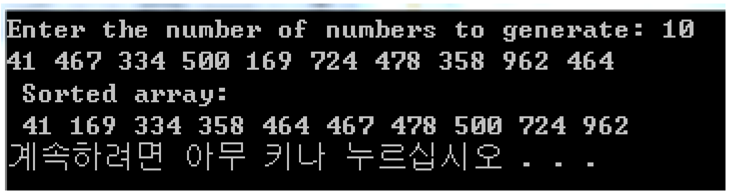
Chap 7. Sorting

Contents

Selection Sort ← Chapter 1.

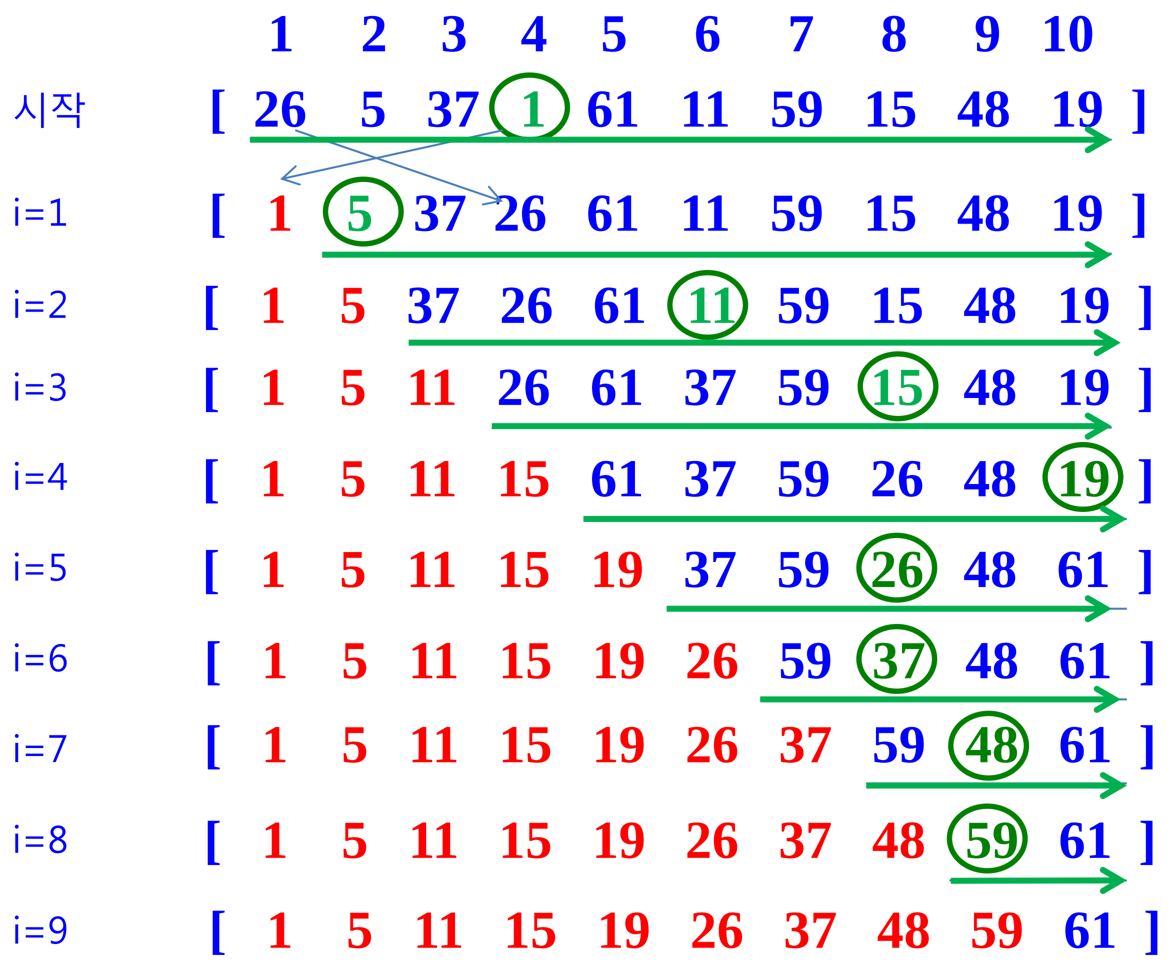
```c
// Program 1.4: Selection sort
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ( (t) = (x), (x) = (y), (y) = (t))
void selectionSort(int [],int);
void main(void)
{
    int i,n;
    int a[MAX_SIZE];
    printf("Enter the number of numbers to generate: ");
    scanf ("%d", &n);
    if( (n < 1) || (n >MAX_SIZE)) {
        fprintf(stderr, "Improper value of n\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < n; i++) {/*randomly generate numbers*/
        a[i] =rand() % 1000;
        printf("%d ", a[i]);
    }
    selectionSort(a, n);
    printf("\n Sorted array:\n ");
    for (i = 0; i < n; i++) /* print out sorted numbers */
        printf("%d ", a[i]);
    printf ( "\n");
}
```

```c
/*selection sort*/
void selectionSort(int a[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n-1; i++) {
        min = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[min]) min = j;
        if (min != i)
            SWAP(a[i],a[min],temp);
    }
}
```



```
Enter the number of numbers to generate: 10
41 467 334 500 169 724 478 358 962 464
 Sorted array:
 41 169 334 358 464 467 478 500 724 962
계속하려면 아무 키나 누르십시오 . . .
```

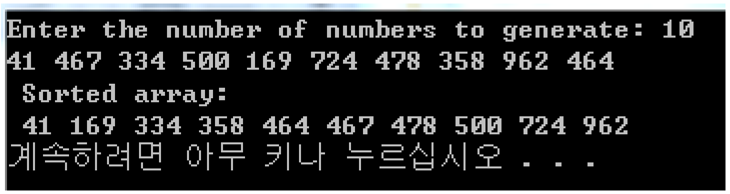
Selection Sort



		1	2	3	4	5	6	7	8	9	10	
시작	[	26	5	37	$\boxed{1}$	61	11	59	15	48	19	]
i=1	[	1	$\boxed{5}$	37	26	61	11	59	15	48	19	]
i=2	[	1	5	37	26	61	$\boxed{11}$	59	15	48	19	]
i=3	[	1	5	11	26	61	37	59	$\boxed{15}$	48	19	]
i=4	[	1	5	11	15	61	37	59	26	48	$\boxed{19}$	]
i=5	[	1	5	11	15	19	37	59	$\boxed{26}$	48	61	]
i=6	[	1	5	11	15	19	26	59	$\boxed{37}$	48	61	]
i=7	[	1	5	11	15	19	26	37	59	$\boxed{48}$	61	]
i=8	[	1	5	11	15	19	26	37	48	$\boxed{59}$	61	]
i=9	[	1	5	11	15	19	26	37	48	59	61	]

```c
// Program : Bubble sort
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ( (t) = (x), (x) = (y), (y) = (t))
void selectionSort(int [],int);
void main(void)
{
    int i,n;
    int a[MAX_SIZE];
    printf("Enter the number of numbers to generate: ");
    scanf ("%d", &n);
    if( (n < 1) || (n >MAX_SIZE)) {
        fprintf(stderr, "Improper value of n\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < n; i++) {/*randomly generate numbers*/
        a[i] =rand() % 1000;
        printf("%d ", a[i]);
    }
    bubbleSort(a, n);
    printf("\n Sorted array:\n ");
    for (i = 0; i < n; i++) /* print out sorted numbers */
        printf("%d ", a[i]);
    printf ( "\n");
}
```

```c
/* bubble sort */
void bubbleSort(int a[], int n)
{
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < (n - i) - 1; j++){
            if (a[j] > a[j + 1])
                SWAP(a[j],a[j+1],temp);
        }
    }
}
```



```
Enter the number of numbers to generate: 10
41 467 334 500 169 724 478 358 962 464
 Sorted array:
 41 169 334 358 464 467 478 500 724 962
계속하려면 아무 키나 누르십시오 . . .
```

Bubble Sort

		1	2	3	4	5	6	7	8	9	10	
시작	[	26	5	37	1	61	11	59	15	48	19	]
i=0	[	5	26	1	37	11	59	15	48	19	61	]
i=1	[	5	1	26	11	37	15	48	19	59	61	]
i=2	[	1	5	11	26	15	37	19	48	59	61	]
i=3	[	1	5	11	15	26	19	37	48	59	61	]
i=4	[	1	5	11	15	19	26	37	48	59	61	]
i=5	[	1	5	11	15	19	26	37	48	59	61	]
i=6	[	1	5	11	15	19	26	37	48	59	61	]
i=7	[	1	5	11	15	19	26	37	48	59	61	]
i=8	[	1	5	11	15	19	26	37	48	59	61	]

7.1 Motivation

In this chapter, we use the term ***list*** to mean a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as ***keys***.

One way to search for a record with the specified key is to examine the list of records in left-to-right or right-to-left order. Such a search is known as a ***sequential search***.

We assume that ***the list of records*** is stored in positions 1 through *n* of an **array**. We use array indexes 1 through *n* for our records rather than 0 through *n*-1 because one of the sort methods we develop, ***heap sort***, employs the array representation of a heap.

The datatype of each record is ***element*** and each record is assumed to have an integer field ***key***.

Program 7.1 gives a ***sequential search*** function that examines the records in the list *a* [1:*n*] in left-to-right order.

```
int sequentialSearch(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;
}
```
Program 7.1 Sequential search

- time complexity
  - worst case: **O(*n*)**
    - Why? Program 7.1 makes *n* **key comparisons** when the search is unsuccessful.
  - **average** number of comparisons for **a successful search**:

$$\sum_{i=0}^{n-1}(i+1)/n = \mathbf{(n+1)/2}$$

- **Binary search** (see Chapter 1) is one of the better-known methods for **searching an ordered, sequential list**. A binary search takes only **O(log *n*)** time to search a list with *n* records.

- Sorting
  - Rearrange n elements into **ascending order**
    - 7, 3, 6, 2, 1 → 1, 2, 3, 6, 7
- Sorting is used
  - as an aid in searching
  - as a means for matching entries in lists (comparing two lists)
- If the list is sorted, the searching time could be reduced
  - from $\mathbf{O(}\boldsymbol{n}\mathbf{)}$ to $\mathbf{O(\log_2}\boldsymbol{n}\mathbf{)}$

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 : $<$
  - Transitive relation : $x < y$ and $y < z \Rightarrow x < z$
- ***Sorting Problem*** : finding a **permutation** σ such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$
  - the desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$

- ***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

  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

We characterize sorting methods into two broad categories: (1) internal methods (i.e., methods to be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory) and (2) external methods (i.e., methods to be used on larger lists). The following **internal sorting** methods will be developed: insertion sort, quick sort, merge sort, heap sort, and radix sort. This development will be followed by a discussion of **external sorting**. Throughout, we assume that relational operators have been overloaded so that record comparison is done by comparing their keys.

7.2 Insertion Sort

The basic step in this method is to insert a new record into a sorted sequence of *i* records in such a way that the resulting sequence of size *i* + 1 is also ordered. Function *insert* (Program 7.4) accomplishes this insertion.

```
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);
     }
}
```

Program 7.5: Insertion sort

```
void insert(element e, element a[], int i)
{
     /* insert e into the ordered list a[l:i] such that the
     resulting list a[l:i+l] is also ordered, the array a
     must have space allocated for at least i+2 elements */
      a[0] = e;
     while (a[i].key > e.key) {
          a[i+l] = a[i];
          i--;
     }
     a[i+l] = e;
}
```

Program 7.4: Insertion into a sorted list

Insertion Sort

		1	2	3	4	5	6	7	8	9	10	
시작	[	26	5	37	1	61	11	59	15	48	19	]
i=2	[	5	26	37	1	61	11	59	15	48	19	]
i=3	[	5	26	37	1	61	11	59	15	48	19	]
i=4	[	1	5	26	37	61	11	59	15	48	19	]
i=5	[	1	5	26	37	61	11	59	15	48	19	]
i=6	[	1	5	11	26	37	61	59	15	48	19	]
i=7	[	1	5	11	26	37	59	61	15	48	19	]
i=8	[	1	5	11	15	26	37	59	61	48	19	]
i=9	[	1	5	11	15	26	37	48	59	61	19	]
i=10	[	1	5	11	15	19	26	37	48	59	61	]

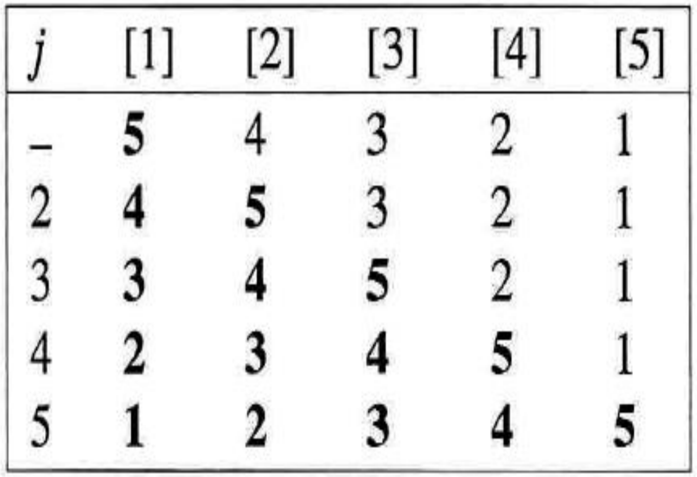
Analysis of *insertionSort*:

- < Method 1 >
  - *insert*(e, a, i) $\Rightarrow$ i+1 comparisons
  - *InsertionSort*(a, n) invokes *insert* for $i = j\text{-}1 = 1, 2, \ldots, n\text{-}1$

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

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

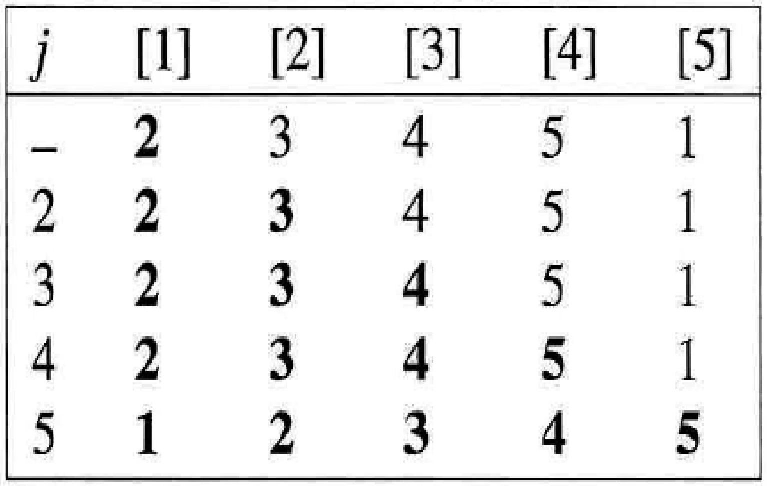
Example 7.1: Assume that *n* = 5 and the input key sequence is 5, 4, 3, 2, l. After each insertion we have



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**

Note that records R_2, R_3, R_4, R_5 are LOO. This input sequence exhibits the worst-case behavior of insertion sort.

Example 7.2: Assume that n = 5 and the input key sequence is 2, 3, 4, 5, l. After each iteration we have



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**

Note that only R_5 is LOO.

It should be fairly obvious that *insertionSort* is **stable**. The fact that the computing time is O(*kn*) makes this method very desirable in sorting sequences in which only a very few records are LOO (i.e., k<<n). The simplicity of this scheme makes it about the fastest sorting method for small *n* (say, n ≤ 30).

Variations

1. ***Binary Insertion Sort***: We can <u>reduce the number of comparisons</u> made in an insertion sort by replacing the sequential searching technique used in *insert* (Program 7.4) with binary search. The number of record moves remains unchanged.
2. ***Linked Insertion Sort***: The elements of the list are represented as a linked list rather than as an array. The number of record moves becomes zero because only the link fields require adjustment. However, we must retain the sequential search used in *insert*.

7.3 Quick Sort

We now tum our attention to a sorting scheme with very good average behavior. The quick sort scheme developed by **C. A. R. Hoare** has the best average behavior among the sorting methods we shall be studying.

devide

In quick sort, we select a **pivot** record from among the records to be sorted. Next, the records to be sorted are reordered so that the keys of records to the left of the pivot are less than or equal to that of the pivot and those of the records to the right of the pivot are greater than or equal to that of the pivot.

Finally, the records to the left of the pivot and those to its right are sorted independently (using the quick sort method recursively).

conquer

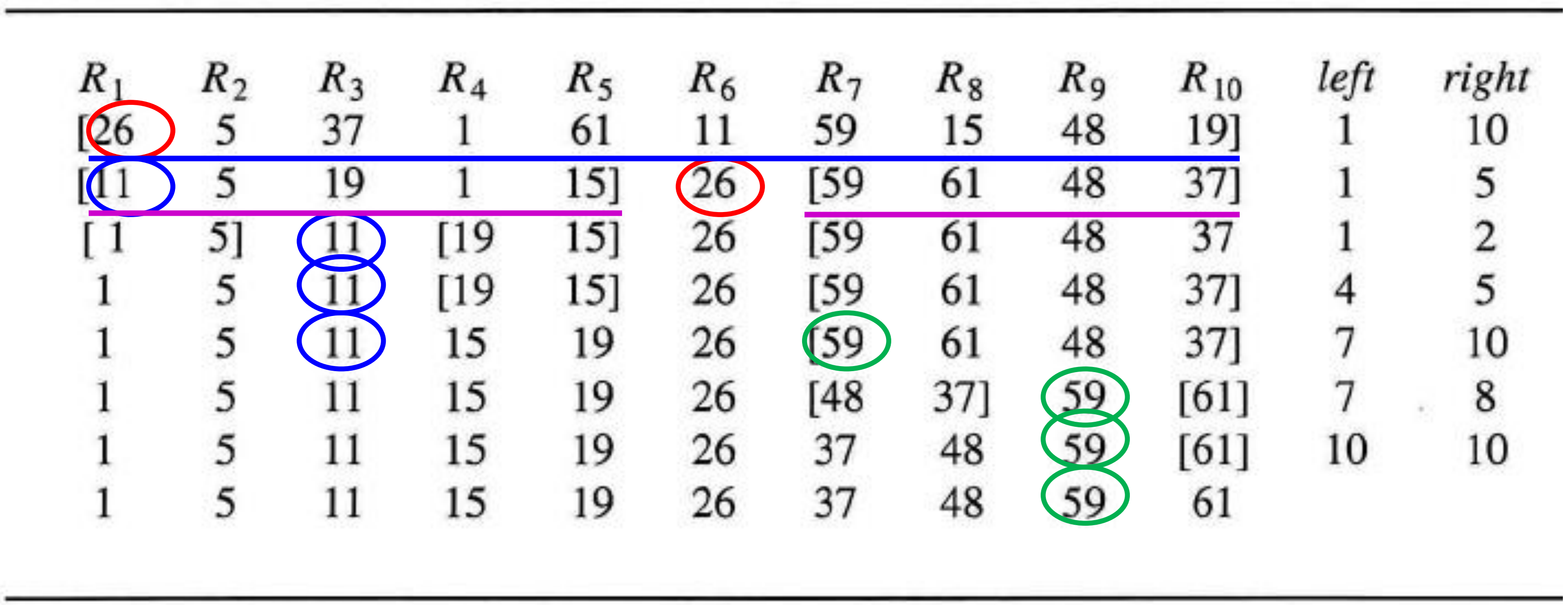
Program 7.6 gives the resulting quick sort function. To sort $a[1:n]$, the function invocation is *quickSort*(a, 1, n). Function *quickSort* assumes that $a[n + 1]$ has been set to have a key at least as large as the remaining keys.

