[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.09.29 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

숙제 1

CSE3081 (2반): 알고리즘 설계와 분석
[숙제 1]

담당 교수: 임 인 성

2020년 9월 22일

마감: 10월 6일 화요일 오후 8시 정각
제출물, 제출 방법, LATE 처리 방법 등: 조교가 과목 게시판에 공지함.

목표: 이번 숙제는 주어진 문제에 대하여 서로 다른 시간 복잡도를 가지는 두 알고리즘을 구현한 후, 다양한 크기의 입력 데이터에 대한 수행 시간을 측정하여, 이론적인 시간 복잡도와 실제 수행 시간 간의 연관 관계를 분석해봄을 목표로 한다.

문제

1. 다음과 같은 Maximum Sum Subarray Problem (1D)을 고려하자.

   Given a 1-dimensional integer array of size n, find the maximum-sum subarray **with at least one element**.

   이 문제를 해결해주는 다음과 같은 시간 복잡도를 가지는 두 가지 알고리즘을 구현하라. 각 방법은 최대 합뿐만 아니라 그에 해당하는 subarray의 처음과 마지막 원소의 인덱스를 찾아주어야 한다.

   - **Algorithm 1**: 시간 복잡도 $O(n \log n)$ $\longleftarrow$ Divide-and-conquer 기법을 적용한 방법
   - **Algorithm 2**: 시간 복잡도 $O(n)$ $\longleftarrow$ Dynamic programming 기법을 적용한 방법 (Kadane's algorithm)

2. 다음과 같은 Maximum Sum Subrectangle Problem (2D)을 고려하자.

   Given a 2-dimensional integer array of size $n \times n$, find the maximum-sum subrectangle **with at least one element**.

   이 문제를 해결해주는 다음과 같은 시간 복잡도를 가지는 세 가지 알고리즘을 구현하라. 각 방법은 최대 합뿐만 아니라 그에 해당하는 subrectangle의 위-왼쪽 모서리와 아래-오른쪽 모서리 원소들의 인덱스를 찾아주어야 한다.

   - **Algorithm 3**: 시간 복잡도 $O(n^4) \longleftarrow$ Summed Area-Table 기법 적용 방법 (강의 설명)
   - **Algorithm 4**: 시간 복잡도 $O(n^3 \log n) \longleftarrow$ 1D 문제를 풀기 위하여 Algorithm 1을 적용한 방법
   - **Algorithm 5**: 시간 복잡도 $O(n^3) \longleftarrow$ 1D 문제를 풀기 위하여 Algorithm 2를 적용한 방법 (강의 설명)

3. 이번 숙제의 목적은

   (a) 2D 문제를 서로 다른 시간 복잡도를 가지는 Algorithm 3, Algorithm 4, 그리고 Algorithm 5로 구현하여,

   (b) **충분히 큰 여러 크기의 입력 크기 n에 대하여** 수행 시간을 측정한 후,

   (c) 과연 그러한 수행시간이 이론적인 시간 복잡도와 일치하는지를 확인하는 것이다.

[주제 3]

