기본 알고리즘
제3장

2016- Fall

국민대학교 컴퓨터공학부 최준수

분할정복기법

- Divide & Conquer
  - Recursion 기반 해결기법
  - 분할정복기법의 문제해결 시나리오

(1) 분할 (Divide)

- 주어진 문제를 두 개 혹은 그 이상의 같은 형식의 작은 문제로 나눈다.

(2) 정복 (Conquer)

- 나누어진 작은 문제는 재귀적으로 해결한다. 즉, 나누어진 작은 문제는 더 이상 나누어서 문제를 해결할 필요가 없이 직접 문제를 해결할 수 있을 때 까지 재귀적으로 계속 분할해가면서 문제를 해결한다.

(3) 통합 (Combine)

- 한 개 이상의 작은 문제들로부터 구한 모든 해답들을 서로 통합해서 원래 문제의 해답을 만든다.

분할정복기법 (2)

- Divide & Conquer
  - 분할정복기법은 Top-Down 문제해결 방법
    - 초기에 큰 문제가 주어졌을 때, 직접 이 문제를 해결할 수 없으므로, 이 문제를 적절한 크기의 작은 문제로 분할하여 해결하는 방법이다. 이러한 방법을 top-down 방법이라고 한다.
  - Recursion 으로 알고리즘 구현
    - Top-down 방법에 의하여 분할된 작은 문제들 또한 직접 문제를 해결할 수 없는 경우에는 또 다시 더 작은 문제로 분할한다.
    - 작은 문제는 문제를 직접 해결할 수 있을 때 까지 계속 분할한다.
    - 이러한 해결 방법은 recursion 으로 쉽게 구현할 수 있다.

분할정복기법 (3)

- Divide & Conquer
  - 분할정복기법 알고리즘의 정확성 증명
    - 수학적 귀납법 사용
  - 분할정복기법 알고리즘의 시간복잡도 계산
    - 시간복잡도 T(n) 을 알고리즘으로 부터 재귀식으로 유도하고, 이 재귀식(점화식)을 풀어서 구함

수학적 귀납법

– 수학적 귀납법(Mathematical Induction)의 원리

Theorem: **Principles of Mathematical Induction**

- Let *P(n)* be a statement that is defined for integers *n*, and let *a* be a fixed integer.
- Suppose the following two statements are true:
  - *P(a)* is true
  - For all integers *k>=a* , if *P(k)* is true then *P(k+1)* is true.
- Then the statement
  - For all integers *n >= a, P(n)*

  is true.

수학적 귀납법 (2)

– 수학적 귀납법에 의한 증명방법

1. Base Step

- Prove *P(a)*

2. Inductive Hypothesis

- Suppose *P(k)* is true for *k>=a*

3. Inductive Step

- Prove P(k+1) is true using the inductive hypothesis.

수학적 귀납법 (3)

- 예
  - P(n) : For all integers $n \geq 1$, $\sum_{i=1}^{n}(-1)^{n-i}i^2 = n(n+1)/2$
  - 증명:

    (1) Base step: P(1)

    Left Part : 1
    Right Part : 1
    Therefore *P(1)* is true

    (2) Inductive Hypothesis

    Suppose that *P(k)* is true for *k>=1*

수학적 귀납법 (4)

- 예
  - P(n) : For all integers *n ≥ 1*, $\sum_{i=1}^{n}(-1)^{n-i}i^2 = n(n+1)/2$

(3) Inductive Step: Prove that *P(k+1)* is true

From the left part of *P(k+1)*, we have

$$\begin{aligned}\sum_{i=1}^{k+1}(-1)^{k+1-i}i^2 &= (k+1)^2 + \sum_{i=1}^{k}(-1)^{k+1-i}i^2 \\ &= (k+1)^2 - \sum_{i=1}^{k}(-1)^{k-i}i^2 \\ &= (k+1)^2 - P(k) \\ &= (k+1)^2 - k(k+1)/2 \\ &= (k+1)(k+2)/2\end{aligned}$$

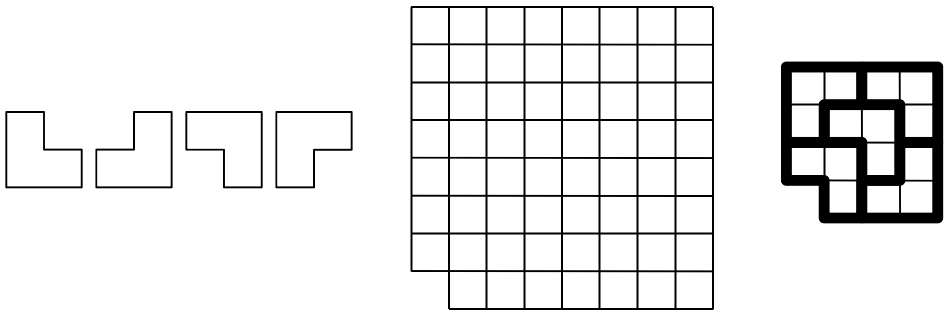
Thus part is equal to the right part of *P(k+1)*.

(결론) Therefore we conclude that the theorem is true by the *theorem of mathematical induction*.

Tromino 타일 채우기

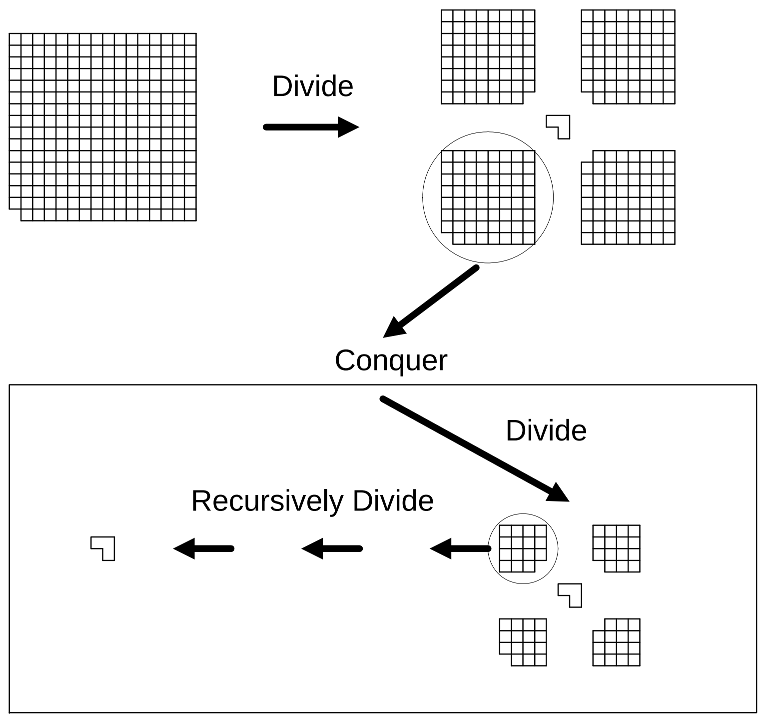
– 문제

- 트로미노(tromino) 는 다음 그림과 같이 크기가 1x1 인 정사각형 세 개로 기역자 모양으로 만들어진 모양으로서, 모양을 회전시키면 아래와 같은 네 가지 모양을 가지고 있다.
- 크기가 NxN ($N=2^k$) 이고, 한 쪽 모퉁이에 1x1 크기의 격자가 떨어져 나간 바둑판 모양의 격자판이 주어졌을 때, 이 격자판을 트로미노 타일로 빈 공간없이, 또한 타일이 서로 겹치지 않게, 모든 격자판을 채우는 방법을 고안하시오. 예를 들어 N=4인 경우는 아래 그림과 같이 채울 수 있다.



Tromino 타일 채우기 (2)

– 해결 알고리즘: (Divide & Conquer)



Tromino 타일 채우기 (3)

– 알고리즘의 정확성 증명

- 위 알고리즘에 의해서는 귀퉁이가 떨어져나간 $2^n x 2^n$ ($n \geq 1$) 크기의 격자판은 항상 트로미노 타일로 채울수 있다.

- 수학적 귀납법에 의한 증명

  (1) Base case : n=1 인 경우

  귀퉁이가 떨어져나간 2x2 크기의 격자판은 1개의 트로미노 타일로 채울 수 있다.

