Chap 7. Sorting (1)

Contents

7.1 Motivation

- Sorting
  - Rearrange *n* elements into ascending(decending) order
    - 7, 3, 6, 2, 1 → 1, 2, 3, 6, 7

- Tow 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:

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

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

```
#define COMPARE(x, y) (((x) < (y)) ? -1 : ((x) == (y)) ? 0 :1 )
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) $K_{\sigma_S(i)} \leq K_{\sigma_S(i+1)}$, $1 \leq i \leq n\text{-}1$

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

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

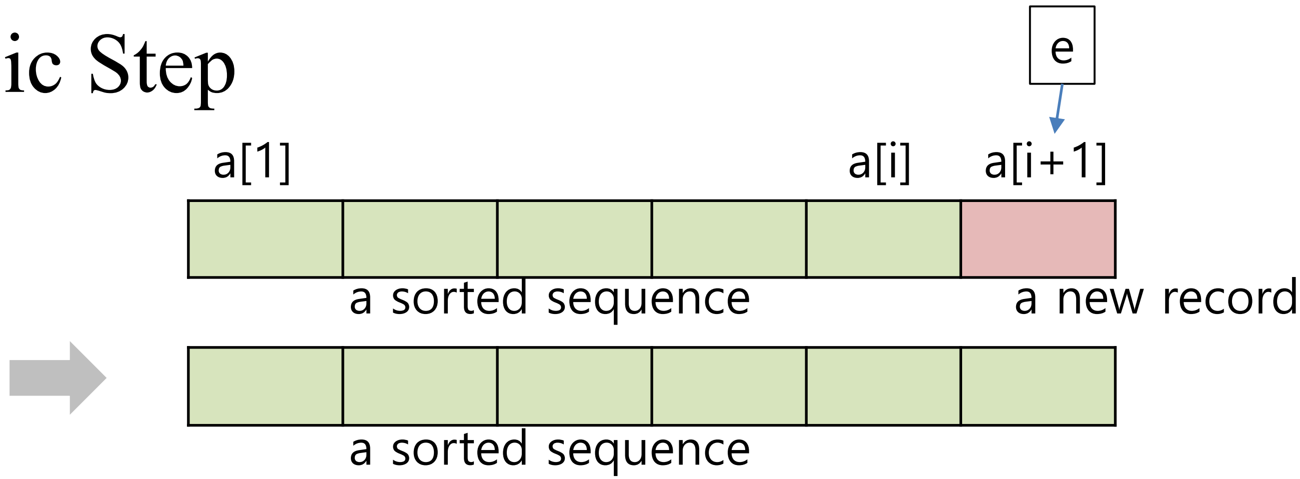
  – stable sorting : 2_1, 2_2, 3, 6, 7, 8

  – 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

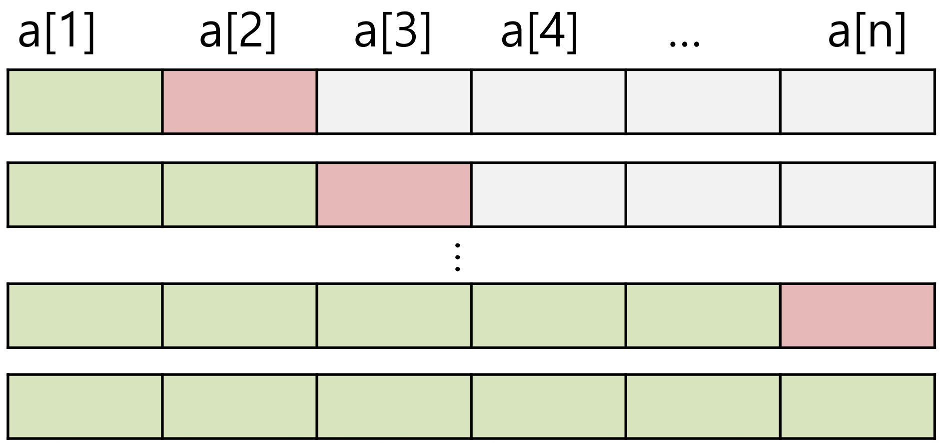
- Basic Step




```
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 $i = j\text{-}1 = 1, \ldots, n\text{-}1$
  - $O(\sum_{i=1}^{n-1}(i+1)) = O(n^2)$

<Method 2>

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

  iff $R_i < \max_{1 \le j < i}\{Rj\}$

- 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 ($n \leq 30$)

