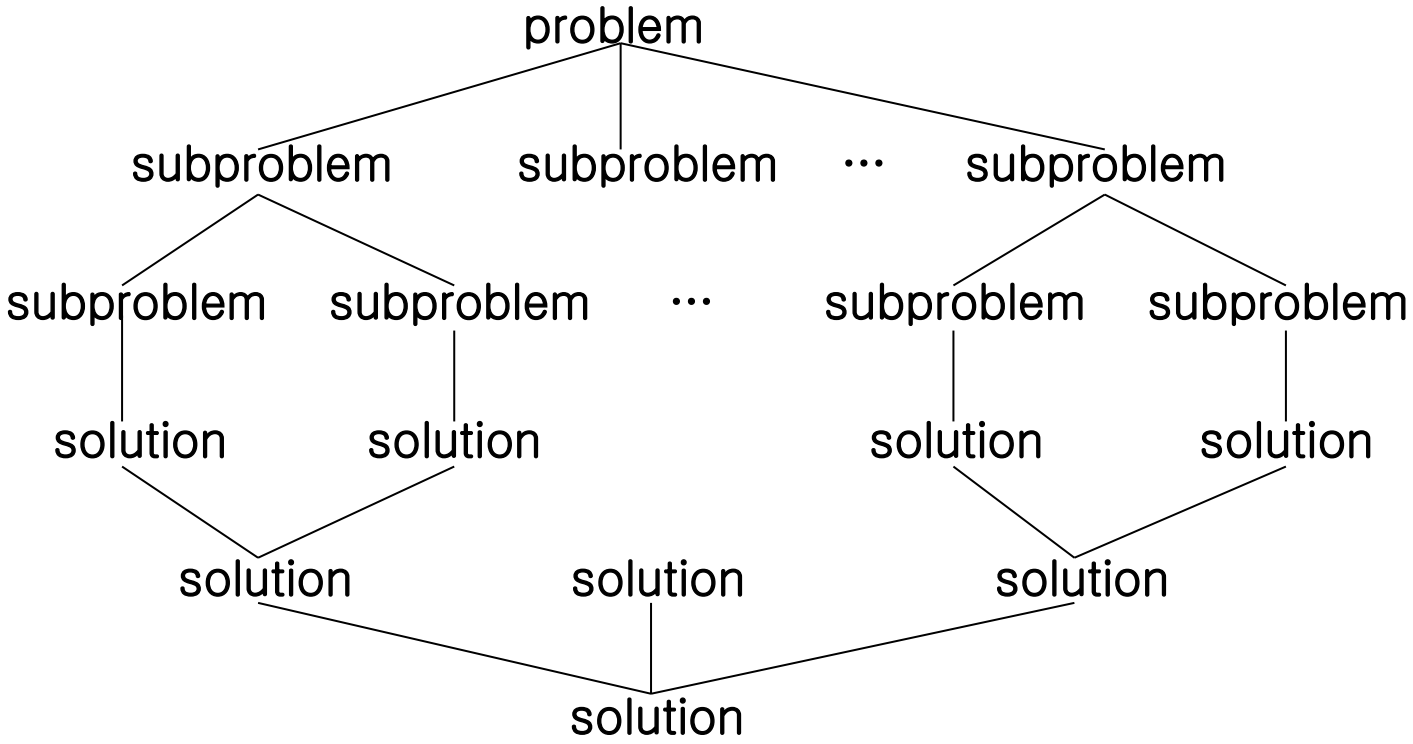
제 2 장 분할정복법 (divide-and-conquer)

- 일반적인 방법
- 이진탐색(이분탐색)
- 합병 정렬

일반적인 방법

- 주어진 문제를 k (1 ≤ k ≤ n) 개의 부분 문제들(subproblems)로 나눈 후, 각 부분 문제들의 해를 구하여 원래 문제의 해답으로 결합하는 방법이다.



- subproblem 들은 일반적으로 원래 problem과 같은 유형이며, 처리해야할 데이터의 크기만 작다.

