🍏

ch.03 분할 정복 알고리즘

분할정복 [분할 → 부분문제 ⋯→ 부분해
부분문제 ⋯→ 부분해
⋮] 정복

Divid and Conquer

: 주어진 문제의 입력을 분할하여 문제를 해결(정복)하는 방식의 알고리즘

- subproblem(부분문제) : 분할된 입력에 대한 문제
  - ex). 입력 크기가 n 부분 문제의 크기를 n/2 라고 하면 k번 분할
    - k = log2n

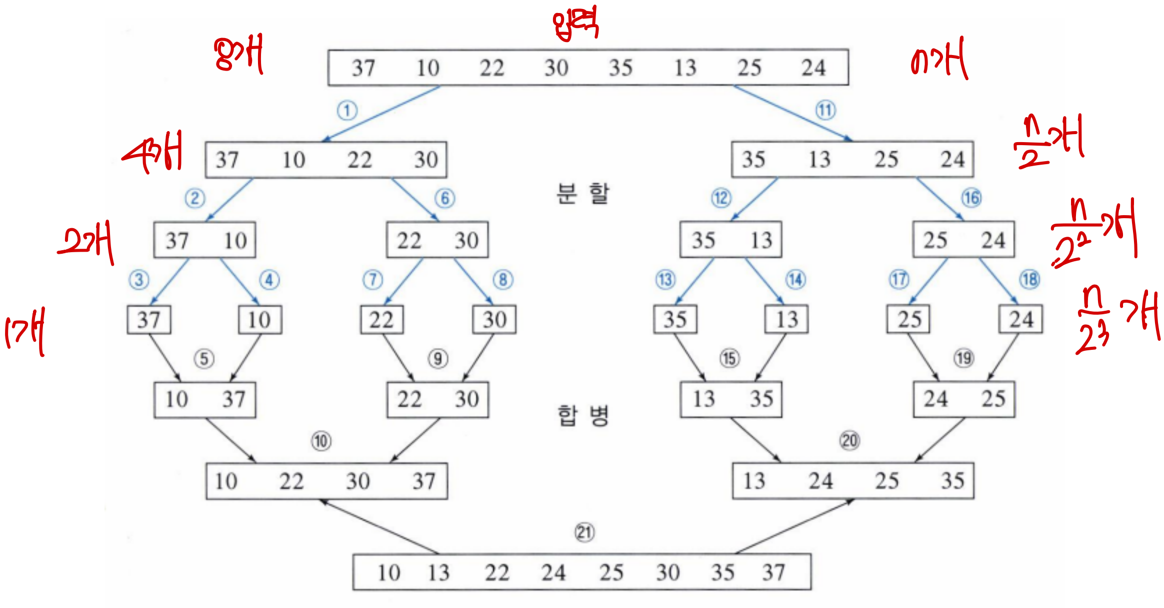
입력크기가 n
개수 a
부분크기 $\frac{n}{b}$
$(a = \log_b n)$ ★

$\boxed{\frac{n}{2^k} = 1} \rightarrow n = 2^k$
$k = \log_2 n$

외부정렬의 기본

3.1 Merge Sort(합병 정렬)

: 입력이 2개의 부분문제로 분할되고, 부분문제의 크기가 1/2로 감소



[그림 3-2]

- n개의 숫자 → n/2씩 2개의 부분문제로 분할(Divide)

→ 각각의 부분문제를 순환적으로 합병 정렬 → 2개의 정렬된 부분을 합병하여 정렬 (Conquer)

→ 배열, idx 0~끝

p=q이면 그 자체로 정렬된 것.

→ 정렬할 부분이 존재해야 함.

```
MergeSort(A,p,q)

//입력 : A[p] ~ A[q]
//출력 : 정렬된 A[p] ~ A[q]

if(p<q){ //배열의 원소의 수가 2개 이상이면
  k=[(p+q)/2] //k=반으로 나누기 위한 중간 원소의 인덱스
  MergeSort(A,p,k) //앞부분 순환 호출
  MergeSort(A, k+1, q) //뒷부분 순환 호출
  A[p]~A[k]와 A[k+1]~A[q]를 합병 //merge
}
```

→ mid (홀수일 경우, 소수점 절삭)

→ 임시배열 B[p]~B[q]를 A[p]~A[q]로 복사.

