Chapter 2

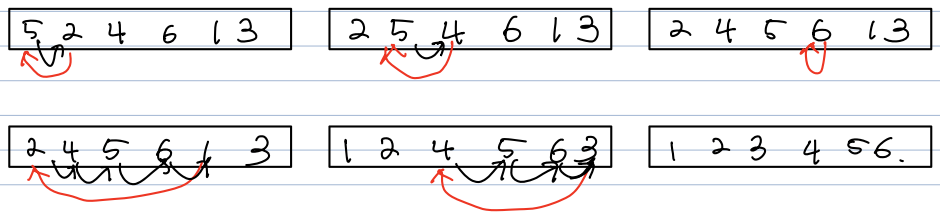
Insertion sort.

Input (입력) : n개 수들의 수열 $\langle a_1, a_2, \cdots a_n \rangle$

Output (출력) : $a_1' \le a_2' \le \cdots a_n'$ 을 만족하는 순열 (재배치)
Not decreasing → 모든 정렬이 이렇다

key : 정렬하고자 하는 숫자
n개인 입력 : A.length

(a)

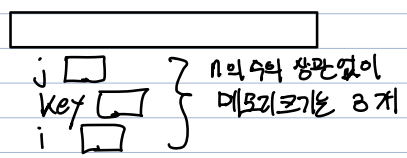


Insertion-Sort(A)

1. for j = 2 to A.length: n-1 번 반복
2. key = A[j]
   // A[j]를 정렬된 배열 A[1, ..., j-1]에 삽입한다.
3. i = j-1
4. while 0 <= i 그리고 key < A[i]: key가 A[i] 보다 작고 1 보다 클때 까지 (배열에 범위라는뜻)
5. A[i+1] = A[i]
6. i = i-1 : i를 뒤에서 부터 비교하면서 바꿔서 나간다.
7. A[i+1] = key

Analysis

Space



In-place sorting

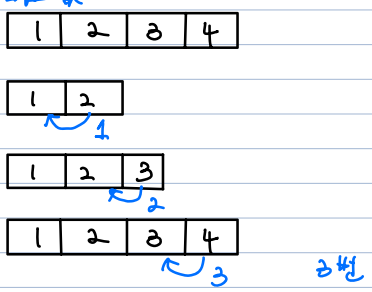
Time

```
1.  for j = 2 to A.length:        n-1 번 반복
2.      key = A[j]
3.      i = j - 1
4.      while 0 <= i 그리고 key < A[i]:
5.          A[i+1] = A[i]
6.          i = i - 1
7.      A[i+1] = key
```

두개의 루프

special 배열과 언제 끝날지 모른다.

최소 값



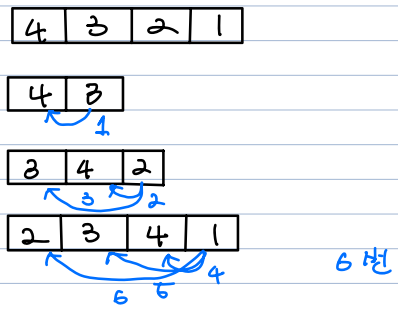
3번

최소 값 : n-1

↓

j 안반복

최대 값



6 번

최대값 : $1+2+3+\cdots+n-1$

$= \frac{n(n-1)}{2}$

Merge Sort

1. Divide (분할) : 정렬할 n개의 원소의 배열을 n/2 개씩 부분수열 두개로 분할한다.
2. Conquer (정복) : 병합 정렬을 이용해 두부분 수열을 재귀적으로 정렬한다.
3. Combine (결합) : 정렬된 두개의 부분 배열을 병합해 정렬된 배열 하나로 만든다.

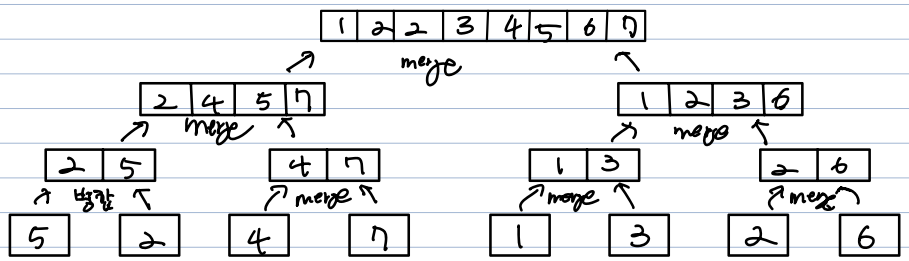
Input (입력) : n개 수들의 수열 $\langle a_1, a_2, \cdots a_n \rangle$

Output (출력) : $a_1' \le a_2' \le \cdots a_n'$ 를 만족하는 순열 (재배치)

