분할 정복

명지대학교
컴퓨터공학과
이 충기 교수

지난 주 강의 내용

- 알고리즘의 효율성
- 시간복잡도
- 알고리즘의 복잡도 분석 방법
- 점근적 표기
  - Big O 표기법
  - Ω 표기법
  - $\boldsymbol{\theta}$ 표기법

이번 주 강의 내용

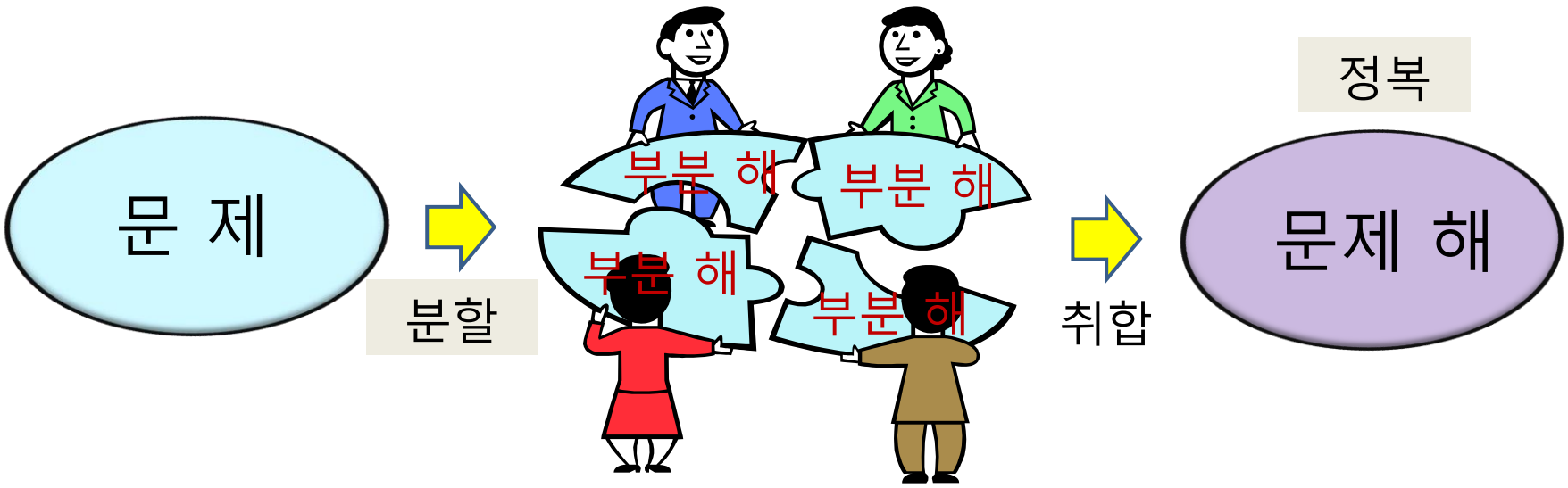
- 분할 정복
- 배열의 최대값과 최소값 찾기
- 합병 정렬
- 빠른 정렬
- 분할 정복이 부적절한 경우

분할 정복(Divide and Conquer)

설계 전략

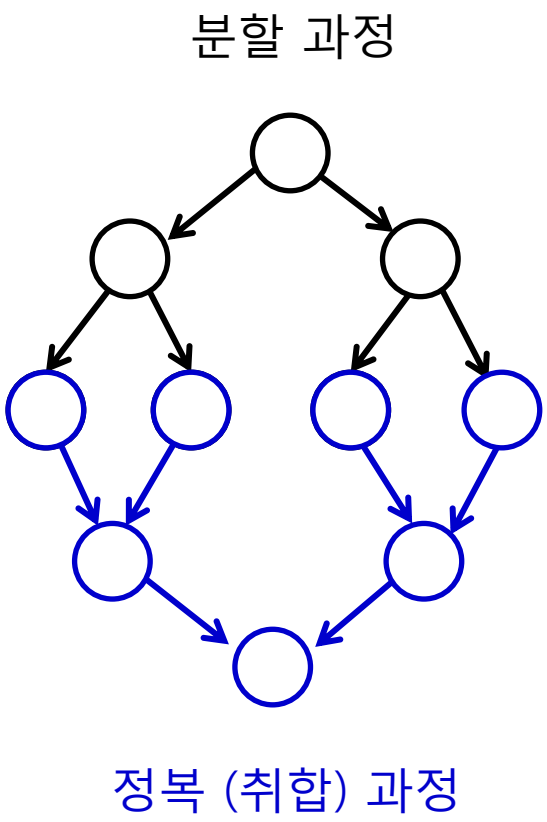
- ____(Divide)
  - 문제를 풀기 쉽도록 여러 개의 더 작은 부분 문제들(subproblem)로 나눈다.
  - 부분문제는 더 이상 분할할 수 없을 때까지 계속 분할한다.
- ____(Conquer)
  - 더 작은 부분 문제들을 _____으로 해결한다.
- ____(Merge)
  - 원래 문제에 대한 해를 구하기 위해 부분 문제들의 해를 합친다.

부분 문제



정복 과정

- 대부분의 분할 정복 알고리즘은 문제를 단순히 분할만 해서는 해를 구할 수 없다.
- 따라서 분할된 부분 문제들을 정복해야 한다. 즉, 부분 해를 찾아야 한다.
- 정복하는 방법은 문제에 따라 다르나 일반적으로 부분 문제들의 해를 취합하여 보다 큰 부분 문제의 해를 구한다.



