Chap 7. Sorting (1)

Contents

7.1 Motivation

- Sorting
  - Rearrange n elements into ascending order
    - 7, 3, 6, 2, 1 → 1, 2, 3, 6, 7

- Two important 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 $O(\log_2 n)$

- Sequential Search

```
int seqSearch(element a[], int k, int n)
{/* search a[1: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;
}
```

Program 7.1 Sequential search

- time complexity
  - worst case: $O(n)$
  - average number of comparisons for a successful search:

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

- Binary Search
  - Assumption: *list*[0].*key* ≤ *list*[1].*key* ≤…≤*list*[*n*-1].*key*

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

  - time complexity: O(log *n*)

Terminology

- Record : $R_1, R_2, \ldots, R_n$
  - A list of records : $(R_1, R_2, \ldots, R_n)$
- R_i has key value K_i
- Ordering relation(<)
  - Transitive relation : $x < y$ and $y < z \Rightarrow x < z$

- ***Sorting Problem*** : finding a permutation σ such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n\text{-}1$
  - the desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$

Terminology

- ***Stable Sorting*** : σ_s

  (1) $K_{\sigma_S(i)} \le K_{\sigma_S(i+1)}$, $1 \le i \le n$-1

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

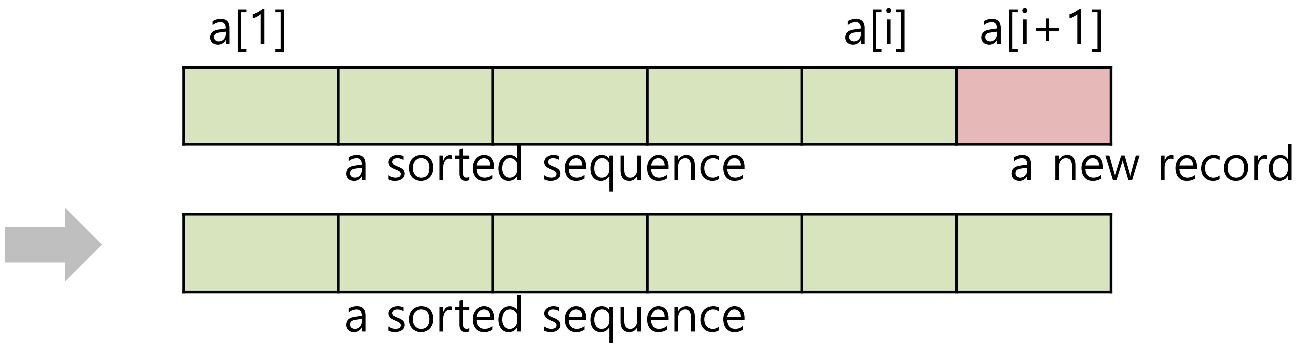
ex) input list : 6, 7, 3, 2_1, 2_2, 8

- stable sorting : 2_1, 2_2, 3, 6, 7, 8
- unstable sorting : 2_2, 2_1, 3, 6, 7, 8

- ***Internal Sorting*** (c.f. external sorting)- the list is small enough to sort entirely in main memory
  - insertion sort
  - quick sort
  - heap sort
  - merge sort
  - radix sort

7.2 Insertion Sort

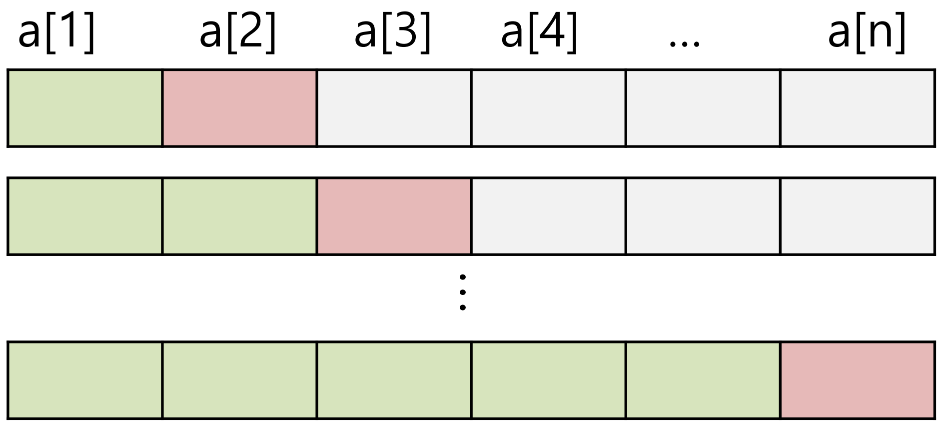
- Basic Step




```
void insert(element e, element a[], int i)
{/* insert e into the ordered list a[1:i] such that the
    resulting list a[1:i+1] is also ordered, the array a
    must have space allocated for at least i+2 elements */
   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

- Insertion Sort




```
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

- **Analysis of *insertionSort:***

< Method 1 >

- Worst case time
  - *insert*(e, a, i) $\Rightarrow$ i+1 comparisons
  - *InsertionSort*(a, n) invokes *insert* for $i = j$-1 = 1, 2, …, n-1
  - O($\sum_{i=1}^{n-1}(i+1)$) = O(n^2)

<Method 2>

- Record R_i is *left out of order*(LOO)

  iff $R_i < \max_{1 \le j < i}\{R_j\}$
- The insertion step is executed only for those records that are LOO
- if number of LOOs = k,
  - computing time : $O(kn)$
  - worst case time : $O(n^2)$