Divide-and-Conquer Techniques

and

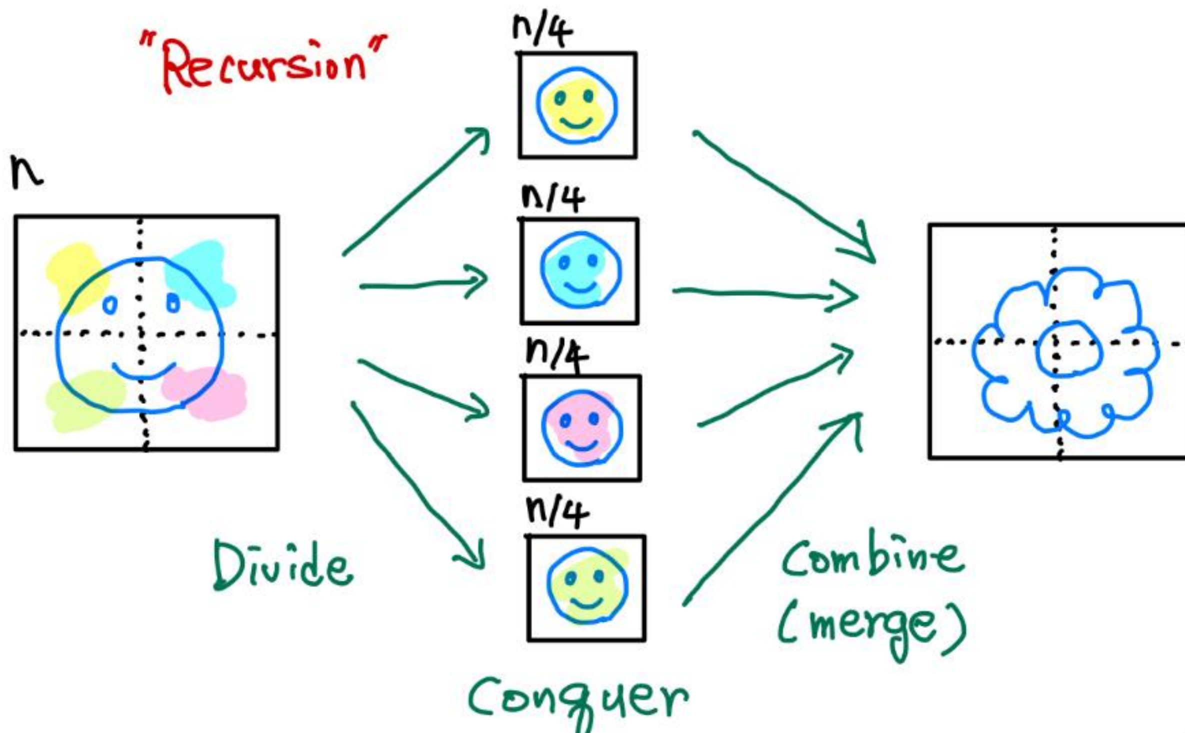
Sorting Techniques

The Divide-and-Conquer Approach

① **Divide** an instance of a problem into one or more smaller instances.

② **Conquer** (Solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.

③ If necessary, **combine** the solutions to the smaller instances to obtain the solution to the original instance.



Merge Sort

Problem: Sort *n* keys in nondecreasing sequence.

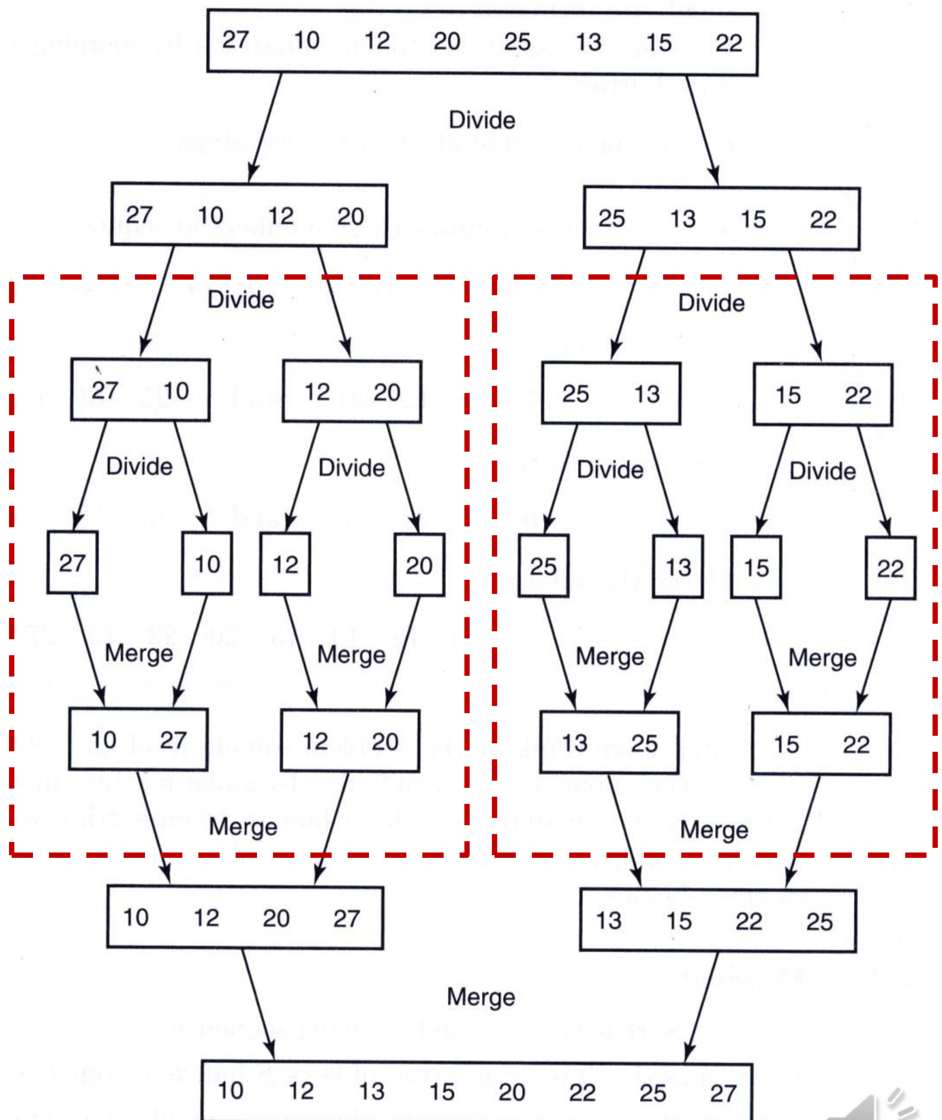
Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