Not decreasing → 모든 정렬이 이렇다

용 : 분할기준의 인덱스 $\frac{p+r}{2}$,

r : A.length
p : 1



Merge-Sort (A, P, r)

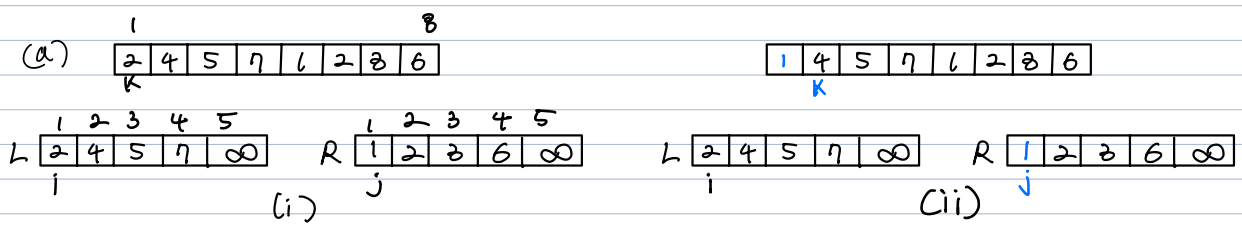
1. if p < r:
2. q = ⌊(p+r)/2⌋ → runtime T.
3. Merge-Sort (A, P, q) T(n/2)
4. Merge-Sort (A, q+1, r) T(n/2)
5. Merge (A, P, q, r) → runtime Cn

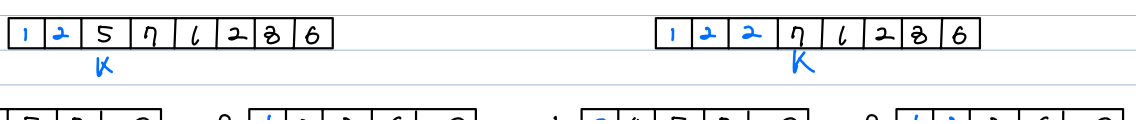
Merge

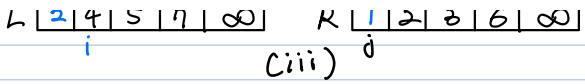
Input (입력) A, p, q, r , p ≤ q < r A[p, q], A[q+1 ~ r] 은 모두 정렬되었다고 가정

Output (출력) p ~ r 인덱스의 정렬된 A

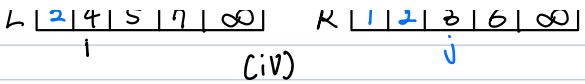
$n_1 = q - p + 1$ part 1의 개수
$n_2 = r - q$ part 2의 개수



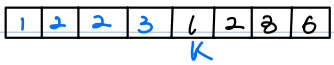


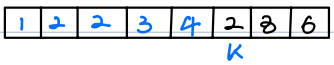


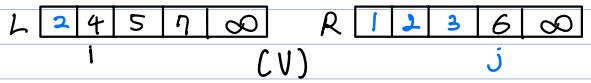
(iii)



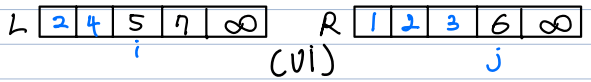
(iv)



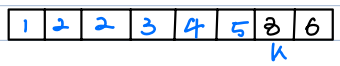


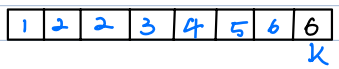


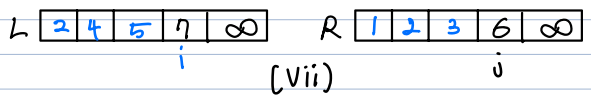
(v)



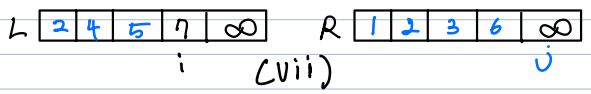
(vi)



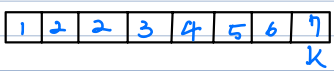


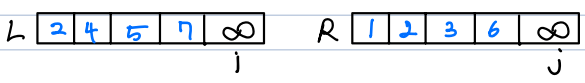


(vii)



(vii)



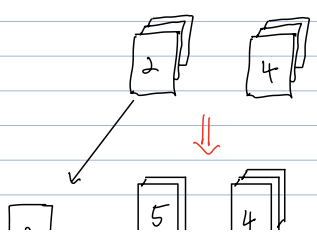


(ix)

Merge(A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. 배열 $L[1 \ldots n_1+1]$ 과 $R[1 \ldots n_2+1]$ 을 생성한다. Space Resource.
4. for i = 1 to n_1
5. $\quad L[i] = A[p+i-1]$
6. for j = 1 to n_2
7. $\quad R[j] = A[q+j]$
8. $L[n_1+1] = \infty$
9. $R[n_2+1] = \infty$
10. i = 1
11. j = 1
12. for k = p to r:
13. $\quad$ if $L[i] \le R[j]$:
14. $\quad\quad A[k] = L[i]$
15. $\quad\quad i = i+1$
16. $\quad$ else $A[k] = R[j]$

두 배열에 숫자 복사



종카드 수만큼 비교와 갈아.