Call *quickSort*(*a*, 1, *n*)

```
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+l].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 (pivot<a[j].key);
               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+l,right);
   }
}
```

Program 7.6: Quick sort

Example 7.3: Suppose we are to sort a list of 10 records with keys (26, 5, 37, 1, 61, 11, 59, 15, 48, 19). Figure 7.1 gives the status of the list at each call of *quickSort*. Square brackets indicate sublists yet to be sorted. □

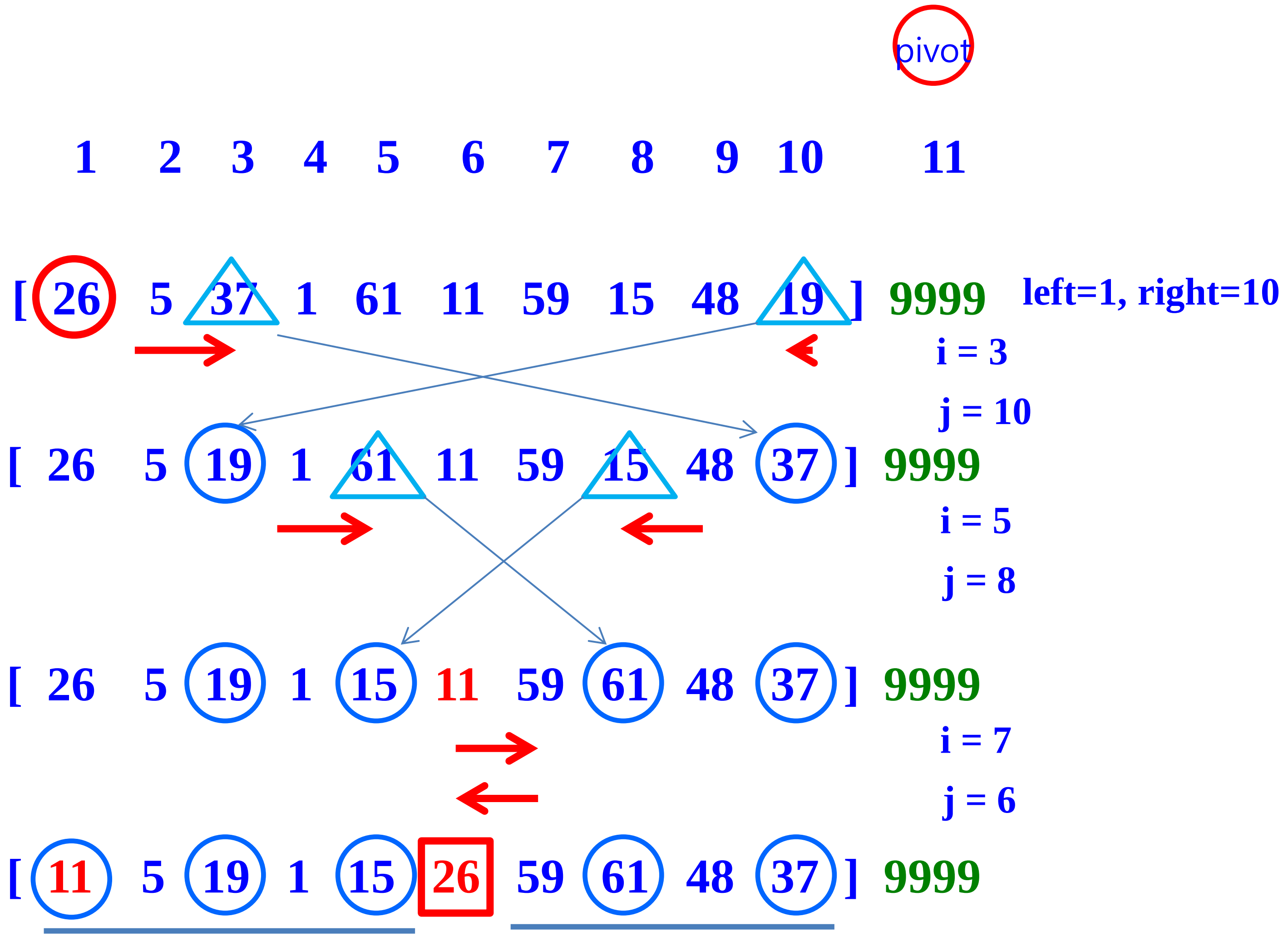


R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]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

Figure 7.1: Quick sort example

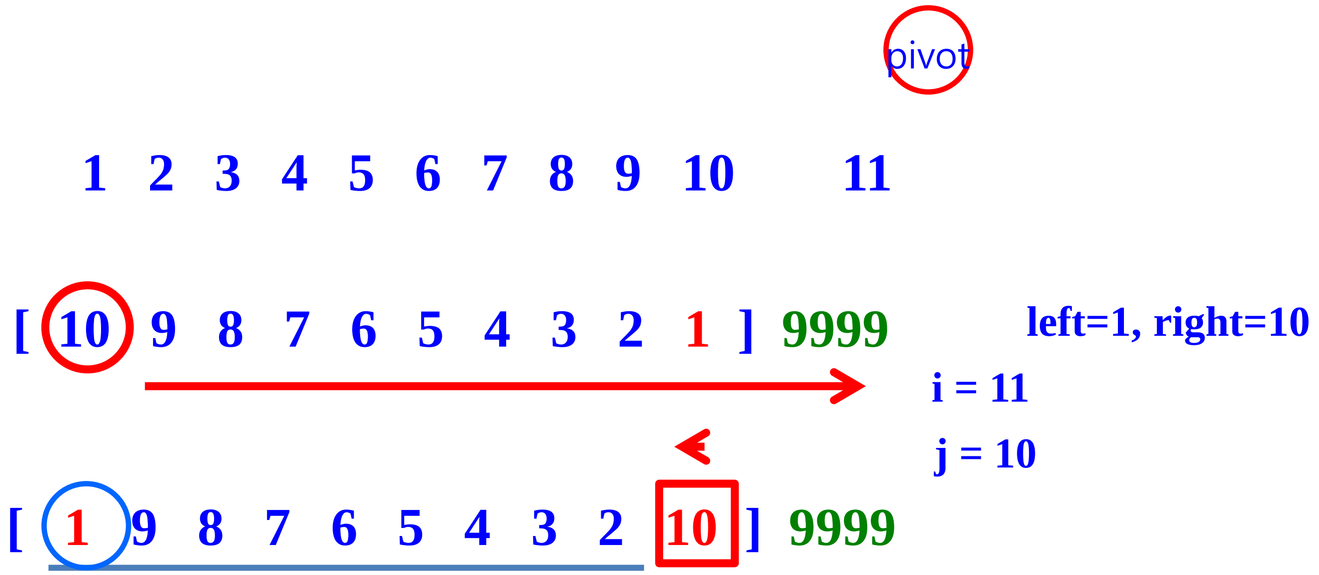
Function *quickSort* assumes that *a*[*n*+1] has been set to have a key at least as large as the remaining keys.

quickSort(*a*, 1, 10)



quickSort(*a*, 1, 5);
quickSort(*a*, 7, 10);

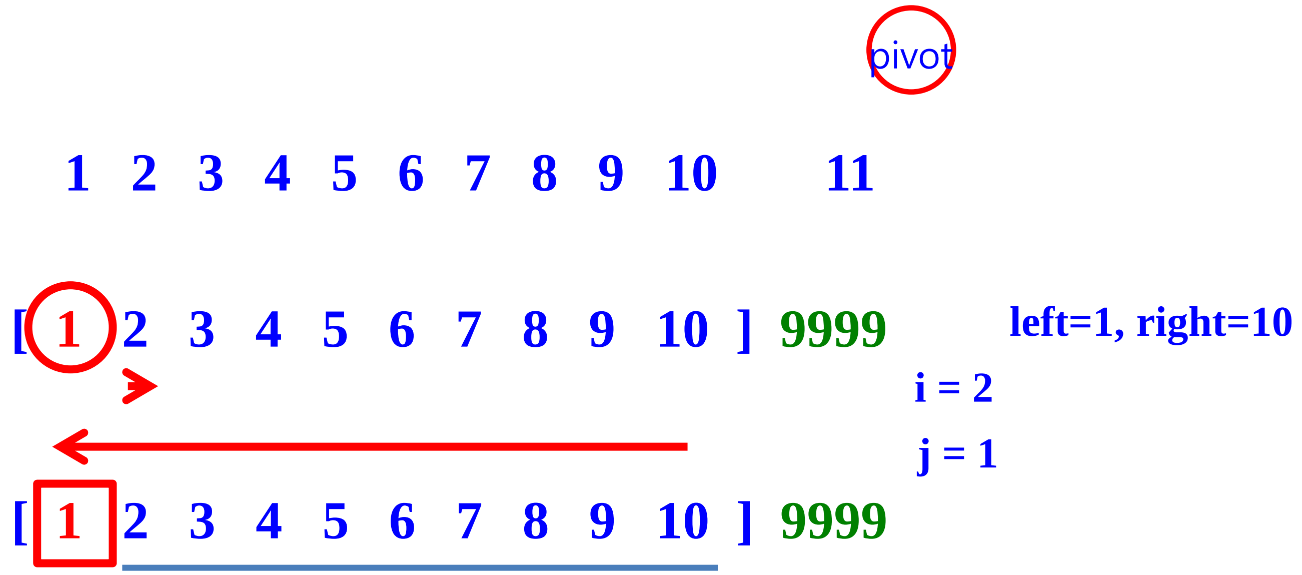
quickSort(*a*, 1, 10)



quickSort(*a*, 1, 9);
quickSort(*a*, 11, 10);

***quickSort*(*a*, 1, 10) :**	***quickSort*(*a*, 1, 9); *quickSort*(*a*, 11, 10);**
***quickSort*(*a*, 1, 9) :**	***quickSort*(*a*, 1, 8); *quickSort*(*a*, 10, 9);**
***quickSort*(*a*, 1, 8) :**	***quickSort*(*a*, 1, 7); *quickSort*(*a*, 9, 8);**
***quickSort*(*a*, 1, 7) :**	***quickSort*(*a*, 1, 6); *quickSort*(*a*, 8, 7);**
***quickSort*(*a*, 1, 6) :**	***quickSort*(*a*, 1, 5); *quickSort*(*a*, 7, 6);**
***quickSort*(*a*, 1, 5) :**	***quickSort*(*a*, 1, 4); *quickSort*(*a*, 6, 5);**
***quickSort*(*a*, 1, 4) :**	***quickSort*(*a*, 1, 3); *quickSort*(*a*, 5, 4);**
***quickSort*(*a*, 1, 3) :**	***quickSort*(*a*, 1, 2); *quickSort*(*a*, 4, 3);**
***quickSort*(*a*, 1, 2) :**	***quickSort*(*a*, 1, 1); *quickSort*(*a*, 3, 2);**
***quickSort*(*a*, 1, 1) :**	**;**

quickSort(*a*, 1, 10)



quickSort(*a*, 1, 0);
quickSort(*a*, 2, 10);

***quickSort*(*a*, 1, 10) :**	***quickSort*(*a*, 1, 0); *quickSort*(*a*, 2, 10);**
***quickSort*(*a*, 2, 10) :**	***quickSort*(*a*, 2, 1); *quickSort*(*a*, 3, 10);**
***quickSort*(*a*, 3, 10) :**	***quickSort*(*a*, 3, 2); *quickSort*(*a*, 4, 10);**
***quickSort*(*a*, 4, 10) :**	***quickSort*(*a*, 4, 3); *quickSort*(*a*, 5, 10);**
***quickSort*(*a*, 5, 10) :**	***quickSort*(*a*, 5, 4); *quickSort*(*a*, 6, 10);**
***quickSort*(*a*, 6, 10) :**	***quickSort*(*a*, 6, 5); *quickSort*(*a*, 7, 10);**
***quickSort*(*a*, 7, 10) :**	***quickSort*(*a*, 7, 6); *quickSort*(*a*, 8, 10);**
***quickSort*(*a*, 8, 10) :**	***quickSort*(*a*, 8, 7); *quickSort*(*a*, 9, 10);**
***quickSort*(*a*, 9, 10) :**	***quickSort*(*a*, 9, 8); *quickSort*(*a*, 10, 10);**
***quickSort*(*a*, 10, 10) :**	**;**

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

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

  - unstable sorting
  - good(best) sorting method
    - average computing time is **O($n\log_2 n$)**

Analysis of quickSort: The worst-case behavior of quickSort is examined in Exercise 2 and shown to be $O(n^2)$. However, if we are lucky, then each time a record is correctly positioned, the sublist to its left will be of the same size as that to its right. This would leave us with the sorting of two sublists, each of size roughly n/2. The time required to position a record in a list of size *n* is O(*n*). If T(*n*) is the time taken to sort a list of *n* records, then when the list splits roughly into two equal parts each time a record is positioned correctly, we have

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

– unstable sorting

Lemma 7.1 shows that the average computing time for function quickSort is O(*n*log *n*). Moreover, experimental results show that as far as average computing time is concerned, Quick sort is the best of the internal sorting methods we shall be studying.

Unlike insertion sort, where the only additional space needed was for one record, quick sort needs stack space to implement the recursion. If the lists split evenly, as in the above analysis, the maximum recursion depth would be log *n*, requiring a stack space of O(log *n*). The worst case occurs when the list is split into a left sublist of size *n* - 1 and a right sublist of size 0 at each level of recursion. In this case, the depth of recursion becomes n, requiring stack space of O(n). The worst-case stack space can be reduced by a factor of 4 by realizing that right sublists of size less than 2 need not be stacked. An asymptotic reduction in stack space can be achieved by sorting smaller sublists first. In this case the additional stack space is at most O(log n).

***Quick sort using a median-of-three*:**

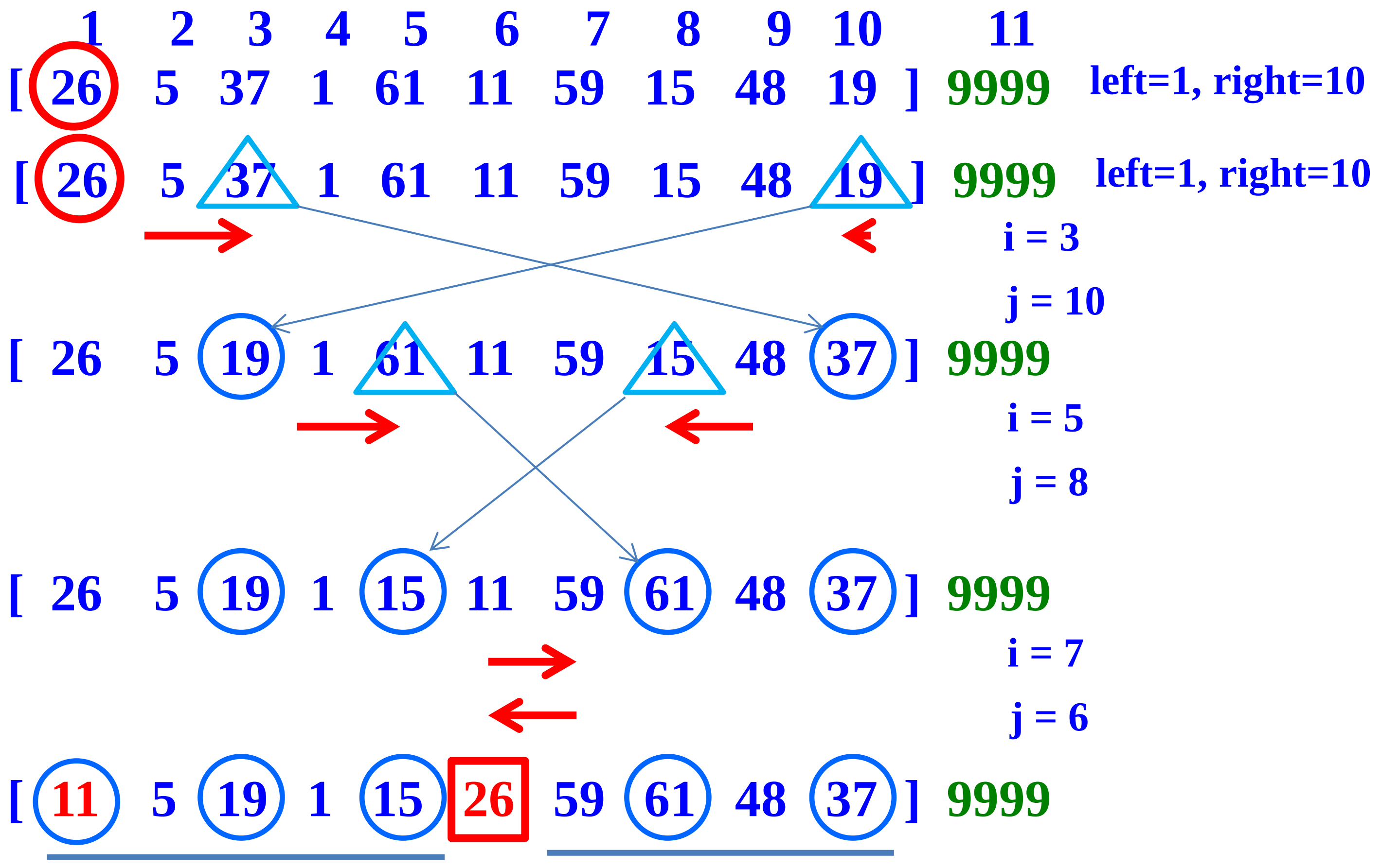
Our version of quick sort always picked the key of the first record in the current sublist as the pivot. A better choice for this pivot is the median of the first, middle, and last keys in the current sublist. Thus, pivot = median $\{K_l, K_{(l+r)/2}, K_r\}$. For example, median{l0, 5, 7} = 7 and median{l0, 7, 7} = 7.