배열의 최대값과 최소값 찾기

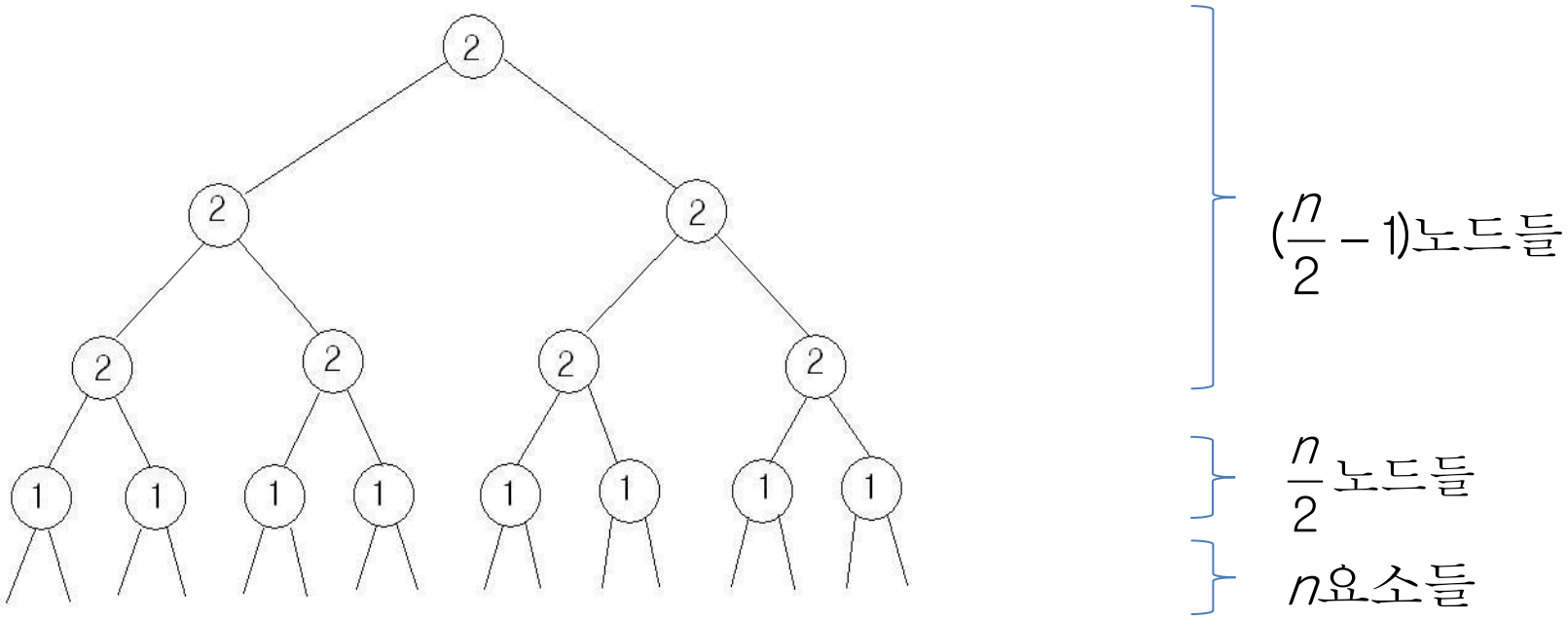
- 명확한 전략
  1. 최대값을 찾는다 . . . n - 1번의 비교 필요
  2. 남은 배열 요소들의 최소 값을 찾는다 . . . n - 2번의 비교 필요
- 명확하지 않은 전략
  1. 배열을 반으로 나눈다.
  2. 양쪽 절반들의 최대값과 최소값을 찾는다.
  3. 전체 배열의 최대값과 최소값을 구하기 위해 두 개의 최대값들과 두 개의 최소값들을 비교한다.

배열의 최대값과 최소값 찾기 알고리즘

```
// 배열 a[i..j] 의 최대값과 최소값을 찾는다
void findMaxMin(int i, int j, int low, int high)
{
        int mid, min1, max1, min2, max2;
        if (i == j) { low = a[i]; high = a[i]; }
        else if (i == j - 1) {
                if (a[i] < a[j]) {
                   low = a[i];
                   high = a[j];
                }
                else {
                   low = a[j];
                   high = a[i];
                }
        }
        else {
                mid= ⌊(i+j)/2⌋ ;
                findMaxMin(i, mid, min1, max1);
                findMaxMin(mid+1, j, min2, max2);
                low = Min(min1, min2);
                high = Max(max1, max2);
        }
}
```

최대값과 최소값 찾기 알고리즘 분석

- 알고리즘에 대한 다음 실행 트리(tree)를 보아라. 각 노드 안의 숫자는 배열 요소들간의 비교 횟수를 나타낸다.



- 위 트리에서 비교 횟수들을 합하면

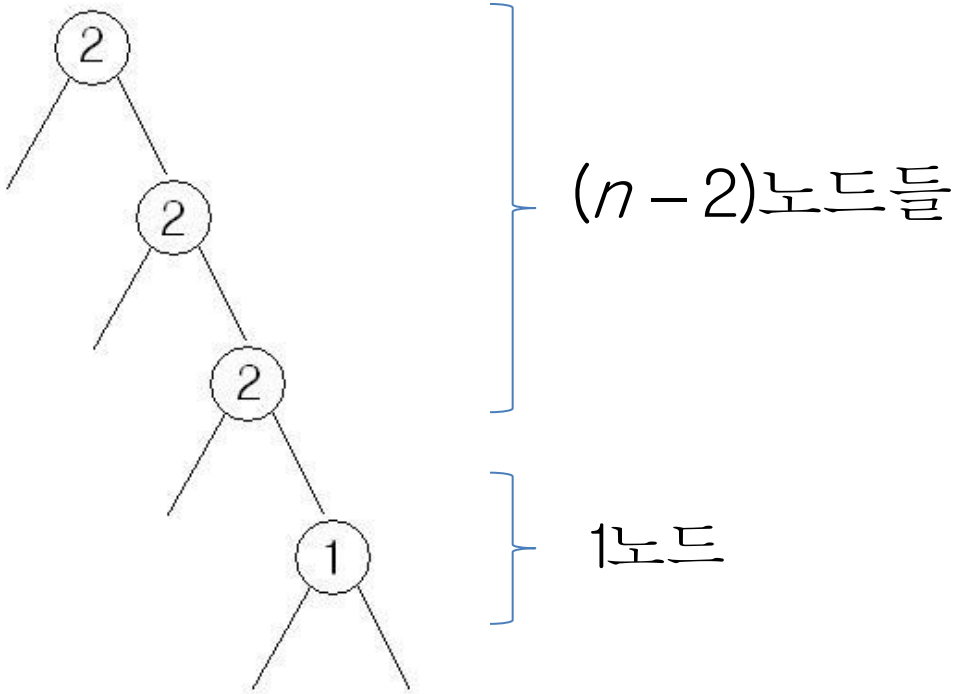
$$2 \times (\frac{n}{2} - 1) + \frac{n}{2} = \frac{3n}{2} - 2$$

(이는 최선의 결과라고 증명할 수 있다.)

주: 모든 개선은 트리의 가장 낮은 수준(level)에서 이루어진다.
우리는 점화식을 이용할 수도 있다.

균형 취하기(balancing)

- 일반적으로 문제를 거의 같은 크기의 하위 문제들로 나누는 것이 가장 좋다. 위 문제에서 우리가 배열의 한 요소를 S_1에 넣고 나머지 요소들을 S_2에 넣는다면 실행 트리는 다음과 같다:



총 비교 횟수 = 2(n-2)+1 = 2n-3

합병 정렬(Merge Sort)

- N개의 숫자들을 정렬하는 명확한 방법
  1. 최대값을 찾는다
  2. 남아있는 숫자들을 재귀적으로 정렬한다.

분석

요구되는 비교 횟수

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}=\theta(n^2)$$

명백히 이 방법은 "_____" 발견법(heuristic) 을 이용하지 않는다.