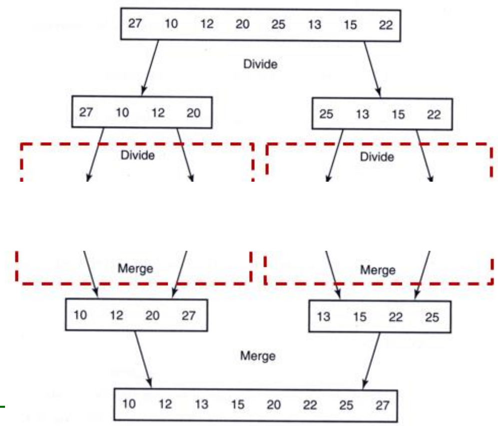
① **Divide** the array into two subarrays each with ~n/2 items.

② **Conquer** each subarray by sorting it recursively.

③ **Combine** the solutions to the subarrays by merging them into a single sorted array.



- A **simple** implementation




```
// Sort a list from A[left] to A[right].
// Should be optimized for higher efficiency!!!

void merge_sort(item_type *A, int left, int right) {
  int middle;

  if (left < right) {
    middle = (left + right)/2;

    merge_sort(A, left, middle);
    merge_sort(A, middle + 1, right);

    merge(A, left, middle, right);
  }
}
```

$T(n)$

Divide $O(1)$

Conquer $2T(n/2)$

Combine $O(n)$

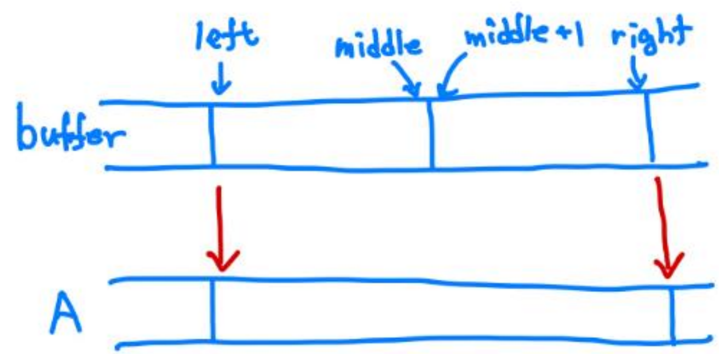
– An example of merging two arrays

k	left	right	merged
1	10 12 20 27	13 15 22 25	10
2	10 12 20 27	13 15 22 25	10 12
3	10 12 20 27	13 15 22 25	10 12 13
4	10 12 20 27	13 15 22 25	10 12 13 15
5	10 12 20 27	13 15 22 25	10 12 13 15 20
6	10 12 20 27	13 15 22 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 25	10 12 13 15 20 22 25
-			10 12 13 15 20 22 25 27

