CHAPTER 7

SORTING

Contents

배열 → 찾고자 하는 key 값

```
int seqSearch(element a[], int k, int n)
{/* search a[l:n]; return the least i such that  a[i].key = k; return 0,
    if k is not in the array*/
  int i;
  for (i = 1; i <= n && a[i].key != k; i++)
          ;
  if (i>n)  return 0;
  return i; → 있으면 index를 반환
}
```

Program 7.1 Sequential search

```
typedef struct {
        int key;
        /* other fields */
} element;
```

Unsuccessful search:

O(...)

Average successful search:

$$\left(\sum_{1 \le i \le n} i \right) / n = (n + 1)/2$$

for문이 세번 되니까..

- Binary Search Sorting이 되어있어야함
  - Assumption: …

n 개의 데이터

$\Rightarrow T(n) = k+1$

$n \times (\frac{1}{2})^k = 1 \Rightarrow k = \log_2 n$

```
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    while (left <= right) {
        middle = (left + right) / 2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1
                     break;
            case 0 : return middle;
            case 1 : right = middle -1;}
    }
    return -1;
}
```

$\therefore T(n) = \log_2 n + 1$

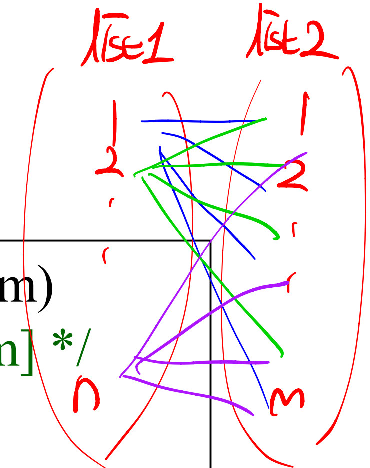
$O(\log n)$

  - Time complexity: O(…)

- List Verification 같은 내용이지만 다른 데이터소스에서 들고옴
  - Verify two lists of records that there is no discrepancy between the two
    - Two lists are essentially the same but have been obtained from two different sources
    - Ex) International Revenue Service (IRS)
      (e.g., employee vs. employer)

정렬X

- Function *verifyl* 두 list가 같은지 확인
  - directly compare the two unsorted lists




```
void verify1(element list1[], element list2[], int n, int m)
{/* compare two unordered lists listl[1:n] and list2[l:m] */
    int i,j, marked[MAX_SIZE];

    for(i=1; i<=m; i++) marked[i] = FALSE;
    for(i=1; i<=n; i++)     {
        if((j == seqSearch(list2, m, list1[i].key)) == 0)
            printf("%d is not in list2 \n", list1[i].key);
        else
            marked[j] = TRUE;
    }
    for(i=1; i<=m; i++)
        if(!marked[i]) printf("%d is not in list1 \n", list2[i].key);
}
```

⇒ $O(n^2)$

→ seqSearch(list2, list1[i].key, m)

$\frac{m+1}{2}$

1한번 수 출력
2한번 수 출력

Time Complexity:

… $O(n^2)$

Program 7.2: Verifying two unsorted lists using a sequential search

정렬O

- Function *verify2*
  - Sort the two lists and then do the comparison

```
void verify2(element list1[], element list2[], int n, int m)
{/*same as verify1, but we sort list1 and list2 first
    int i, j;
    sort(list1, n);  sort(list2, m); i=j=1;
    while( (i<=n) && (j<=m) )
        if (list1[i].key < list2[j].key) {
            printf(“%d is not int list2 \n”, list1[i].key);    i++;
        }
        else if( list1[i].key==list2[j].key ) {
            i++; j++;
        }
        else {
            printf(“%d is not int list1 \n”, list2[i].key);    j++;
        }
    for(; i<=n; i++)        printf(“%d is not in list2 \n”, list1[i].key);
    for(; j<=m; j++)        printf(“%d is not in list1 \n”, list2[j].key);
}
```

Program 7.3: Fast verification of two sorted lists

list1: 1 3 4 6 7 9 10

list2: 1 2 5 6 8 10 12

list1: 1 2 3 4 5

i=6

list2: 4 6 8 9 10

j=2

```
void verify2(element list1[], element list2[], int n, int m)
{/*same as verify1, but we sort list1 and list2 first
    int i, j;
    sort(list1, n);  sort(list2, m); i=j=1;
    while( (i<=n) && (j<=m) )
        if (list1[i].key < list2[j].key) {
            printf(“%d is not int list2 \n”, list1[i].key);     i++;
        }
        else if( list1[i].key==list2[j].key ) {
           i++; j++;
        }
        else {
            printf(“%d is not int list1 \n”, list2[i].key);     j++;
        }
   for(; i<=n; i++)          printf(“%d is not in list2 \n”, list1[i].key);
   for(; j<=m; j++)          printf(“%d is not in list1 \n”, list2[j].key);
}
```