- **time complexity**
  - 일반적으로 숫자의 비교 횟수로 나타냄
  - divide : O(1)

    1) 배열의 중간 인덱스 계산

    O(n)

    2) 2번의 recursion 호출 → stack 공간복잡도만 차지, 시간복잡도는 constant time
  - Conquer : O(n+m)

    1) 최대 비교 횟수 : 2개의 정렬된 배열의 크기가 각각 n, m 이라면 → n+m-1
      - 가장 마지막에 저장된 숫자는 비교할 숫자가 없으므로 비교 횟수에서 빠짐
  - Merge Sort : O(nlogn)

    1) 합병 한 번에서의 비교 횟수 : 각 층에서 수행된 비교 횟수는 O(n) (입력된 모든 숫자 참여)

    → 모든 숫자가 합병에 참여 ⇒ O(n)

    층마다 모든 원소 확인 ⇒ O(n) = O(n)

    2) 층 수 : log2n

    3) O(n) * log2n = O(nlogn)

    X 층이 log2n

    → O(nlogn) ★
- pros and cons

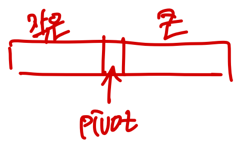
- 공간 복잡도 → O(n)
    - 대부분의 정렬 알고리즘들은 입력을 위한 메모리 공간과 O(1) 크기의 메모리 공간만을 사용
    - O(1) 크기의 메모리 공간: 입력 크기와 상관 없는 공간
    - Merge Sort

      +stack이 사용
        - 입력을 위한 메모리 공간 외에 추가로 입력과 같은 크기의 공간(임시 배열이) 필요
        - 합병된 결과를 저장할 공간이 필요

But. 보조배열 없이 구현 가능은 함. But Overhead가 ↑

3.2 Quick sort

: 정복 후 분할하는 알고리즘 → 문제를 2개의 부분문제로 분할
각 부분문제의 크기가 일정하지 않은 형태의 분할정복 알고리즘



- idea
    - pivot을 기준으로 작은 숫자는 왼편, 큰 숫자는 오른편에 위치하도록 분할
    - 분할된 부분문제들에 대하여 순환적으로 수행하여 정렬

→ 분할된 부분문제들에 포함되지 X

```
QuickSort(A, left, right)

//입력 : 배열 A[left]~A[right]
//출력 : 정렬된 배열 A[left]~A[right]

if(left < right){ //left와 right가 같으면 더 이상 정렬할 수 없는 경우 -> 원소가 하나
  pivot을 A[left]~A[right] 중에서 선택 -> pivot을 A[left]와 자리를 바꿈
  pivot과 배열의 각 원소를 비교하여 pivot보다 작은 숫자들은 A[left]~A[p-1]로 올김
  pivot보다 큰 숫자들은 A[p+1] ~ A[right]로 옮기며, pivot은 A[p]
  (계속 swap하면서 pivot을 기준으로 low와 high를 switch)

  //p가 pivot의 idx
  QuickSort(A, left, p-1)
  QuickSort(A, p+1, right) //배열의 크기가 커서 stack 공간 낭비 시 iteration
}
```

정렬이 끝남

정렬

pivot보다 작지만 가장 오른쪽 ⇒ next pivot!!

- **time complexity**
    - pivot 선택이 좌우함 → pivot의 크기에 따라 치우치는 정렬의 여부가 결정됨.
        - 최악 : pivot이 가장 작은 숫자가 선택되는 경우

- 비교 횟수 : (n-1) + (n-2) + (n-3) + … + 2 + 1 = n(n-1)/2 = O(n^2)

- 최선 : pivot이 중앙값으로 선택되는 경우 ⇒ 1/2씩 분할
  - 비교 횟수 : O(nlog2n) = $O(n) * O(\log_2 n)$
    - 각 층에서의 원소가 1회씩 비교 → O(n)
    - 층 수 → log2n

최악 : $O(n^2)$

최선, 평균 → $O(n\log_2 n)$

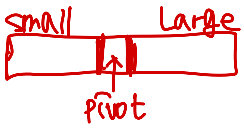
pivot에 대한 선택 확률이 같음.