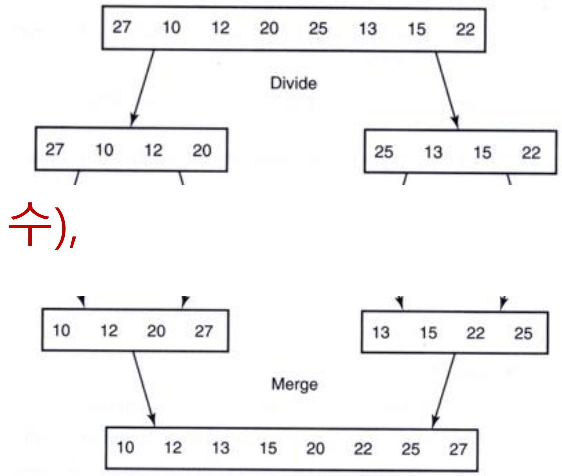
```c
item_type *buffer; // extra space for merge sort, allocated beforehand

void merge(item_type *A, int left, int middle, int right) {
  int i, i_left, i_right;

  memcpy(buffer + left, A + left, sizeof(item_type)*(right - left + 1));

  i_left = left;
  i_right = middle + 1;
  i = left;

  while ((i_left <= middle) && (i_right <= right)) {
    if (buffer[i_left] < buffer[i_right])
      A[i++] = buffer[i_left++];
    else
      A[i++] = buffer[i_right++];
  }

  while (i_left <= middle)
    A[i++] = buffer[i_left++];
  while (i_right <= right)
    A[i++] = buffer[i_right++];
}
```

$O(r - l + 1)$

$\boxed{O(n)}$



left = 0, right = n-1



- **Worst-case time complexity**
  - 편의상 $n = 2^m$이라 할 경우 (m은 0보다 같거나 큰 정수),

$$\boxed{\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ T(1) &= 1 \end{aligned}}\quad , n \geq 2$$

Number of subproblems

Subproblem size

$$\boxed{T(n) = O(n \log n)}$$

Merge Sort		
Divide	**Conquer**	**Combine**
O(1)	2T(n/2)	O(n)

  - n개의 원소를 k개와 l개로 나누어 진행한다고 가정하면 ($n = k + l$),

$$T(n) = T(k) + T(l) + cn \ (k \approx l)$$

    - $n = 2^m$이 아닌 일반적인 경우에도 같은 시간 복잡도를 가짐을 증명할 수 있음.

Solving Recurrence Equations

$a_n = \alpha a_{n-1} + \beta n,\ a_0 = 1$

- Solve the following recurrences T(n) for given $T(1) = 1$:

① $T(n) = aT(n\text{-}1) + bn$

② $T(n) = T(n/2) + bn\log n$

③ $T(n) = aT(n\text{-}1) + bn^2$

④ $T(n) = aT(n/2) + bn^2$

⑤ $T(n) = T(n/2) + c\log n$

⑥ $T(n) = T(n/2) + cn$

⑦ $T(n) = 2T(n/2) + cn$

⑧ $T(n) = 2T(n/2) + cn\log n$

⑨ $T(n) = T(n\text{-}1) + T(n\text{-}2)$, for $T(1) = T(2) = 1$

Some Derivations

1. $T(n) = 2 \cdot T(n/2) + c \cdot n, \;\; T(1) = 1$

   (Assum $n = 2^m$, i.e., $m = \log_2 n$ for some integer $m \geq 0$)

$$
\begin{aligned}
T(2^m) &= 2 \cdot T(2^{m-1}) + c \cdot 2^m \\
&= 2\{2 \cdot T(2^{m-2}) + c \cdot 2^{m-1}\} + c \cdot 2^m \\
&= 2^2 \cdot T(2^{m-2}) + 2 \cdot c \cdot 2^m \\
&= 2^2\{2 \cdot T(2^{m-3}) + c \cdot 2^{m-2}\} + 2 \cdot c \cdot 2^m \\
&= 2^3 \cdot T(2^{m-3}) + 3 \cdot c \cdot 2^m \\
&\;\;\vdots \\
&= 2^m \cdot T(2^0) + m \cdot c \cdot 2^m \\
&= n \cdot 1 + (\log_2 n) \cdot c \cdot n \\
&= O(n \log n)
\end{aligned}
$$

2. $T(n) = T(n-1) + c \cdot n, \;\; T(1) = 1$

$$3.\ T(n) = 2 \cdot T(n/2) + c \cdot n^2,\ \ T(1) = 1$$
$$(\text{Assum } n = 2^m \text{ for some nonnegative integer } m)$$

$$\begin{aligned}
T(2^m) &= 2 \cdot T(2^{m-1}) + c \cdot 2^{2m} = 2\{2 \cdot T(2^{m-2}) + c \cdot 2^{2(m-1)}\} + c \cdot 2^{2m} \\
&= 2^2 \cdot T(2^{m-2}) + c\{2^{2m-1} + 2^{2m}\} \\
&= 2^2\{2 \cdot T(2^{m-3}) + c \cdot 2^{2(m-2)}\} + c\{2^{2m-1} + 2^{2m}\} \\
&= 2^3 \cdot T(2^{m-3}) + c\{2^{2m-2} + 2^{2m-1} + 2^{2m}\} \\
&\ \ \vdots \\
&= 2^m \cdot T(2^{m-m}) + c\{2^{2m-(m-1)} + \cdots + 2^{2m-2} + 2^{2m-1} + 2^{2m}\} \\
&\ \ \vdots \\
&= 2^m + 2 \cdot c \cdot 2^{2m} - 2 \cdot c \cdot 2^m \\
&= 2 \cdot c \cdot n^2 - (2 \cdot c - 1)n \\
&= O(n^2)
\end{aligned}$$