Program 7.3: Fast verification of two sorted lists

Time Complexity:

$\rightarrow O(t_{sort}(n) + t_{sort}(m) + \underbrace{n + m}_{\text{두 개의 list가 disjoint 할 때}})$

$\rightarrow O(\max\{n\log n, m\log m\})$

Sorting

- Rearrange *n* elements into ascending order
  - 7, 3, 6, 2, 1 → 1, 2, 3, 6, 7
- Uses of sorting
  - An aid in searching
  - A means for matching entries in lists (comparing two lists)
- If the list is sorted
  - The searching time could be reduced from O(*n*) to O(log *n*)

Terminology

- Record : $R_1, R_2, \ldots, R_n$
  – List of records : $(R_1, R_2, \ldots, R_n)$
- Key value : K_i
- Ordering relation : $<$
  – for any three values *x, y,* and *z*
  $x < y$ and $y < z \Rightarrow x < z$
- Sorting Problem
  – Finding a permutation σ such that
  $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$
  – The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$

☆ 원래 데이터의 순서를 유지

- Stable sorting : σ_s

(1) $K_{\sigma_s(i)} \leq K_{\sigma_s(i+1)}$, $1 \leq i \leq n-1$

(2) If $i < j$ and $K_i == K_j$, R_i precedes R_j in the sorted list

⇒ key가 같은 record가 있다

- ex) Input list : 6, 7, 3, 2_1, 2_2, 8

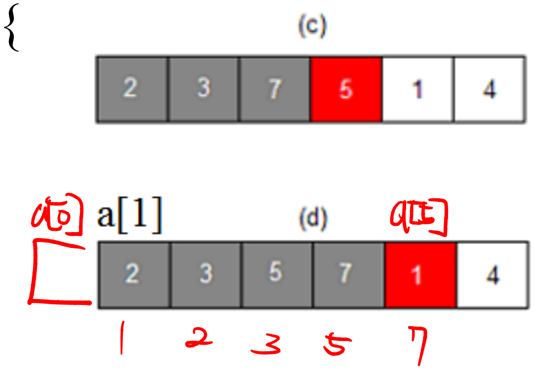
이 순서를 유지

  - Stable sorting : 2_1, 2_2, 3, 6, 7, 8
  - Unstable sorting : 2_2, 2_1, 3, 6, 7, 8

- Sorting methods : internal or external methods
- Internal methods
  - Used when the list to be sorted is small enough so that the entire sort can be carried out in main memory
  - Insertion sort, quick sort, merge sort, heap sort, radix sort Main memory에 올릴 수 있는 적당한 데이터들
- External methods
  - Used on larger lists → 데이터가 어마어마하게 많을때
  - Merge sort

7.2 INSERTION SORT

```
void insert(element e, element a[], int i)
{/* insert e into a[1:i]
     such that a[1:i+1] is also ordered */
   a[0] = e;
   while(e.key <a[i].key)     {
       a[i+1] = a[i];
       i--;
   }
   a[i+1] = e;
}
```

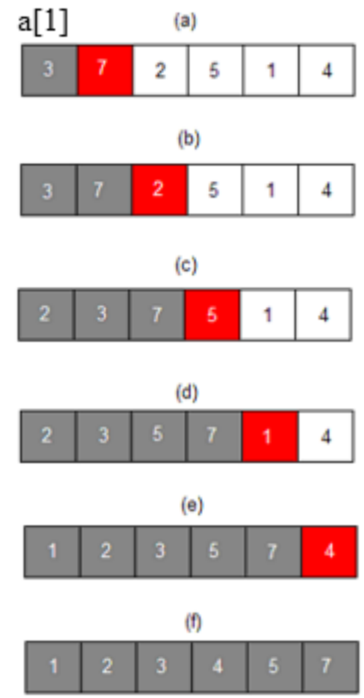


Program 7.4: Insertion into a sorted list

The complexity of *Insert*:
O(..)

```
void insertionSort(element a[], int n)
{/* sort a[1:n] into nondecreasing order */
    int j;
    for(j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
```

Program 7.5: Insertion sort



- Write the status of the list (12, 2, 16, 30, 8, 28, 4) at the end of each iteration of the for loop of *insertionSort*

```
void insertionSort(element a[], int n)
{/* sort a[l:n] into nondecreasing order */
    int j;
    for(j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
```

- Analysis of *insertionSort:*
  - The complexity of *insert*: O(i)
  - The complexity of *insertionSort:*

$$O\left(\sum_{i=1}^{n-1}(i+1)\right) = O(n^2).$$

- We can also obtain an estimate of the computing time of insertion sort
  - based on the relative disorder in the input list

정렬해야 하는 데이터

- R_i is *left out of order* (LOO) iff $R_\mathrm{i} < \max\limits_{1 \le \mathrm{j} < \mathrm{i}} \{R_\mathrm{j}\}$