Shell Sort: A Better Insertion Sort

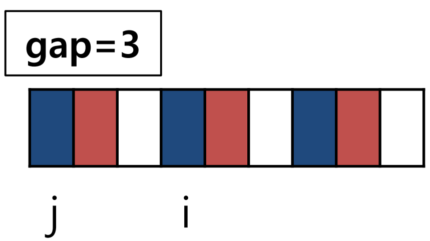
- Shell sort is a variant of insertion sort
  - Reduces work by moving elements farther earlier
- Performs:
  - $O(n^2)$ comparisons
- Divide and conquer approach to insertion sort
  - Sort many smaller subarrays using insertion sort
  - Sort progressively larger arrays
  - Finally sort the entire array
- These subarrays are elements separated by a *gap*
  - Start with *large gap*
  - Decrease the gap on each "pass"

Shell Sort: The Varying Gap

	10	8	6	20	4	3	22	1	0	15	16
	10					3					16
		8					22				
gap 5			6					1			
				20					0		
					4					15	
	3					10					16
		8					22				
			1					6			
				0					20		
					4					15	
	3	8	1	0	4	10	22	6	20	15	16
	3			0			22			15	
gap 3		8			4			6			16
			1			10			20		
	0			3			15			22	
		4			6			8			16
			1			10			20		
	0	4	1	3	6	10	15	8	20	22	16
gap 1	0	1	3	4	6	8	10	15	16	20	22

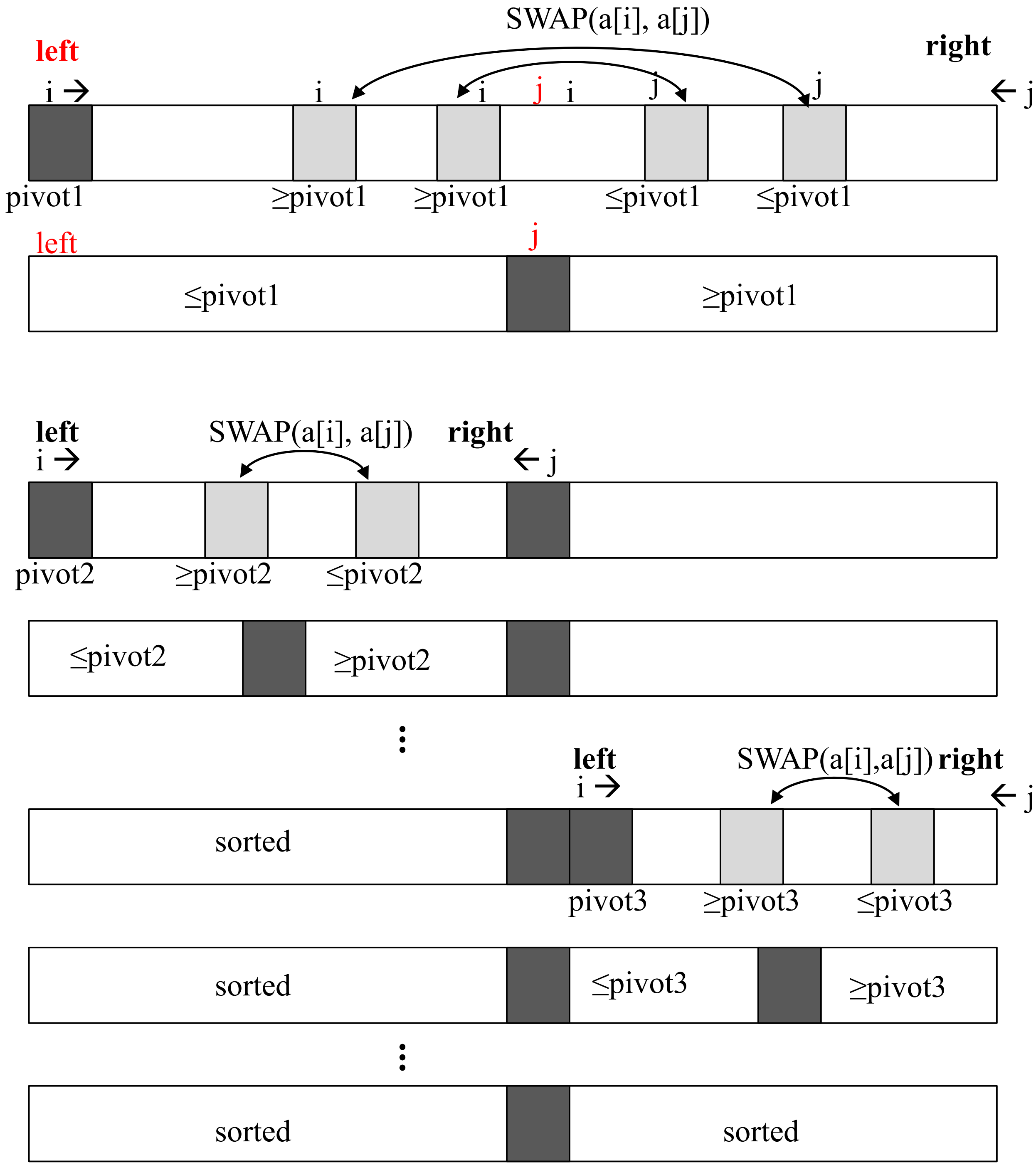
Shell Sort Algorithm

```
// insertion sort with gap
// range of sort is from first to last
inc_insertion_sort(int list[], int first, int last, int gap)
// sublist sorting
{
        int i, j, key;
        for(i=first+gap; i<=last; i=i+gap){
                key = list[i];  // i-th element is sorted
                for(j=i-gap; j>=first && key<list[j];j=j-gap)
                        list[j+gap]=list[j];
                list[j+gap]=key;
        }
}
//
void shell_sort( int list[], int n )   // n = size
{
        int i, gap;
        for( gap=n/2; gap>0; gap = gap/2 ) {
                if( (gap%2) == 0 ) gap++;
                for(i=0;i<gap;i++)       // number of sublists : gap
                        inc_insertion_sort(list, i, n-1, gap);
        }
}
```