17. $j = j+1$

Merge 함수의 runtime → n

Analysis

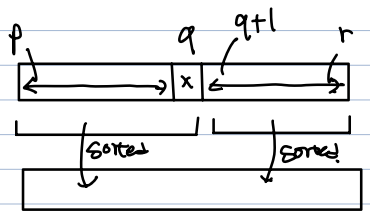
Space

Merge (A, p, q, r)

need an extra space (merge)

100개가 있으면 100개 만큼을 다시 copy

(A)



Time.

Merge-Sort (A, P, r)

```
1.  if p < r:
2.      q = ⌊(p+r)/2⌋
3.      Merge-Sort (A, p, q)
4.      Merge-Sort (A, q+1, r)
5.      Merge (A, p, q, r)
```

→ runtime 1. (c)

$T(\frac{n}{2})$

$T(\frac{n}{2})$

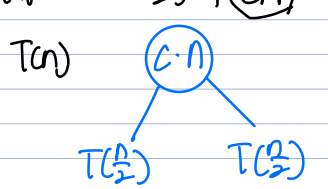
→ runtime cn

$cn + c = cn$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ c & n = 1 \end{cases}$$

$T(n) = c$ divide, combine

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



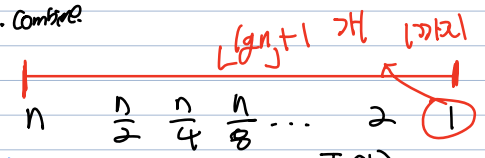
D & C

1. d
2. c
3. combine

$\log_2 8 = 3$

8 4 2 1

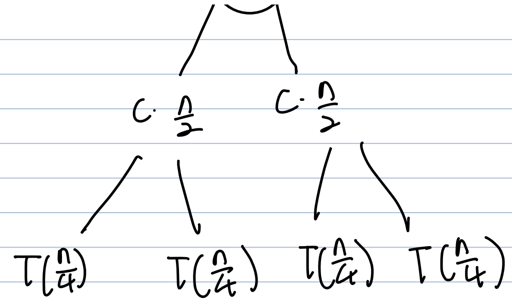
$\log n + 1$ 개 까지



$\log n + 1$

$$T\left(\frac{n}{2^k}\right) \quad \frac{n}{2^k} = 1$$

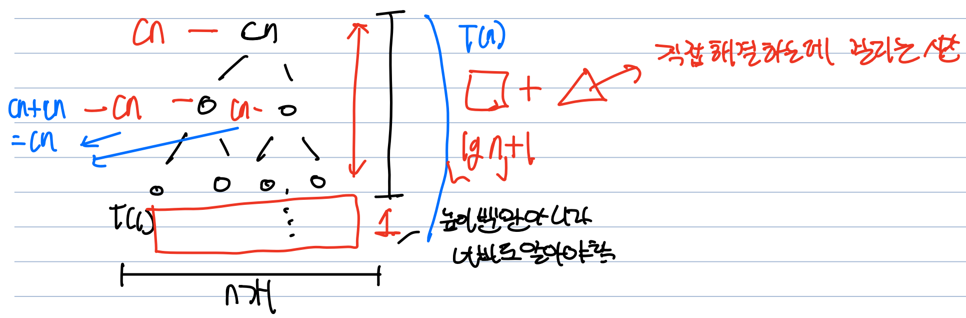
$k=0$ $T(n)$



$k=1 \quad T(\frac{n}{2}) \qquad 2^k = n$

$k=2 \quad T(\frac{n}{4}) \qquad \log_2 2^k = \log_2 n$

$k = \lg n$



높이가 2일 때 2^2

$T(n) = cn \cdot \boxed{\lg n} + c \cdot 1 \cdot \boxed{n}$

각 높이마다 모든 비용은 cn으로 같다.