* ⇒ 정렬되어야 하는 친구들

44	55	12	42	94	18	06	67
		*	*		*	*	*

- k : # of LOO records

  → computing time: O(kn)

- Ex 7.1
  - $n = 5$, input key: 5, 4, 3, 2, 1 최악의 경우
  - LOO Records: R_2, R_3, R_4, R_5

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

$$O\left(\sum_{i=1}^{n-1}(i+1)\right) = O(n^2).$$

- Ex 7.2
  - $n = 5$, input key: $\overset{j=1}{\overset{\downarrow}{2}}$, 3, 4, 5, $\overset{j=5}{\overset{\downarrow}{1}}$
  - Only R_5 is LOO;
  - j = 2, 3, 4 → O(1)
  - j = 5 → O(n)

j	[1]	[2]	[3]	[4]	[5]
–	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

- Insertion Sorting
  - Stable
  - Desirable in sorting sequences in which only a very few records are LOO ⇒ LOO가 적으면 efficient 하게 동작함
  - Fastest sorting method for small *n* ($n \le 30$)

7.3 QUICK SORT

- Algorithm : Quick Sorting
  - 1) Select a record, called pivot 기준점
  - 2) Reorder the records
    - the left of the pivot are less than(or equal) to that of the pivot
    - the right of the pivot are greater than(or equal) to that of the pivot
  - 3) Use the quick sort method recursively for sublists
    - the records to the left of the pivot and those to its right are sorted independently

- Ex 7.3)
  - A list of 10 of records with keys:
    (26, 5, 37, 1, 61, 11,59, 15, 48, 19)

26 5 19 1 61 11 59 15 48 37

26 5 19 1, 15 11 59 61 48 37

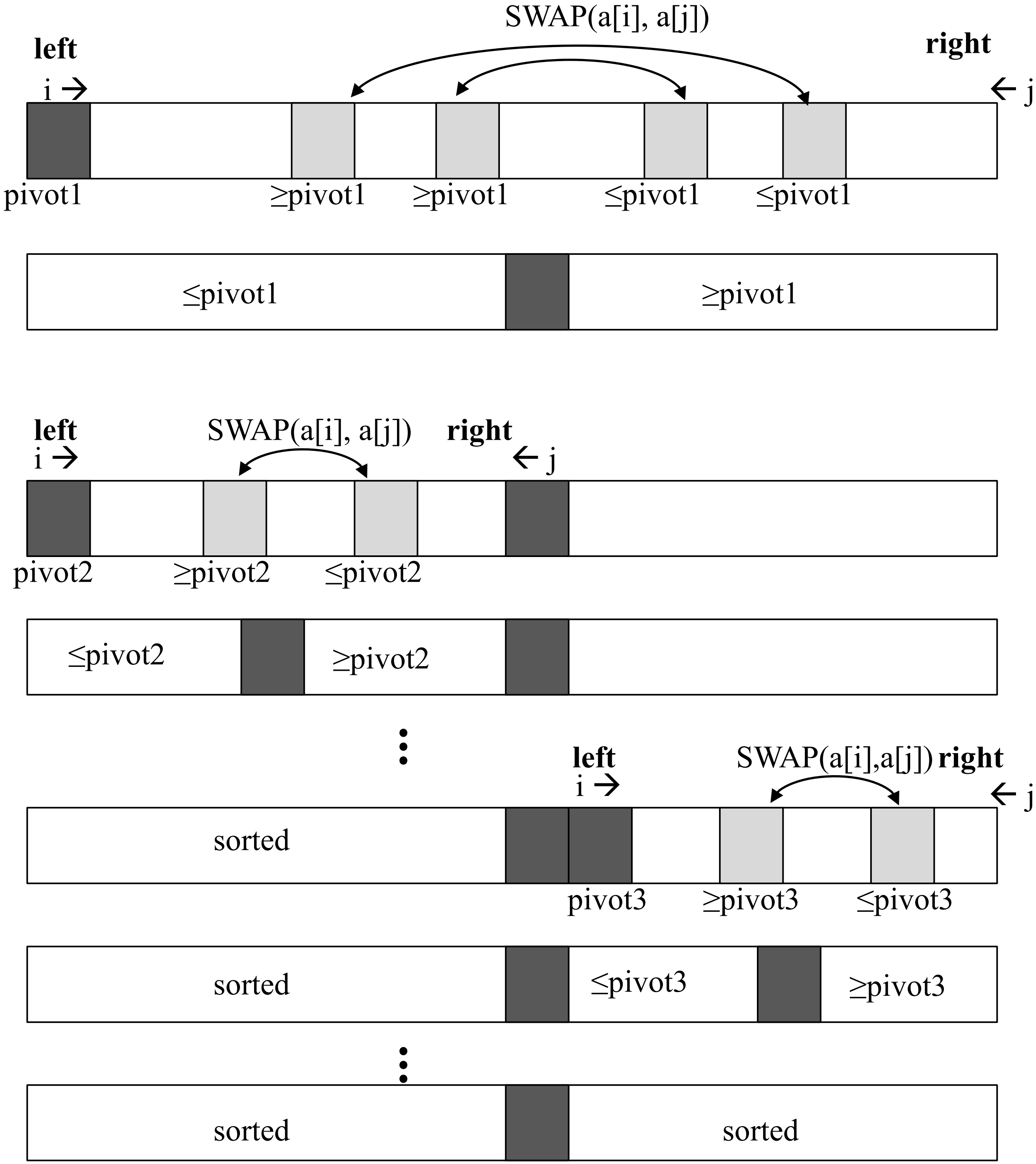
	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
pivot	[26	5	37	1	61	11	59	15	48	19]	1	10
	[11	5	19	1	15]	26	[59	61	48	37]	1	5
	[1	5]	11	[19	15]	26	[59	61	48	37	1	2
	1	5	11	[19	15]	26	[59	61	48	37]	4	5
	1	5	11	15	19	26	[59	61	48	37]	7	10
	1	5	11	15	19	26	[48	37]	59	[61]	7	8
	1	5	11	15	19	26	37	48	59	[61]	10	10
	1	5	11	15	19	26	37	48	59	61		