- 평균 : pivot이 랜덤하게 선택되는 경우
  - O(nlog2n)
- pivot 선정 방법
  - random
  - median of Three : 가장 왼쪽, 중간, 가장 오른쪽 중 중앙값
  - median of medians : 입력을 3등분하여 각 부분에서의 중앙값 중 중앙값
- 입력의 크기가 매우 클 때 성능 향상을 위해 삽입 정렬이 동시에 사용되기도 함
- 입력의 크기가 작을 때는 퀵정렬이 삽입정렬보다 빠르지는 않음(cuz. recursion)

3.3 선택 문제

: n개의 숫자들 중에서 k번째로 작은 숫자를 찾는 문제

- **time complexity**
  - 최악 : O(nlogn)
    1. 최소 숫자를 k번 찾음 → 최소를 찾은 뒤에는 입력에서 그 숫자를 제거 → O(kn)
    2. 숫자들을 오름차순으로 정렬 후 k번째 숫자를 찾음 → O(nlogn)



각 그룹의 크기 (숫자 개수)를 알면 k번째 작은 숫자가 어디에 있는지!

- 선택 문제(숫자 찾기 문제)
  - binary search → 임의의 숫자를 효율적으로 찾기 위함, 입력 중 절반만 비교

```
Selection(A, left, right, k)

//입력 : A[left]~A[right]와 k, 단 1<=k<=|A|, |A|=right-left+1
//출력 : A[left]~A[right]에서 k번째 작은 원소
```

```
//1. 정렬
//Quick sort(line2)와 동일
pivot을 A[left]~A[right]에서 random하게 선택
pivot과 A[left]의 자리를 swap
pivot과 배열의 각 원소를 비교하여 pivot보다 작은 숫자는 A[left]~A[p-1]로 옮김
pivot보다 큰 숫자는 A[p+1]~A[right]로 옮기며, pivot은 A[p]에 놓는다.

//2. 해당되는 숫자 search
S=(p-1)-left+1 //S = small group의 크기 <- 가장 오른쪽 원소의 idx : p-1
small ①if(k<=S) //small group에서 찾기
   Selection(A, left, p-1, k)
pivot ②else if(k=S+1)  //pivot = k번째 작은 숫자
   return A[p]
large ③else    //large group에서 찾기
   Selection(A, p+1, right, k-S-1)
                             pivot
                           small group
```

- 분할 정복 알고리즘이기도 하지만 random 알고리즘
    - pivot 선택이 너무 한쪽으로 치우치면 알고리즘의 수행 시간이 길어짐.

- **time complexity**
    - 치우쳐지게 정하는 pivot의 경우의 수 : n/2 → 즉, good or bad 분할이 될 확률이 같음
    - pivot을 random하게 정했을 때 good 분할이 될 확률 → 1/2 ⇒ 평균 2회 연속하면 good
    - 평균 :
        1. 입력을 두 그룹으로 분할 → O(n)
        2. 분할 후 큰 부분의 최대 크기 : (3/4n-1) → 편의상 3/4n
        3. 연속적인 분할 후

           → O(n + 3/4n + (3/4)^2n + (3/4)^3n … + (3/4)^(i-1)n + (3/4)^in) = O(n)

           2배 이상보단 작음.
        4. 2 * O(n) = O(n)

           평균 2회 연속.

- 선택 알고리즘 : 이진 탐색과 매우 유사한 성격

- 이진탐색 : 분할 과정을 진행하면서 범위를 1/2씩 좁혀가며 찾고자 하는 숫자 탐색
- 선택 알고리즘 : pivot으로 분할하여 범위를 좁혀감

→ 정렬을 다 하지 않고 k번째 작은 수를 선형 시간 내에 찾음.

⇒ 부분 문제들을 취합하는 과정이 별도로 필요 없다는 공통점!