Quick sort using a median-of-three : 26, 61, 19 = 26

pivot

quickSort(*a*, 1, 10)

a[left] 이 median-of-three 를



```
   1   2   3   4   5   6   7   8   9   10      11
[ 26   5  37   1  61  11  59  15  48  19 ]  9999   left=1, right=10
[ 26   5  37   1  61  11  59  15  48  19 ]  9999   left=1, right=10
                                                   i = 3
                                                   j = 10
[ 26   5  19   1  61  11  59  15  48  37 ]  9999
                                                   i = 5
                                                   j = 8
[ 26   5  19   1  15  11  59  61  48  37 ]  9999
                                                   i = 7
                                                   j = 6
[ 11   5  19   1  15  26  59  61  48  37 ]  9999
```

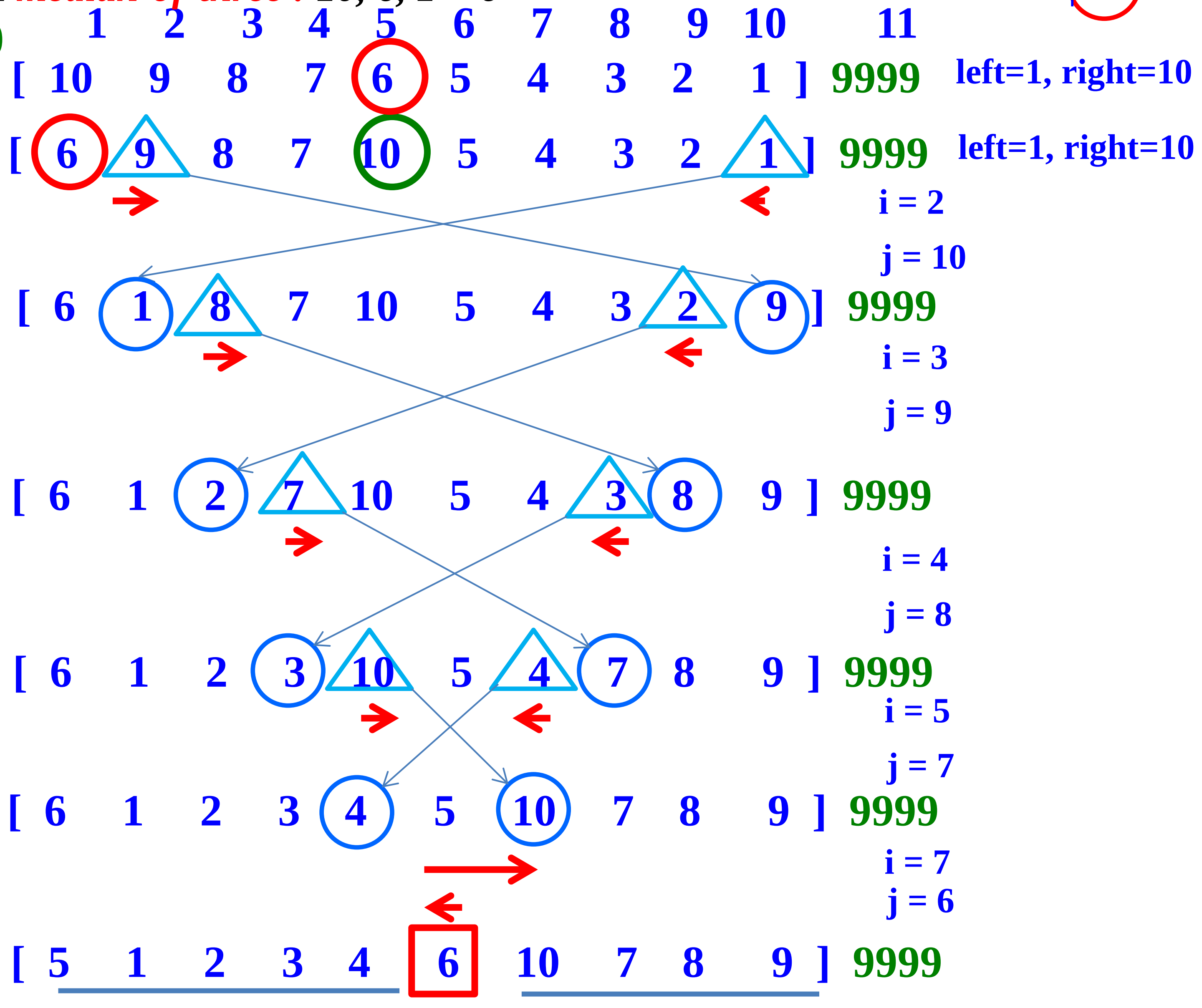
quickSort(*a*, 1, 5);
quickSort(*a*, 7, 10);

Quick sort using a median-of-three : 10, 6, 1 = 6

pivot

quickSort(*a*, 1, 10)

a[left] 에 median-of-three 를

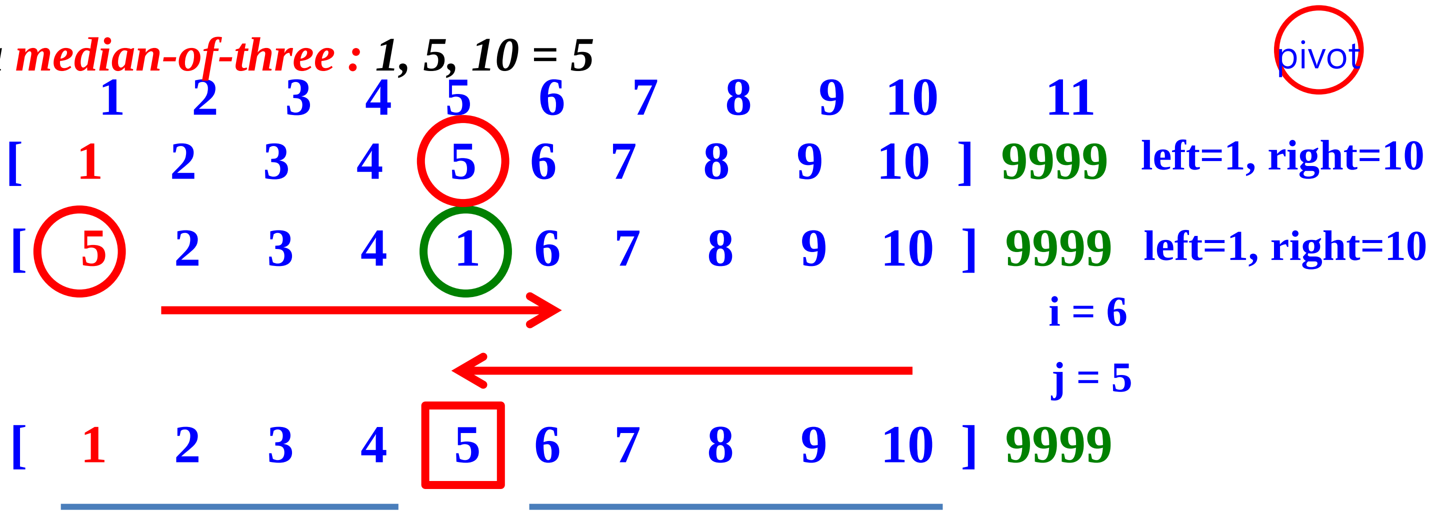


quickSort(*a*, 1, 5);
quickSort(*a*, 7, 10);

Quick sort using a median-of-three : 1, 5, 10 = 5

quickSort(*a*, 1, 10)

a[left] 에 *median-of-three* 를


pivot
1 2 3 4 5 6 7 8 9 10 11
[1 2 3 4 5 6 7 8 9 10] 9999 left=1, right=10
[5 2 3 4 1 6 7 8 9 10] 9999 left=1, right=10
i = 6
j = 5
[1 2 3 4 5 6 7 8 9 10] 9999

quickSort(*a*, 1, 4);
quickSort(*a*, 6, 10);

7.4 How Fast Can We Sort?

Both of the sorting methods we have seen so far have a worst-case behavior of $O(n^2)$. It is natural at this point to ask the question, What is the best computing time for sorting that we can hope for? The theorem we shall prove shows that if we restrict our question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n\log n)$ is the best possible time.

The method we use is to consider a tree that describes **the sorting process**. Each vertex of the tree represents a key comparison, and the branches indicate the result. Such a tree is called a ***decision tree***. A path through a decision tree represents a sequence of computations that an algorithm could produce.

Example 7.4: Let us look at the decision tree obtained for insertion sort working on a list with three records (Figure 7.2). The input sequence is R_1, R_2, and R_3, so the root of the tree is labeled [1, 2, 3]. Depending on the outcome of the comparison between keys K_1 and K_2, this sequence may or may not change. If $K_2 < K_1$, then the sequence becomes [2, 1, 3]; otherwise it stays [1, 2, 3]. The full tree resulting from these comparisons is given in Figure 7.2.

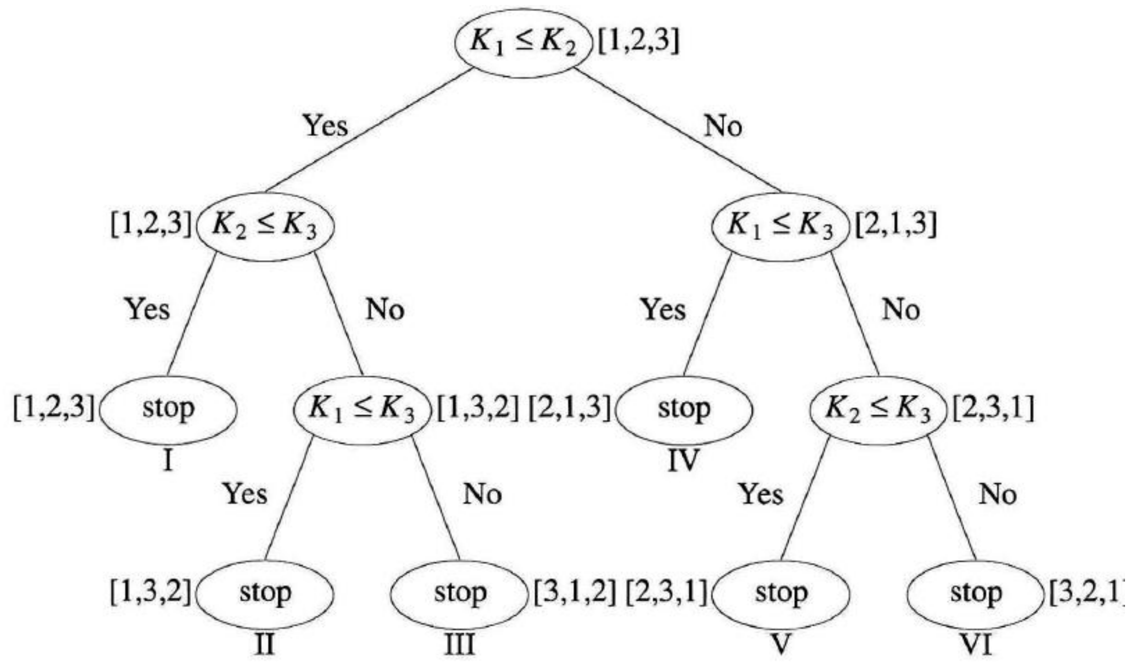


Figure 7.2: Decision tree for insertion sort

leaf	permutation	sample input key values that give the permutation
I	1 2 3	[7, 9, 10]
II	1 3 2	[7, 10, 9]
III	3 1 2	[9, 10, 7]
IV	2 1 3	[9, 7, 10]
V	2 3 1	[10, 7, 9]
VI	3 2 1	[10, 9, 7]

Permutation 3! = 6
the maximum depth = 3

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Proof: When sorting n elements, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves. But a decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is k. Therefore, the height must be at least $\log_2 n! + 1$. □

$2^h \geq n! \Rightarrow \quad h \geq \log_2(n!)$

Corollary: Any algorithm that sorts by comparisons only must have a worst case computing time of $\Omega(n\log_2 n)$.

Proof: We must show that for every decision tree with $n!$ leaves, there is a path of length $cn\log_2 n$, where c is a constant. By the theorem, there is a path of length $\log_2(n!)$.

Now

$$n! = n\,(n - 1)(n - 2)\ \cdot\cdot\cdot\ (3)(2)(1) \geq (n/2)^{n/2}$$

So, $\log_2(n!) \geq \log_2 (n/2)^{n/2} = \Omega(n\log_2 n)$. □

Theorem: Any decision tree sorting *n* elements has height (*n* log *n*)

Proof:

- Assume elements are the (distinct) numbers 1 through *n*
- There must be *n*! leaves (one for each of the *n*! permutations of *n* elements)
- Tree of height *h* has at most 2^h leaves

$$2^h \geq n! \Rightarrow \quad h \geq \log_2(n!)$$
$$= \log_2(n\,(n - l)(n - 2) \cdot \cdot \cdot (3)(2)(1))$$
$$\geq \log_2((n/2)^{n/2})$$
$$= \Omega(n\log_2 n). \ \square$$

Using a similar argument and the fact that binary trees with 2^n leaves must have an average root-to-leaf path length of $\Omega(n\log_2 n)$, we can show that the average complexity of comparison-based sorting methods is $\Omega(n\log_2 n)$.

EXERCISE 1 (for Sorting)

Sort할 data를 n, n-1, n-2,· · ·, 1의 sequence로 생성하여 배열에 저장하시오. Sorting algorithm은 (1) Selection sort, (2) Insertion sort, (3) Quick sort, (4) Quick sort using a median-of-three(이 알고리즘은 교재의 알고리즘을 기반으로 우리가 작성하여야 한다)를 사용한다.

우리는 n을 10, 100, 1000, 10000, 100000에 대하여 수행하고자 한다. 각 n에 대하여 각 algorithm의 수행 시간을 우리가 Chapter 1에서 수행한 방법으로 측정하시오.

(1) a[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};로 하였을 때 정확히 동작하는 지를 확인하고 시간을 측정한다.

(2) a[100], a[1000]에도 정확히 동작하는 지를 확인하고 시간을 측정한다.

(3) a[10000]에는 동작하는 지를 확인하고 시간을 측정한다. 만약 동작하지 않는 알고리즘이 있다면 그 이유를 분석하시오. Hint. Quick Sort의 경우, 동작하지 않을 수 있다. 무엇 때문일까? 이 경우, Quick sort using a median-of-three는 동작하는가?

(4) a[100000]에는 동작하는 지를 확인하고 시간을 측정한다. 만약 동작하지 않는 알고리즘이 있다면 그 이유를 분석하시오. Hint. Quick Sort의 경우, 동작하지 않을 수 있다. 무엇 때문일까? 이 경우, Quick sort using a median-of-three는 동작하는가?

이 분석은 수요일의 과제에도 유사하게 사용된다.

7.5 Merge Sort

7.5.1 Merging

Before looking at the merge sort method to sort *n* records, let us see how one may merge two sorted lists to get a single sorted list. Program 7.7 gives the code for this. The two lists to be merged are *initList* [*i*:*m*] and *initList* [*m* + 1:*n*]. The resulting merged list is *mergedList* [*i* :*n*].

Example

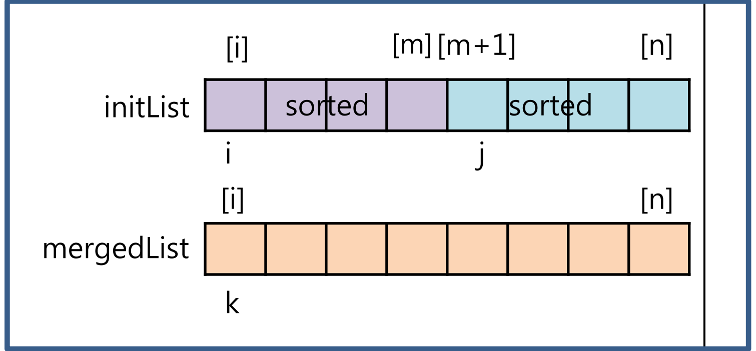
	A	B	C
1	2, 5, 6	1, 3, 8, 9, 10	
2	2, 5, 6	3, 8, 9, 10	1
3	5, 6	3, 8, 9, 10	1, 2
4	5, 6	8, 9, 10	1, 2, 3
5	6	8, 9, 10	1, 2, 3, 5
6		8, 9, 10	1, 2, 3, 5, 6
7			1, 2, 3, 5, 6, 8, 9, 10

- Compare the smallest elements of A and B and merge the smaller into C.
- When one of A and B becomes empty, append the other list to C.

2-way merge

```
void merge(element initList[], element mergedList[], int LeftRunStart, int LeftRunEnd, int RightRunEnd)
{
    /* the sorted lists initList[LeftRunStart:LeftRunEnd] and initList[LeftRunEnd+l:RightRunEnd] are merged
       to obtain the sorted list mergedList[LeftRunStart:RightRunEnd] */
    int left, right, target;
    left = LeftRunStart;
    right = LeftRunEnd+l;
    target = LeftRunStart;
    while (left <= LeftRunEnd && right <= RightRunEnd) {
        if (initList[left].key <= initList[right].key)
            mergedList[target++] = initList[left++];
        else
            mergedList[target++] = initList[right++];
    }
    if  (left > LeftRunEnd) {
        for (; right <= RightRunEnd; )
            mergedList[target++] = initList[right++];
    } else {
        for (; left <= LeftRunEnd; )
            mergedList[target++] = initList[left++];
    }
}
```

Program 7.7: Merging two sorted lists



- Analysis of *merge*:
  - Total increment in *k* is *n*-*i*+1.
  - O(*n*-*i*+1) ➔ O(*n*)
  - Stable sorting

7.5.2 Iterative Merge Sort

This version of **merge sort** begins by interpreting the input list as comprised of *n* sorted sublists, each of size 1. In the first merge pass, these sublists are merged by pairs to obtain *n*/2 sublists, each of size 2 (if *n* is odd, then one sublist is of size 1). In the second merge pass, these *n*/2 sublists are then merged by pairs to obtain *n*/4 sublists. Each merge pass reduces the number of sublists by half. Merge passes are continued until we are left with only one sublist. The example below illustrates the process.