Another Implementation of Merge Sort

[Horowitz 7.6.3]

```
int rmerge(element list[], int lower, int upper)
/* sort the list, list[lower],..., list[upper]. The link
field in each record is initially set to -1. */
{
  int middle;
  if (lower >= upper)
     return lower;
  else {
     middle = (lower + upper) / 2;
     return listmerge(list,rmerge(list,lower,middle),
                           rmerge(list,middle+1,upper));
  }
}
```

Program 7.11: Recursive merge sort

`rmerge` returns an integer that points to the start of the sorted list.
`start = rmerge(list, 0, n-1);`

```
typedef struct {
   int key;
   int link;
} element;
```

`start = 3`

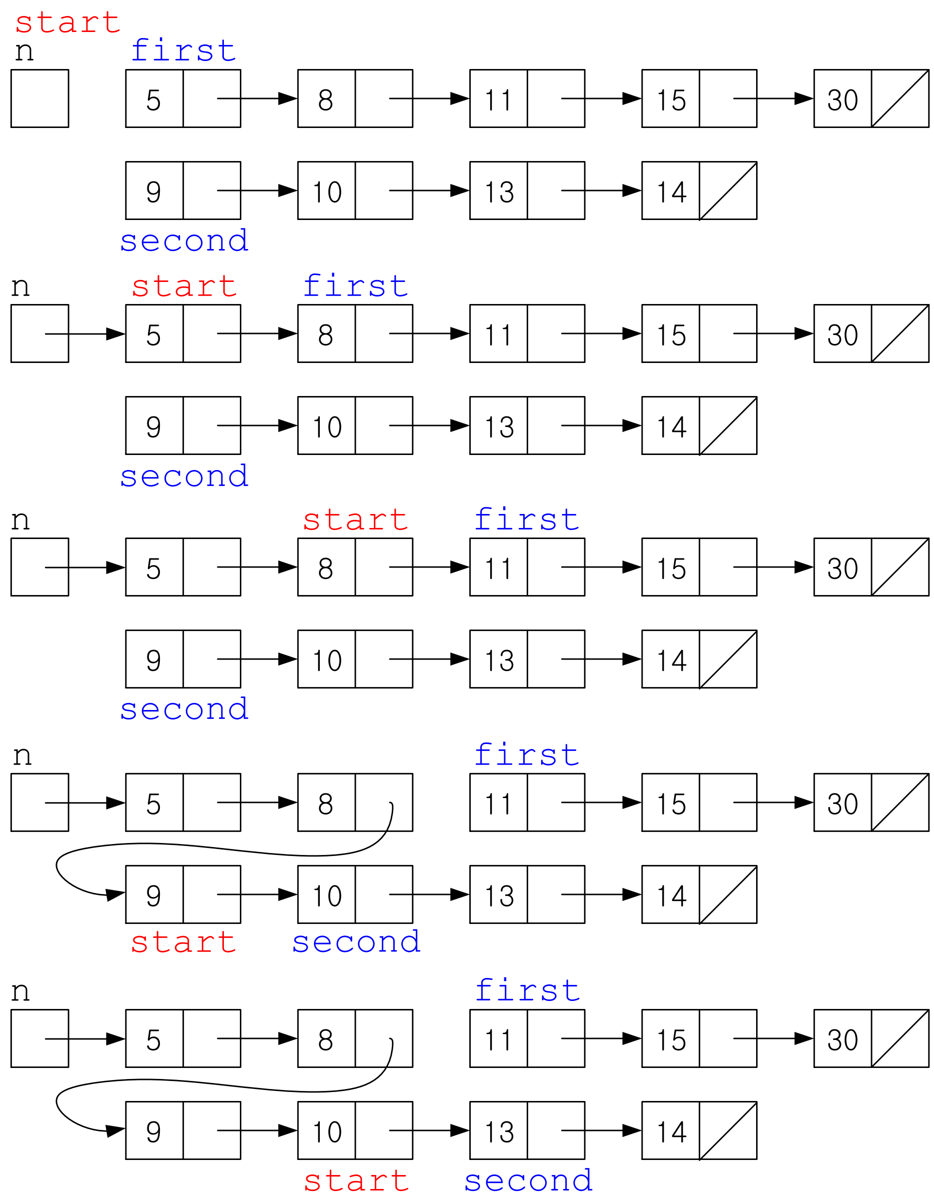
i	0	1	2	3	4	5	6	7	8	9
key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0

```
int listmerge(element list[], int first, int second)
/* merge lists pointed to by first and second */
{
  int start = n;
  while (first != -1 && second != -1)
    if (list[first].key <= list[second].key) {
    /* key in first list is lower, link this element to
    start and change start to point to first */
      list[start].link = first;
      start = first;
      first = list[first].link;
    }
    else {
    /* key second list is lower, link this element into
    the partially sorted list */
      list[start].link = second;
      start = second;
      second = list[second].link;
    }
  /* move remainder */
  if (first == -1)
    list[start].link = second;
  else
    list[start].link = first;
  return list[n].link; /* start of the new list */
}
```