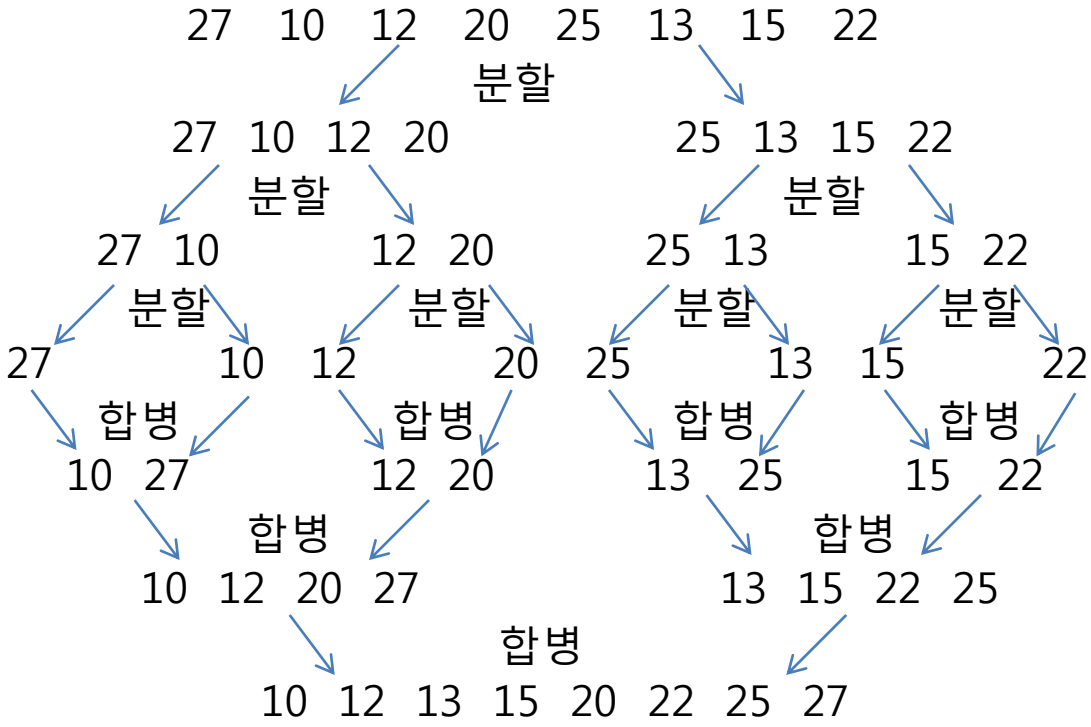
다음 방법은 "_____" 발견법을 이용하고 단지 $\theta(n \log n)$ 비교 횟수를 요구 한다.

합병 정렬(계속)

전략

- 배열을 반으로 나눈다
- 두 개의 반들을 각각 정렬한다
- 정렬한 두 개의 반들을 합병한다.

예제

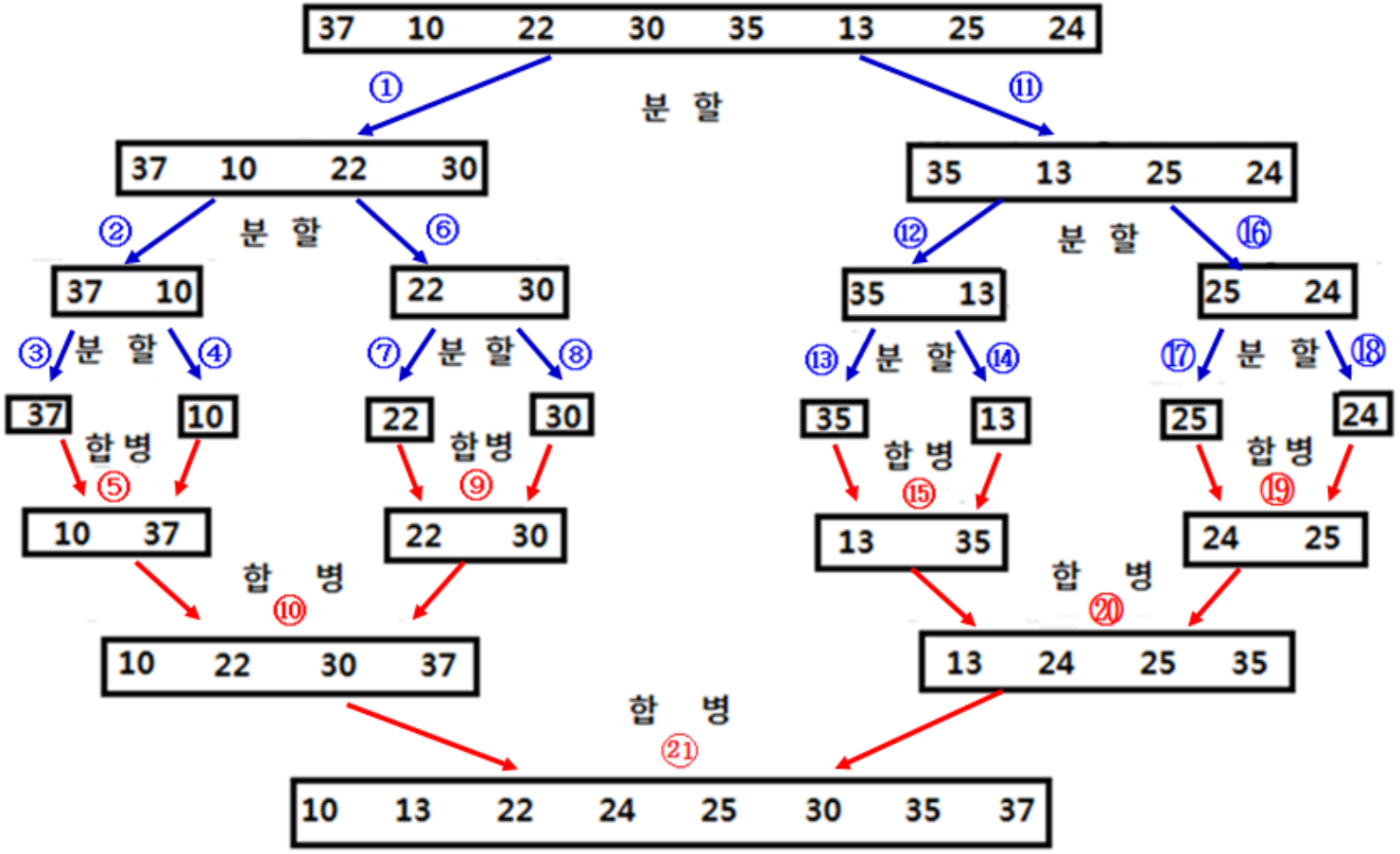


합병 정렬 알고리즘

```
// 배열 A[i .. j]를 정렬한다.
void Mergesort(int i, int j)
{
    int m;
    if (i< j) {
            m = ⌊(i+j)/2⌋;
            Mergesort(i, m);
            Mergesort(m+1, j);
            Merge(i, m, j);
    }
}
```