  -> 순환 호출을 사용하는 것이 유리한다.
- 제어 추상화: 제어의 흐름만을 구체적으로 보여줌

```
Type DAndC(P)
{
    if Small(P) return S(P);
    else {
        divide P into smaller instances P1, P2, …, Pk, k ≥ 1;
        apply DAndC to each of these subproblems;
        return Combine( DAndC(P1), DAndC(P2) ,…, DAndC(Pk), ));
     }
 }
```

- Small(P)는 입력의 크기가 분할하지 않고도 그 답이 계산될 수 있을 정도로 충분히 작은지의 여부를 판단하는 조건문이다.
- Combine 은 k개의 부분 문제들에 대한 해를 사용하여 P의 해를 결정해준다.
- DAndC의 계산시간

$$T(n) = \begin{cases} g(n) & \text{small } n \\ T(n_1) + T(n_2) + \ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

- 순환 관계식 풀기

  분할-정복 알고리즘의 복잡도는 일반적으로 다음의 순환식으로 표시된다.

$$T(n) = \begin{cases} T(1) & \text{n = 1} \\ aT(\frac{n}{b}) + f(n) & \text{n > 1} \end{cases}$$

여기서 a 와 b는 이미 결정된 상수들이며, $T(1)$ 은 이미 알려져 있고, n 은 b의 멱수(power)로 가정한다.

순환식 풀기 예제

- 예제 (치환법)

  $a = 2, b = 2, T(1) = 2, f(n) = n$ 일 때,

  $T(n) = 2T(n/2) + n$

  $= 2[2T(n/4) + n/2] + n$

  $= 4T(n/4) + 2n$

  $= 4[2T(n/8) + n/4] + 2n$

  $= 8T(n/8) + 3n$

  $= \cdots$

  * 모든 $1 \le i \le \log_2 n$ 에 대해 $T(n) = 2^i T(n/2^i) + i \cdot n$ 이므로

    $i = \log_2 n$ 이면 $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log_2 n = n \log_2 n + 2n$

2.1 이진 탐색(이분탐색)

- 문제: $a_0 \le a_1 \le \ldots \le a_{n-1}$ 인 리스트 $a_0, a_1, \ldots, a_{n-1}$ 에서 x의 값이 존재한다면 (즉 $x = a_j$) 그 위치 j를 결정하라.

  ➔ $P = (n, a_0, a_1, \ldots, a_{n-1}, x)$
- 분할-정복 기법 적용
  - n = 1 이면 Small(P)가 true, 그리고 $x = a_i$ 이면 $S(P) = i$
    $x \neq a_i$ 이면 $S(P) = -1$
  - n > 1 이면, 어떤 index k 에 대하여

    $P1 = (k, a_0, a_1, \ldots, a_{k-1}, x)$

    $P2 = (1, a_k, x)$

    $P3 = (n-k-1, a_{k+1}, a_{k+2}, \ldots, a_{n-1}, x)$

    $x = a_k$ 이면 P는 해결되고, $x < a_k$ 이면 P1, $x > a_k$ 이면 P3

  * $k = \lfloor (n+1)/2 \rfloor$ 이면 (즉, 리스트의 중간) 이진 탐색이라 부른다.

프로그램(lec4-1)

```
int BinSrch(int a[], int i, int l, int x)
// Given an array a[i:l] of elements in nondecreasing
// order, 0<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return -1.
{
    if (l==i) { // If Small(P)
            if (x==a[i]) return i;
            else return -1;
    }
    else { // Reduce P into a smaller subproblem.
            int mid = (i+l)/2;
            if (x == a[mid]) return mid;
            else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
            else return BinSrch(a,mid+1,l,x);
    }
}
```

- 이진 탐색에서는 combine 이 필요없다.

- 예제

  −15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

1) x = 151

low	high	mid
0	13	6
7	13	10
11	13	12
13	13	13 (found)

2) x = −14

low	high	mid
0	13	6
0	5	2
0	1	0
1	1	1

not found

3) x = 9

low	high	mid
0	13	6
0	5	2
3	5	4 (found)

- 정리: 함수 BinSrch(a, 0, n−1, x)는 정확히 동작한다.