  (2) Inductive Hypothesis: n=k 인 경우

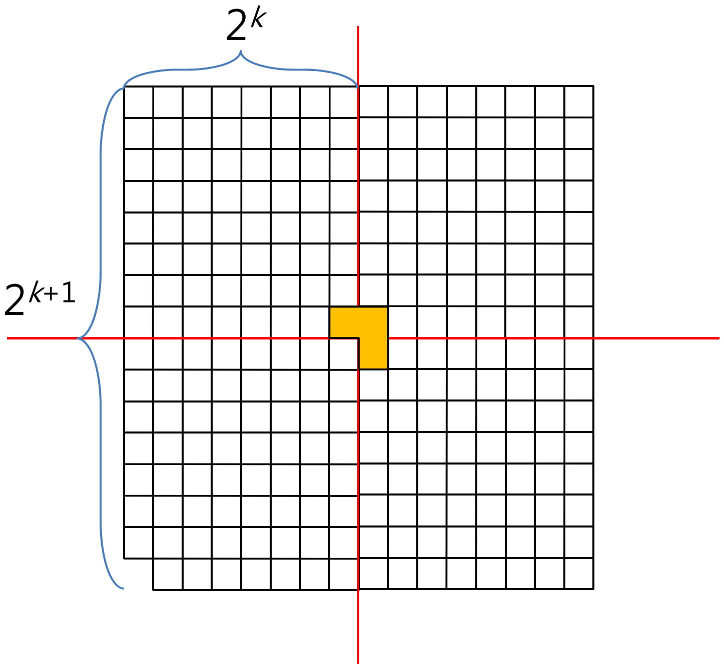
  귀퉁이가 떨어져나간 $2^k x 2^k$ 크기의 격자판은 항상 트로미노 타일로 채울 수 있다라고 가정하자.

Tromino 타일 채우기 (4)

– 알고리즘의 정확성 증명

(3) Inductive Step : n=k+1 인 경우

(2) 번 단계의 가정을 이용하여, 귀퉁이가 떨어져나간 2^{k+1}x2^{k+1} 크기의 격자판은 다음과 같이 트로미노 타일로 채울 수 있다라고 증명할 수 있다.

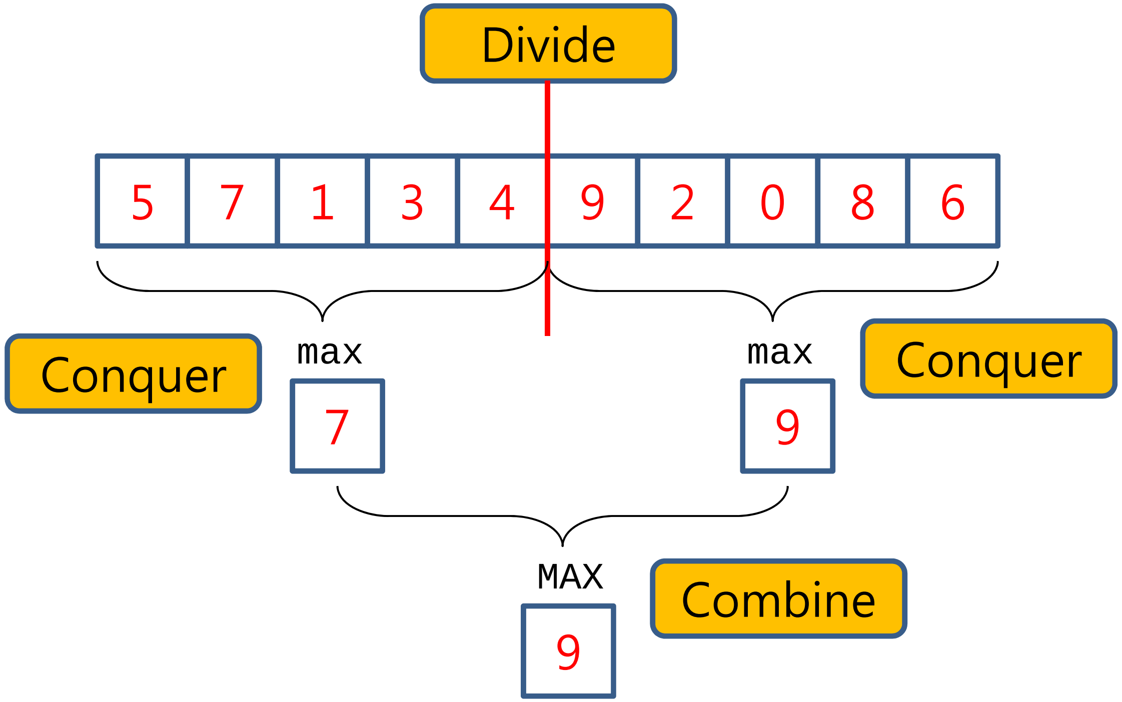


(결론) 따라서 모든 2^nx2^n ($n \geq 1$) 크기의 격자판은 위 알고리즘에 따라 트로미노로 채울 수 있다.

Finding Max

- Finding Max(recursive)

$$\max([a_1,a_2,a_3,\cdots,a_{n-1},a_n]) = \begin{cases} a_1 & n=1 \\ MAX(\max([a_1,\cdots,a_k]),\max([a_{k+1},\cdots,a_n])) & n>1 \end{cases}$$



Finding Max (2)

- Finding Max (Divide & Conquer)

$$\max([a_1, a_2, a_3, \cdots, a_{n-1}, a_n]) = \begin{cases} a_1 & n = 1 \\ MAX(\max([a_1, \cdots, a_k]), \max([a_{k+1}, \cdots, a_n])) & n > 1 \end{cases}$$

```
#define MAX_SIZE 100

#define MAX(a,b) ((a)>(b)?(a):(b))

int recurMax(int a[], int left, int right)
{
    int half;

    if (left == right)
        return a[left];
    else
    {
        half = (left+right)/2;
        return MAX(recurMax(a, left, half),
                   recurMax(a, half+1, right));
    }
}
```

```
void main(void)
{
    int v[MAX_SIZE] = { 5, 3, 9, 2, 4, 8, 1, 6, 0, 7 };
    printf("%d\n", recurMax(v, 0, 9));
}
```

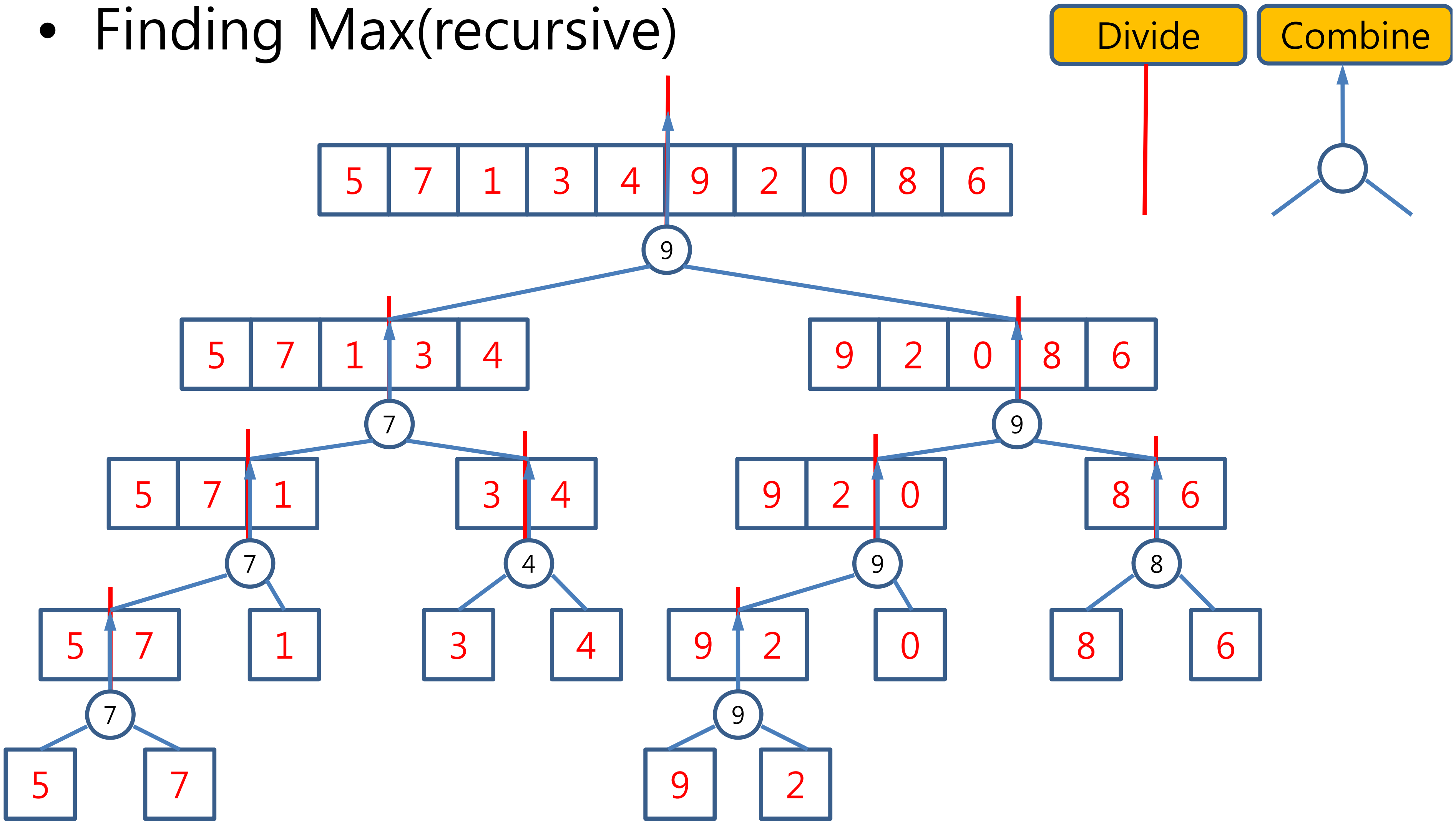
Divide

Conquer

Combine

Finding Max

- Finding Max(recursive)



Finding Max (3)

- Finding Max (Divide & Conquer)
  - Analysis
    - Basic operation : MAX( )에서 두 수를 비교하는 연산자
    - T(n) : n 개의 정수 중에서 최대값을 구할 때, basic operation 의 수행 횟수.