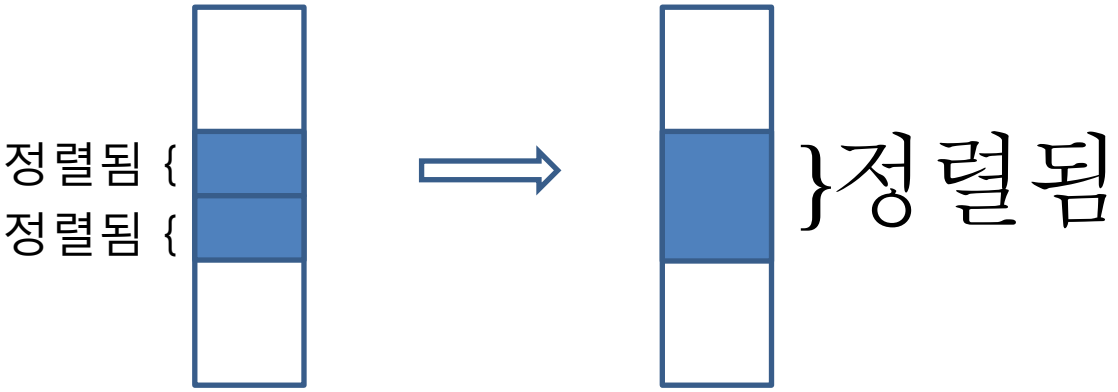
주: 첫 호출: Mergesort(1, n)

- 입력 크기가 n=8인 배열 A=[37, 10, 22, 30, 35, 13, 25, 24]에 대하여



합병 알고리즘

Merge(x, y, z)



- 임시 저장을 위해 다른 배열을 사용한다.
- 크기가 m 인 부분과 크기가 n인 부분을 합병하는 것은 최악의 경우에 _____번의 비교가 필요하다.

Merge 메소드

```
// 정렬된 배열 부분 A[low . . mid] 과 A[mid+1 . . high]를 합병한다.
void MERGE(int low , int mid, int high)
{
      int[ ]   B= new int [high]; // 합병된 결과를 저장하기 위한 임시 배열
      int      h, i , j, k;
      h = low;  i = low;  j = mid +1;

      while (h <= mid &&  j <= high) {
          if (A[h] <= A[j] { B[i] = A[h]; h = h + 1;}
          else { B[i] = A[j]; j = j + 1; }
          i = i+1;
      }
      if (h > mid) {
         for (k = j; k <= high; k++) {
             B[i] = A[k];
             i = i+1;
         }
      }
      else
          for (k = h; k <= mid; k++) {
             B[i] = A[k];
              i = i+1;
          }
      for (k = low; k <= high; k++)
        A[k] = B[k];
}
```

합병 정렬 알고리즘 시간복잡도 분석

T(n): n개의 배열 요소들을 정렬하기 위해 필요한 비교 횟수

$T(n) = 2T(\frac{n}{2}) + n - 1,\ n \geq 2$

$T(1) = 0$

가정: $n = 2^k,\ k>1$

$2T(\frac{n}{2})$: 크기가 반인 두 배열을 재귀적으로 정렬하는데 필요한 비교 횟수

$n-1$: Merge 하기 위해 최악의 경우에 필요한 비교 횟수

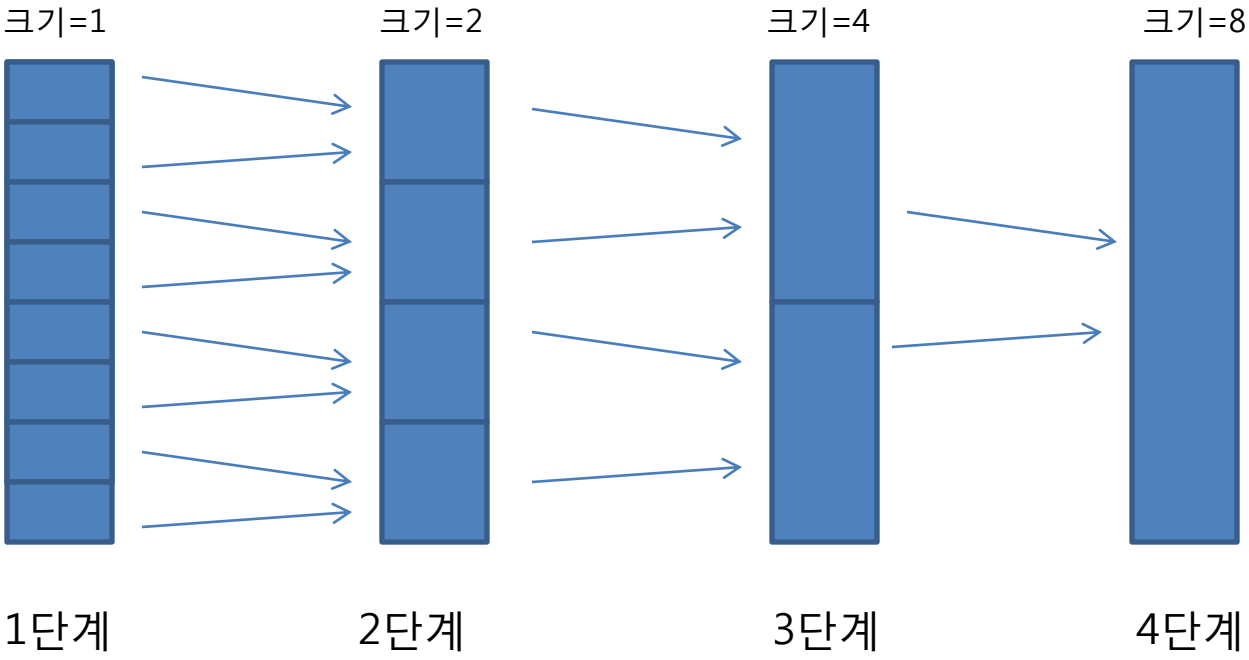
$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n - 1 \\
&= 2(2T(\frac{n}{2^2}) + \frac{n}{2} - 1) + n - 1 = 2^2 T(\frac{n}{2^2}) + n - 2 + n - 1 \\
&= 2^2 T(\frac{n}{2^2}) + 2n - (2+1) = 2^2(2T(\frac{n}{2^3}) + \frac{n}{2^2} - 1) + 2n - (2+1) \\
&= 2^3 T(\frac{n}{2^3}) + 3n - (2^2 + 2 + 1) \\
&\vdots \\
&= 2^k T(\frac{n}{2^k}) + kn - \sum_{i=0}^{k-1} 2^i \\
&= 2^k T(1) + kn - \frac{2^k - 1}{2-1} \\
&= kn - 2^k + 1 \\
&= n\log n - n + 1 \\
&= \Omega(n \log n)
\end{aligned}
$$

합병 정렬 알고리즘 분석

관찰

- 자주 분할정복 프로그램에서 재귀를 제거하면 더 효율적인 프로그램을 얻을 수 있다.

비 재귀적인 Mergesort



아이디어: Mergesort에서 배열 부분들을 합병하는 방법을 이용한다.