$cn(\lg n + 1) = cn\lg n + cn \qquad O(cn\lg n + cn) = O(n\lg n)$

Chapter 6

Heapsort

max-heapify 시간성

Build-heap = (lgn)

그래프이론

자료구조 · 배열 / graph / tree

graph (undirected / directed)

사이클 원이 그려질때, (마지막 것이 첫번째것)

O ⟶ O

V = 노드 E = 엣지

Input (입력) : n개 수들의 수열 $\langle a_1, a_2, \cdots a_n \rangle$

Output (출력) : $a_1' \le a_2' \le \cdots a_n'$ 을 만족하는 순열 (재배치)

Not decreasing → 모든 점점이 이웃함

A.heapsize = A의 원소중 heap에 포함되는 원소의 개수

A.heapsize $\le$ A.length

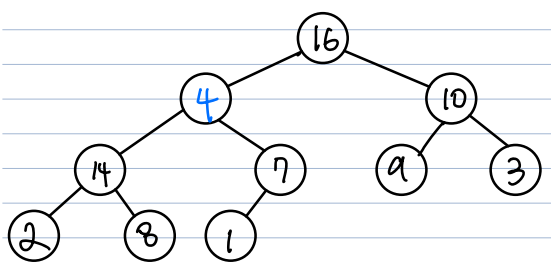
i = 원소의 인덱스

parent(i) = $\lfloor \frac{i}{2} \rfloor$

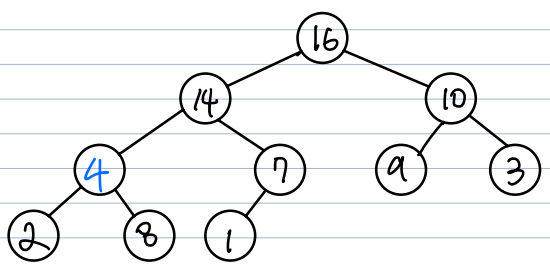
left(i) = $2i$

right(i) = $2i+1$

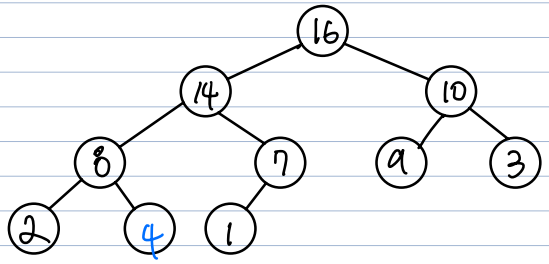
높이 : h 노드개수가 n개일때 높이는 lgn



(i)



(ii)



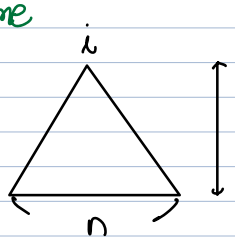
(iii)

Max-Heapify(A, i)

1. l = Left(i) left(i) = 2i
2. r = Right(i) right(i) = 2i+1
3. if l ≤ A.heap-size and A[l] > A[i]
4. largest = l] index를 저장
5. else largest = i
6. if r ≤ A.heap-size and A[r] > A[i]
7. largest = r
8. if largest ≠ i
9. exchange A[i] with A[largest]
10. Max-Heapify(A, largest) → 아래 인덱스로 밑으로 차츰 바꾸기

Analysis

Time



$O(\lg n)$

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

↓

build max-heap

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

$= \frac{n}{2} \sum_{h=0}^{\lg n} O(\frac{h}{2^h}) < \sum_{h=0}^{\infty} O(\frac{h}{2^h})$

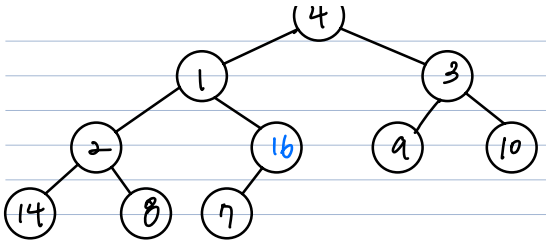
$\frac{1}{(1-\frac{1}{2})^2} = 2$

$\therefore O(n)$

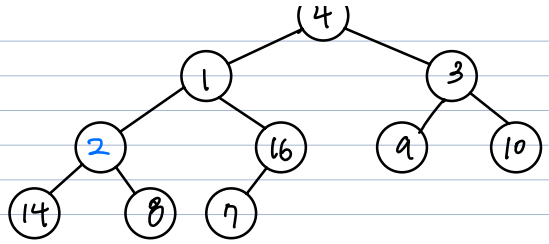
Build Max-heap

A

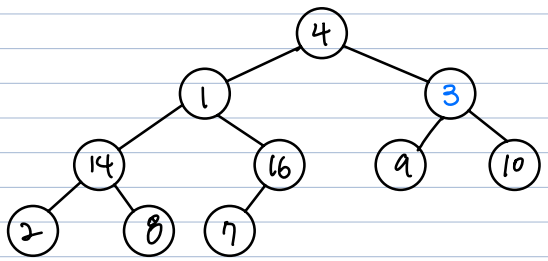
4	1	3	2	16	9	10	14	8	7



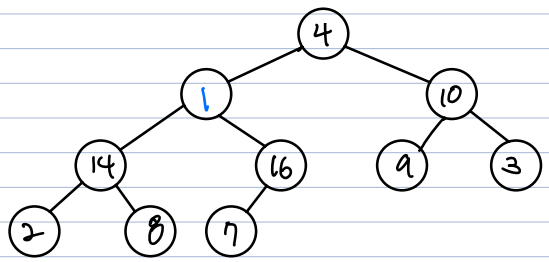
(i)