이진탐색의 순환관계식

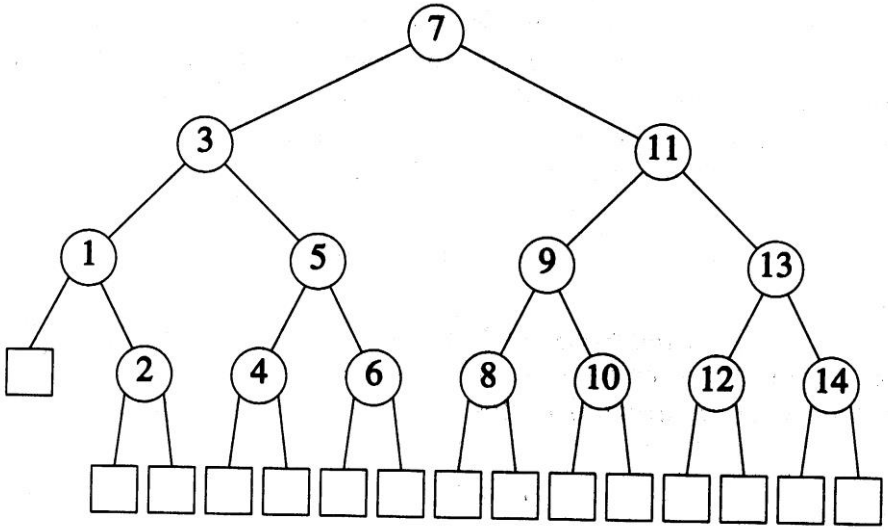
- n 이 2의 멱수일 경우

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

이때, a = 1, b = 2, f(n) = c
치환법에 의해, $T(n) = T(n/2^i) + c \cdot \log_2 n$ $(1 \le i \le \log_2 n)$
따라서, T(n) = O(log n) 이다.

이진 탐색의 결정 트리

- n = 14 일 때



- 시간 복잡도 O(log n)

반복적 구조를 갖는 이진 탐색(lec4-2)

```
int BinSearch(int a[], int n, int x)
// Given an array a[0:n-1] of elements in nondecreasing
// order, n>=0, determine whether x is present, and
// if so, return j such that x == a[j]; else return -1.
{
    int low = 0, high = n-1;
    while (low <= high){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid - 1;
        else if (x > a[mid]) low = mid + 1;
        else return mid;
    }
    return -1;
}
```

반복당 한번의 비교 사용(lec4-3)

```
Int BinSearch1(int a[], int n, int x)
// Same specifications as BinSearch except n > 0
{
   int low=0, high=n;
   // high is one more than possible.
   while (low < (high-1)) {
            int mid = (low + high)/2;
            if (x < a[mid]) high = mid; // Only one comparison
                                        // in the loop
            else low = mid; // x >= a[mid]
    }
    if (x == a[low]) return(low); // x is present.
    else return -1; // x is not present.
}
```

- BinSearch는 간혹 BinSearch1보다 두 배의 원소 비교를 한다 (예를 들어, x > a[n] 인 경우).
- 성공적인 탐색에 대해서 BinSearch1은 BinSearch보다 원소 비교를 (log n)/2 더 많이 한다(예를 들어, x == a[mid]인 경우).

	n	5000	10000	15000	20000	25000	30000
성공	Binsearch	51.30	67.95	67.72	73.85	76.77	73.40
	Binsearch1	47.68	53.92	61.98	67.46	68.95	71.11
실패	Binsearch	50.40	66.36	76.78	79.54	78.20	81.15
	Binsearch1	41.93	52.65	63.33	66.86	69.22	72.26

2.2 합병 정렬

- 최악의 경우에 대한 복잡도는 O(n log n)이다.
- n 개의 원소(키)를 갖는 리스트 (a[1], a[2], ... a[n])를 두 개의 부분 리스트 (a[1], ..., a[⌊n/2⌋])와 (a[⌊n/2⌋+1], ..., a[n])로 분할하여 각각을 정렬한다. 그 결과인 정렬된 부분 리스트를 합병하여 전체 n 개의 원소를 갖는 정렬된 리스트로 만든다.
- 각각의 부분 리스트에 대해 이러한 분할과 합병을 반복한다.
- 리스트에 한 개의 원소만 존재할 때 Small(P) 가 true로 되며, 이 경우 이미 리스트는 정렬된 것으로 생각할 수 있으므로 바로 return 한다.