비 재귀적인 Mergesort 프로그램

```
size = 1;

while (size < n) {
   for (i = 1;  i <= n;  i = i + 2 * size)
      Merge(i , i + size - 1, i + 2 * size - 1)
   size = size * 2;
}
```

분석: $\theta(\log n)$ 단계

각 단계는 θ(n) 비교가 필요하다.

따라서 비교 횟수는 $\theta(n \log n)$ 이다.

합병 정렬의 단점

- 합병 정렬의 공간 복잡도: O(n)
- 입력을 위한 메모리 공간 (입력 배열)외에 추가로 입력과 같은 크기의 공간 (임시 배열)이 별도로 필요.
- 2개의 정렬된 부분을 하나로 합병하기 위해, 합병된 결과를 저장할 곳이 필요하기 때문

응용

- 합병 정렬은 외부정렬의 기본이 되는 정렬 알고리즘이다.
- 연결 리스트에 있는 데이터를 정렬할 때에도 퀵 정렬이나 힙 정렬 보다 훨씬 효율적이다.
- 멀티코어 (Multi-Core) CPU와 다수의 프로세서로 구성된 그래픽 처리 장치 (Graphic Processing Unit)의 등장으로 정렬 알고리즘을 병렬화하는 데에 합병 정렬 알고리즘이 활용

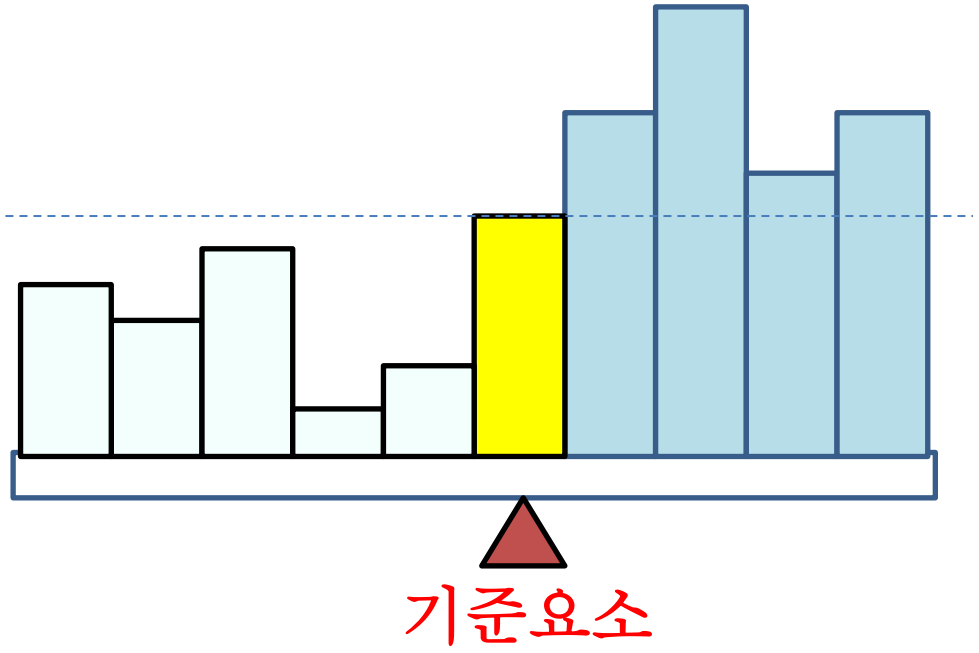
빠른 정렬(Quicksort)

- 1962년에 영국의 Hoare가 개발
- 빠른정렬(Quicksort)란 이름이 오해의 여지가 있다. 왜냐하면 사실 절대적으로 가장 빠른 정렬 알고리즘이라고 할 수는 없기 때문이다. 차라리 "분할교환정렬(partition exchange sort)"라고 부르는 게 더 정확하다.
- 평균 시간복잡도: $\theta(n \log n)$
  최악 시간복잡도: $\theta(n^2)$

빠른 정렬 알고리즘

- 문제: n 개의 정수를 올림차순으로 정렬
- 입력: 정수 $n > 0$, 크기가 n 인 배열 S[1..n]
- 출력: 올림차순으로 정렬된 배열 S[1..n]
- 기본 아이디어
  - 배열을 두 부분으로 분할한다. 분할할 때 배열 내의 한 기준 요소보다 작거나 같은 요소들은 그 앞부분에 위치시키고, 그 기준 요소보다 큰 요소들은 모두 그 뒷부분에 위치시킨다.
  - 각 분할된 부분을 재귀적으로 정렬한다.

- 빠른 정렬은 기준요소(pivot)이라 일컫는 배열의 요소를 기준으로 기준요소보다 작거나 같은 숫자들은 왼편으로, 기준요소보다 큰 숫자들은 오른편에 위치하도록 분할하고, 기준요소를 그 사이에 놓는다.
- 빠른 정렬은 분할된 부분문제들에 대하여 위와 동일한 과정을 재귀적으로 수행하여 정렬