Example 7.5: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). The tree of Figure7.4 illustrates the sublists being merged at each pass. □

run size=1

run size=2

run size=4

run size=8

run size=16

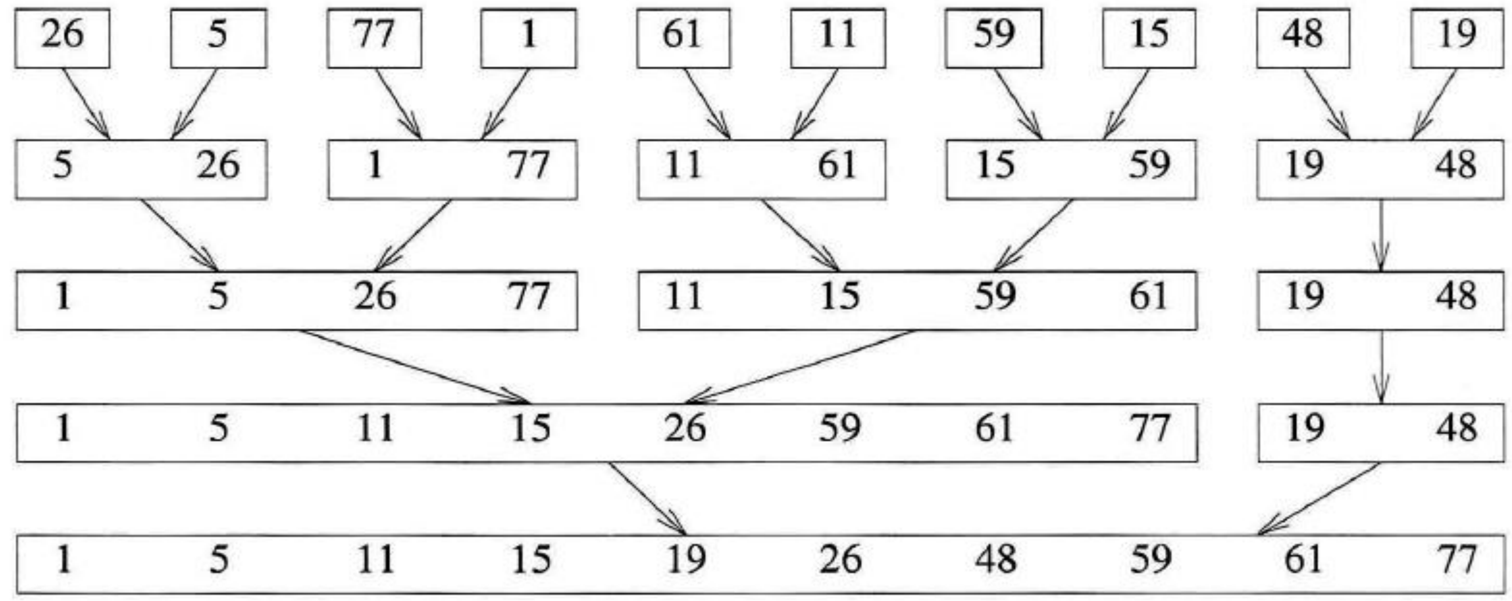


Figure 7.4: Merge tree

2-way merge

2-way merge sort

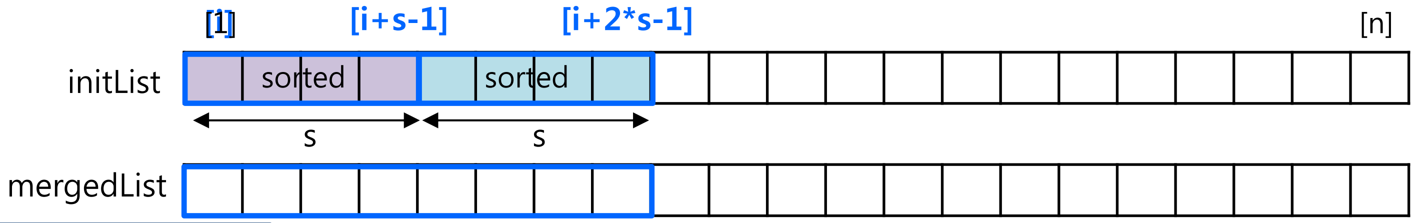
```
void mergeSort(element orig[], int EndIndex)
{
    /* sort orig[l:n] using the merge sort method */
    int runSize = 1;
    element extra[MAX_SIZE];

    while (runSize < EndIndex) {
        mergePass(orig, extra, EndIndex, runSize);
        runSize *= 2;
        mergePass(extra, orig, EndIndex, runSize);
        runSize *= 2;
    }
}
```

Program 7.9: Merge sort

```
void mergePass(element initList[], element mergedList[], int EndIndex, int runSize)
{
    int left, target;
    left = 1;
    // Merge 2 runs of equal size
    for (; left <= (EndIndex - 2*runSize + 1); left += 2*runSize)
        merge(initList,mergedList,left,left+runSize-l,left+2*runSize-1);
    if (left + runSize - 1 < EndIndex) {
        // merge 2 sublists of  size s and size less than s
        merge(initList,mergedList,left,left+runSize-l, EndIndex);
    } else {
        // 1 remaining sublist of size less than s
        for (target = left; target <= EndIndex; target++)
            mergedList[target] = initList[target];
    }
}
```

Program 7.8: A merge pass

call mergeSort(a, n)



Analysis

- Sorted segment size is 1, 2, 4, 8, …
- Number of merge passes is ⌈$\log_2 n$⌉.
- Each merge pass takes O(n) time.
- Total time is **O(n log n)**.
- Need O(n) additional space for the merge.

7.5.3 Recursive Merge Sort

In the recursive formulation we divide the list to be sorted into two roughly equal parts called the left and the right sublists. These sublists are sorted recursively, and the sorted sublists are merged.

Example 7.6: The input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) is to be sorted using the recursive formulation of merge sort. If the sublist from *left* to *right* is currently to be sorted, then its two sublists are indexed from *left* to $\lfloor(left + right)/2\rfloor$ and from $\lfloor(left + right)/2\rfloor + 1$ to *right*. The sublist partitioning that takes place is described by the binary tree of Figure 7.5. Note that the sublists being merged are different from those being merged in *mergeSort*. □

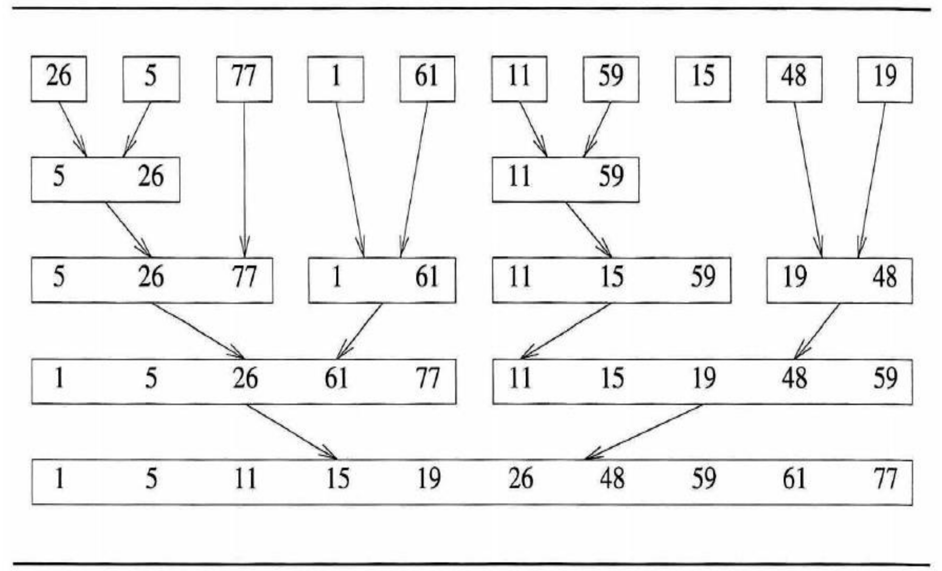
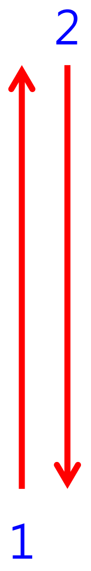


Figure 7.5: Sublist partitioning for recursive merge sort



- ***Downward pass*** over the recursion tree.
  - Divide large instances into small ones.
- ***Upward pass*** over the recursion tree.
  - Merge pairs of sorted lists.
- Number of leaf nodes is ***n***.
- Number of nonleaf nodes is ***n-1***.

call rmergeSort(a, link, 1, n)

```
int rmergeSort(element a[], int link[], int left, int right)
{
    /* a[left:right] is to be sorted,
       link[i] is initially 0 for all i,
       returns the index of the first element
       in the sorted chain */
    if (left >= right) return left;
    int mid = (left + right) / 2;
    return listMerge(a, link,
                        rmergeSort(a, link, left, mid),
                        rmergeSort(a, link, mid+1, right));
}
```

Program 7.10: Recursive merge sort

```
int listMerge(element a[], int link[], int LeftStart, int RightStart)
{
   /* sorted chains beginning at LeftStart and RightStart, respectively, are merged;
      link[0] is used as a temporary header; returns start of merged chain */
    int left, right, target = 0;
    left = LeftStart; right = RightStart;
    for (; left && right;) {
        if (a[left] <= a[right]) {
            link[target] = left;
            target = left; left = link[left];
        } else {
            link[target] = right;
            target = right; right = link[right];
        }
    }
    /* attach remaining records to result chain */
    if (left == 0) link[target] = right;
    else link[target] = left;
    return link[0];
}
```

Program 7.11: Merging sorted chains

- ***Downward pass*** over the recursion tree: rmergeSort()를 호출하는 동안의 과정이다.
- ***Upward pass*** over the recursion tree: rmergeSort()가 return되면서 listMerge()가 수행되는 과정이다.

Analysis

– Downward pass
  - O(1) time at each node.
  - O(n) total time at all nodes.

– Upward pass
  - O(n) time merging at each level that has a nonleaf node.
  - Number of levels is O(log n).
  - **Total time is O(n log n).**

To eliminate the record copying that takes place when *merge* (Program 7.7) is used to merge sorted sublists we associate an integer pointer with each record. For this purpose, we employ an integer array *link*[1:*n*] such that *link*[*i*] gives the record that follows record *i* in the sorted sublist. In case *link*[*i*] = 0, there is no link record. With the addition of this array of links, record copying is replaced by link changes and the runtime of our sort function becomes independent of the sizes of a record. Also the additional space required is O(*n*). By comparison, the iterative merge sort described earlier takes O(*sn*log*n*) time and O(*sn*) additional space. On the down side, the use of an array of links yields a sorted chain of records and we must have a follow up process to physically rearrange the records into the sorted order dictated by the final chain. We describe the algorithm for this physical rearrangement in Section 7.8.

We assume that initially *link*[*i*] = 0, $1 \le i \le n$. Thus, each record is initially in a chain containing only itself. Let *start*1 and *start*2 be pointers to two chains of records. The records on each chain are in nondecreasing order. Let *listMerge*(*a*, *link*, *start*1, *start*2) be a function that merges two chains *start*1 and *start*2 in array *a* and returns the first position of the resulting chain that is linked in nondecreasing order of key values. The recursive version of merge sort is given by function *rmergeSort* (Program 7 .10). To sort the array *a*[1:*n*] this function is invoked as *rmergeSort*(*a*, *link*, 1, *n*). The start of the chain ordered as described earlier is returned. Function *listMerge* is given in Program 7.11.

***Variation*- *Natural Merge Sort*:** We may modify *mergeSort* to take into account the prevailing order within the input list. In this implementation we make an initial pass over the data to determine the sub lists of records that are in order. Merge sort then uses these initially ordered sublists for the remainder of the passes. Figure 7.6 shows natural merge sort using the input sequence of Example 7.6.

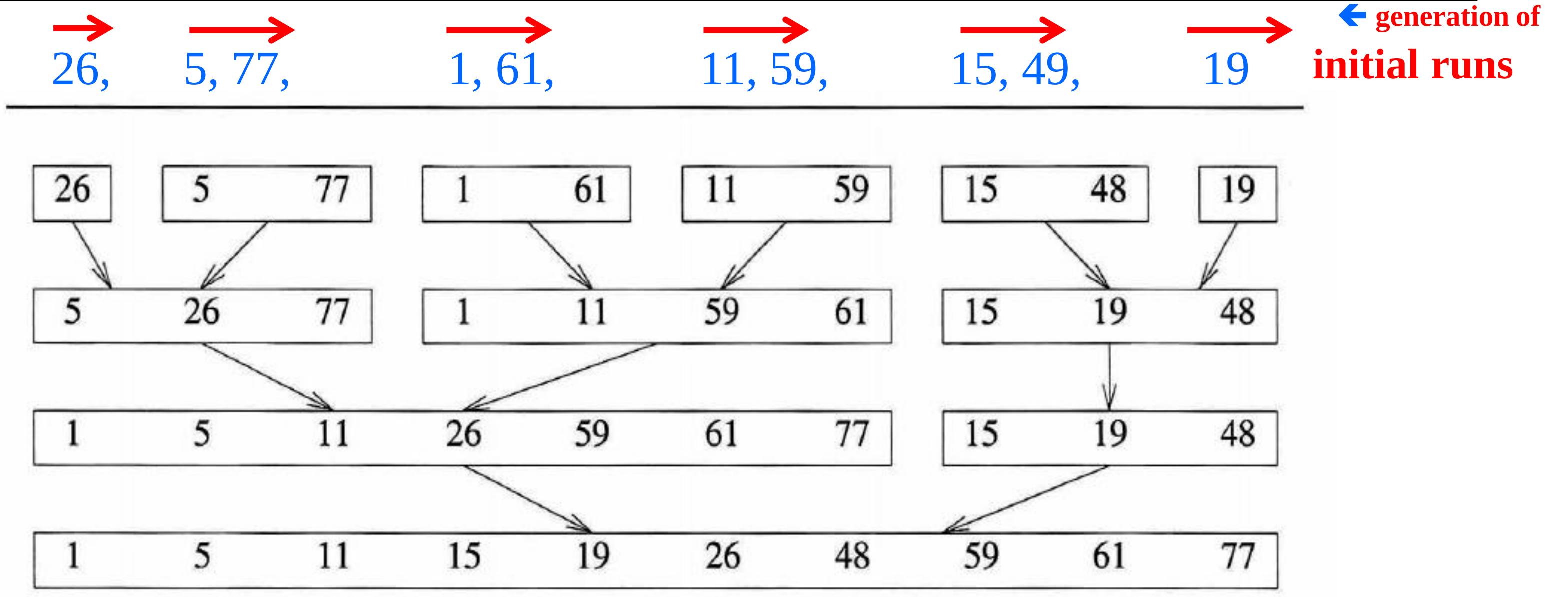


Figure 7.6: Natural merge sort

7.6 Heap Sort

Although **the merge sort** scheme discussed in the previous section has a computing time of O(*n* log *n*), both in the worst case and as average behavior, it requires additional storage proportional to the number of records to be sorted. The sorting method we are about to study, **heap sort**, requires only a fixed amount of additional storage and at the same time has as its worst-case and average computing time O(*n* log *n*). However, heap sort is slightly slower than merge sort.

In heap sort, we utilize the max-heap structure introduced in Chapter 5. The deletion and insertion functions associated with max heaps directly yield an O(*n* log *n*) sorting method. The *n* records are first inserted into an initially empty max heap. Next, the records are extracted from the max heap one at a time. It is possible to create the max heap of *n* records faster than by inserting the records one by one into an initially empty heap. For this, we use the function *adjust* (Program 7.12), which starts with a binary tree whose left and right subtrees are max heaps and rearranges records so that the entire binary tree is a max heap. The binary tree is embedded within an array using the standard mapping. If the depth of the tree is *d*, then the **for** loop is executed at most *d* times. Hence the computing time of adjust is O(*d*).

To sort the list, first we create a max heap by using *adjust* repeatedly, as in the first **for** loop of function *heapSort* (Program 7 .13). Next, we swap the first and last records in the heap. Since the first record has the maximum key, the swap moves the record with maximum key into its correct position in the sorted array. We then decrement the heap size and readjust the heap. This swap, decrement heap size, readjust heap process is repeated *n* - 1 times to sort the entire array *a*[1:*n*]. Each repetition of the process is called a ***pass***. For example, on the first pass, we place the record with the highest key in the *n*th position; on the second pass, we place the record with the second highest key in position *n* - 1; and on the *i*th pass, we place the record with the *i*th highest key in position *n*- *i* + 1.

call heapSort(a, n)

```
void heapSort(element a[], int n)
{
     /* perform a heap sort on a[l:n] */
    int i,j;
    element temp;

    /* build the initial heap */
    for (i = n/2; i > 0; i--)
        adjust(a,i,n);
    /* sort */
    for (i = n-1; i > 0; i--) {
        SWAP(a[l],a[i+l],temp);
        adjust(a,l,i);
    }
}
```

Program 7.13: Heap sort

```
void adjust(element a[], int root, int n)
{
    /* a[root]를 제외하고 n까지의 그 하위 tree들은 max-heap이 구성되어 있다 */
    /* 이제 a[root]를 포함하여 그 tree의 max-heap을 구성하라 */
    int child,rootkey;
    element temp;
    temp= a[root];
    rootkey = a[root].key;
    child = 2*root;                    /* the left child */
    while (child <= n) {
        if ((child < n) && (a[child].key< a[child+l].key))
            child++;
        if (rootkey > a[child].key)  break;
        else {
            a[child/2] = a[child]; /* move child to parent */
            child *= 2;
        }
    }
    a[child/2] = temp;
}
```

Program 7.12: Adjusting a max heap

Analysis

- **average case : O($n \cdot \log_2 n$)**
- **function *adjust* : O(d), where d: depth of tree**
- **worst case :**

  $\lfloor \log_2 n \rfloor + \lfloor \log_2 (n-1) \rfloor + \ldots + \lfloor \log_2 2 \rfloor =$ **O($n \cdot \log_2 n$)**

Example 7.7: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). If we interpret this list as a binary tree, we get the tree of Figure 7.7(a). Figure 7.7(b) depicts the max heap after the first for loop of *heapSort*. Figure 7.8 shows the array of records following each of the first seven iterations of the second for loop. The portion of the array that still represents a max heap is shown as a binary tree; the sorted part of the array is shown as an array. □