Figure 7.1: Quick sort example

(26, 5, 37, 1, 61, 11,59, 15, 48, 19)

```
                                           1      10
void quickSort(element a[], int left, int right)
{
      int pivot, i, j; element temp;
             1     10
      if(left < right) {
         i = left; j = right +1;       i=1, j=11
         pivot = a[left].key;      pivot=26
         do{
            do i++;     while(a[i].key < pivot);     i=5
            do j--;     while(a[j].key > pivot);     j=9
            if(i < j)   SWAP(a[i], a[j], temp);
          } while(i < j);
          SWAP(a[left], a[j],temp);
          quickSort(a, left, j-1);
          quickSort(a, j+1, right);
      }
}
```

Program 7.6: Quick sort




```
i=left; j = right+1
pivot=a[left].key;
do{
    do i++; while(a[i].key<pivot);
    do j--; while(a[j].key>pivot);
    if(i<j) SWAP(a[i], a[j]);
}while( i<j );
SWAP(a[left], a[j]);
quickSort(a, left, j-1);
quickSort(a, j+1, right);
```

- Draw a figure similar to Fig 7.1 starting with the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) i: j:

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
[illegible]	5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

pivot= [illegible]

```
void quicksort(element list[], int left, int right)
{
    int pivot,i,j;
    element temp;
    if (left < right) {
        i = left;      j = right + 1;
        pivot = list[left].key;
        do {
            do
               i++;
            while (list[i].key < pivot);
            do
               j--;
            while (list[j].key > pivot);
            if (i < j)
               SWAP(list[i],list[j],temp);
        } while (i < j);
        SWAP(list[left],list[j],temp);
        quicksort(list,left,j-1);
        quicksort(list,j+1,right);
    }
}
```

Q: How many function calls for [1 5] ? 3번

```
void quickSort(element a[], int left, int right)
{
    int pivot, i, j; element temp;
    if(left < right) {
        i = left; j = right +1;
        pivot = a[left].key;
        do{
            do i++;     while(a[i].key < pivot);
            do j--;     while(a[j].key > pivot);
            if(i < j)   SWAP(a[i], a[j], temp);
        } while(i < j);
        SWAP(a[left], a[j],temp);
        quickSort(a, left, j-1);
        quickSort(a, j+1, right);
    }
}
```

Program 7.6: Quick sort

- Analysis of quickSort
  - Worst case : $O(n^2)$
  - Optimal case : $O(n\log n)$ → pivot이 적절한 중간값을 가질때
    - $T(n)$: the time taken to sort a list of n records

$$
\begin{aligned}
T(n) &\le cn + 2T(n/2) \qquad \text{c: constant} \\
&\le cn + 2(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\;\;\vdots \\
&\le cn\log_2 n + nT(1) = O(n\log n)
\end{aligned}
$$

  - Unstable sorting (insertion sort는 stable sort)
  - Average computing time: $O(n\log n)$
  - The best of the internal sorting methods

- Stack Space:
  - Average/best case: $O(\log n)$
  - Worst case: $O(n)$
- Variations
  - A better choice for the pivot ✓
    - The median of the first, middle, and last keys in the current sublist;
    - pivot = median$\{K_l, K_{(l+r)/2}, K_r\}$
  - Ex)
    - median$\{10, 5, 7\} = 7$
    - median$\{10, 7, 7\} = 7$

Contents