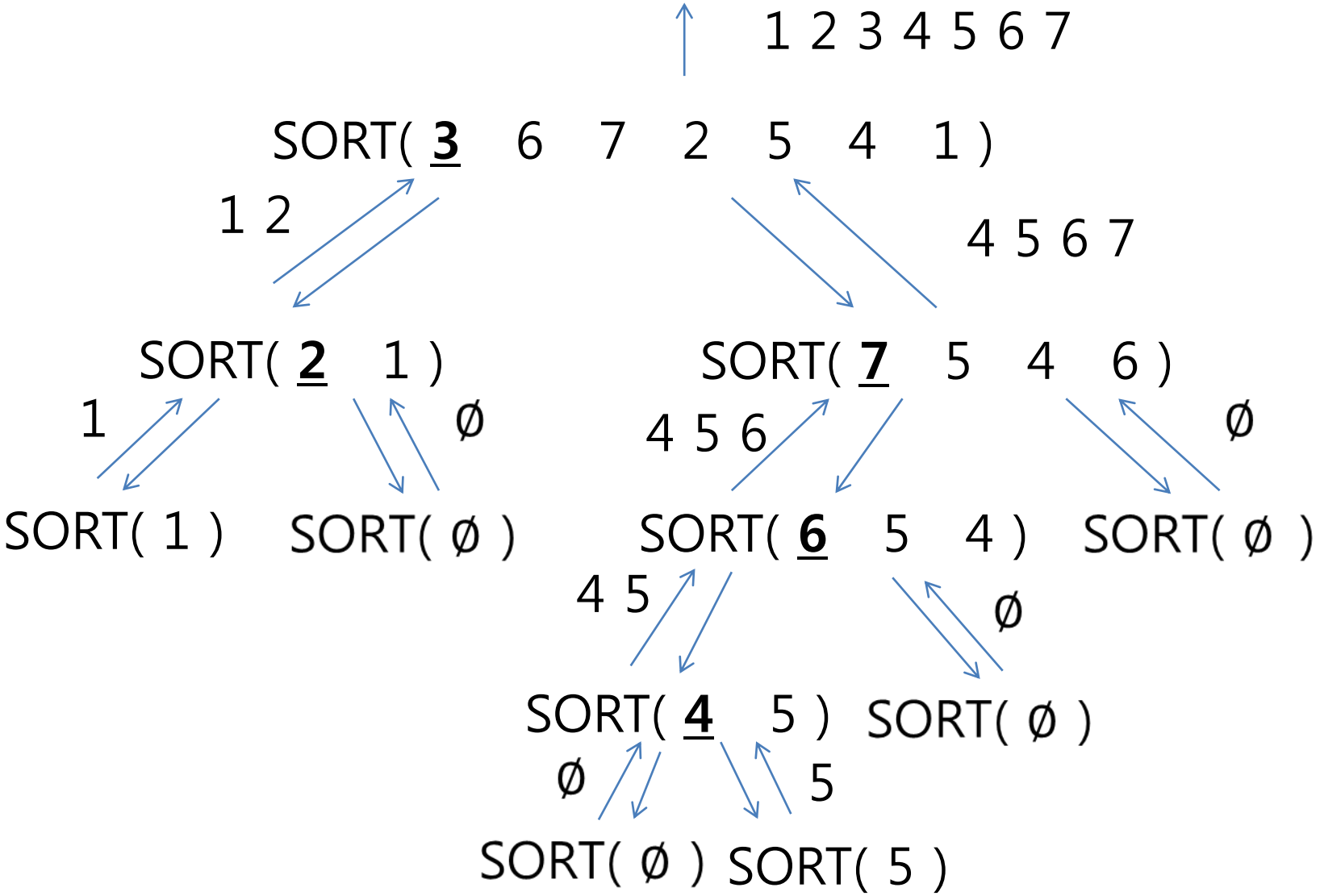
빠른 정렬

- 배열을 어떻게 분할하는가?
  - 임의의 한 요소 x를 선택한다(보통 첫 번째 요소 선택)
  - 배열을 두 부분으로 나눈다
    -> x보다 작거나 같은 요소들 => S_1
    -> x보다 큰 요소들 => S_2
- 부분 문제들을 어떻게 푸는가?
  - 물론 재귀적으로 푼다
- 부분 문제들에 대한 해들을 어떻게 합치는가?
  - 아주 쉽다. 즉, 정렬한 S_1을 처음에
  - 다음으로 x
  - 다음으로 정렬한 S_2을 마지막에

빠른 정렬 알고리즘

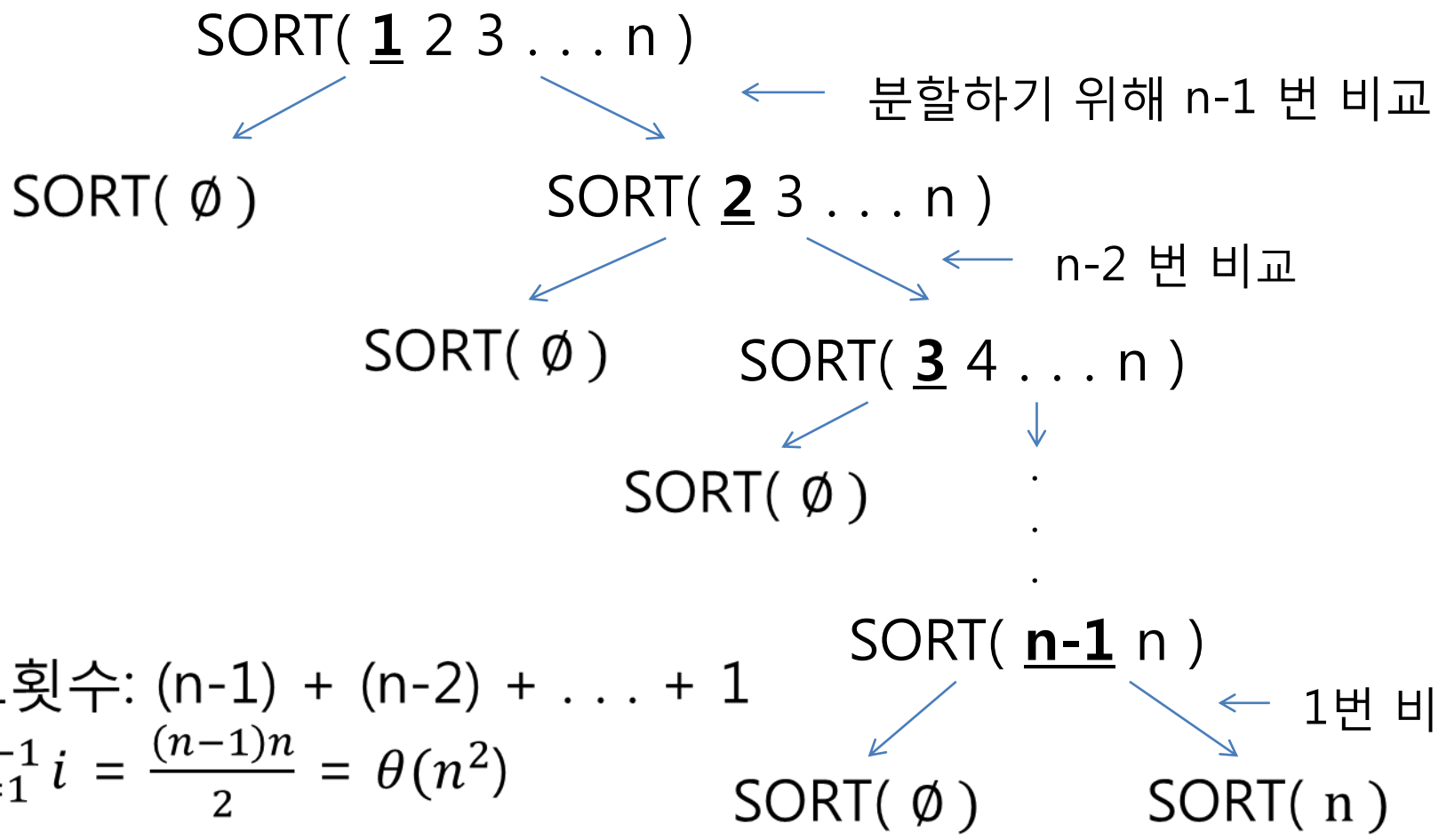
```
// 입력: 배열 S(정렬되지 않음)
// 출력: 정렬된 배열
procedure quicksort(S) {
        if (S가 1개 이하의 요소들을 포함한다)
        then return (S)
        else {
           x = S의 임의의 한 요소
           S1 = x보다 작거나 같은 S의 요소들의 부분배열
           S2 =  x보다 큰 S의 요소들의 부분배열
           return (quicksort(S1) followed by x
                     followed by quicksort(S2) )
         }
}
```

예: 집합에서 임의의 요소는 첫 번째 요소를 선택한다고 가정하자.