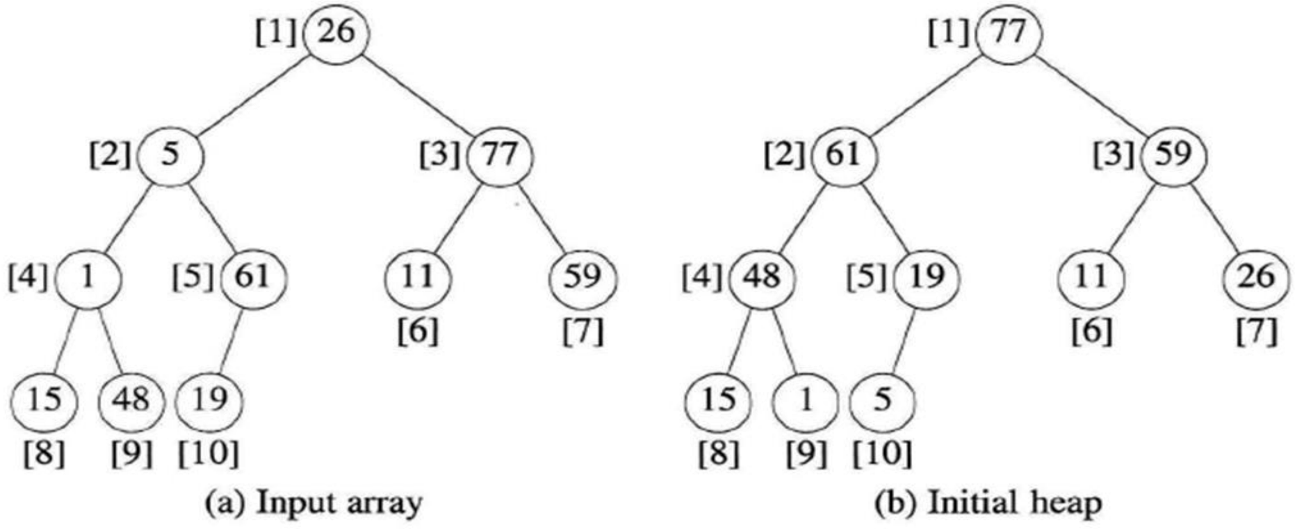


Figure 7.7: Array interpreted as a binary tree

0	1	2	3	4	5	6	7	8	9	10
	26	5	77	1	61	11	59	15	49	19

Input Array

0	1	2	3	4	5	6	7	8	9	10
	77	61	59	48	19	11	26	15	1	5

Initial heap

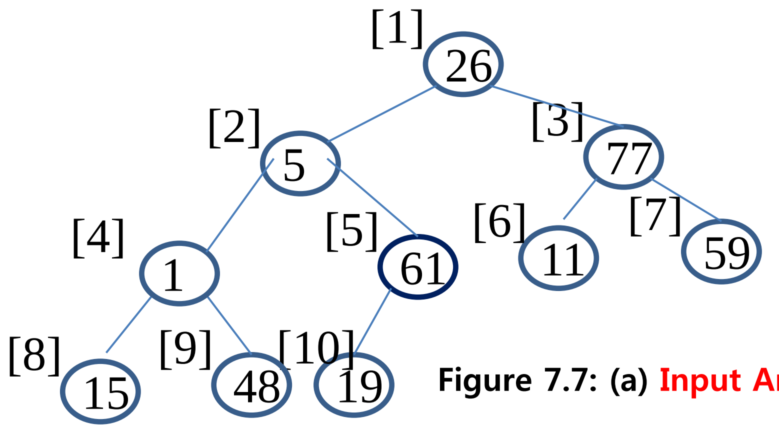
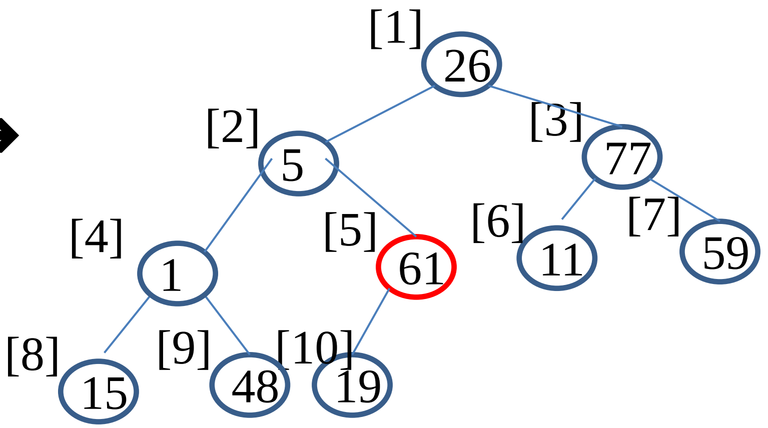
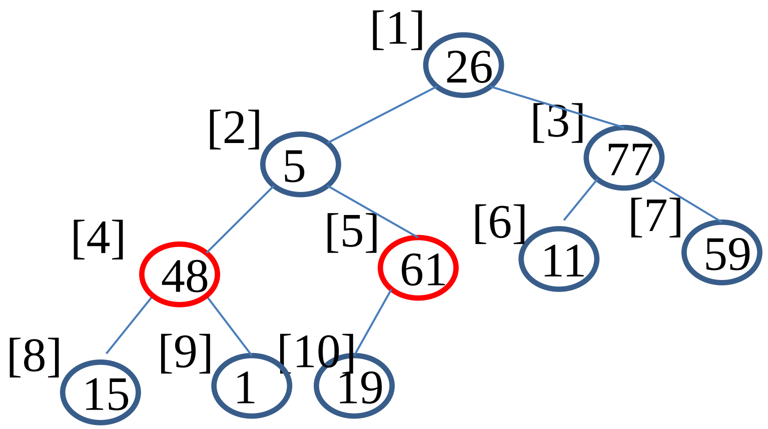


Figure 7.7: (a) Input Array

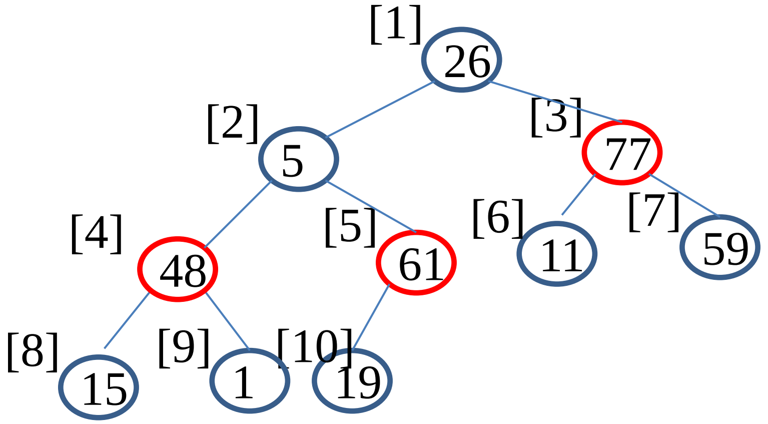
➔



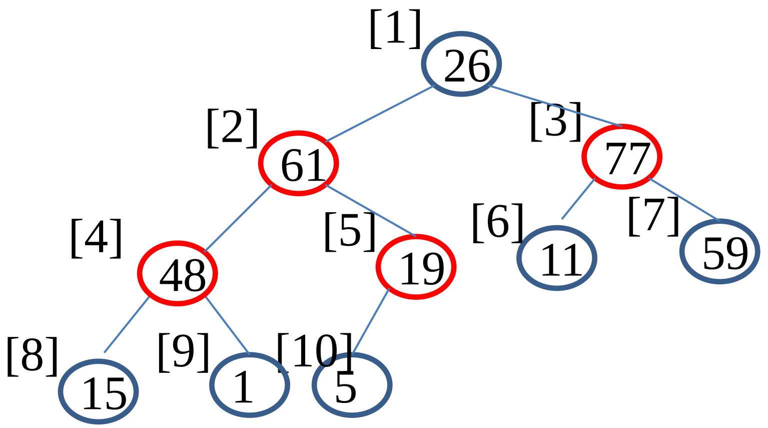
➔



➔



➔



➔

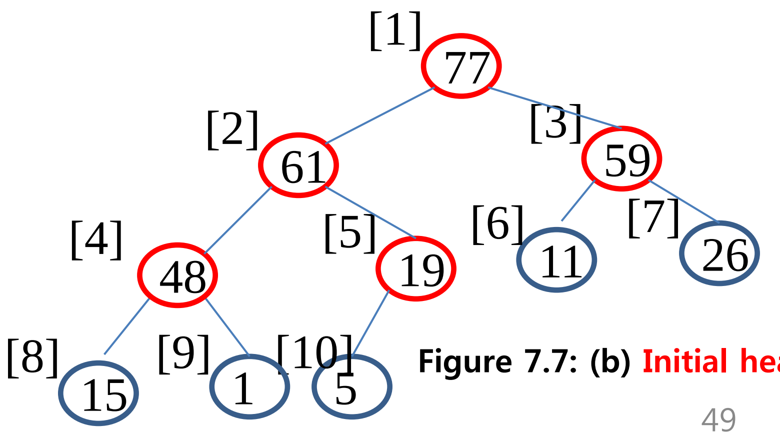


Figure 7.7: (b) Initial heap

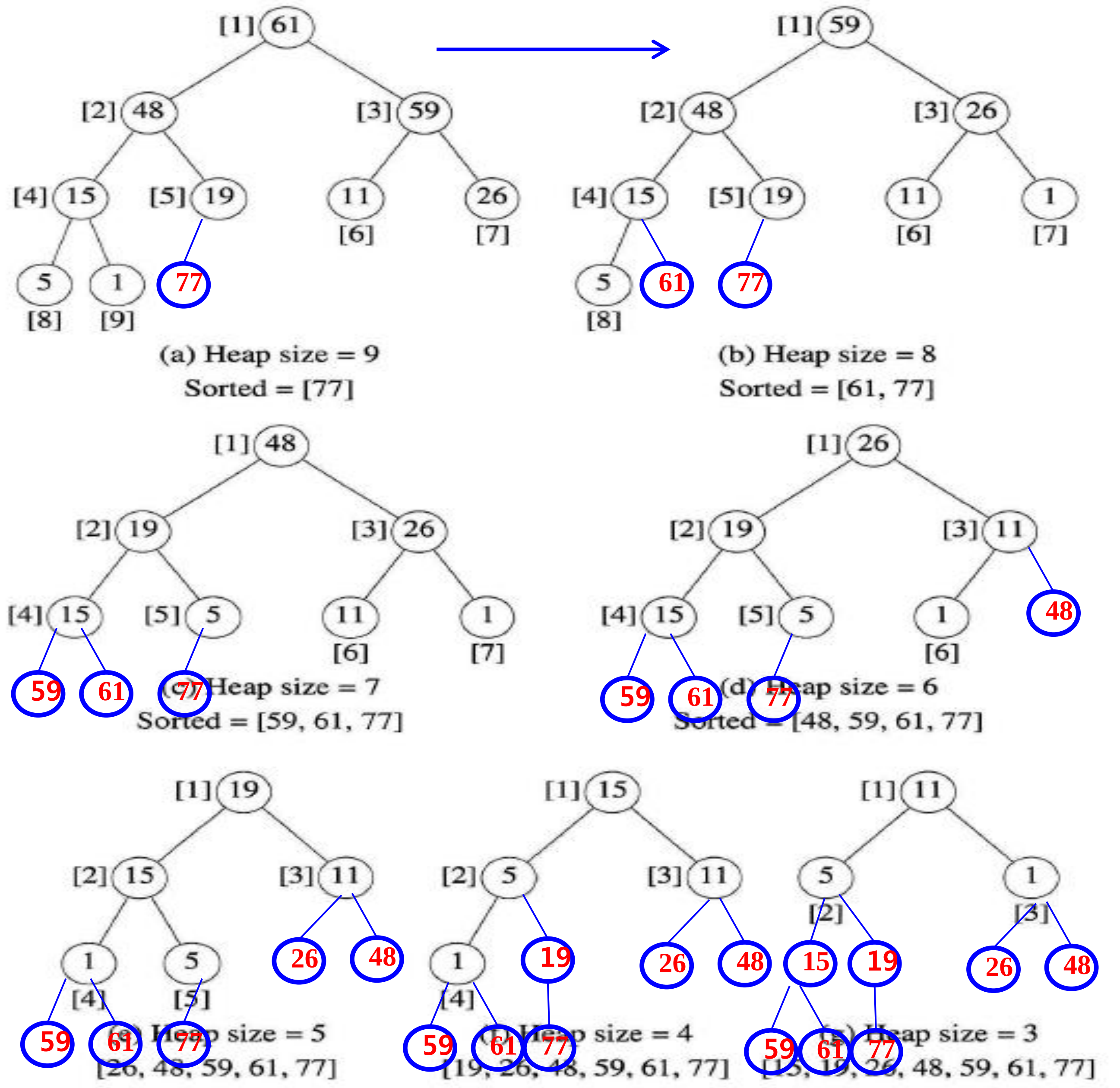
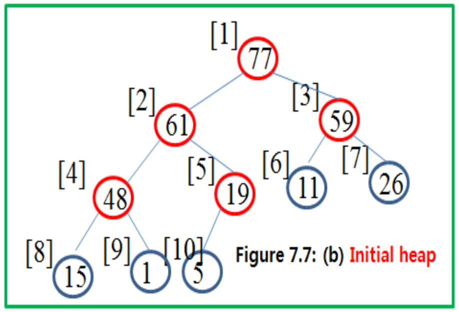
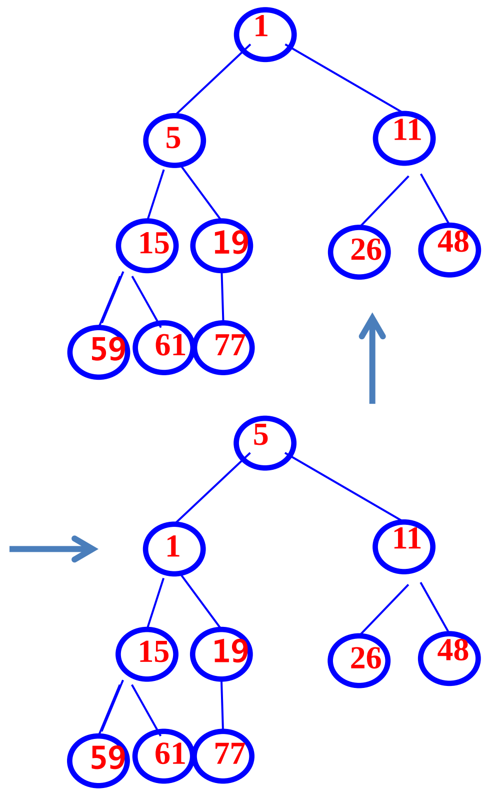


Figure 7.8: Heap sort example





7.9 Summary of Internal Sorting

- No one method is best under all circumstances
  - Some are good for small *n*, others for large *n*

- Insertion sort is good when
  - The list is partially ordered
  - Best sorting method for small *n*

- Merge sort has the best worst case behavior
  - But requires more storage than heap sort

- Quick sort has the best average behavior
  - But worst case behavior is $O(n^2)$

Method	Worst	Average
Insertion sort	n^2	n^2
Heap sort	$n \log n$	$n \log n$
Merge sort	$n \log n$	$n \log n$
Quick sort	n^2	$n \log n$

Figure 7.15: Comparison of sort methods

n	*Insert*	*Heap*	*Merge*	*Quick*
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Times are in milliseconds

Figure 7.16: Average times for sort methods

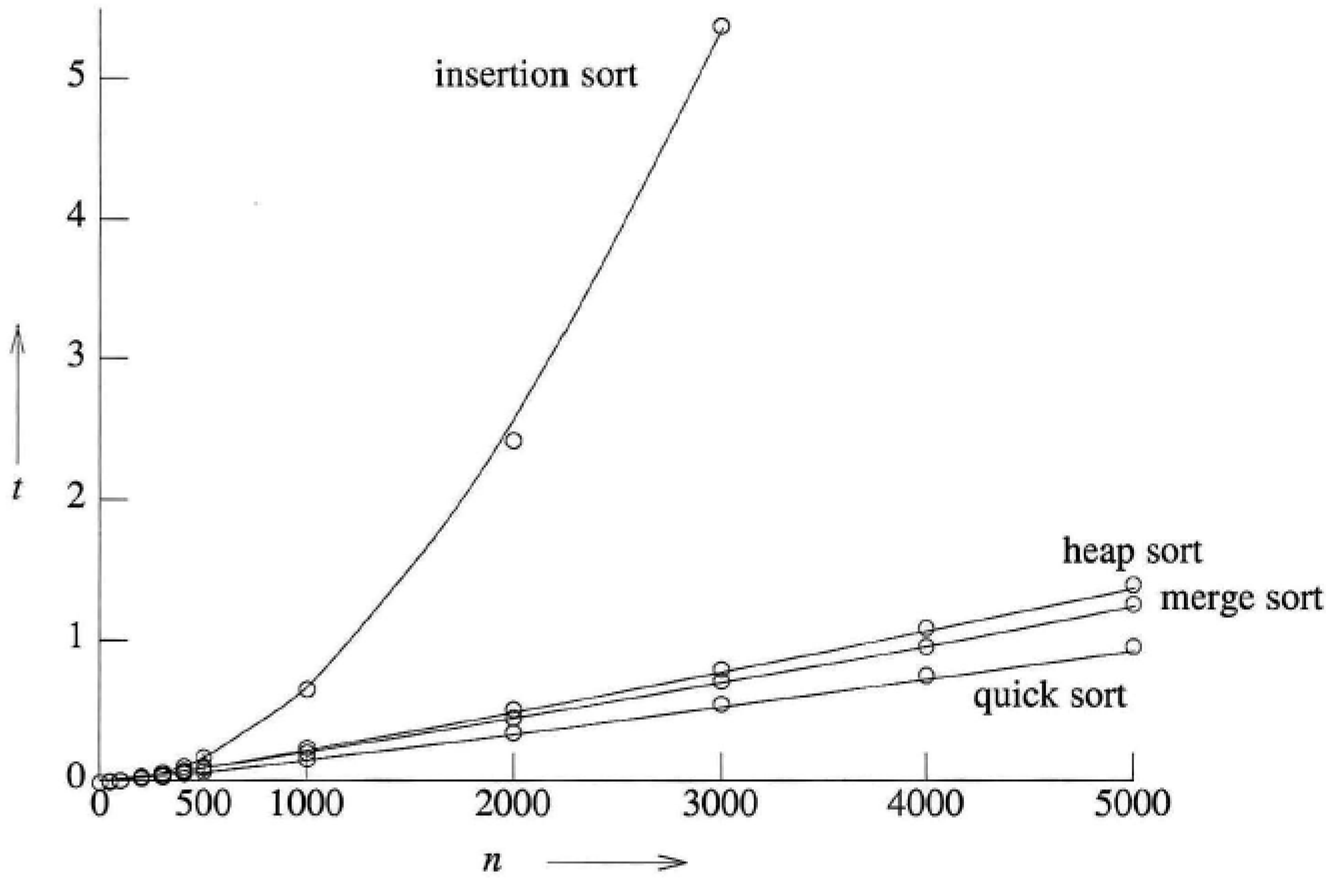


Figure 7.18: Plot of average times (milliseconds)

EXERCISE (for Sorting)

Sort할 data를 random number generator를 이용하여 1과 10000 사이의 정수들로 생성하시오.

Sort algorithm은 (1) selection sort, (2) merge sort, (3) heap sort를 사용하시오.

(1) a[10], a[100]으로 하였을 때 정확히 동작하는 지를 확인한다.

(2) a[1000]에 정확히 동작하는 지를 확인하고 시간을 측정한다.

(3) a[10000]에 동작하는 지를 확인하고 시간을 측정한다.

(4) a[100000]에 동작하는 지를 확인하고 시간을 측정한다.

측정된 시간을 1장의 그래프 그리기로 나타내시오.

나는 개인적으로 selection sort와 merge sort를 좋아한다.

7.10 External Sorting

7.10.1 Introduction

Assume that the lists to be sorted are so large that an entire list cannot be contained in the internal memory of a computer, making an internal sort impossible.

We shall assume that the list (or file) to be sorted resides on a disk.

The term ***block*** refers to the unit of data that is read from or written to a disk at one time.

A block generally consists of several records.

For a disk, there are three factors contributing to the read/write time:

(l) ***Seek time***: time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

(2) ***Latency time***: time until the right sector of the track is under the read/write head.