`listmerge` takes two sorted chains, `first` and `second`, and returns an integer that points to the start of a new sorted chain that includes the first and second chains.

Program 7.12: Merging linked lists

listmerge 함수 수행 예



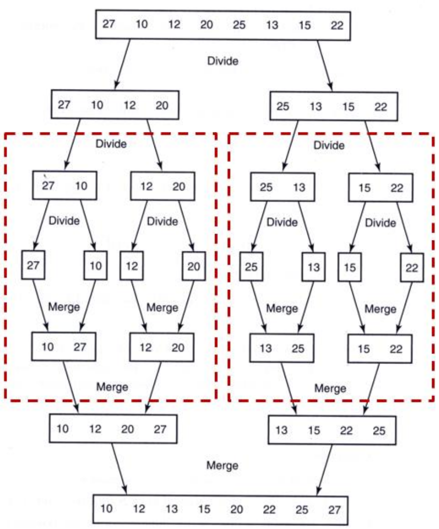
Quick Sort

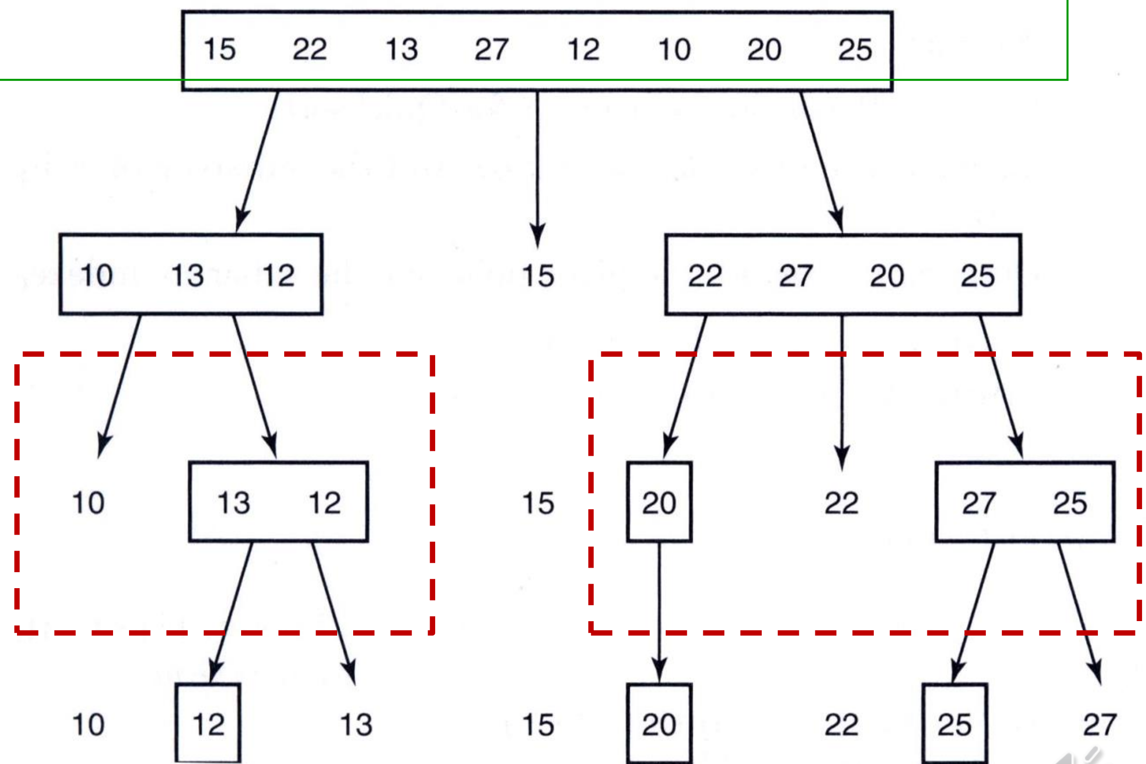
Pivot strategy

① **Divide:** Select a **pivot element**, and then divide the array into two subarrays such that ….

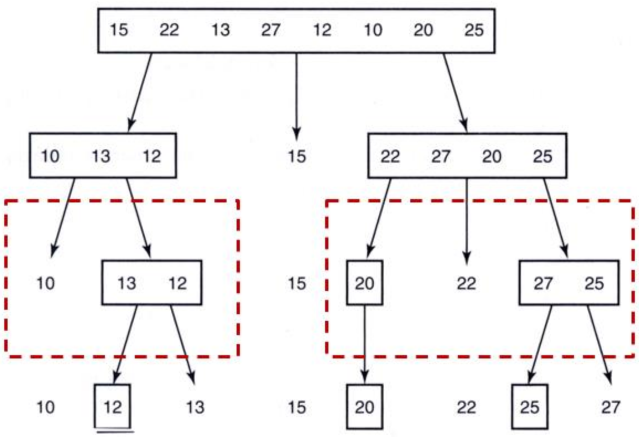
② **Conquer:** sort each subarray recursively.

③ **Combine:** do nothing.





- A **simple** implementation




```
// Sort a list from A[left] to A[right].
// Should be optimized for higher efficiency!!!

void quick_sort(item_type *A, int left, int right) {
  int pivot;

  if (right - left > 0) {
    pivot = partition(A, left, right);