최악 시간복잡도 분석

주어진 집합이 정렬된 경우



총 비교횟수: (n-1) + (n-2) + . . . + 1

$= \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \theta(n^2)$

빠른 정렬 알고리즘

```
// 배열 S[low] . . . S[high]를 정렬한다
void quicksort (int low, int high)
{
        int pivotpoint;
        if (high > low) {
            pivotpoint = partition(low, high);  // 분할
            quicksort(low, pivotpoint - 1);
            quicksort(pivotpoint + 1, high);
        }
}
```

주: 배열은 quicksort(1, n)을 호출함으로써 정렬된다.

분할 알고리즘

2단계 알고리즘

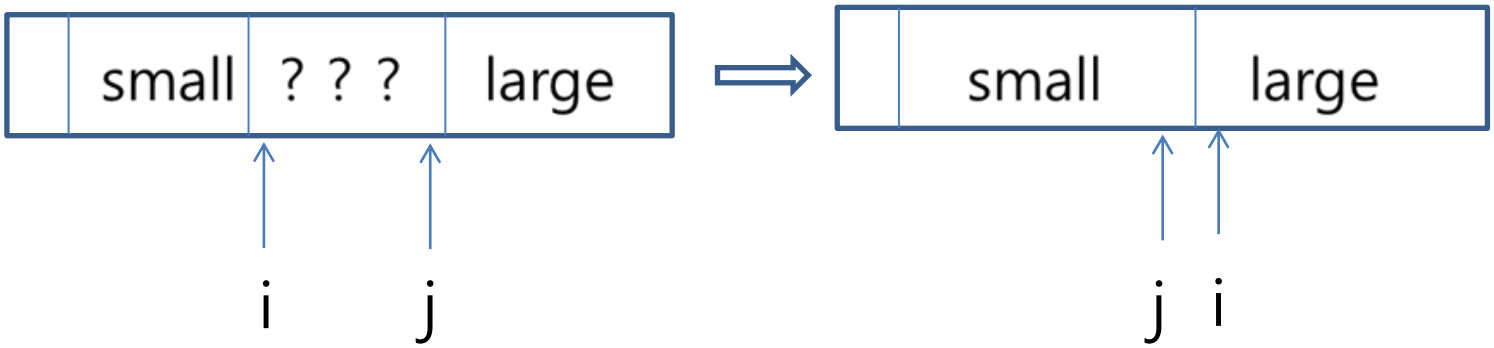
1. 배열을 다음과 같이 재배열한다:

S[low] ≤ S[low] > S[low]

2. S[low]를 "≤ 구역"의 마지막 요소와 교환한다.

분할 알고리즘

주 아이디어



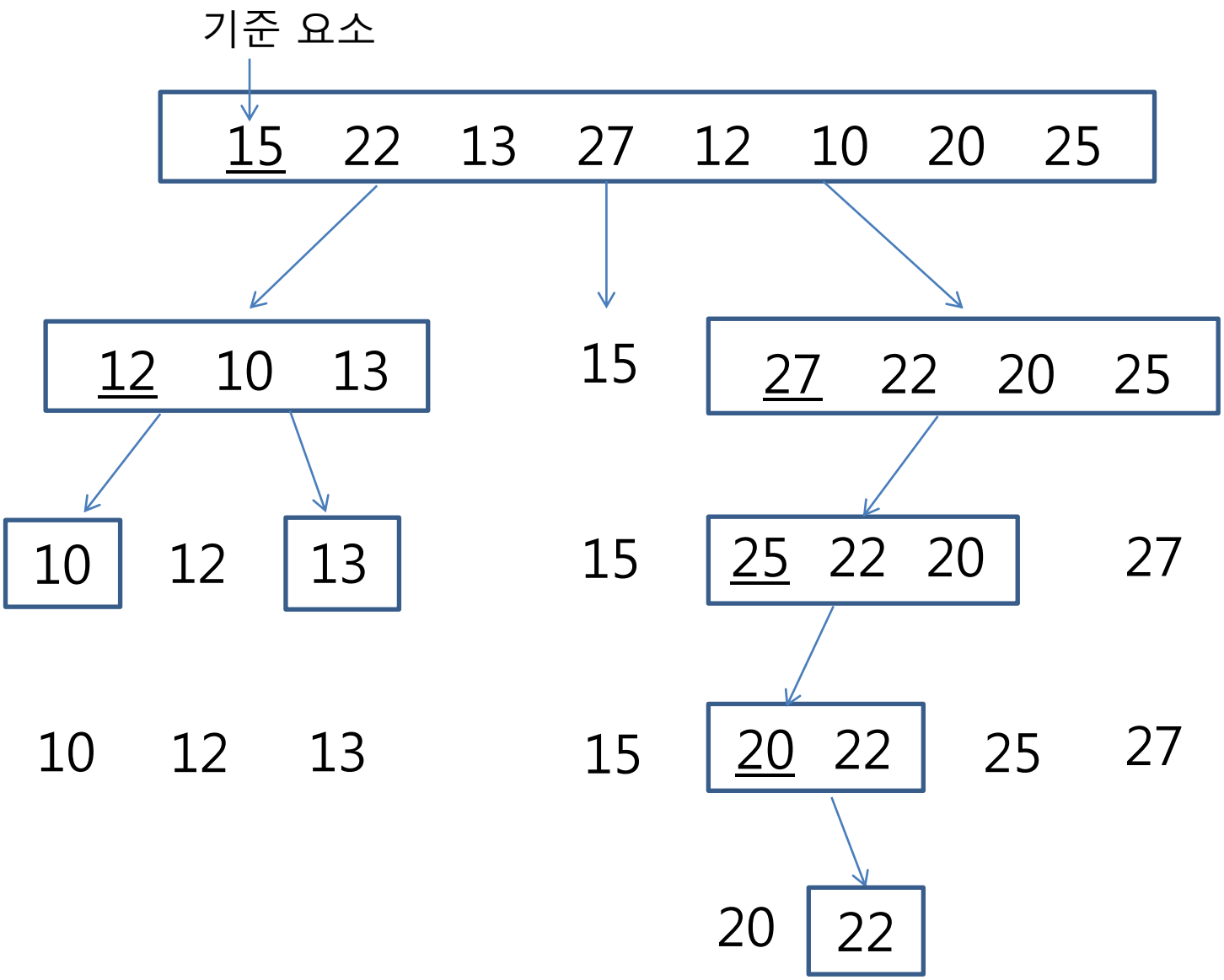
경우 1: S[i] ≤ S[low] 이면 i = i + 1
S[j] > S[low] 이면 j = j - 1

