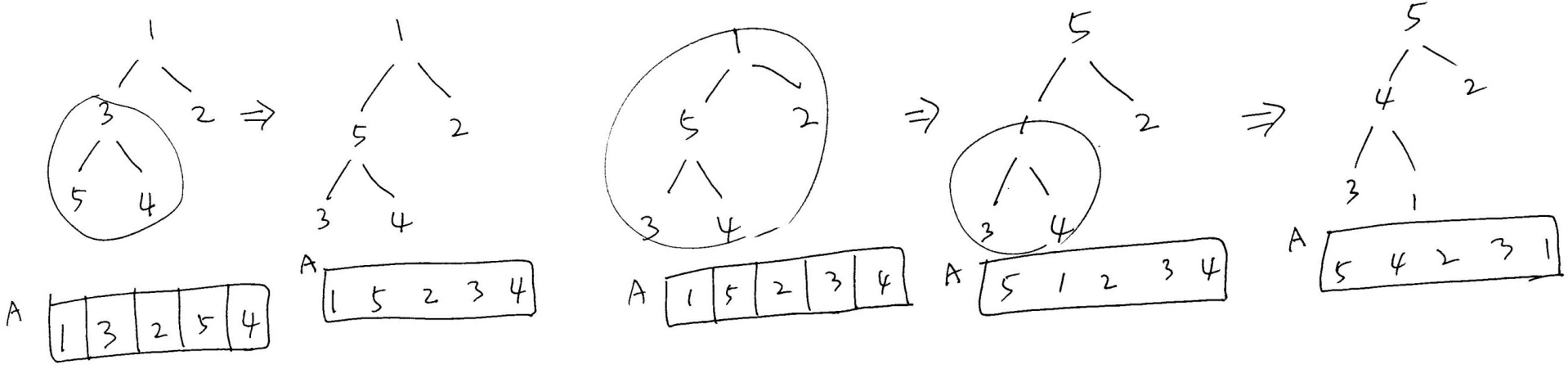
There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h

in any n-element heap.

Time complexity of MAX-HEAPIFY - O(lg n)

BUILD-MAX-HEAP(A)

1 $heap\text{-}size[A] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A, i)



BUILD-MAX-HEAP(A)

1 $heap\text{-}size[A] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A, i)

Time complexity

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

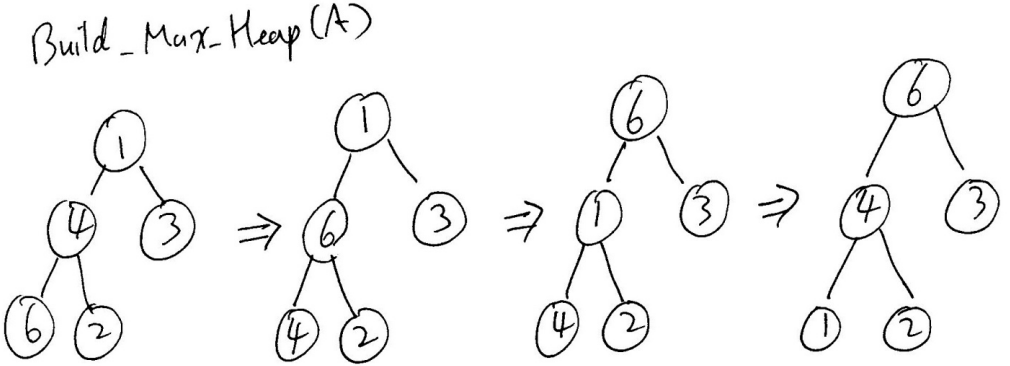
$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

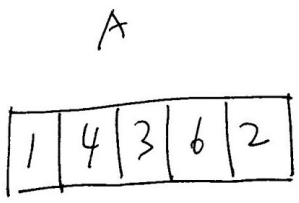
$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$

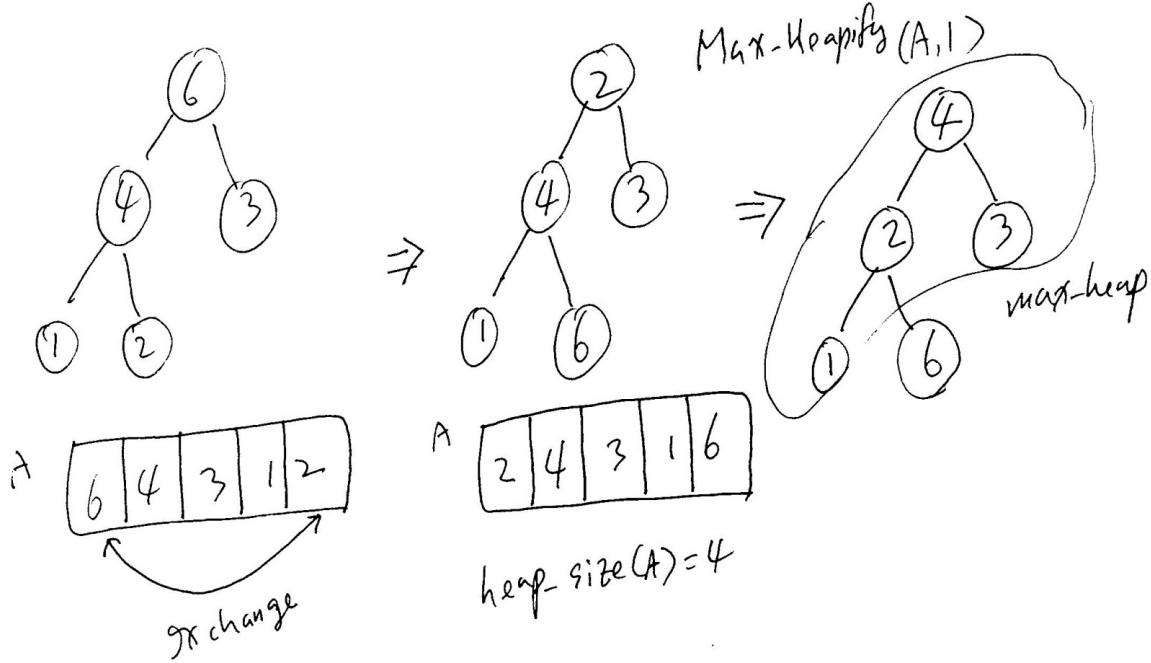
$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n) .$$