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 & n > 1 \end{cases}$$

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

Binary Search

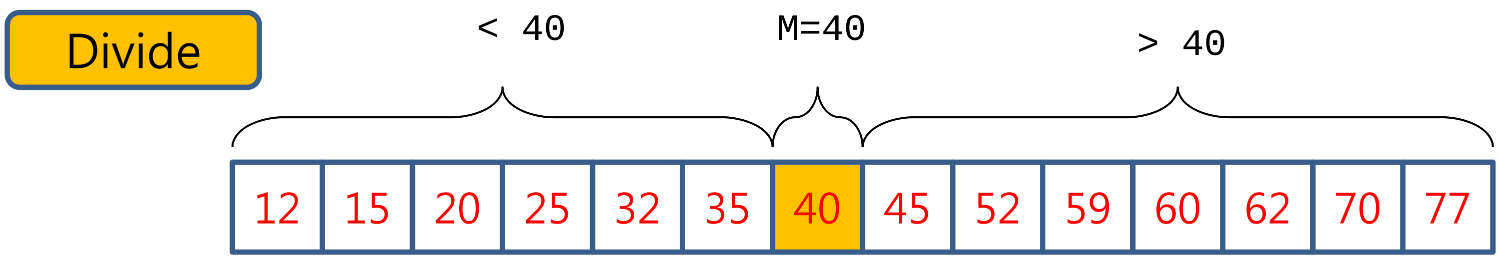
- Binary Search
  - 문제
    - 오름차순으로 정렬된 n 개의 정수가 저장된 1차원 배열에 주어진 정수 x 가 들어 있는지를 검사하시오.

x ?

12	15	20	25	32	35	40	45	52	59	60	62	70	77

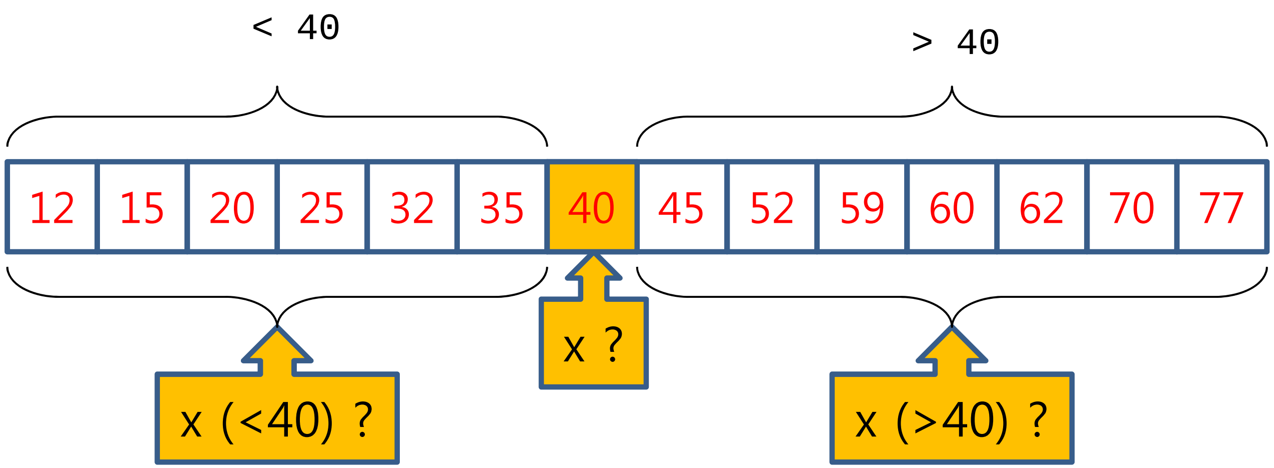
Binary Search (2)

- Binary Search
  - Divide & Conquer 알고리즘
    - Divide
      - 가장 중앙에 있는 정수 M과 M을 중심으로 M 보다 작은 그룹과 M 보다 큰 그룹 등의 세 그룹으로 나눈다



Binary Search (3)

- Conquer
  - x = M
    - » 가장 중앙에 있는 정수 M과 찾고자 하는 정수 x를 비교한다.
    - » 두 수가 같은 경우에는 종료한다.
  - x < M
    - » M 보다 작은 그룹에서 x 를 재귀적으로 검색한다.
  - x > M
    - » M 보다 큰 그룹에서 x 를 재귀적으로 검색한다.



Binary Search (4)

- Combine
  - Conquer 단계에서 분할된 작은 문제들에 대하여 구한 해답을 원래 문제의 해답으로 정한다.

```
int binarySearch(int a[], int left, int right, int value)
{
    int mid;

    if (left > right)
        return -1; /* not found */
    else
    {
        mid = (left+right)/2;

        if (a[mid] == value)
            return mid;
        else if (a[mid] > value)
            return binarySearch(a, left, mid-1, value);
        else
            return binarySearch(a, mid+1, right, value);
    }
}
```

```
void main(void)
{
    int v[] = { 1, 3, 4, 7, 9, 11, 15 };
    printf("%d\n", binarySearch(v, 0, 6, 11));
}
```

Binary Search (5)

– Analysis
  - basic operation : x 와 배열에 있는 수를 비교하는 연산
  - T(n) : n 개의 정수에서 x 를 검색할 때의 basic operation 수

$$T(n) = \begin{cases} 1 & n = 1 \\ \max\left\{ T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right), T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \right\} + 1 & n > 1 \end{cases}$$

– 위 식은 아래와 같이 단순화하여 생각할 수 있다.

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

Binary Search (6)

– Analysis

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

- 위 식은, n = 2^k 일 때, 아래와 같이 order 를 구할 수 있다.

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 1 \\ &= T(2^{k-2}) + 1 + 1 \\ &= T(2^{k-3}) + 1 + 1 + 1 \\ &\cdots \\ &= T(2^{k-k}) + 1 + \cdots + 1 \\ &= k + 1 \\ &= \log n + 1 \\ &\in O(\log n) \end{aligned}$$

Find Peak Value

- Find Peak Value
  - 오르내리막 수열
    - 최고점 (Peak Value)

12	15	20	25	32	35	40	45	31	26	12	9	7	3

    - n 개의 정수로 구성된 오르내리막 수열이 있을 때, 이 수열의 최고점을 빠르게 계산하는 알고리즘을 제시하시오.