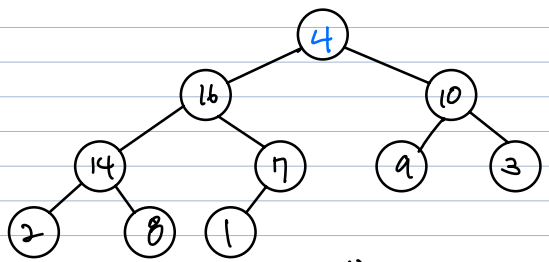
(ii)



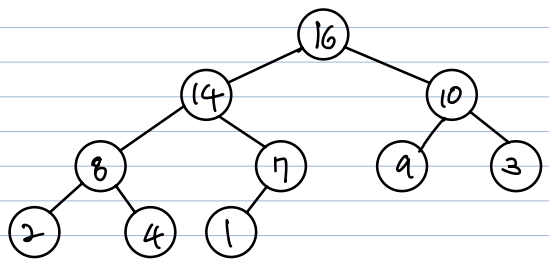
(iii)



(iv)



(v)



(vi)

Build-MAX-Heap(A)

1. A.heapsize = A.length
2. for i = $\lfloor \frac{A.length}{2} \rfloor$ down to 1 // child가 있는 노드만 recursion
3. Max-Heapify(A, i) → // O(lgn)

Analysis

Time

Max-heapify의 Time = O(lgn) = O(h)

원소가 n일때의 힙의 높이 = $\lfloor \lg n \rfloor$ = 높이 h

높이가 h 인 노드의 수는 최대 $\lceil \frac{n}{2^{h+1}} \rceil$

높이가 h 일때, Max-heapify 의 수행시간은 O(h)

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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} O\left(h \cdot \frac{1}{2^h}\right) < \sum_{h=0}^{\infty} O\left(h \cdot \frac{1}{2^h}\right)$$

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

$\therefore O(n)$

Heapsort (A)

1. Build Max-heapsort (A) → $O(n)$
2. for i = A.length down to (2) → $n-1$ 개
3. exchange A[1] with A[i]
   A.heapSize = A.heapSize − 1
   Max-Heapify (A, 1) → $O(\lg n)$

Time = $O(n) + (n-1)(O(\lg n))$

$= O(n \lg n) - O(\lg n) + O(n)$

↓

최대차항.

Runtime of Heapsort : $O(n \lg n)$

Chapter 7

Quick Sort

Input (입력): n개 수들의 수열 $\langle a_1, a_2, \cdots a_n \rangle$

Output (출력): $a_1' \le a_2' \le \cdots a_n'$ 을 만족하는 순열 (재배치)
Not decreasing → 모든 정렬이 이점자

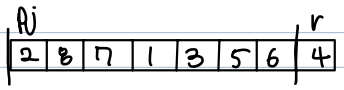
1. Divide (분할): 배열 A[p..r]을 A[p, q-1]과 A[q+1, r]로 분할, A[p..q-1]에는 A[q]보다 작은원소 배열, A[q+1, r]은 큰원소 배열로. 반이상 어지게 배치.
2. Conquer (정복): 퀵정렬을 재귀호출, 두 배열을 정렬
3. Combine (결합): 나뉜 배열은 이미 정렬되어 있어 결합할 필요가 없다.

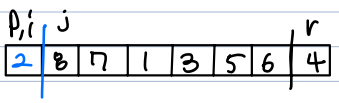
Partition.

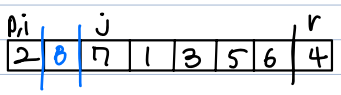
pivot: A를 나눌 기준원소 A[r]을 선택

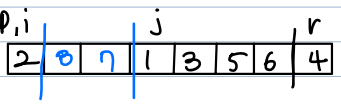
$p \le i \le r-1 \le r$

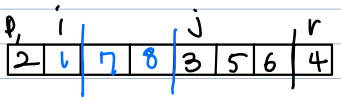
1. $p \le k \le i$ 이면 $A[k] \le x$
2. $i+1 \le k \le r-1$ 이면 $A[k] > x$
3. $k = r$ 이면 $A[k] = r$

(i)