(3) ***Transmission time***: time to transmit the block of data to/from the disk.

The most popular method for sorting on external storage devices is ***merge sort***. This method consists of two distinct phases.

Sort-Merge Strategy

Phase 1 (Sort Phase).

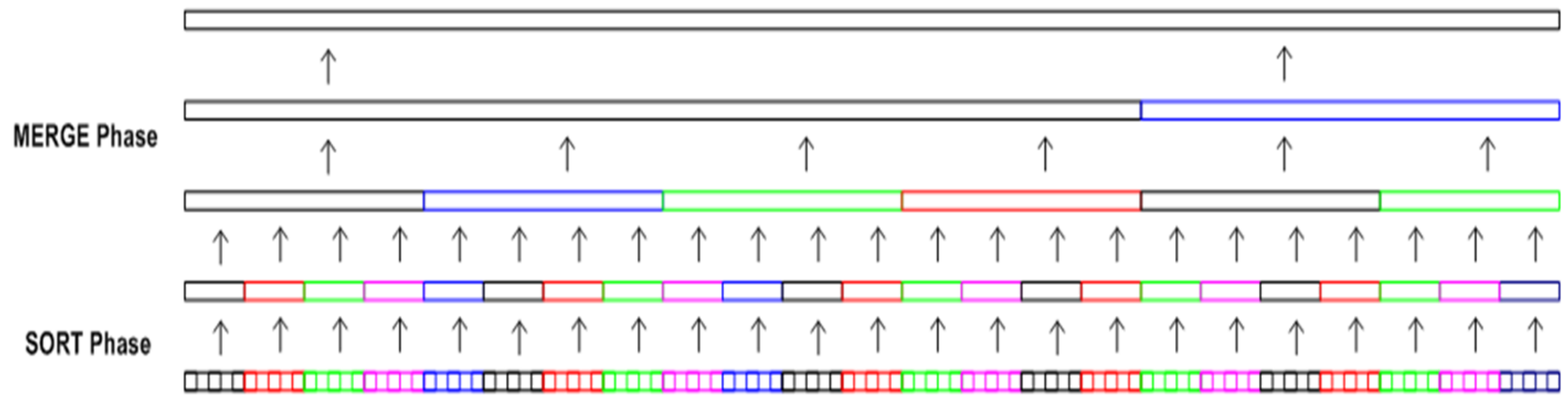
(1) Divide the file into ***runs*** (such that the size of a run is small enough to fit into main memory),

(2) **sort** each run in main memory using a fast in-memory sorting algorithm, and

(3) write each run onto a file (or files) as they are generated.

Phase 2 (Merge Phase).

Merge the resulting runs together into successively bigger runs, until only one run is left.



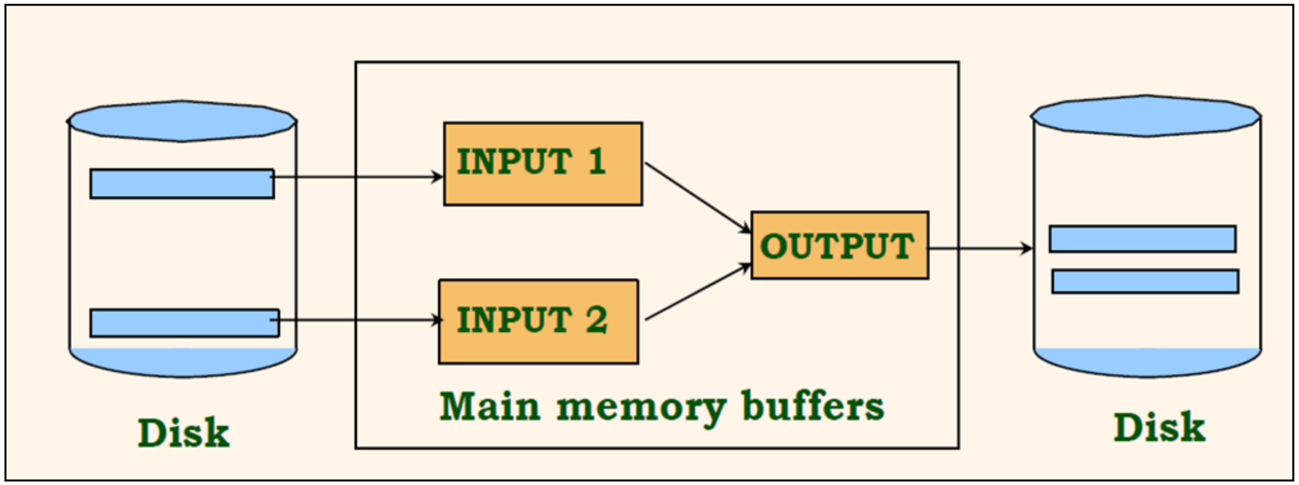
External Sorting: b = 115, nB = 5,

nR = 23, dM = 4 (four-way merging), 23 → 6 → 2 → 1 : 3 passes, the number of block accesses = 2*115 + 2*115*3 = 2*115*4= 920

Example 7.12: A list containing 4500 records is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input list is maintained on disk and has a block length of 250 records. We have available another disk that may be used as a scratch pad. The input disk is not to be written on. One way to accomplish the sort using the general function outlined above is to

(1) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R_1 to R_6. A method such as heap sort, merge sort, or quick sort could be used. These six runs are written onto the scratch disk (Figure 7.19).

(2) Set aside three blocks of internal memory, each capable of holding 250 records. Two of these blocks will be used as input buffers and the third as an output buffer. Merge runs R_1 and R_2. This merge is carried out by first reading one block of each of these runs into input buffers. Blocks of runs are merged from the input buffers into the output buffer. When the output buffer gets full, it is written onto the disk. If an input buffer gets empty, it is refilled with another block from the same run. After runs R_1 and R_2 are merged, R_3 and R_4 and finally R_5 and R_6 are merged. The result of this pass is three runs, each containing 1500 sorted records or six blocks. Two of these runs are now merged using the input/output buffers set up as above to obtain a run of size 3000. Finally, this run is merged with the remaining run of size 1500 to obtain the desired sorted list (Figure 7.20). □



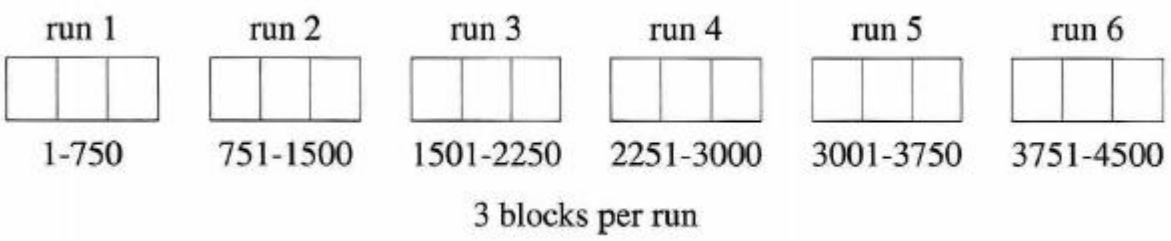


Figure 7.19: Blocked runs obtained after internal sorting

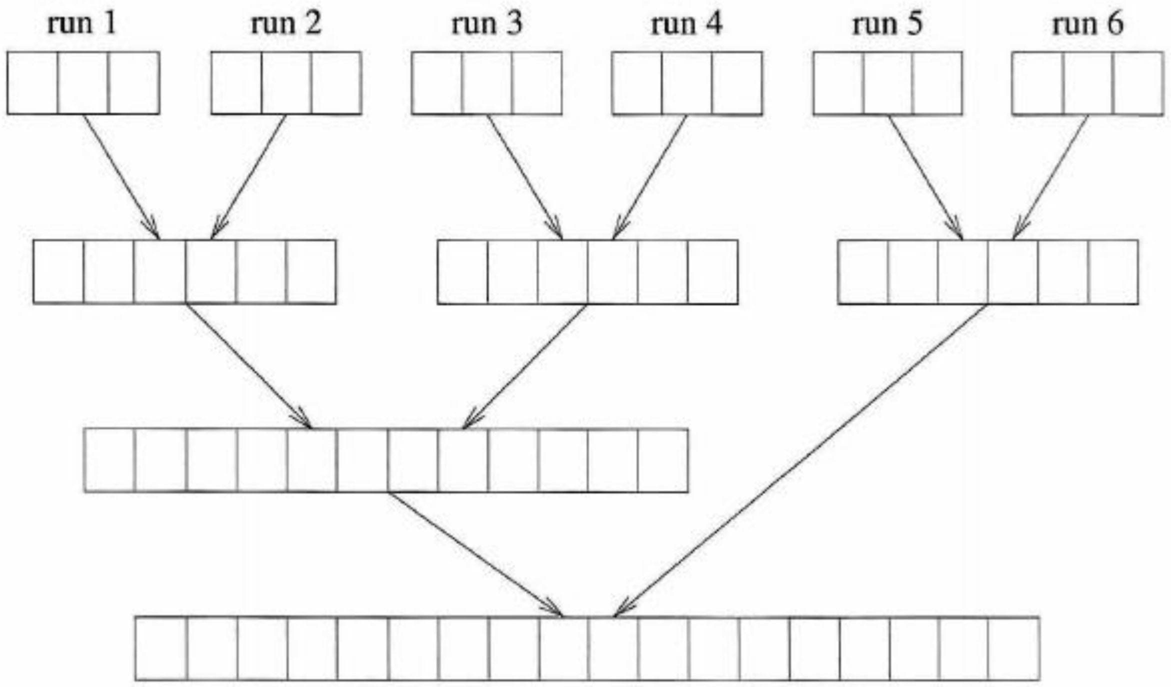
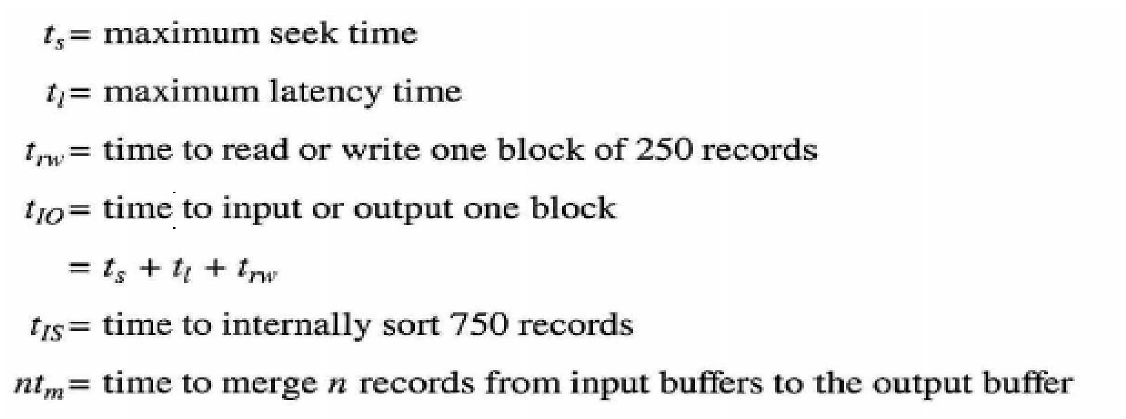


Figure 7.20: Merging the six runs

To analyze the complexity of external sort, we use the following notation:



t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

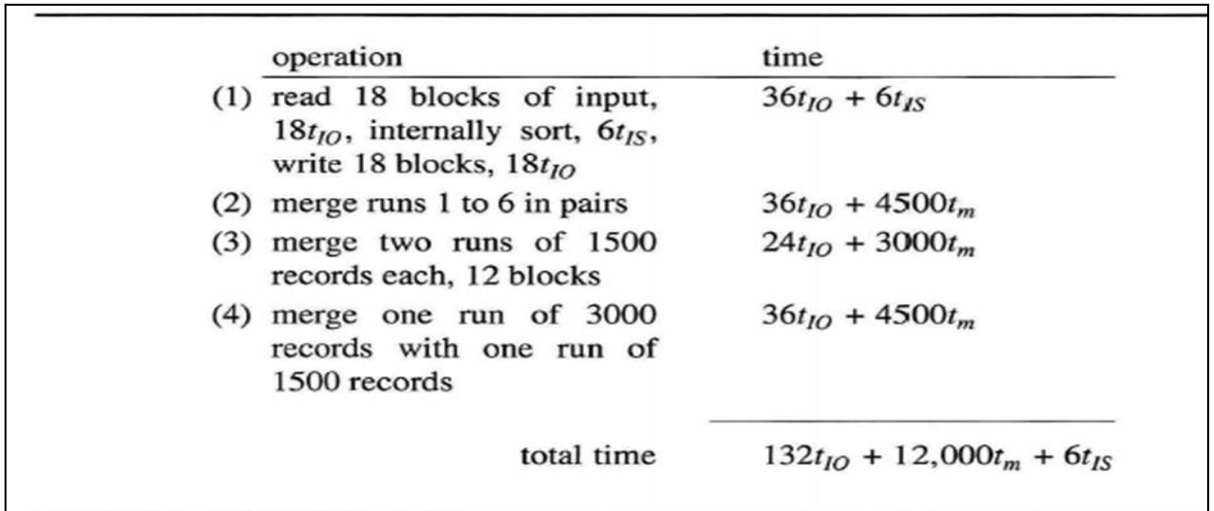
t_{IO} = time to input or output one block

$= t_s + t_l + t_{rw}$

t_{IS} = time to internally sort 750 records

nt_m = time to merge n records from input buffers to the output buffer

We shall assume that each time a block is read from or written onto the disk, the maximum seek and latency times are experienced. Although this is not true in general, it will simplify the analysis. The computing times for the various operations in our 4500-record example are given in Figure 7.21.



	operation	time
(1)	read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2)	merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(3)	merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4)	merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
	total time	$132t_{IO} + 12,000t_m + 6t_{IS}$

Figure 7.21: Computing times for disk sort example

7.10.2 k-Way Merging

The **two-way merge** function *merge* (Program 7.7) is almost identical to the merge function just described (Figure 7 .20). In general, if we start with *m* runs, the merge tree corresponding to Figure 7.20 will have $\lceil \log_2 m \rceil + 1$ levels, for a total of $\lceil \log_2 m \rceil$ passes over the data list. The number of passes over the data can be reduced by using a higher-order merge (i.e., *k*-way merge for $k \geq 2$). In this case, we would simultaneously merge *k* runs together. Figure 7.22 illustrates a **four-way merge** of 16 runs. The number of passes over the data is now two, versus four passes in the case of a two-way merge. In general, a *k*-way merge on *m* runs requires $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher-order merge.

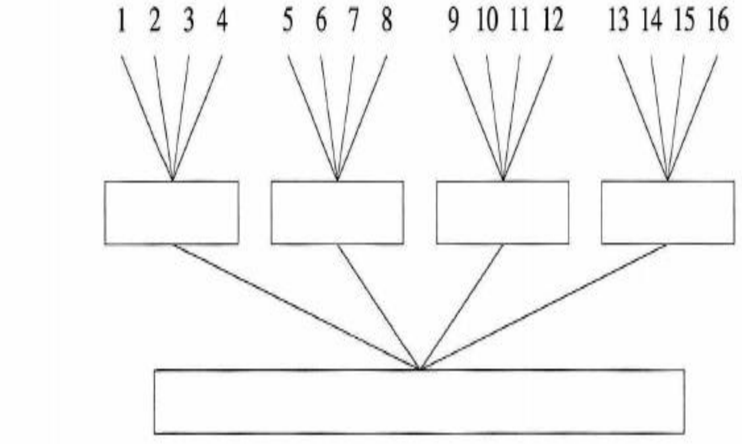


Figure 7.22: A four-way merge on 16 runs

The use of a higher-order merge, however, has some other effects on the sort. To begin with, k runs of size $s_1, s_2, s_3, \cdots, s_k$ can no longer be merged internally in $O(\sum_{1}^{k} s_i)$ time.

In a k-way merge, as in a two-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make k - 1 comparisons to determine the next record to output. The computing k time for this is $O((k-1)\sum_{1}^{k} s_i)$.

Since $\log_k m$ passes are being made, the total number of key comparisons is $n(k$ -l) $\log_k m = n(k$ -l)$\log_2 m/\log_2 k$, where n is the number of records in the list. Hence, (k- l)/$\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be outweighed by the resulting increase in CPU time needed to perform the k-way merge.

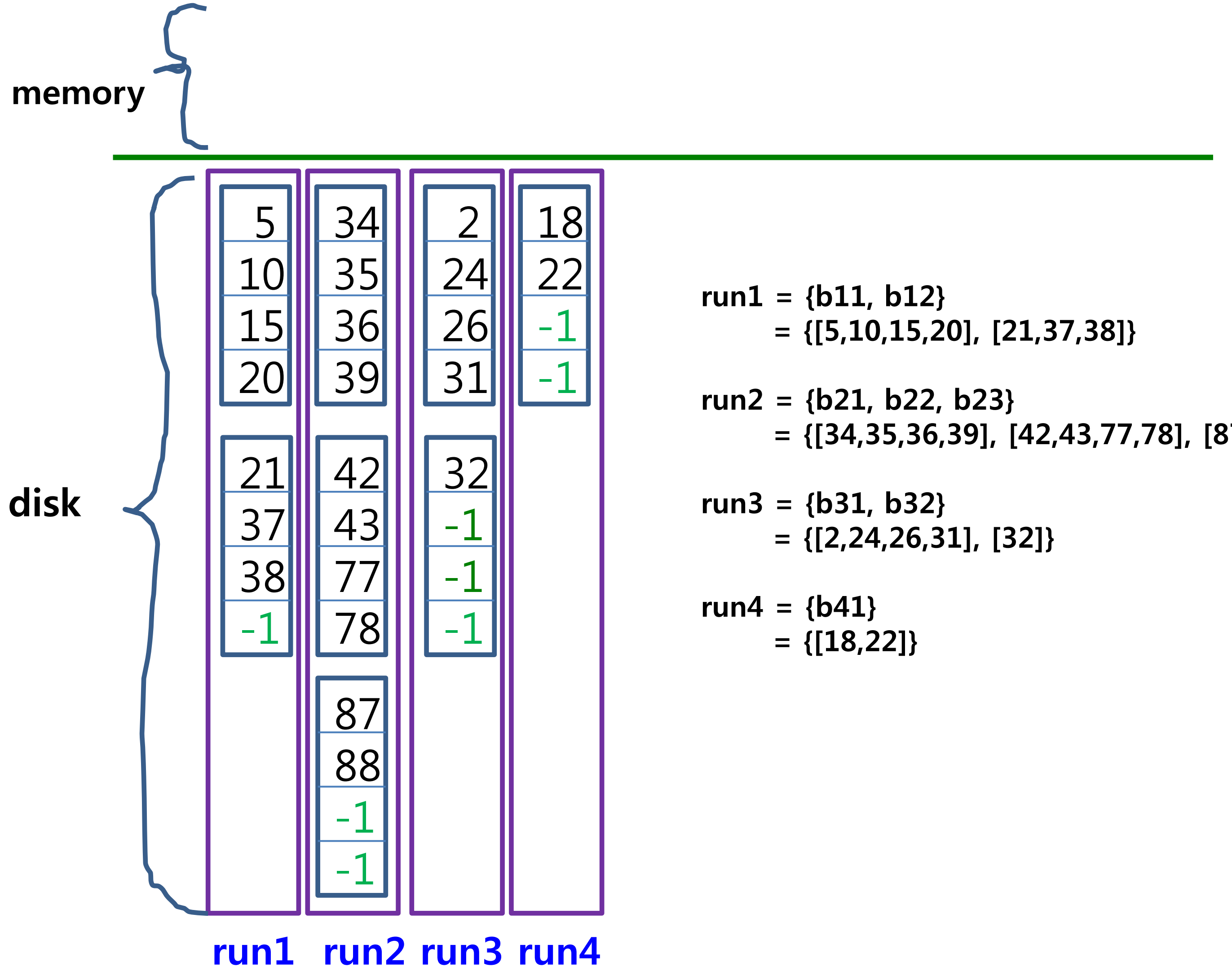
For large k (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using **a loser tree with *k* leaves** (see Chapter 5). In this case, the total time needed per level of the merge tree is $O(n \log_2 k)$. Since the number of levels in this tree is $O(1og_k m)$, the asymptotic internal processing time becomes $O(n\ 1og_2 k \log_k m)$ = **$O(n\log_2 m)$**. This is independent of k.