Merge Sorting

- Merge Sorting
  - 1차원 배열에 저장된 n 개의 데이터를 오름차순으로 정렬

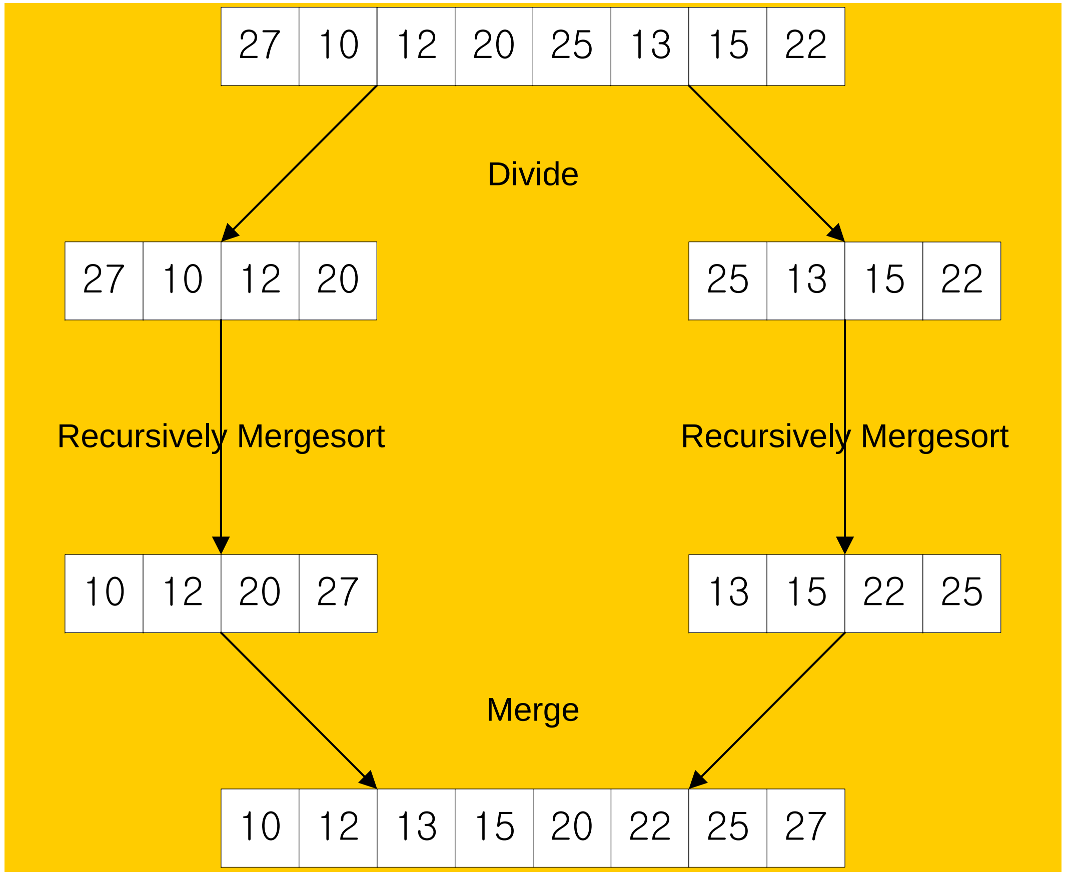
  – Divide
    - 배열을 각각 n/2 개의 데이터로 만들어진 두 개의 부분배열로 분할한다.

  – Conquer
    - 나누어진 부분배열에 대하여 재귀적으로 합병 정렬을 수행한다.
    - 단, 데이터의 개수가 1개인 경우에는 그 자체로 정렬된 상태이다.

  – Combine (Merge)
    - 두 개의 이미 정렬된 부분배열을 통합하여 n 개의 정렬된 배열로 만든다.

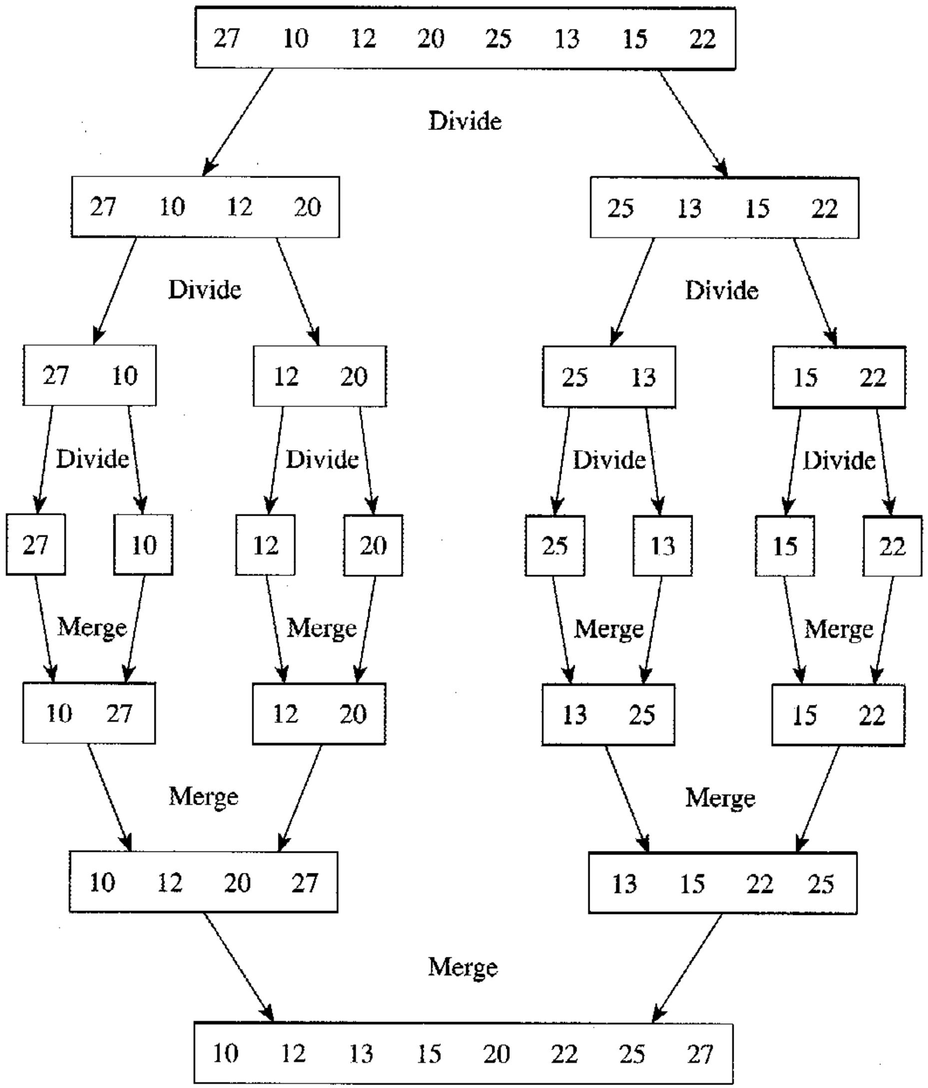
Merge Sorting (2)

- 예



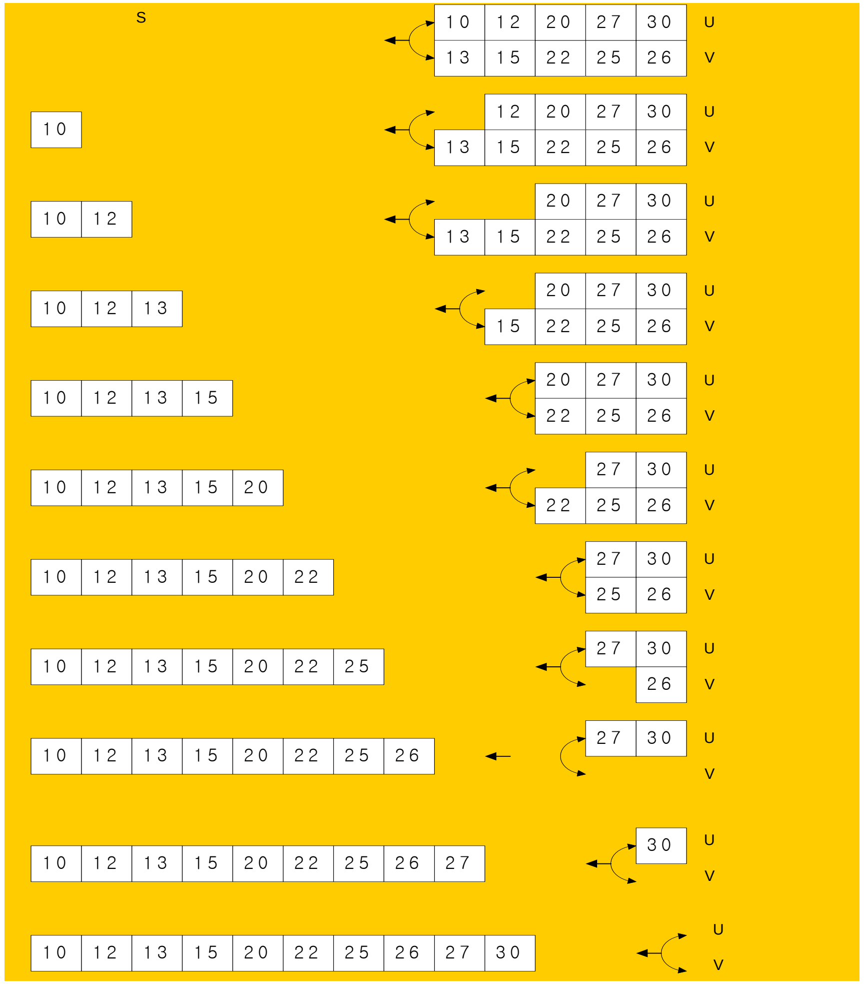
Merge Sorting (3)

- 예



Merge Sorting (4)

– Merge
  - 이미 정렬된 두 개의 배열이 주어졌을 때, 이 두 배열을 합쳐서 정렬함.