- Example 7.1
  - n = 5
  - input key (5, 4, 3, 2, 1)
  - records R_2, R_3, R_4, R_5 are LOO

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**

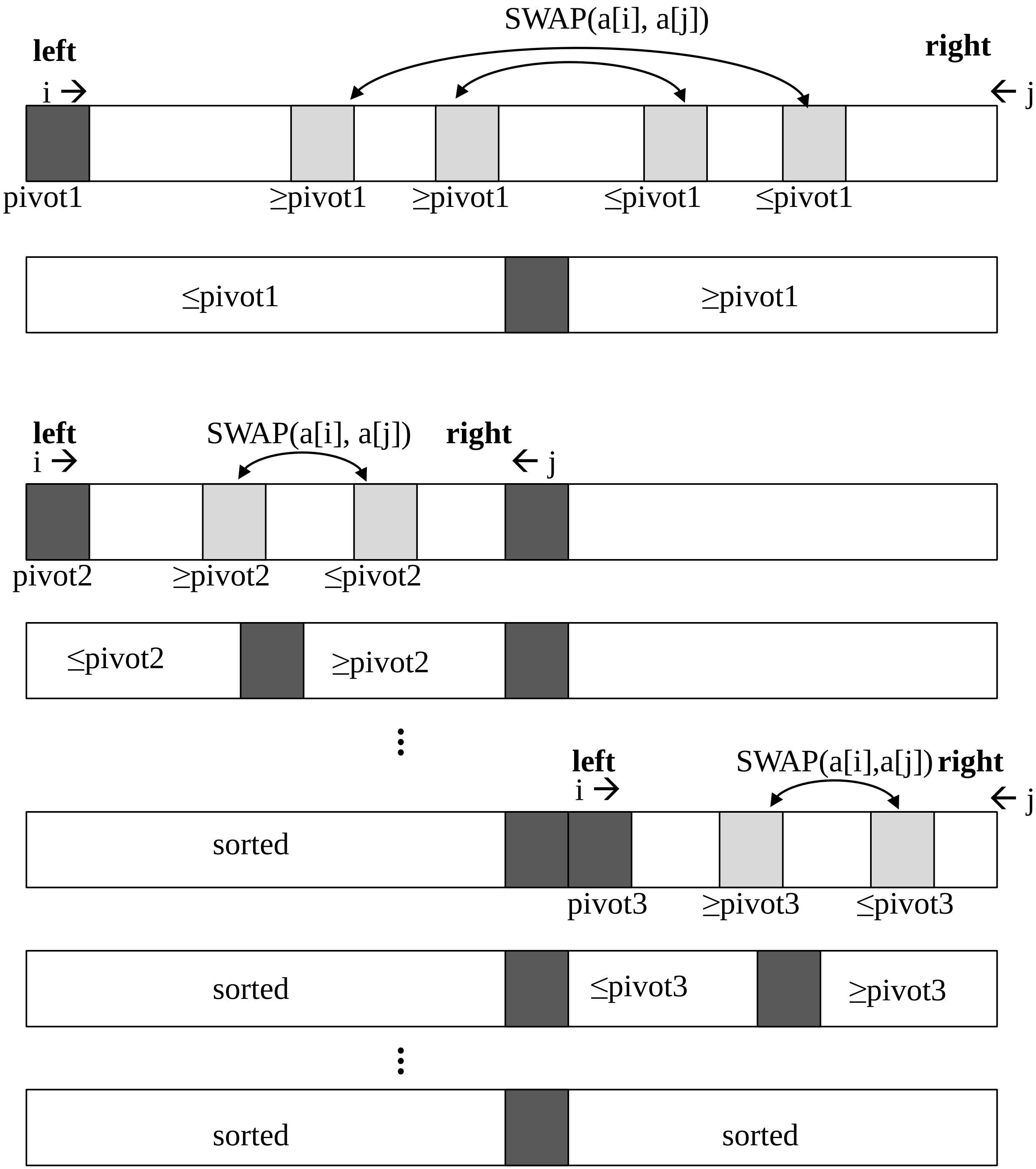
- Example 7.2
  - n = 5
  - input key (2, 3, 4, 5, 1)
  - only R_5 is LOO

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**

- O(kn) makes this method very desirable in sorting sequences in which only a very few records are LOO(i.e., $k << n$).
- *Stable sorting* method
- Useful for small size sorting ($n \leq 30$)

7.3 Quick Sort

- Divide and conquer
  - two phase
  - split and control
- Use *recursion* : stack is needed
- Best average time : $O(n \cdot \log_2 n)$




```
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]);
```

```
void quickSort(element a[], int left, int right)
{/* sort a[left:right] into nondecreasing order
    on the key field; a[left].key is arbitrarily
    chosen as the pivot key;  it is assumed that
    a[left].key <= a[right+1].key */
   int pivot,i,j;
   element temp;
   if (left < right) {
     i = left; j = right + 1;
     pivot = a[left].key;
     do {/* search for keys from the left and right
            sublists, swapping out-of-order elements until
            the left and right boundaries cross or meet */
        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

- Example 7.3

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[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

- Analysis
  - Worst case : $O(n^2)$
    - in the case of sorted input
  - Optimal case :

$$
\begin{aligned}
T(n) &\le cn + 2T(n/2) \\
&\le cn + 2(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\quad\vdots \\
&\le cn\log_2 n + nT(1) = O(n\log n)
\end{aligned}
$$

  - *unstable sorting*
  - good(best) sorting method
    - average computing time is $O(n\log n)$