3.4 최근접 점의 쌍 찾기(Closest Pair)

: 2차원 평면상의 n개의 점이 입력으로 주어질 때, 거리가 가장 가까운 한 쌍의 점을 찾는 문제

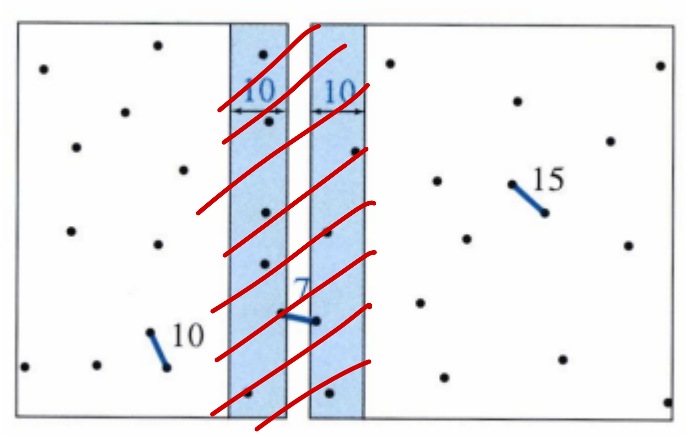
- idea

① n개의 점이 주어졌을 때

  - 그 사이의 거리 경우의 수 → nC2 = n(n-1)/2 → O(n^2)
  - 한 쌍의 거리 계산 → O(1)
  - time complexity : O(n^2) * O(1) = O(n^2)

② n개의 점이 주어졌을 때 → 분할정복 이용★ (부분 문제의 크기가 1/2로 감소)

  - n개의 점 → 1/2로 분할하여 각각의 부분문제에서 최근접 점의 쌍을 찾음
  - 취합할 때 반드시 부분문제 사이의 거리를 고려해야 함



[그림 3-9]

- 중간 영역에 있는 점들을 찾는 방법

- d = min {왼쪽 부분의 최근접 점의 쌍 사이의 거리, 오른쪽 부분의 ‘’ }
- arr = {점들이 x좌표 오름차순으로 정렬, y는 생략}

왼쪽 부분문제의 가장 오른쪽 점 → 4

오른쪽 부분문제의 가장 왼쪽 점 → 5

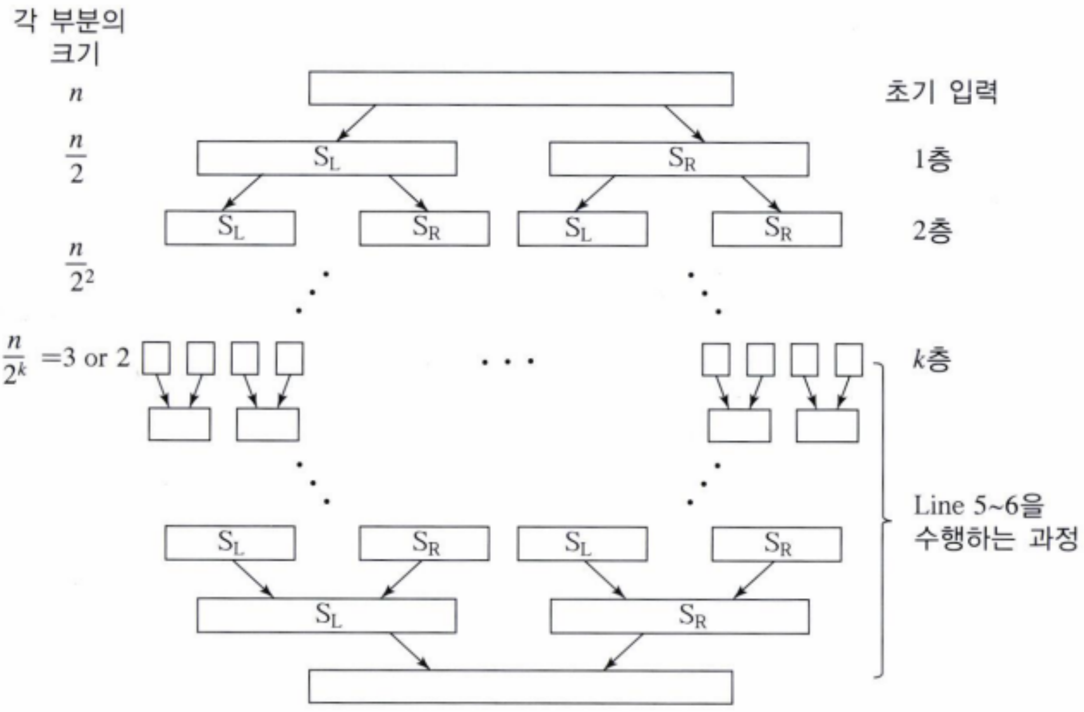
0	1	2	3	4	5	6	7	8	9
(1, -)	(13, -)	(17, -)	(25, -)	(26, -)	(28, -)	(30, -)	(37, -)	(45, -)	(56, -)