Merge Sorting (5)

– Merge
  - 이미 정렬된 두 개의 배열이 주어졌을 때, 이 두 배열을 합쳐서 정렬함.

```
#define MAX_SIZE 100

void merge(int a[], int low, int mid, int high)
{
    int i, j, k;
    int tmp[MAX_SIZE];

    for(i=low; i<=high; i++)
        tmp[i] = a[i];

    i = k = low;
    j = mid+1;

    while(i<=mid && j<=high)
        if(tmp[i] <= tmp[j])
            a[k++] = tmp[i++];
        else
            a[k++] = tmp[j++];

    while(i<=mid)
        a[k++] = tmp[i++];
    while(j<=high)
        a[k++] = tmp[j++];
}
```

Merge Sorting (6)

– Merge Sorting

```
void mergeSort(int v[], int low, int high)
{
    int mid;

    if(low == high)
        return; /* base case */

    mid = (low + high) / 2;

    mergeSort(v, low, mid);
    mergeSort(v, mid+1, high);
    merge(v, low, mid, high);
}
```

```
void main(void)
{
    int i, v[MAX_SIZE] = { 5, 6, 9, 4, 0, 2, 1, 7, 3, 8 };

    mergeSort(v, 0, 9);
}
```

Merge Sorting (7)

– Analysis

- basic operation : merge( ) 에서 배열에 있는 두 수를 비교하는 연산
- T(n) : n 개의 정수를 merge sorting 할 때의 basic operation 수

$$T(n)=\begin{cases} 0 & n=1 \\ \underbrace{T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+T\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{왼쪽반, 오른쪽반}}+\underbrace{(n-1)}_{n-1\text{은 merge}} & n>1 \end{cases}$$

– 위 식은 아래와 같이 단수화하여 생각할 수 있다.

$$T(n)=\begin{cases} 0 & n=1 \\ 2T(n/2)+n & n>1 \end{cases}$$

n−1은 n으로 취급해도 문제 x

Merge Sorting (8)

– Analysis

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

- 위 식은, n = 2^k 일 때, 아래와 같이 order 를 구할 수 있다.

$$\begin{aligned} T(n) &= 2T(n/2) + n = 2\{2T(n/4) + n/2\} + n \\ &= 4T(n/4) + 2n = 4\{2T(n/8) + n/4\} + 2n \\ &= 8T(n/8) + 3n \\ &\cdots \\ &= nT(1) + (\log n)n \\ &= n \log n \\ &\in O(n \log n) \end{aligned}$$

Max. Conti. Subseq. Sum (MCSS)

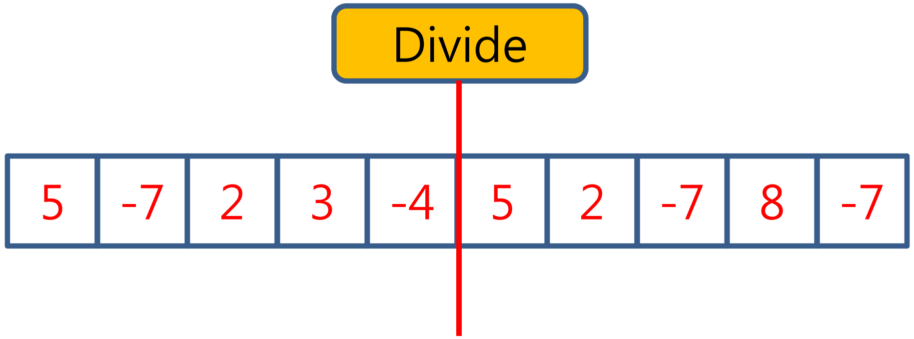
- Maximum Contiguous Subsequence Sum
  - n 개의 정수 $a_1, a_2, \ldots, a_n$ 이 주어졌을 때, 연속적인 부분 수열의 합 $\sum_{k=i}^{j} a_k$ 이 최대가 되는 구간 (i, j) 와 그 구간의 합을 계산하시오.

5	-7	2	3	-4	5	2	-7	8	-7

$$\sum_{k=3}^{9} a_k = 9$$

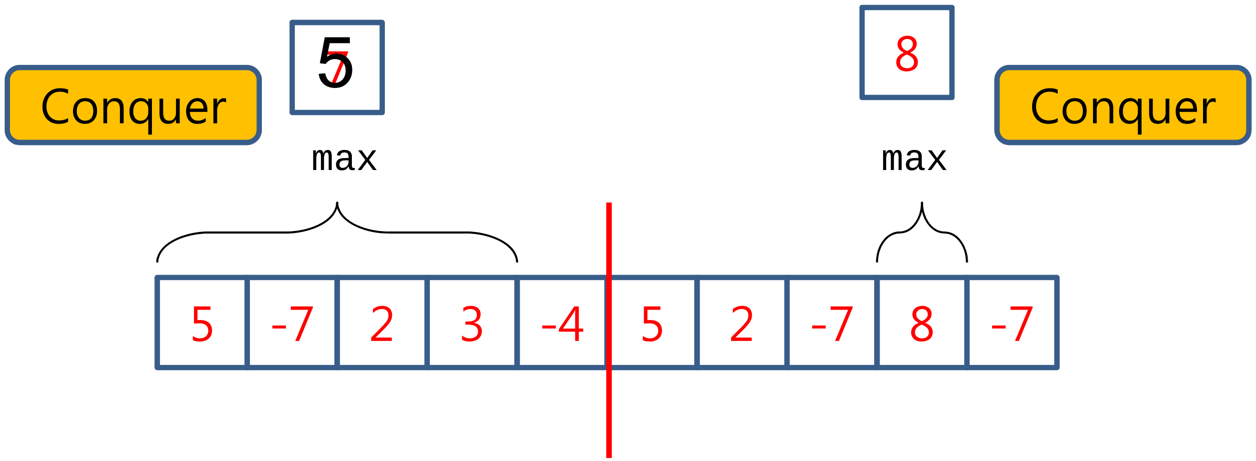
Max. Conti. Subseq. Sum (MCSS) (2)

- Divide & Conquer
  - Divide :
    - n 개의 정수배열을 크기가 각각 n/2 개인 부분배열로 나눈다.



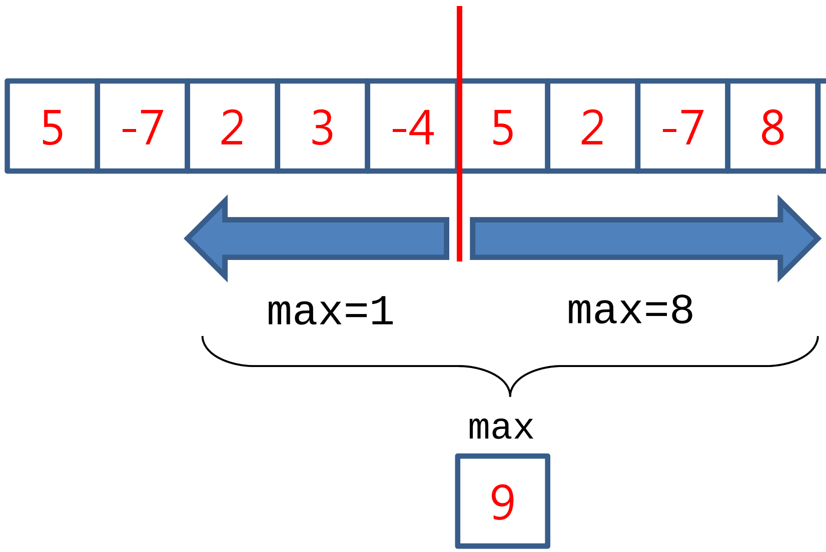
Max. Conti. Subseq. Sum (MCSS) (3)

- Divide & Conquer
  - Conquer:
    - 크기가 각각 n/2 개인 부분배열에 대하여 recursive 하게 문제를 해결한다.



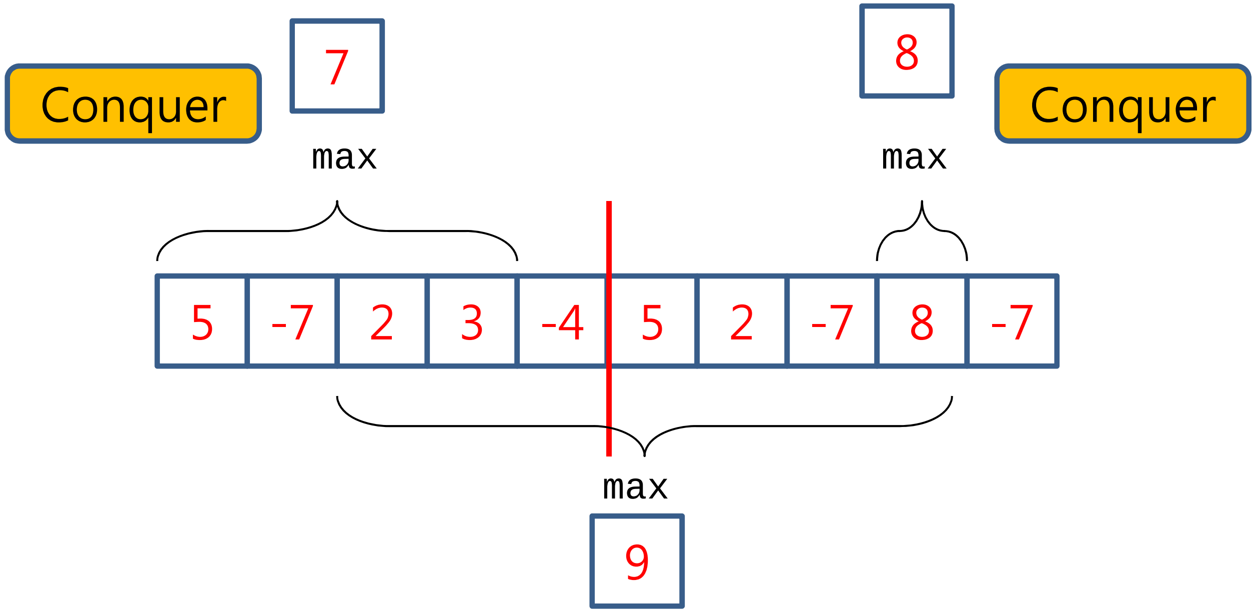
Max. Conti. Subseq. Sum (MCSS) (4)