경우 2: S[i] > S[low] 이고 S[j] ≤ S[low] 이면
S[i]와 S[j]를 교환하고 i = i + 1, j = j - 1

분할 알고리즘

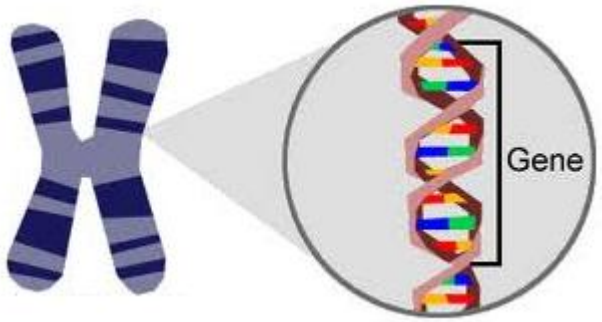
```
int partition (int low, int high) {
    int i, j;
    i = low + 1;
    j = high;
    while (i <= j) {
        if (S[i] <= S[low]) i = i + 1;
      else if (S[j] > S[low]) j = j - 1;
      else {
          S[i] ↔ S[j];
          i = i + 1;
          j = j - 1;
      }
    }
    S[low] ↔ S[j];
    return j;
}
```

빠른 정렬


기준 요소
15 22 13 27 12 10 20 25
12 10 13
15
27 22 20 25
10
12
13
15
25 22 20
27
10 12 13
15
20 22
25
27
20
22

응용

- 빠른 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘이다.
- 빠른 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능을 보인다.
- 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미 배열(suffix array)과 함께 빠른 정렬이 활용된다



분할 정복 적용 시 주의할 점

- 분할 정복이 부적절한 경우는 입력이 분할될 때마다 분할된 부분문제의 입력 크기의 합이 분할되기 전의 입력 크기보다 매우 커지는 경우이다.

토끼 문제

한 쌍의 토끼들은 한 달이 될 때에 한 쌍의 토끼를 낳고 두 달이 될 때에 또 다른 한 쌍의 토끼를 낳는다. 내가 방금 한 쌍의 새로 태어난 토끼들을 샀다면 지금부터 *n*번째 달에 몇 쌍의 토끼들이 태어날까?

주: *n* 이 주어지면 새로 태어난 토끼 쌍들의 수를 구해야 한다. *n*은 매개변수이다.

토끼 문제의 설계와 분석

f_i 를 지금부터 i 번째 달에 태어난 토끼 쌍들의 수라고 하자.

i --> 0 1 2 3 4 5 6 ...

f_i --> 1 1 2 3 5 8 13 ...

f_i 는 다음 점화식을 만족시킨다:

$$f_i = f_{i-1} + f_{i-2}, \; i \geq 2$$

$$f_0 = 1$$

$$f_1 = 1$$

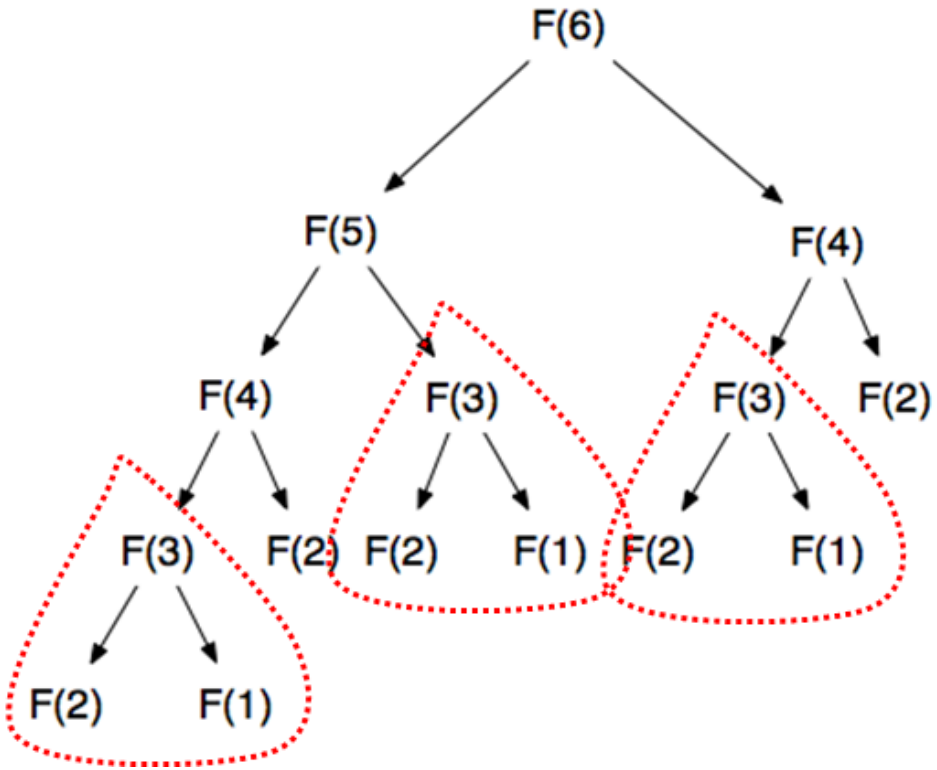
위 수열은 피보나찌(Fibonacci) 수열이라고 부른다.

토끼 문제 해

```
int fib1(int n)
{
    if (n <= 1)
        return 1;
    else
        return  fib1(n-1) + fib1(n-2);
}
```

토끼 문제 해 분석

- 분할 정복이 부적절한 경우는 입력이 분할될 때마다 분할된 부분문제의 입력 크기의 합이 분할되기 전의 입력 크기보다 매우 커지는 경우이다.



- 예를 들어, n 번째의 피보나치 수를 구하는데 F(n) = F(n-1) + F(n-2)로 정의되므로 재귀 호출을 사용하는 것이 자연스러워 보이나, 이 경우의 입력은 1개이지만, 사실상 n의 값 자체가 입력 크기인 것이다.
- 따라서 n이라는 숫자로 인해 2개의 부분 문제인 F(n-1)과 F(n-2)가 만들어지고, 2개의 입력 크기의 합이 (n-1) + (n-2) = (2n-3)이 되어서, 분할 후 입력 크기가 거의 2배로 늘어난다. 이전 슬라이드의 그림은 피보나치 수 F(5)를 구하기 위해 분할된 부분 문제들을 보여준다. F(2)를 5번이나 중복하여 계산해야 하고, F(3)은 3번 계산된다.

토끼 문제 해 2

```
int fib2(int n)
{
        int f[0..n];
        f[0] =1;  f[1] = 1;
        for (int i = 2;  i <= n; i++)
          f[i] = f[i-1] + f[i-2];  // f[n]이 답이다.
}
```

주: 시간복잡도: n – 1 = O(n)

요약

균형취하기를 적용한 분할정복은 풀어야 할 하위 문제들이 독립적이라면 (중복 계산이 없다면) 유용한 기법이다. 또한 분할 단계와 합병단계는 효율적이어야 한다.

1. 피보나찌 문제는 하위문제들이 독립적이지 않은 예이다.

2. 대개 분할 또는 합병 단계 중 하나가 쉽다.

3. 문제는 보통 2개의 하위 문제들로 나뉘어지나 항상 그렇지는 않다.
   - 예: 이진검색에서는 하나의 하위 문제들로 나누어지는 경우도 있다.
   - 예: 두 개보다 많은 하위 문제들로 나누어지는 경우도 있다.