프로그램(lec4-4)

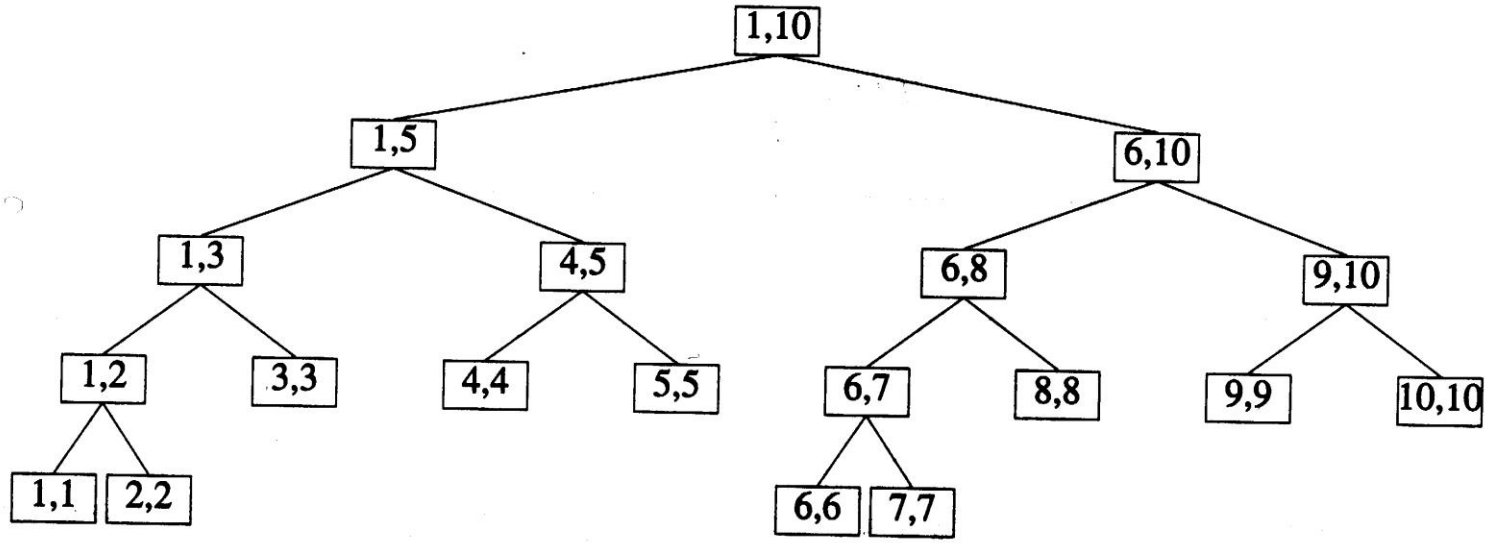
```
void MergeSort(int low, int high)
  // a[low : high] is a global array to be sorted.
  // Small(P) is true if there is only one element to
  // sort. In this case the list is already sorted.
  {
     if (low < high) { // If there are more than one element
        // Divide P into subproblems.
          // Find where to split the set.
            int mid = (low + high)/2;
        // Solve the subproblems.
          MergeSort(low, mid);
          MergeSort(mid + 1, high);
        // Combine the solutions.
          Merge(low, mid, high);
     }
  }
```

보조 기억장소를 사용하는 합병(lec4-4)
(프로그램에서 배열 b[]가 보조 기억장소)