- Divide & Conquer
  - Combine:
    - “Conquer” 단계에서 계산한 값과 다음과 같이 연속된 구간이 중앙을 걸쳐서 존재할 수 있으므로 이런 경우도 해결하여야 한다.
    - 이를 위해서는 중앙을 중심으로 왼쪽으로 연속적으로 최대가 되는 값과 중앙을 중심으로 오른쪽으로 연속적으로 최대가 되는 값을 구하여 합한다.



Max. Conti. Subseq. Sum (MCSS) (5)

- Divide & Conquer
  - Combine:
    - 따라서, 전체 데이터 중에서 최대의 합을 만드는 연속적인 구간은 다음 세 값 중의 하나다.
      - 왼쪽 n/2 데이터에서 최대연속구간의 합
      - 오른쪽 n/2 데이터에서 최대연속구간의 합
      - 중앙에 걸쳐서 최대연속구간의 합



Max. Conti. Subseq. Sum (MCSS) (6)

- Divide & Conquer
  - Analysis
    - basic operation : 배열의 숫자를 더하는 연산
    - 이 알고리즘의 "combine" 단계에서 중앙을 거쳐서 최대가 되는 연속구간을 계산하는 데 걸리는 basic operation 은 n-1 번 수행된다.
    - 따라서, 이 알고리즘의 time complexity는 다음과 같이 "merge sorting"과 같은 재귀식으로 나타난다.

$$T(n) = \begin{cases} 0 & n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1) & n > 1 \end{cases}$$

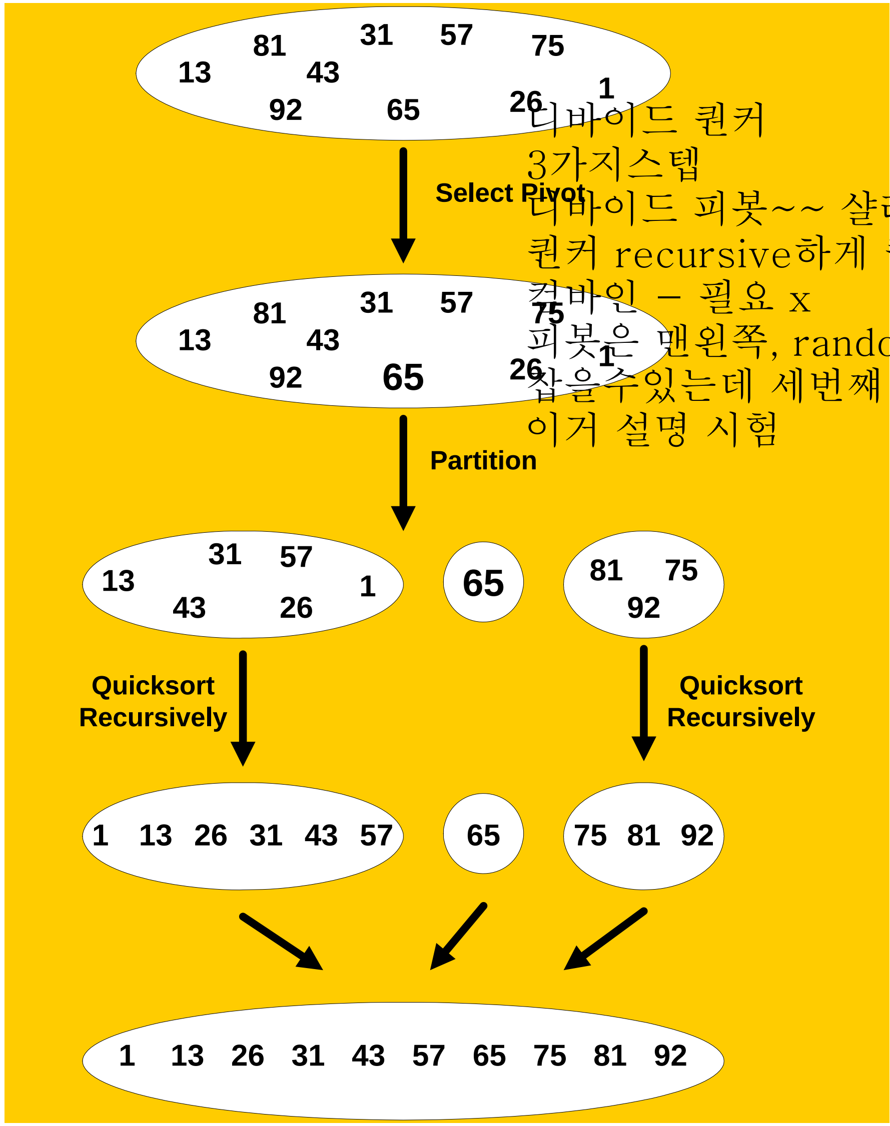
    - 결과적으로, 본 알고리즘의 time complexity 는 "merge sorting"의 time complexity $O(n\log n)$ 과 동일하다.

Quick Sorting

- Quick Sorting
  - Divide & Conquer 알고리즘
    - Divide
      - 먼저 배열 중에 임의의 데이터를 pivot (피봇)으로 선택한다.
      - 배열에 있는 데이터들을 pivot 보다 작은 데이터 그룹과 pivot 보다 큰 데이터 그룹으로 나눈다.
    - Conquer
      - Pivot 을 중심으로 나누어진 두 그룹을 recursive 하게 quick sorting 을 수행한다.
    - Combine
      - Combine 작업는 따로 필요하지 않다 (왜?)

Quick Sorting (2)

- 예



디바이드 퀀커
3가지스텝
디바이드 피봇~~ 샬라샬라
퀀커 recursive하게 퀵소트 하는 파트
컴바인 – 필요 x
피봇은 맨왼쪽, random, 맨왼쪽,맨우,중간 의 중간값으로
잡을수있는데 세번째 방법이 가장 많이 쓰이는 방법이다.
이거 설명 시험

KMU 국민대학교 KOOKMIN UNIVERSITY

Algorithmic Engineering Laboratory

Quick Sorting (3)

– Divide
  - Quick sorting 의 divide 부분을 특별히 “partition”이라고 부른다.
  - 먼저 pivot 데이터를 선택한다.
    - 가장 간단하게는 가장 왼쪽에 있는 데이터를 선택
    - Random 하게 선택하고 가장 왼쪽 데이터와 교환
    - 가장 왼쪽, 가장 오른쪽, 가장 중앙에 있는 세 데이터 중에 크기가 가장 중간인 데이터를 선택하고 가장 왼쪽에 있는 데이터와 교환

피봇 뽑을 때
앞에세개 안하는이유는
소팅이 되어있는 경우가 있는데
이경우 worstcase이므로
안정적으로 왼,오,중을 뽑는것이다.