    quick_sort(A, left, pivot - 1);
    quick_sort(A, pivot + 1, right);
  }
}
```

Divide

Conquer

```c
#define SWAP(a, b) { item_type tmp; tmp = a; a = b; b = tmp; }

int partition(item_type *A, int left, int right) {
  int i, pivot;

  pivot = left;
  for (i = left; i < right; i++) {
    if (A[i] < A[right]) {
      SWAP(A[i], A[pivot]);
      pivot++;
    }
  }
  SWAP(A[right], A[pivot]);
  return(pivot);
}
```

How is the pivot element chosen in this function?

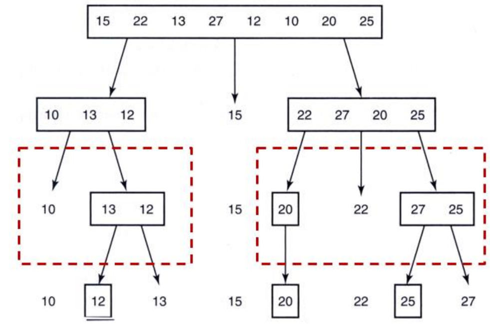
18 20 28 0 38 8 2 16 10 14 24 30 34 12 32 22 6 4 36 26

18 20 0 8 2 16 10 14 24 12 22 6 4 **26** 32 38 30 34 36 28

(13)

Cost Analysis

Quick Sort		
Divide	**Conquer**	**Combine**
O(n)	T(m1) +T(m2)	O(1)



- **Cost**

$$T(n) = T(m_1) + T(m_2) + cn \ (m_1 + m_2 = n-1) \text{ if } n > 1$$
$$T(1) = 1$$