- 중간 영역에 속한 점 :
  - 왼쪽 부분문제의 가장 오른쪽 x좌표 -d
  - 오른쪽 부분문제의 가장 왼쪽 점 x좌표 +d

$d = 10$

0	1	2	3	4	5	6	7	8	9
(1, -)	(13, -)	(17, -)	(25, -)	(26, -)	(28, -)	(30, -)	(37, -)	(45, -)	(56, -)

$26-d = 16$ $28+d = 38$



```
ClosetPair(S)

//입력 : x좌표의 오름차순으로 정렬된 배열 S에는 i개의 점(단, (x,y)로 표현됨)
//출력 : S에 있는 점들 중 최근접 점의 쌍의 거리

if(i<=3) //3개 이하면 더 이상 분할하지 않는다. 2개 혹은 3개에 대한 최근접 점 쌍 return
  return (2 or 3개의 점들 사이의 최근접 쌍)

정렬된 S를 같은 크기의 SL과 SR로 분할한다.(단, |SL|가 홀수이면, |SL| = |SR| + 1)

CPL = ClosetPair(SL)
CPR = ClosetPair(SR)

d = min{dist(CPL), dist(CPR)} //dist() : 두 점 사이의 거리
CPC = 중간 영역에 속하는 점들 중 최근접 점의 쌍

return(CPL, CPC, CPR 중에서 거리가 가장 짧은 쌍)
```

→ 전처리 과정 필요

Left ①
Right ②
Mid ③

- **time complexity**
  - 입력 S에 n개의 점이 있다고 가정
  - 전처리(preprocessing) 과정 : S의 점을 x-좌표로 정렬 → O(nlogn) 합병 OR Heap 정렬 사용
  - 3개의 점이 있는 경우 → O(1)
  - 정렬된 S 분할 → O(1)
  - ClosetPair 호출 → merge sort와 동일 → 분할 후 호출
  - 중간 영역에 속하는 점들 중 최근접 점 쌍 찾기
    1. y 좌표 기준으로 정렬하기 → O(nlogn) 전체 n개
    2. 아래 위로 올라가며 각 점에서 주변 점들 사이의 거리 계산 → O(1) 하나당

       (주변 점의 개수는 O(1)개임 → d보다 큰 거리는 포함되지 않기 때문 → n을 넘지 않음)

       ⇒ O(n) n개

    ⇒ O(nlogn) + O(n)
  - CPL, CPC, CPR 중에서 거리가 가장 짧은 쌍 → O(1)

⇒ O(nlogn)

- 평균 ☆
  1. 합병 정렬 중 합병 과정 → O(n)

2. 해를 취합하여 올라가는 과정 → O(nlogn)

    a. 층 수 < log2n(cuz. 점의 개수가 3 or 2일 때 분할을 중단하기 때문)

    b. 층 수 당 수행 시간 → O(nlogn)

⇒ O(nlogn) * logn = O(nlog(^2)n)

순환호출되어 중간 영역에 속하는 점 중 최근접 점 쌍의 거리를 찾을 때
y 좌표로 중간 영역 점들 정렬 필요

- 최선

1. 입력의 점들을 y좌표 기준으로 전처리 과정에서 미리 정렬하여 다른 배열에 저장

2. 중간 영역에 속한 점들에 대해서 정리할 때 해당 배열을 활용하면 매번 정렬 x

3.5 분할 정복을 적용하는 데 있어서 주의할 점

- 분할
    - 입력이 분할될 때마다 분할된 부분문제의 입력 크기의 합이 분할되기 전의 입력 크기보다 매우 커지는 경우에는 적합하지 못 함.
- 정복
    - 기하에 관련된 다수의 문제들이 효율적인 분할 정복 알고리즘으로 해결됨.