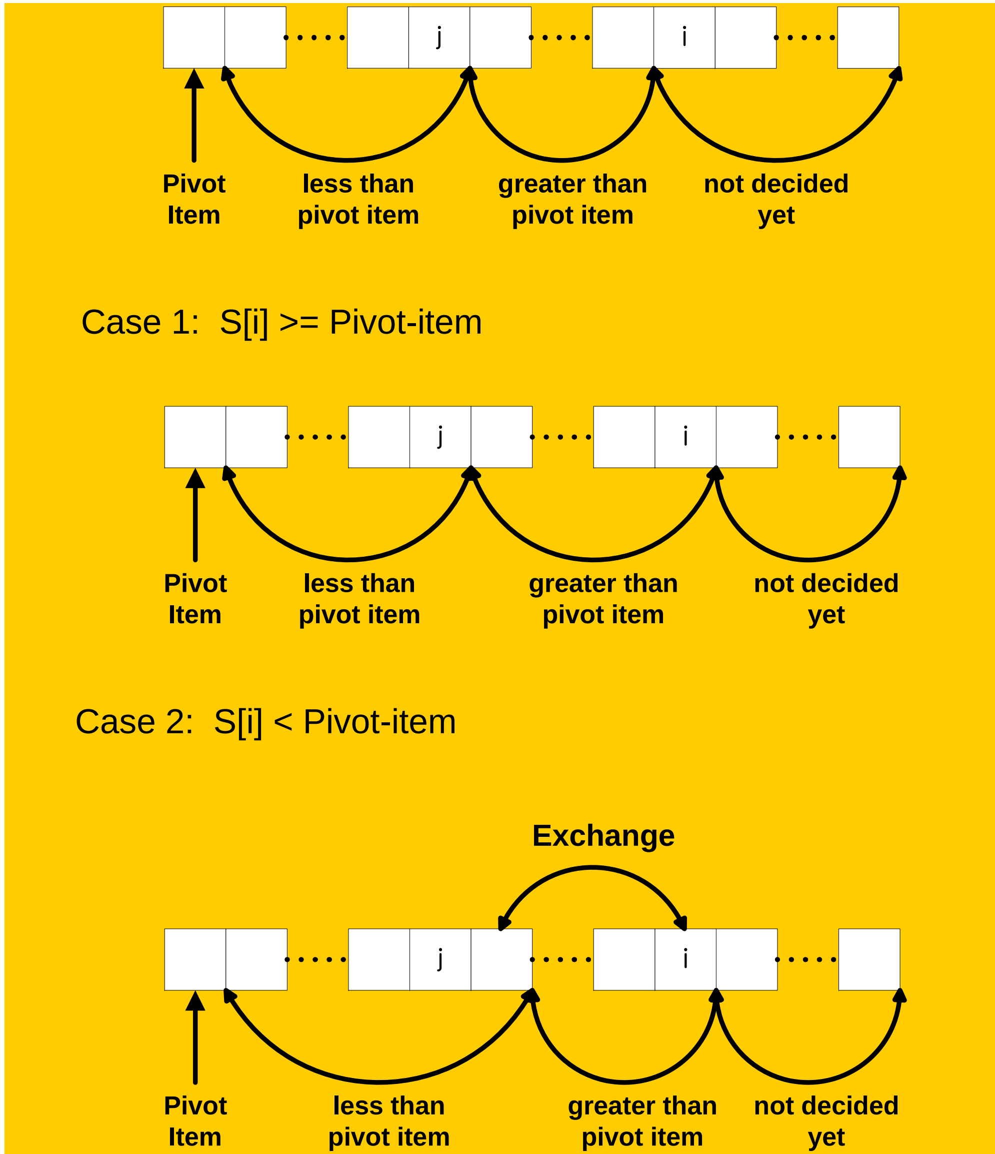
Quick Sorting (4)

– Divide (partition)

```
int partition(int a[], int low, int high)
{
    int i, j;
    int pivot, pivotPos, tmp;

    pivot = a[low];
    j = low;

    for(i=low+1; i<=high; i++)
    {
        if(a[i] < pivot)
        {
            j++;
            /* swap */
            tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }
    }

    pivotPos = j;
    tmp = a[low]; a[low] = a[pivotPos]; a[pivotPos] = tmp;

    return pivotPos;
}
```

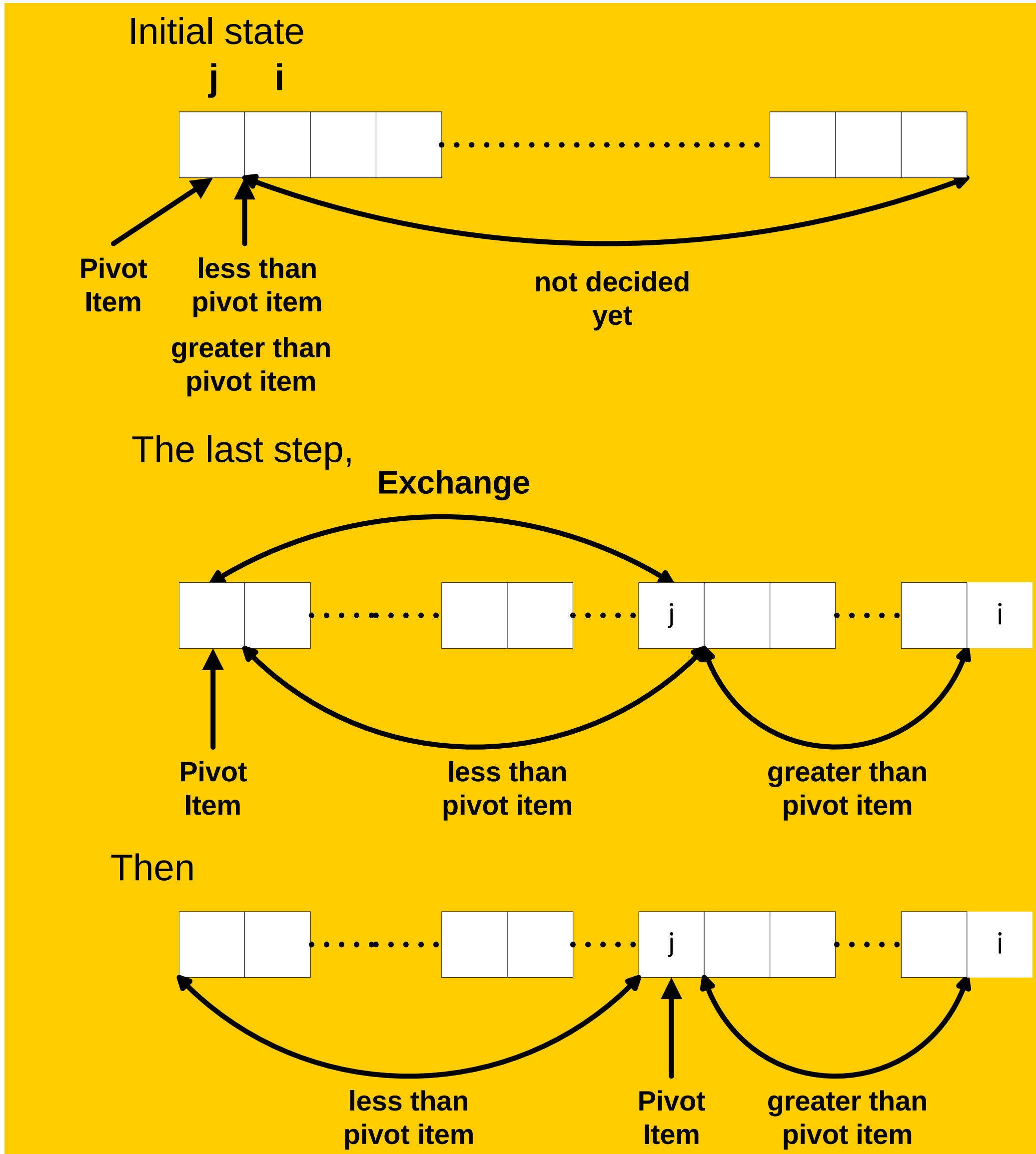
Quick Sorting (5)

– Divide (partition)



Quick Sorting (6)