- **Worst-case time complexity**
  - 매 단계에서 선택한 pivot element가 가장 크거나 가장 작을 경우,

$$T(n) = T(0) + T(n-1) + cn$$

Skewed vs well-balanced trees

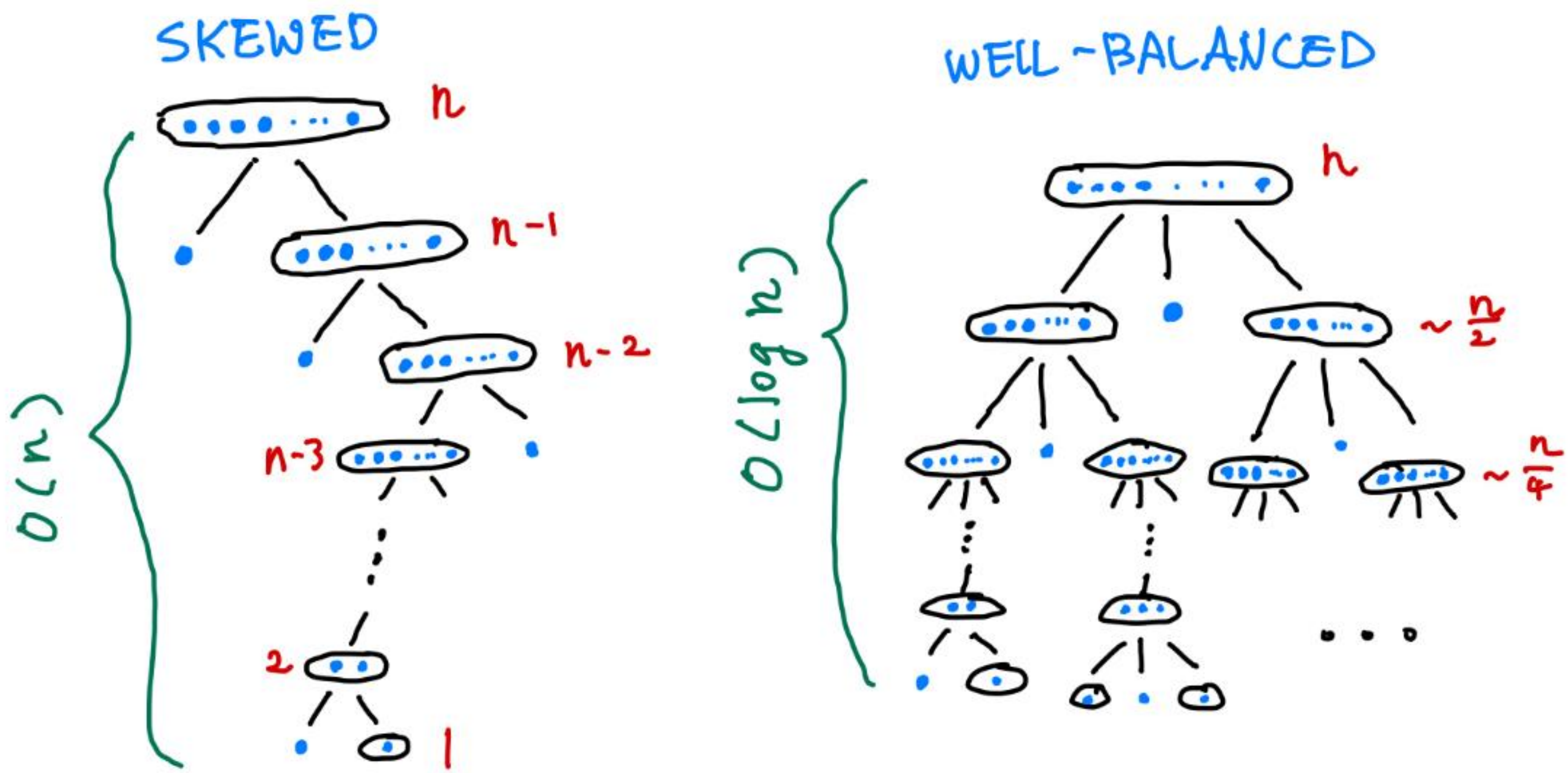
$$\boxed{\begin{array}{l} T(n) = T(n-1) + cn, \text{ if } n > 1 \\ T(1) = 1 \end{array}} \longrightarrow \boxed{T(n) = O(n^2)}$$

- **Average-case time complexity**

$$\boxed{\begin{array}{l} T(n) = \Sigma_{p=1}^{n} \frac{1}{n}\{T(p-1) + T(n-p)\} + cn \\ \\ T(0) = 1 \end{array}} \longrightarrow \boxed{T(n) = O(n \log n)}$$

직관적인 시간 복잡도 추정

$$T(n) = T(m_1) + T(m_2) + cn \ (m_1 + m_2 = n - 1) \text{ if } n > 1$$
$$T(1) = 1$$



$n + (n-1) + (n-2) + \cdots + 2$
$= O(n^2)$

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