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

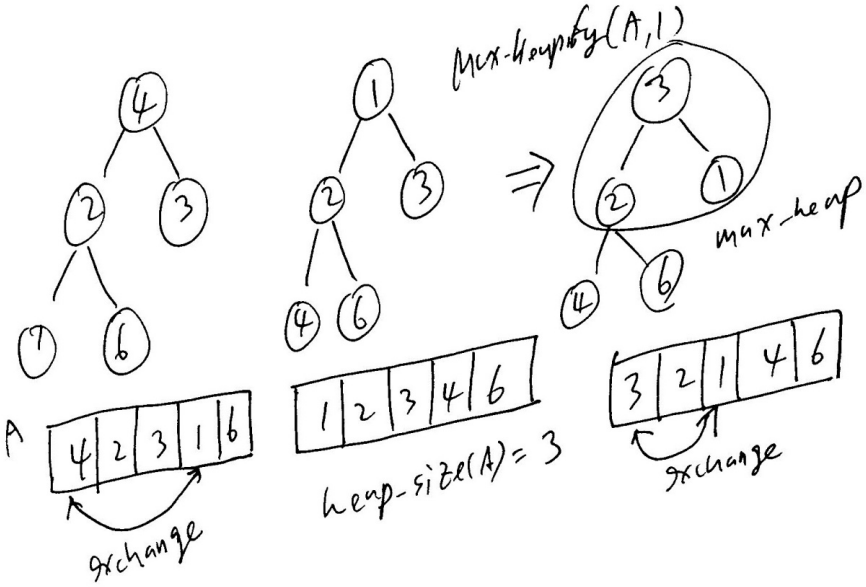




```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```



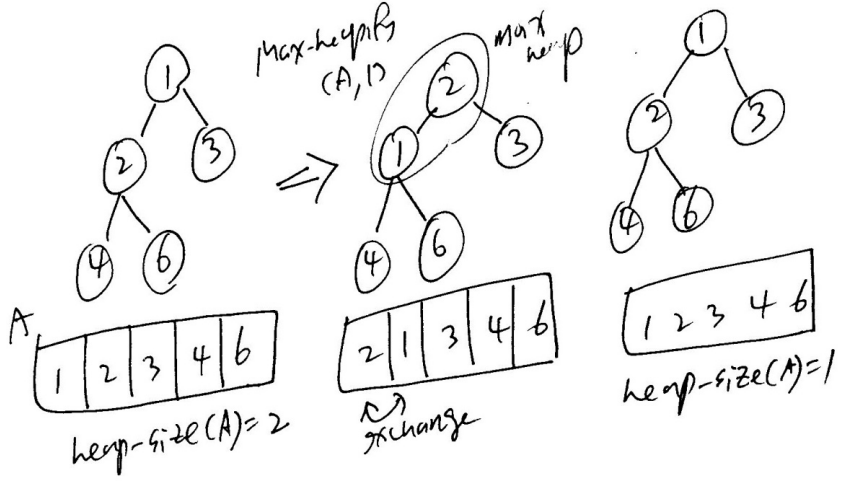
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```



Time complexity

1 O(n)
2 O(n) iterations
  O(lg n) for each iteration

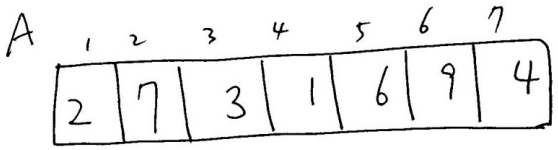
Therefore O(n * lg n)



```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

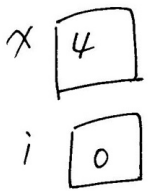
```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



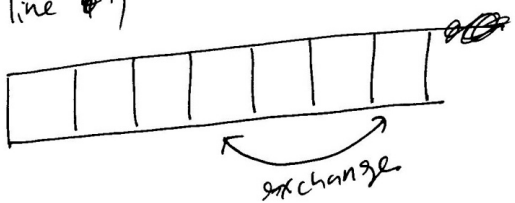
partition (A, 1, 7) p=1 r=7

Assume that the pivot element = 4



line 3 ~ line 6
find the number of elements ≤ x

line 7



return 4

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

A

2	7	3	1	6	9	4

partition(A, 1, 7) p = 1, r = 7

x: 4 i: 0

j = 1 line 4 : true, line 5 : i ← 1, line 6 : A[1] ↔ A[1]

j = 2 line 4 : false

j = 3 line 4 : true, line 5 : i ← 2, line 6 : A[2] ↔ A[3]

A

2	3	7	1	6	9	4

j = 4 line 4 : true, line 5 : i ← 3, line 6 : A[3] ↔ A[4]

A

2	3	1	7	6	9	4

j = 5 line 4 : false

j = 6 line 4 : false

line 7 : A[4] ↔ A[7]

A

2	3	1	4	6	9	7

line 8 : return 4