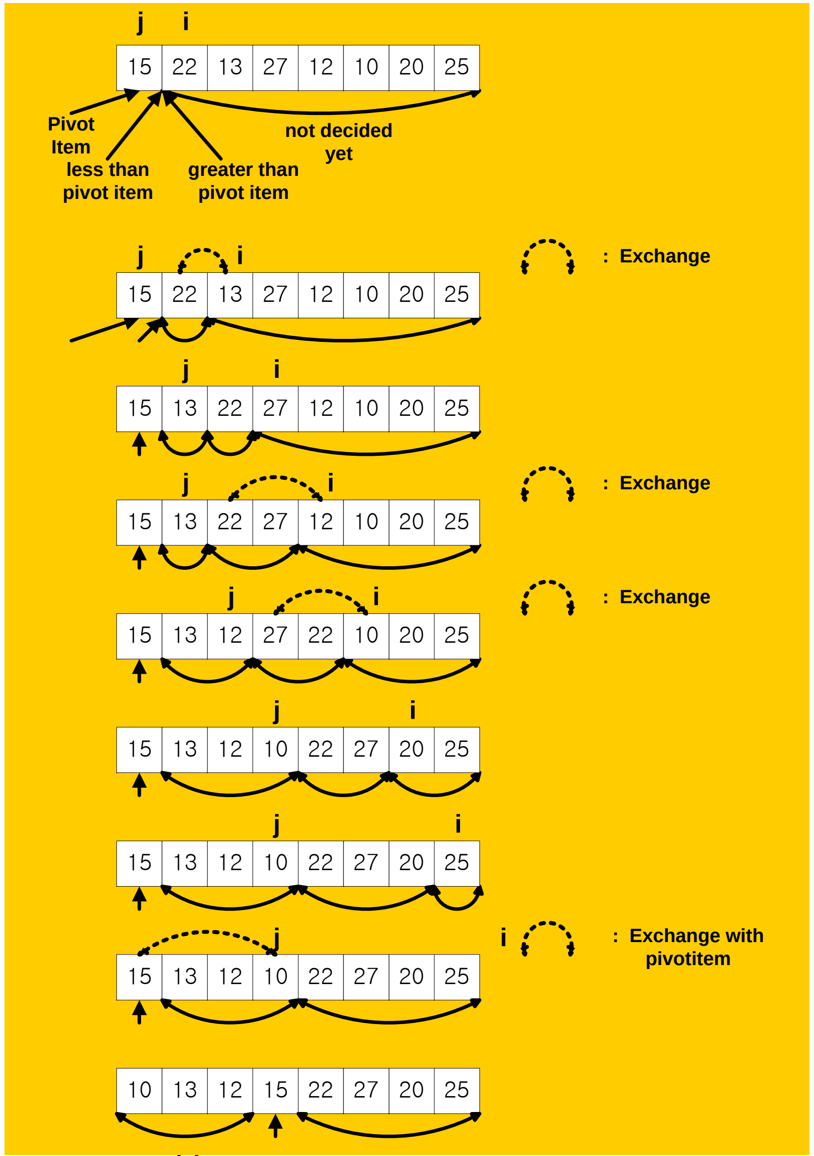
– Divide (partition)



파티션이 1번될때마다
피봇은 제자리를 찾기때문에
더이상 움직이지않음

Quick Sorting (7)

– Divide (partition)



Quick Sorting (8)

- Divide (partition)
  - Analysis
    - Basic operation : 배열에 저장된 데이터와 pivot 과 비교하는 연산
    - T(n) : n 개의 데이터에 대하여 partition 작업을 수행하는 basic operation 의 횟수

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

Quick Sorting (9)

– Quick Sorting 알고리즘

```
void quickSort(int v[], int low, int high)
{
    int pivotPos;

    if(high > low)
    {
        pivotPos = partition(v, low, high);
        quickSort(v, low, pivotPos-1);
        quickSort(v, pivotPos+1, high);
    }
}
```

```
void main(void)
{
    int i, v[MAX_SIZE] = { 5, 6, 9, 4, 0, 2, 1, 7, 3, 8 };

    quickSort(v, 0, 9);
}
```

Quick Sorting (10)

– Quick Sorting 알고리즘

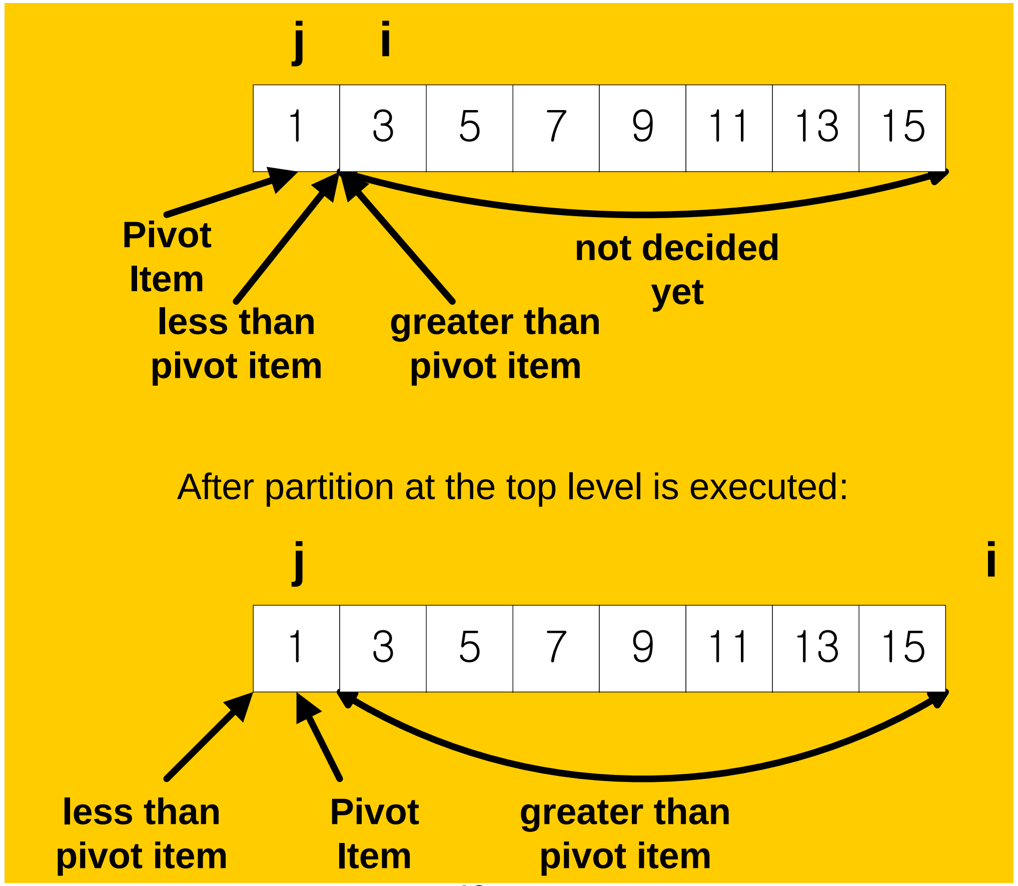
- Analysis
  - Basic operation : 배열에 저장된 데이터와 pivot 과 비교하는 연산
  - T(n) : n 개의 데이터에 대하여 quick sorting 을 수행하는 basic operation 의 횟수
  - worst case 데이터는 미리 정렬된 데이터가 입력되는 경우이다.

KMU 국민대학교 KOOKMIN UNIVERSITY

Algorithmic Engineering Laboratory

Quick Sorting (11)

– Quick Sorting 알고리즘
  - Worst-Case Analysis
    – worst case 데이터는 미리 정렬된 데이터가 입력되는 경우이다.



Quick Sorting (12)

– Quick Sorting 알고리즘

- Worst-Case Analysis
  - worst case 데이터는 미리 정렬된 데이터가 입력되는 경우이다.
  - $T(n) = T(0) + T(n\text{-}1) + n\text{-}1$

Time to sort left subarray

Time to sort right subarray

Time to partition

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n - 1 & n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + (n-1) \\ &= T(n-2) + (n-2) + (n-1) \\ &\dots \\ &= T(1) + 1 + \dots + (n-2) + (n-1) \\ &= n(n-1)/2 \quad \in \quad O(n^2) \end{aligned}$$

Quick Sorting (13)

– Quick Sorting 알고리즘

- Best-Case Analysis
  - best case 는 merge sorting 에서와 같이 나누어지는 부분배열의 크기가 각각 n/2 인 경우이다. 즉, 똑 같은 크기의 두 부분배열로 계속 나누어지는 경우이다.
  - 이 경우의 time complexity 는 merge sorting의 time complexity와 같으므로 그 order는 O($n\log n$) 이다.

- Average-Case Analysis
  - 배열에 속하는 데이터가 모두 같은 확률로 Pivot 으로 선택된다고 가정하여 분석함.
  - 이 경우의 order는 O($n\log n$) 임. (생략)

Merge / Quick Sorting

– Merge Sorting, Quick Sorting 비교

	Merge Sorting	Quick Sorting
Divide	• 매우 간단함 • 단순히 반으로 나눈다	• Partition( ) 수행 • Pivot 을 중심으로, Pivot 보다 작은 데이터그룹과 큰 데이터 그룹 두 부분으로 나눔 • O(n) 시간을 요함
Conquer	Recursive 작업	Recursive 작업
Combine	• Merge( ) 수행 • Conquer 단계에서 정렬된 두 배열을 병합하는 작업 수행 • O(n) 시간을 요함	• 필요없음

Bolts & Nuts 문제

- Bolts & Nuts
  - 문제
    - 크기가 모두 다른 n 개의 너트와 각 너트의 크기에 꼭 맞는 n 개의 볼트가 마구 섞여져 있다. 다음과 같은 작업만이 가능하다고 할 때, 크기가 꼭 맞는 n 개의 볼트-너트 조합을 만드시오.
      - 한 개의 너트와 한 개의 볼트를 끼워보고, 너트가 볼트에 비해 크기가 크다, 작다, 혹은 꼭 맞다를 판별할 수 있다.
      - 그러나, 너트끼리 그 크기를 비교할 수 없으며, 또한 볼트끼리도 그 크기를 비교할 수 없다.
  - 해결 알고리즘
    - Quick sorting 알고리즘과 유사한 방법을 사용