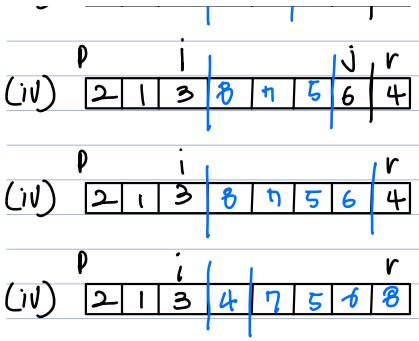
(ii)


(iii)


(iv)


(iv)


(iv)

partition (A, p, r)

1. pivot = A[r]
2. i = p-1
3. for j = p to r-1
4. if A[j] ≤ pivot → partition의 수행시간을 결정
5. i = i+1
6. A[i] ↔ A[j] 교환
7. A[i+1] ↔ A[r] 교환
8. return i+1

Analysis

Time $\Theta(n)$ → 전부 하나하나 pivot과 비교하기 때문

space array 내에서 정렬 / No memory need

Quicksort (A, p, r)

1. if p < r
2. q = partition (A, p, r) → $\Theta(n)$
3. Quicksort (A, p, q-1)
4. Quicksort (A, q+1, r)

Analysis

Time

Worst case

분할을 n-1과 0개씩이라고 나눌때.

$T(n) = T(n-1) + T(0) + \Theta(n)$
$= T(n-1) + \Theta(n)$
$= T(n^2)$

Best case

분할을 각각 $\frac{n}{2}$ 과 $\frac{n}{2}$ 로 될때

$T(n) = 2T(n/2) + \Theta(n)$
$= \Theta(n \log n)$

균등하게 반으로 나눈다.

랜덤화된 퀵정렬

Randomized - partition (A, p, r)

1. i ← Random (p, r)
2. exchange A[r] ↔ A[i]
3. return Partition (A, p, r)

Randomized-Quicksort (A, p, r)

1. if p < r
2. q = Randomized-partition (A, p, r) → O(n)
3. Randomized-Quicksort (A, p, q-1)
4. Randomized-Quicksort (A, q+1, r)

Analysis

Time

Worst case

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q+1)) + \Theta(n)$$

$T(n) < cn^2$ → worst case에서는 quicksort

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q+1)^2) + \Theta(n)$$

$$= c \cdot \max_{0 < q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

상한값은 $\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2$

$$= n^2 - 2n + 1$$

$T(n) \le cn^2 - c(2n-1) + O(n)$

$\le cn^2$

평균 수행시간

partition (A, p, r)

1. pivot = A[r]
2. i = p-1
3. for j = p to r-1
4. if A[j] ≤ pivot → partition의 수행시간을 결정
5. i = i+1
6. A[i] ↔ A[j] 교환
7. A[i+1] ↔ A[r] 교환
8. return i+1

→ 4번째 라인에서 비교하는 횟수를 X라고하자

그러면 수행시간은 $O(n+X)$ 가된다.

ex) $X=n^2 \rightarrow O(n^2)$
$X=\lg n \rightarrow O(n)$

$z_{ij} = \{ \ 3$

2 3 1 5 4
A = 4 5 1 8 7 6

$z_{2,4} = \{4, 5, 6\}$
↑
2번째로 작은수 ~ 4번째로작은수

- Exp (X) X =

ind random var

$x_{ij} \begin{cases} 1 & \text{if } z_i, z_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$

기대값을구해 4 line에서 몇번비교하는지 알아낸다.

$E[X] = E\left[\sum_{\ }^{n-1} \sum_{\ }^{n} x_{ij}\right]$

Ex) {1, ... 10} 이라고할때
첫 pivot이 2라고가정했을때

{1, 2, ... 6} 과 {8, 9, 10} 으로
나뉘어진다 7은 1~10까지 자신을
제외한 모든 원소와 비교하지만,
2와 9는 절대 비교될일이 없다.

$\Big[\; i=0 \quad j=i+1 \qquad \Big]$

$$= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} E[x_{ij}]$$

↓ $z_i < x < z_j$ 일때

z_i 와 z_j 는 절대 비교될일이 없다.

z_i와 z_j가 비교되려면

$\{z_i, z_{i+1}, \ldots z_j\}$ 일때

첫 pivot은 z_i 아니면 z_j이어야함

$$E[x_{ij}] = Pr[x_i] + Pr[x_j]$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

$$= \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}, \quad k = j-i$$

$$= \sum_{i=0}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1} < \sum_{i=0}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=0}^{n-1} O(\log n)$$

$$= O(n \log n)$$

Chapter 1.

Halting problem

Computing machine A → solves problems in arithmetic (산술)

Input: Receives a problem.