In going to a higher-order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $1\text{og}_k m$ passes. This is so because the number of input buffers needed to carry out a *k*-way merge increases with **k**. Although $k + 1$ buffers are sufficient, in the next section we shall see that the use of $2k + 2$ buffers is more desirable. Since the internal memory available is fixed and independent of *k*, the buffer size must be reduced as *k* increases. This in tum implies a reduction in the block size on disk. With the reduced block size, each pass over the data results in a greater number of blocks being written or read. This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain *k* value the input/output time will increase despite the decrease in the number of passes being made. The optimal value for *k* depends on disk parameters and the amount of internal memory available for buffers.

4-way merge sort

The mark for the end of a run: Sentinel Value = -1

run = a sequence of blocks



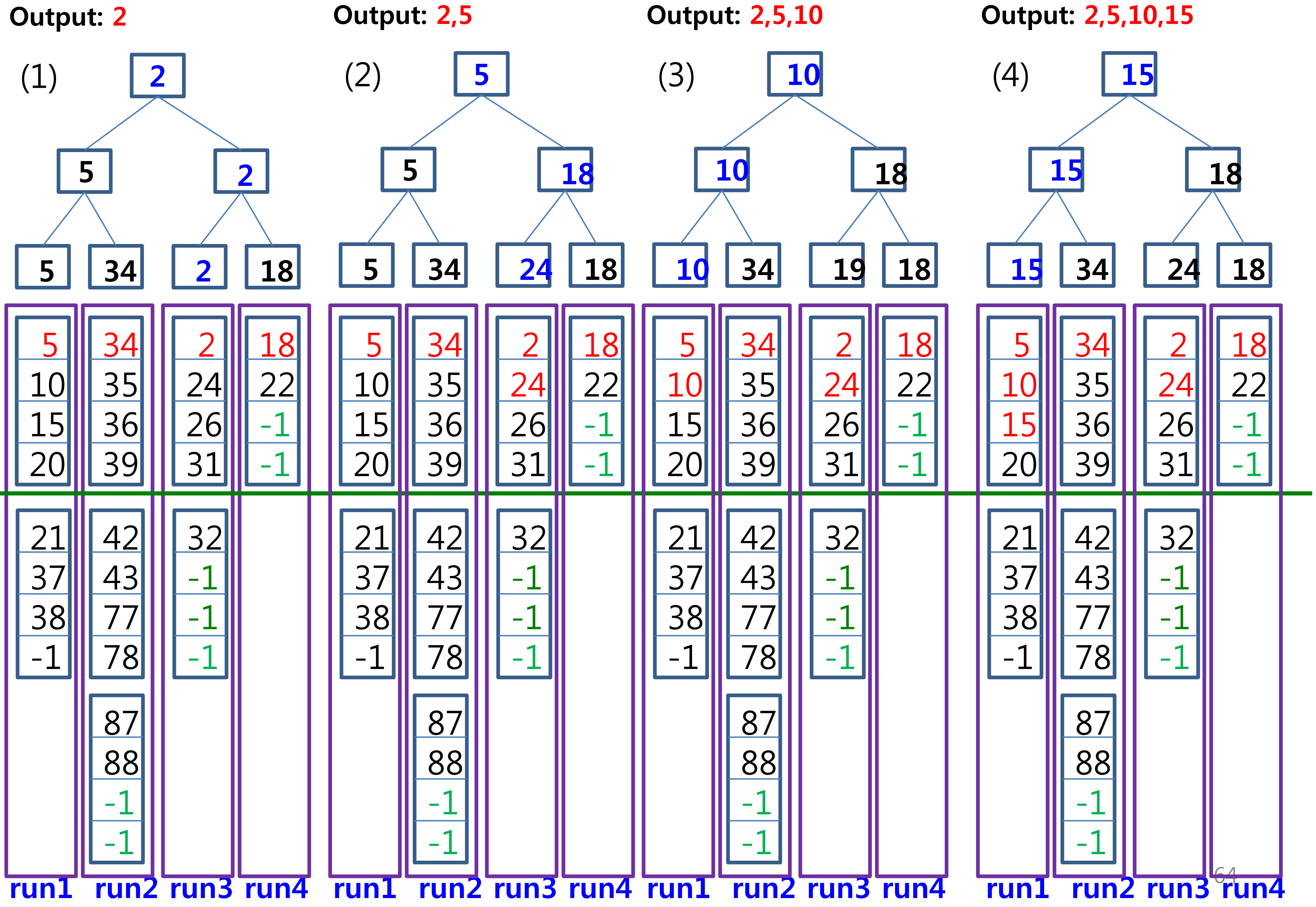
run1 = {b11, b12}
= {[5,10,15,20], [21,37,38]}

run2 = {b21, b22, b23}
= {[34,35,36,39], [42,43,77,78], [87,88]}

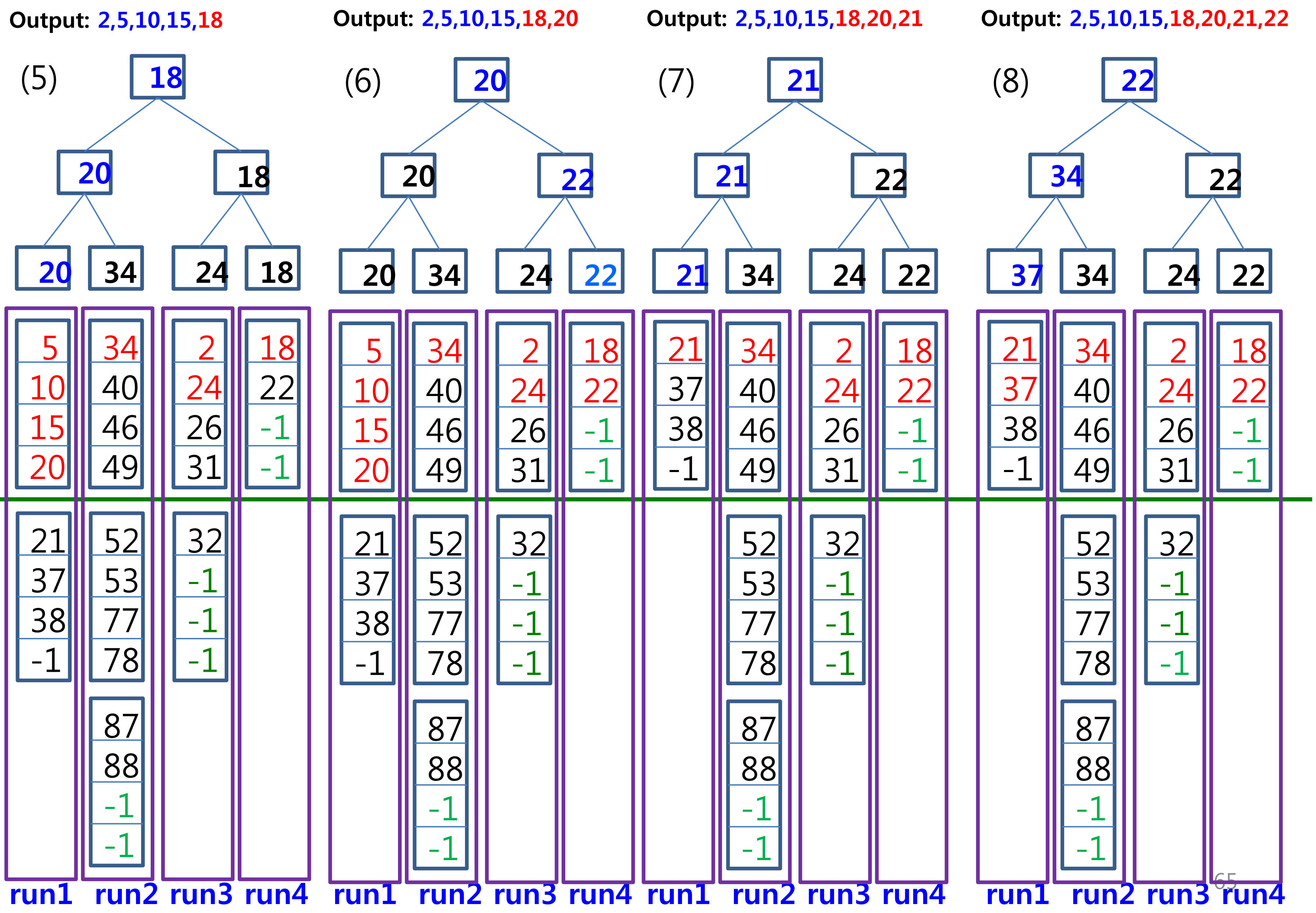
run3 = {b31, b32}
= {[2,24,26,31], [32]}