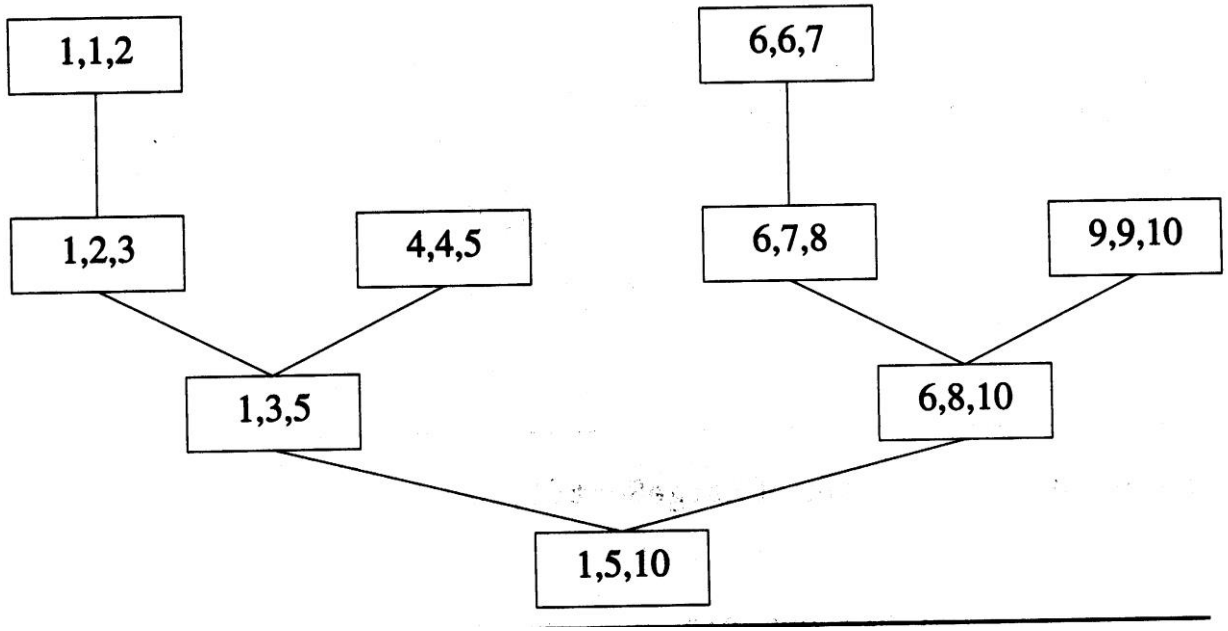
```
void Merge(int low, int mid, int high)
  // a[low:high] is a global array containing two sorted
  // subsets in a[low:mid] and in a[mid+1:high]. The goal
  // is to merge these two sets into a single set residing
  // in a[low:high]. b[] is an auxiliary global array.
  {
     int h = low, i = low, j = mid+1, k;
     while ((h <= mid) && (j <= high)) {
        if (a[h] <= a[j]) { b[i] = a[h]; h++; }
        else { b[i] = a[j]; j++; } i++;
     }
     if (h > mid) for (k=j; k<=high; k++) {
                     b[i] = a[k]; i++;
                  }
     else for (k=h; k<=mid; k++) {
             b[i] = a[k]; i++;
          }
     for (k=low; k<=high; k++) a[k] = b[k];
  }
```

예 제

- 10개의 원소를 갖는 배열 a[1:10]
  (310 285 179 652 351 423 861 254 450 520)
- 호출 과정



Merge의 호출에 대한 트리



정렬 과정

(310 | 285 | 179 | 652 351 | 423 861 254 450 520)
(285 310 | 179 | 652 351 | 423 861 254 450 520)
(179 285 310 | 652 351 | 423 861 254 450 520)
(179 285 310 | 351 652 | 423 861 254 450 520)
(179 285 310 351 652 | 423 861 254 450 520)
(179 285 310 351 652 | 423 | 861 | 254 | 450 520)
(179 285 310 351 652 | 423 861 | 254 | 450 520)
(179 285 310 351 652 | 254 423 861 | 450 520)
(179 285 310 351 652 | 254 423 450 520 861)
(179 254 285 310 351 423 450 520 652 861)

MergeSort의 분석

- MergeSort의 순환 관계식:

$$T(n) = \begin{cases} a & n = 1, \text{ 상수 } a \\ 2T(n/2) + cn & n > 1, \text{ 상수 } c \end{cases}$$

- n0이 2의 멱수일 경우, $n = 2^k$가 되는 양수 k가 존재한다.