Output: prints the answer (Always the right answer)

Computing machine C → plays checkers

Input: Receives a correct state of the board

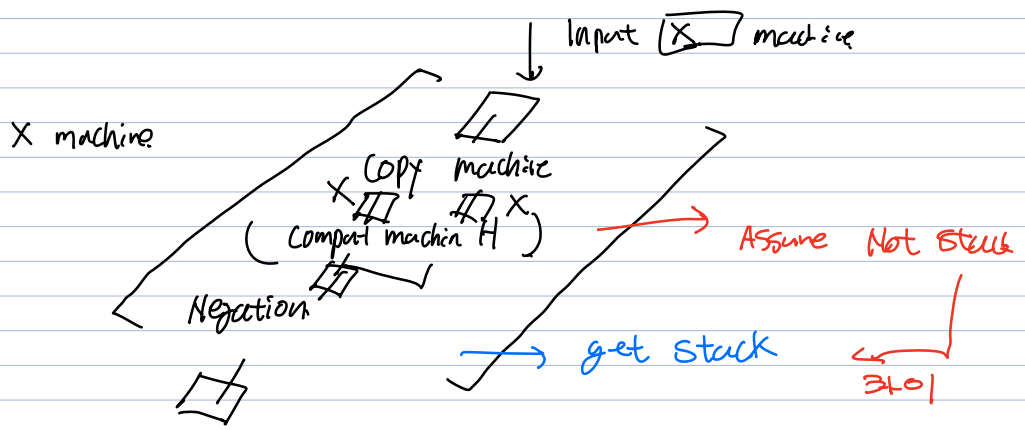
output: prints how to move one of the best pieces (Never lose a game)

When computing machine A receives a checker board it Stucks
When computing machine C receives a arithmetic it stucks.

Computing machine H → solves Halting problem

Inputs: program A, Input x

output: determine the given program on the given input will stuck or not.
(Always the right answer)



Chapter 15

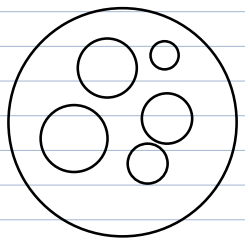
Dynamic programming

① optimal substructure property
② Overlapping substructure property
③ greedy choice property

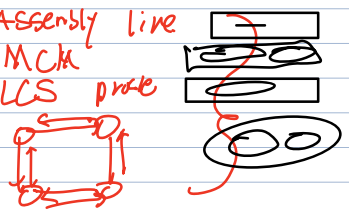
→ DP property

1. An optimal substructure property. "Shape"

한개의 문제를 여러개로 나누어서
sub 문제의 해도 optimal solution이
나오는



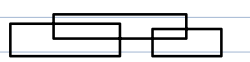
① Assembly line
② MCM
③ LCS prob
④



Overlapping subproblem property

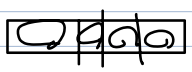
$nC_r = {}_{n-1}C_r + {}_{n-1}C_{r-1}$

DP

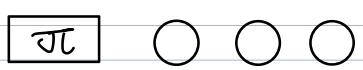


structure name
overlapping subproblem

D & C



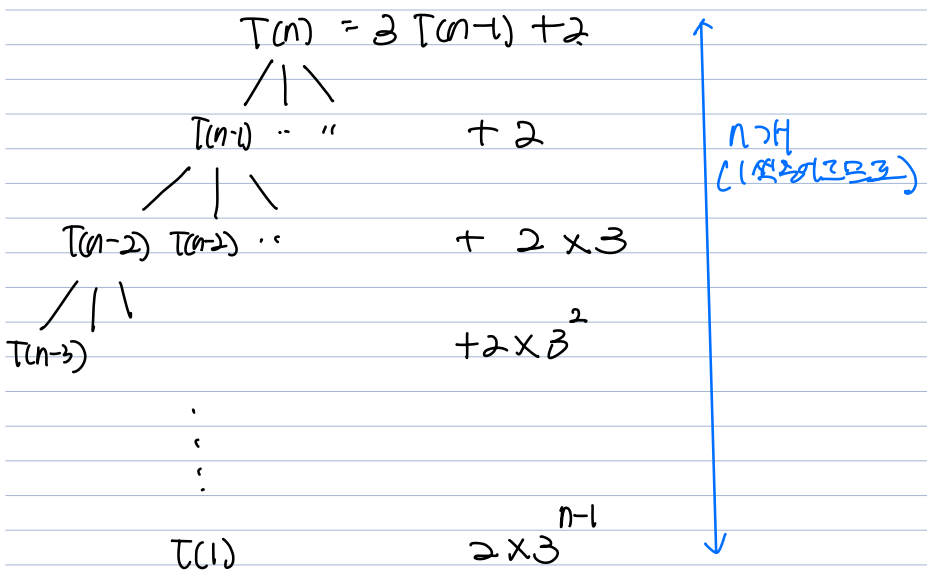
do not share
independently



check overlapping part

알고리즘 기초

1. $T(n) = 3T(n-1) + 2$, $T(1) = 1$ 이 있다면, $T(n)$ 을 정확히 구하여라.