run4 = {b41}
= {[18,22]}

4-way merge sort



4-way merge sort



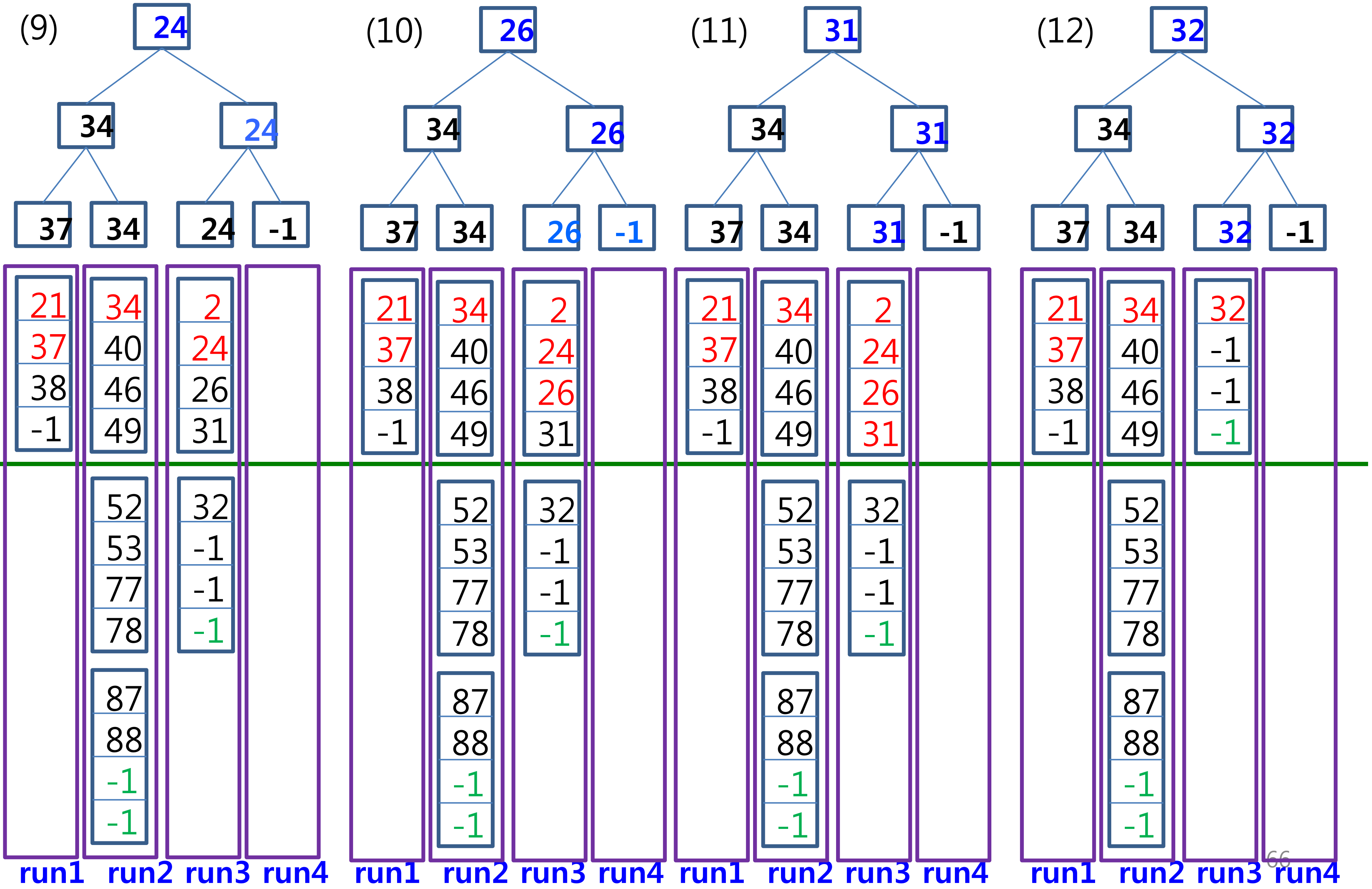
4-way merge sort

Output: 2,5,10,15,18,20,21,22,24

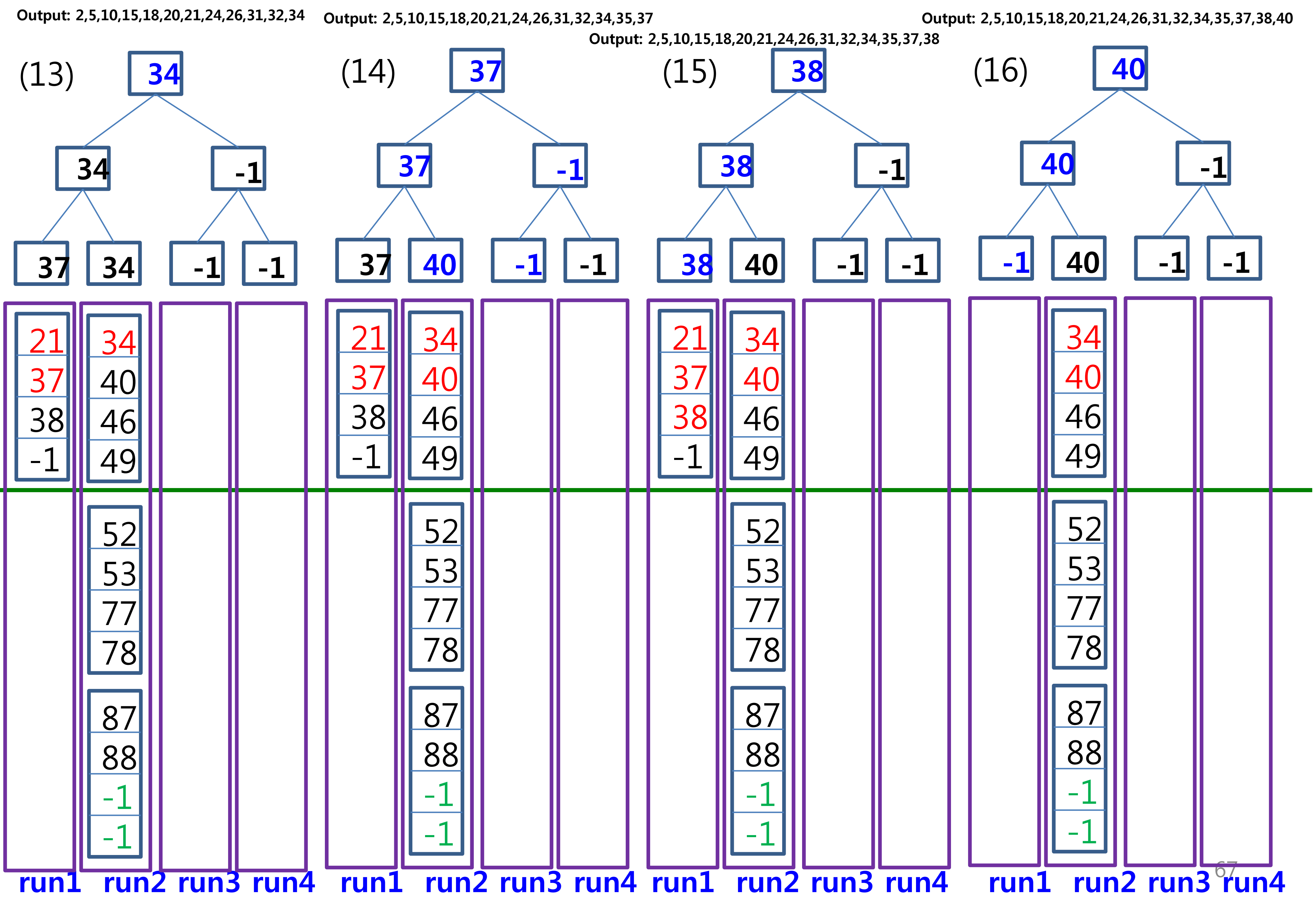
Output: 2,5,10,15,18,20,21,22,24,26

Output: 2,5,10,15,18,20,21,24,26,31

Output: 2,5,10,15,18,20,21,24,26,31,32



4-way merge sort

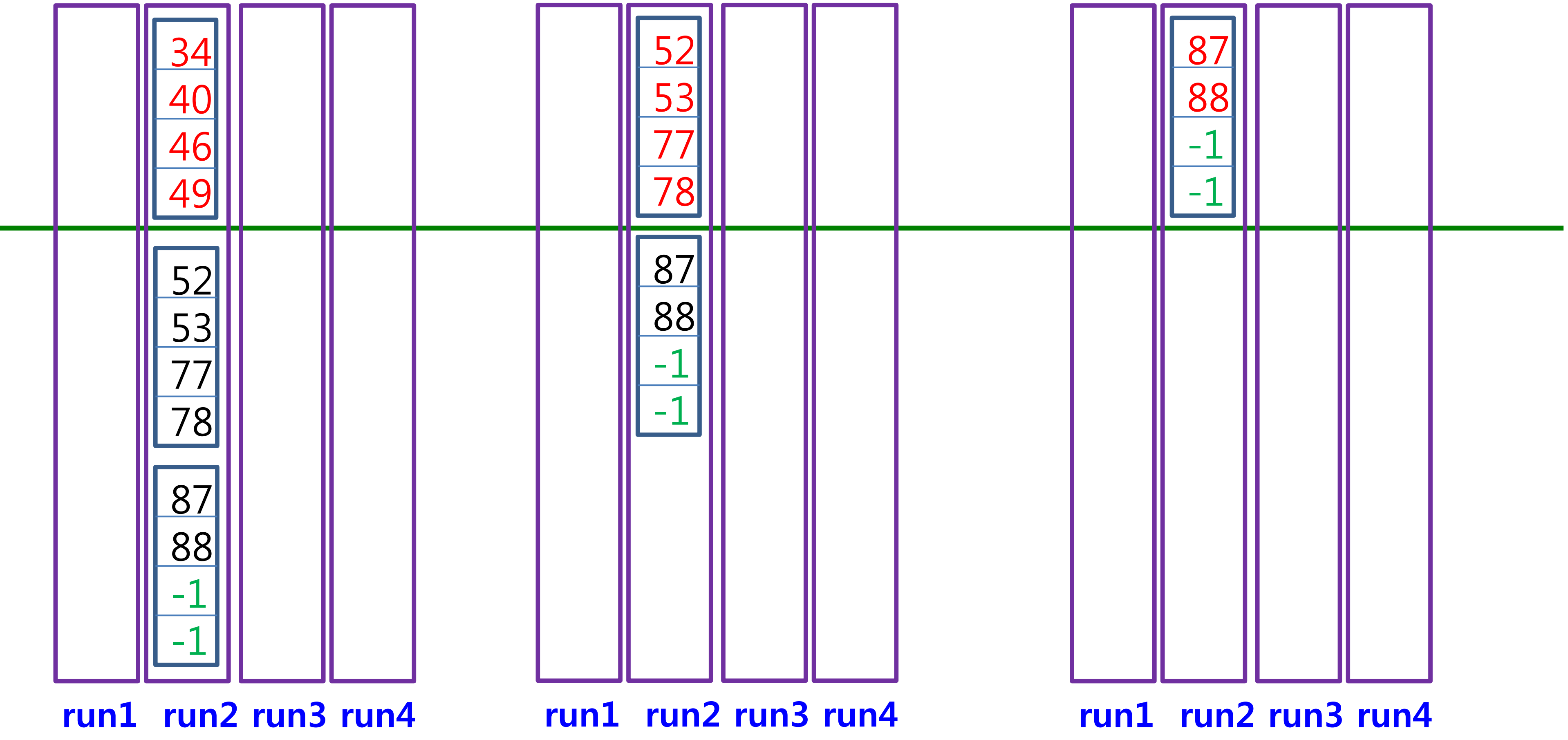


4-way merge sort

Output: 2,5,10,15,18,20,21,24,26,31,32,34,35,37,38,40,46,49

Output: 2,5,10,15,18,20,21,24,26,31,32,34,35,37,38,40,46,49,
52,53,77,78

Output: 2,5,10,15,18,20,21,24,26,31,32,34,35,37,38,40,46,49,
52,53,77,78,87,88



EXERCISE (External Sorting with 4-way merge sort) 10점

우리는 제 5장의 과제 4에서 Winner Tree를 array를 이용하여 8-way merge sort를 수행하는 것을 실습하였다. 그를 기초로 아래 작업을 수행하는 C 프로그램을 작성하시오.

(1) Data Generation
   1) 1 ~ 2,000,000 범위의 200만개의 정수 key들을 random number generator를 이용하여 생성한다.
   2) 생성된 key들을 File origin.dat에 기록한다.

(2) Initial Run Generation Phase
   1) File origin.dat에서 차례로 10,000개씩의 key들을 읽어 정렬하여 File temp1.dat에 차례로 기록한다. 즉, 초기에 모두 200개의 run들이 생성된다.
   2) 기록되는 각 run은 10,000개의 key들을 가지며 그 key들은 10개의 block들에 저장되는데 각 block의 크기는 4096 bytes이다. 각 block의 구성은 다음과 같다. 하나의 key의 크기를 4-byte로 할 경우이다.

▪ 그 block이 key들로 가득 찬 경우(첫 9개의 block들):	[K0, K1, …, K1023]
▪ 그 block의 일부만 key를 저장한 경우(마지막 block):	[K0, K1, …, Kx, -1, -1, …, -1]

      즉, 하나의 block 내에서 -1을 만나면 그 block은 그 run의 마지막 block이며 그 block의 첫 -1 앞 까지가 (만약 key가 있다면) 유효한 key들임을 나타낸다.

(3) Merge Phase : 4-way merge를 수행하여 정렬된 결과를 위에서 제시한 block 단위로 file final.dat에 저장한다.
   1) Temp1.dat의 초기 200개의 run들을 4 개씩 block 단위로 차례로 읽어 merge하여 50개의 run들을 생성하여 temp2.dat에 저장한다. 생성된 각 run의 크기는 40 block이다.
   2) Temp2.dat의 50개의 run들을 4 개씩 block 단위로 차례로 읽어 merge하여 13개의 run들을 생성하여 temp1.dat에 저장한다. 이 경우, 첫 12개의 run들은 각각 4개의 run들을 merge한 것이며 크기가 160 block이 되며 마지막 하나의 run은 2개의 run들을 merge한 것으로써 80 block이다.
   3) Temp1.dat의 13개의 run들을 4 개씩 block 단위로 차례로 읽어 merge하여 4개의 run들을 temp2.dat에 저장한다. 이 경우, 첫 3개의 run들은 각각 4개의 run들을 merge한 것이며 크기가 640 block이 되며 마지막 하나의 run은 1개의 run을 그대로 copy한 것으로 크기가 80 block이다.
   4) Temp2.dat의 4개의 run들을 block 단위로 차례로 읽어 merge하여 하나의 run을 temp1.dat에 저장한다. 그 run의 크기는 2000 block이다.
   5) Temp1.dat의 file name을 final.dat로 변경한다.

(4) file final.dat의 내용을 출력한다.

과제 4 (Section 5.8 SELECTION TREES)

아래 작업을 수행하는 C 프로그램을 작성하시오.

Figure 5.32의 Winner Tree를 array를 이용하여 8-way merge sort를 수행하고자 한다. 아래 사항들을 수행하시오.

(1) 8개의 run을 다음과 같이 작성하시오. 각 run을 하나의 배열로 수행한다. Run1 부터 run 8 까지 차례로 다음과 같이 레코드들을 각 run에 차례로 insert 한 후 각 run에 보관된 값들을 출력하시오. [1점]

0	1	2	3	4	5	6	7
15	14	13	12	11	10	9	8
16	17	18	19	20	21	22	23
31	30	29	28	27	26	25	24
32	33						

의 방법으로 100 까지 각 run에 배치하시오.

(2) 그 run들을 대상으로 Figure 5.32의 Winner Tree를 array를 이용하여 8-way merge sort를 수행하고자 한다. 그 결과를 배열 result에 넣으시오. 정렬이 모두 끝난 후 배열 result의 내용을 출력하시오. [3점]

EXERCISE (External Sorting with 4-way merge sort) : Homework

아래 작업을 수행하는 C 프로그램을 작성하시오.

[Programming Project] 우리는 아래 element 자료형의 data를 다룬다.

```
struct student {
    int id;
    char name[40];
    char address[100];
};
typedef struct student element;
```

(1) File test.dat를 생성한다.

(2) 우리학교의 총 학생들의 수는 6900명이다. 그 학생들의 정보를 생성하여 file test.dat에 저장한다. 학생들의 생성에서 id는 random number generator를 이용하여 생성하며 name은 그 숫자를 이용하여 생성하며 address 또한 그 숫자를 이용하여 생성한다. 편의상 여러 학생들이 동일 id를 가지는 것을 허용한다.

(3) File test.dat에서 생성된 학생들의 정보를 읽어 print하여 정확하게 생성되었음을 확인한다.

(4) 그 학생들의 정보를 external sorting하여 정렬한다. 정렬된 자료는 file sort.dat에 저장한다. 그 정렬은 반드시 아래 방법으로 수행되어야 한다.

    1) 60명의 정보를 하나의 block(묶음)으로 처리한다. 즉, 전체는 6900/60 = 115 개의 block이 있다.
    2) 5개의 block을 읽어 (즉, 300명의 정보를 읽어) 그들을 정렬한 후 disk에 기록함으로써 하나의 run을 생성하도록 하는 방법으로 전체 115/5 = 23 개의 run을 생성하여 file sort.dat에 각 run을 차례로 저장한다.
    3) File sort.dat에 저장된 23 개의 run을 대상으로 그들을 4-way merge를 수행하여 (즉, 4개의 run을 읽어 하나의 run을 생성한다) 그들을 정렬하여 file sort.dat에 저장한다.

(5) File sort.dat에서 정렬된 학생들의 정보를 읽어 print하여 정확하게 정렬되었음을 확인한다.

제약사항: disk read/write 연산에서 한번에 1 block의 크기로 수행하여야 한다.

파일 처리

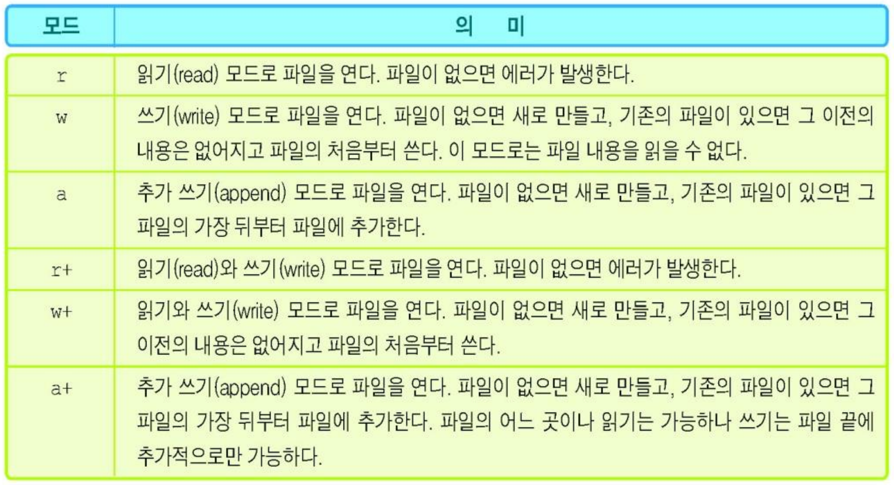
- **텍스트 (text) 파일**
  텍스트 파일의 내용은 모두 지정된 문자 코드(일반적으로 아스키 코드) 값을 갖고 있어, 텍스트 편집기(editor)로 그 내용을 확인할 수 있을 뿐만 아니라 인쇄가 가능하다.
  - ✓ 텍스트 파일을 위한 파일 열기: FILE *f = fopen(fileName, "w");
  - ✓ 파일 입출력을 처리하려면 함수 fprintf()와 fscanf()를 이용

- **이진 (binary) 파일**
  이진 파일은 C 언어의 자료형을 유지하면서 바이트 단위로 저장하는 파일
  이진 파일의 대표적인 예가 실행파일. 이진 파일은 일반 편집기로는 그 내용 확인 불가능
  - ✓ 이진 파일을 위한 파일 열기 모드는 파일 열기 모드에서 문자 'b'를 추가: FILE *f = fopen(fileName, "wb");
  - ✓ 이진 모드로 입출력을 처리하려면 함수 fwrite()와 fread()를 이용

```
size_t  fwrite(const void *, size_t, size_t, FILE *);
size_t  fread(void *, size_t, size_t, FILE *);
```

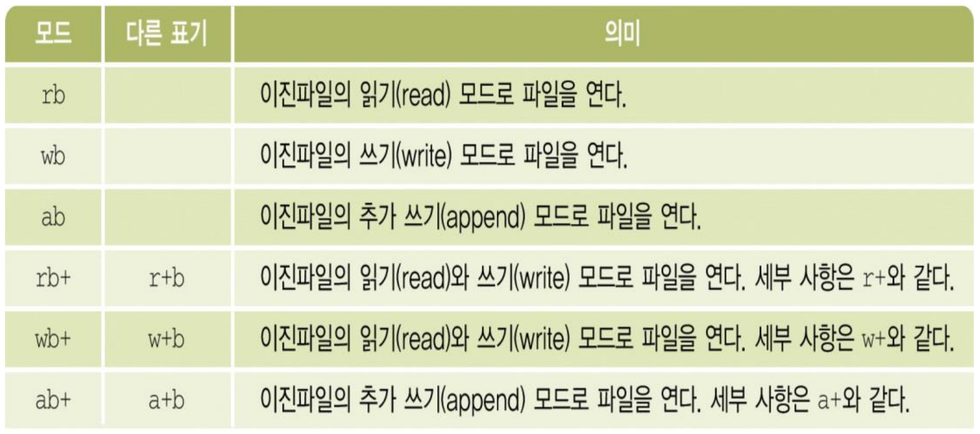
  - ➢ 첫 번째 인자는 출력될 자료의 주소 값,
  - ➢ 두 번째 인자는 출력될 자료의 바이트 크기,
  - ➢ 세 번째 인자는 출력될 단위 자료의 묶음 개수,
  - ➢ 마지막 인자는 출력될 파일 포인터
  - ➢ 반환값은 파일에 출력된 단위 자료형의 개수

❖ 함수 fwrite()에 의하여 출력된 자료는 함수 fread()를 이용하여 입력해야 그 자료형 유지가 가능



모드	의 미
r	읽기(read) 모드로 파일을 연다. 파일이 없으면 에러가 발생한다.
w	쓰기(write) 모드로 파일을 연다. 파일이 없으면 새로 만들고, 기존의 파일이 있으면 그 이전의 내용은 없어지고 파일의 처음부터 쓴다. 이 모드로는 파일 내용을 읽을 수 없다.
a	추가 쓰기(append) 모드로 파일을 연다. 파일이 없으면 새로 만들고, 기존의 파일이 있으면 그 파일의 가장 뒤부터 파일에 추가한다.
r+	읽기(read)와 쓰기(write) 모드로 파일을 연다. 파일이 없으면 에러가 발생한다.
w+	읽기와 쓰기(write) 모드로 파일을 연다. 파일이 없으면 새로 만들고, 기존의 파일이 있으면 그 이전의 내용은 없어지고 파일의 처음부터 쓴다.
a+	추가 쓰기(append) 모드로 파일을 연다. 파일이 없으면 새로 만들고, 기존의 파일이 있으면 그 파일의 가장 뒤부터 파일에 추가한다. 파일의 어느 곳이나 읽기는 가능하나 쓰기는 파일 끝에 추가적으로만 가능하다.

표 17.1 파일 열기 함수 fopen( )의 모드 종류



모드	다른 표기	의미
rb		이진파일의 읽기(read) 모드로 파일을 연다.
wb		이진파일의 쓰기(write) 모드로 파일을 연다.
ab		이진파일의 추가 쓰기(append) 모드로 파일을 연다.
rb+	r+b	이진파일의 읽기(read)와 쓰기(write) 모드로 파일을 연다. 세부 사항은 r+와 같다.
wb+	w+b	이진파일의 읽기(read)와 쓰기(write) 모드로 파일을 연다. 세부 사항은 w+와 같다.
ab+	a+b	이진파일의 추가 쓰기(append) 모드로 파일을 연다. 세부 사항은 a+와 같다.

표 13.3 이진파일 열기 함수 fopen( )의 모드 종류

자료	모드	종류	표준 입출력	파일 입출력
문자	텍스트 이진	입력	int getchar(void)	int getc(FILE *) int fgetc(FILE *)
		출력	int putchar(int)	int putc(int, FILE *) int fputc(int, FILE *)
문자열	(텍스트)	입력	char * gets(char *)	char * fgets(char *, int, FILE *)
		출력	int puts(const char *)	int fputs(const char *, FILE *,)
형식 자료	(텍스트)	입력	int scanf(const char *, ...)	int fscanf(FILE *, const char *, ...)
		출력	int printf(const char *, ...)	int fprintf(FILE *, const char *, ...)

표 17.5 자료에 따른 입출력 함수

표준 파일	키워드	장치
표준입력	stdin	키보드
표준출력	stdout	모니터 화면
표준에러	stderr	모니터 화면

- **char * fgets(char *, int, FILE *);** 문자열을 개행문자(**\n**)까지 읽어 개행문자도 함께 입력 문자열에 저장한다. 첫 번째 인자는 문자열이 저장될 문자 포인터이고, 두 번째 인자는 입력할 문자의 최대 수이며, 세 번째 인자는 입력 문자열이 저장될 파일이다.
- **int fputs(char *, FILE *);** 문자열을 그대로 출력 한다. 첫 번째 인자는 출력될 문자열이 저장된 문자 포인터이고, 두 번째 인자는 문자열이 출력되는 파일이다.
- **int feof(FILE *);** 파일의 위치가 파일의 마지막(**end of file**)인지를 검사하여, 파일의 마지막이면 **0**이 아닌 값을, 파일의 마지막이 아니면 **0**을 반환한다.
- **feof(stdin)**의 검사는 **ctrl+Z**의 입력을 검사하는 것이다. 이 경우, 사용자의 입력을 **scanf(), fscanf()**로 수행한 경우에는 동작하지 않는다. **gets(), fgets()**로 수행한 경우에 정상적으로 동작한다.

자료	모드	종류	파일 입출력
블록	이진	입력	size_t fread(void *, size_t, size_t, FILE *)
		출력	size_t fwrite(const void *, size_t, size_t, FILE *)
정수(int)	이진	입력	int getw(FILE *)
		출력	int putw(int, FILE *)

표 17.6 블록과 정수의 파일 입출력 함수

int sscanf(const char *, const char *, …);

- 문자열에서 자료를 추출하는 함수이다. 헤더 파일 **stdio.h** 파일을 포함
- 첫 번째 인자: 탐색될 문자열이 저장된 문자 포인터,두 번째 인자: 입력되는 문자열, 다음 인자들: 입력될 변수 목록
- The function returns the total number of items successfully matched, which can be less than the number requested.

함수	기능
int fseek(FILE *fptr, long offset, int mode)	파일 fptr에서 기준점 mode에서 offset만큼 떨어진 곳으로 파일 포인터를 위치하는 함수
long ftell(FILE *)	파일의 현재 파일 포인터를 반환하는 함수
void rewind(FILE *)	파일의 현재 포인터를 0 위치(파일의 시작점)로 이동하는 함수

기호	값	의미
SEEK_SET	0	파일의 시작 위치
SEEK_CUR	1	파일의 현재 위치
SEEK_END	2	파일의 끝 위치

기 능	함수 원형
파일 삭제	int remove(const char *)
파일 이름 바꾸기	int rename(const char *, const char *)

rename("oldname.txt", "newname.txt");

표 17.7 파일 삭제 함수