$T(n) = 2T(n/2) + cn$

$= 2(2T(n/4) + cn/2) + cn$

$= 4T(n/4) + 2cn$

$= 4(2T(n/8) + cn/4) + 2cn$

…

$= 2^kT(1) + kcn$

$= an + cn \log n$

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

개선 사항

- 비효율적인 부분들을 개선하며, 시간 복잡도는 여전히 O(n log n)이다.
- 문제점 :

  보조 기억장소 b[]를 사용하여 합병하므로 Merge의 각 호출에서 b[low:high]에 저장된 결과를 a[low:high]로 다시 복사해야 한다.

  → 키를 포함하는 레코드의 크기가 큰 경우 많은 시간을 소요
- 개선점 :

  해당 레코드를 가리키는 링크 값을 사용하여 레코드를 이동시키지 않고 링크 값만을 이동한다. 일반적으로 링크 값을 저장하는 필드는 전체 레코드에 비해 크기가 작으므로 이동 시간을 줄일 수 있을 뿐만 아니라 좀더 적은 공간을 사용한다.

- 예:

a:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	-	50	10	25	30	15	70	35	55
link:		0	3	4	1	7	0	8	6

  하나의 리스트 시작 q = 2, 이것은 (10, 25, 30, 50) 을 의미하고,
  다른 리스트 시작 r = 5, 이것은 (15, 35, 55, 70) 을 의미한다.
  합병 후,

link:	8	5	4	7	3	0	1	6

  합병 리스트의 시작 p = 2, 이것은 (10, 15, 25, 30, 35, 50, 55, 70)을 의미한다.
- 여기서 링크 값은 변수의 주소가 아니라 원소를 저장하고 있는 배열의 첨자값이다.
- 링크 값이 0인 경우 이것은 리스트의 끝을 의미한다.

개선된 합병 정렬(lec4-5)

```
int MergeSort1(int low, int high)
  // The global array a[low : high] is sorted in
  // nondecreasing order using the auxiliary array
  // link[low:high]. The values in link  will
  // represent a list of the indices low through
  // high giving a[] in sorted order. A pointer
  // to the beginning of the list is returned.
  {
       if (low < high)
       {
            int mid = (low + high)/2;
            int q = MergeSort1(low, mid);
            int r = MergeSort1(mid+1, high);
            return(Merge1(q,r));
       }
       else  return low;
  }
```

링크를 사용하는 합병(lec4-5)

```
int Merge1(int q, int r)
  // The lists q and r are merged and
  // a pointer to the beginning of the merged list is returned.
  {
     int i=q, j=r, k=0;
     // The new list starts at link[0].  → p 에 해당함.
     while (i != 0 && j != 0) { // While both lists are nonempty do
        if (a[i] <= a[j]) { // Find the smaller key.
           link[k] = i; k = i; i = link[i]; // Add a new key to the list.
        }
        else {
           link[k] = j; k = j; j = link[j];
        }
     }
     if (i == 0) link[k] = j;
     else link[k] = i;
     return(link[0]);
  }
```

예 제

a:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
	–	50	10	25	30	15	70	35	55	
link:	0	0	0	0	0	0	0	0	0	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10,50)
3 4 3	3	0	1	4	0	0	0	0	0	(10,50), (25,30)
2 3 2	2	0	3	4	1	0	0	0	0	(10,25,30,50)
5 6 5	5	0	3	4	1	6	0	0	0	(10,25,30,50)(15,70)
7 8 7	7	0	3	4	1	6	0	8	0	(10,25,30,50)(15,70)(35,55)
5 7 5	5	0	3	4	1	7	0	8	6	(10,25,30,50)(15,35,55,70)
2 5 2	2	8	5	4	7	3	0	1	6	(10,15,25,30,35,50,55,70)