$$= 2(1 + 3^1 + 3^2 + \cdots 3^{n-1})$$

$$= a\frac{r^n - 1}{r - 1} = 2\frac{3^n - 1}{3 - 1} = 2 \times \frac{3^n - 1}{2} = 3^n - 1$$

$$\therefore O(3^n - 1) = O(3^n)$$

$$X = a(1 + r + r^2 + r^3 + \cdots r^{n-1})$$

$$Xr = a(r + r^2 + r^3 + \cdots r^n)$$

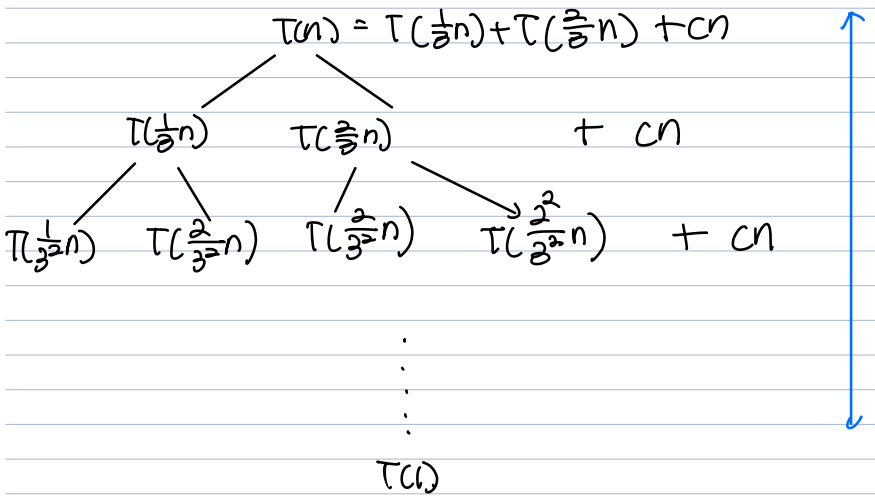
$$Xr - X = a(r^n - 1)$$

$$X(r - 1) = a(r^n - 1)$$

$$a(r^n - 1)$$

$x = \frac{\text{비교}}{r-1}$

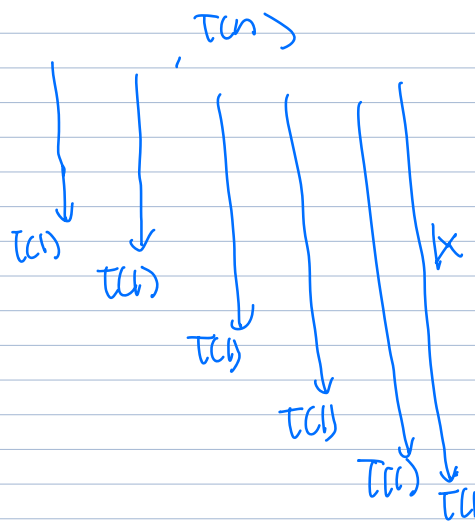
(2) $T(n) = T(\frac{1}{3}n) + T(\frac{2}{3}n) + cn$



$n \times (\frac{1}{3})^k \longrightarrow 1$

$n \times (\frac{2}{3})^k \longrightarrow 1$

들중 1에서 가장 가깝게 수렴하는 것은 $(\frac{1}{3})$ 그러므로



공식으로

$n \times (\frac{2}{3})^k = 1$

$n (\frac{2^k}{3^k}) = 1$

$n = \frac{3^k}{2^k} = (\frac{3}{2})^k$

$k = \log_{\frac{3}{2}} n$

$\therefore O(\log_{\frac{3}{2}} n) \times O(cn)$

$= O(n \log_{\frac{3}{2}} n)$

2. 알고리즘을 디자인하는 법 공부하기

1. Incremental Approach.

Incremental Approach to design an algorithm is given a sequence of input, and finds a sequence of solutions that build incrementally while adapting to the changes in the input.

Insertion sort → Each loop change sorting from the front to rear as incremental approach

2. Divide & conquer.

merge & quick sort

1. Divide (분할) : 정렬할 n개의 원소의 배열을 n/2 개씩 부분수열 두개로 분할한다.
2. Conquer (정복) : 병합 정렬을 이용해 두부분수열을 재귀적으로 정렬한다.
3. Combine (결합) : 정렬된 두개의 부분 배열을 병합해 정렬된 배열 하나로 만든다.