7.3 Quick Sort

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




```
do{  …
   if(i<j) SWAP(a[i], a[j]);
}while( i<j )
SWAP(a[left], a[j]);
```

```
#define SWAP(x, y, temp) ((temp) = (x), (x) = (y), (y)=(temp))


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) && (i < right));
                   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);
                printList(a, num);
                quickSort(a, left, j - 1);
                quickSort(a, j + 1, right);
        }
}
```

- 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 : $T(n)$

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

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

7.5 Merge Sort

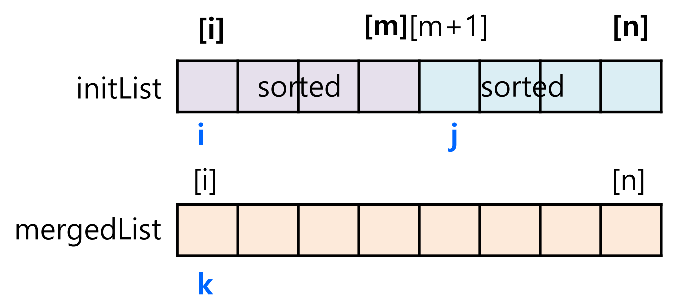
- Merge *two sorted lists* to *a single sorted list*.
  - initList[i:m] and initList[m+1:n]➔mergedList[i:n]
- Example

	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.

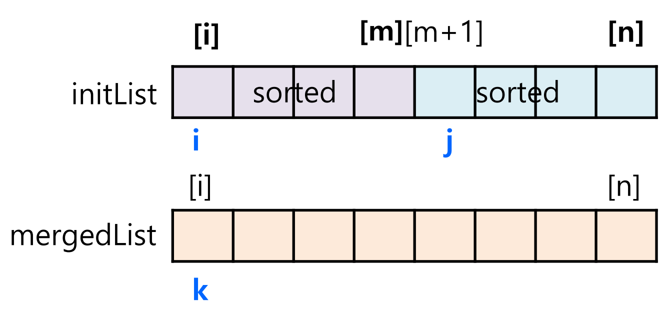
```
void merge(element initList[], element mergedList[],
           int i, int m, int n)
{/*  the sorted lists initList[i:m] and initList[m+1:n] are
     merged to obtain the sorted list mergedList[i:n] */
  int j,k,t;
  j = m+1;          /* index for the second sublist */
  k = i;            /* index for the merged list */

  while (i <= m && j <= n) {
    if (initList[i].key <= initList[j].key)
      mergedList[k++] = initList[i++];
    else
      mergedList[k++] = initList[j++];
  }
  if (i > m)
  /* mergedList[k:n] = initList[j:n] */
    for (t = j; t <= n; t++)
      mergedList[t] = initList[t];
  else                K++
  /* mergedList[k:n] = initList[i:m] */
    for (t = i; t <= m; t++)
      mergedList[k+t-i] = initList[t];
}                      K++
```

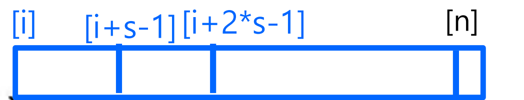


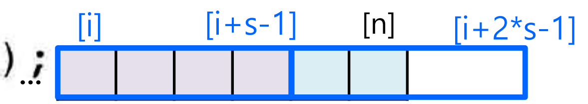
Program 7.7: Merging two sorted lists

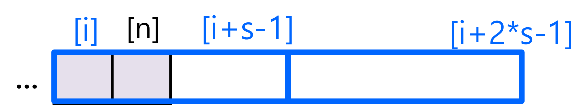
- **Analysis of *merge*:**
  - Total increment in k is n-i+1.
  - O(n-i+1) ➔ O(n)
  - Stable sorting



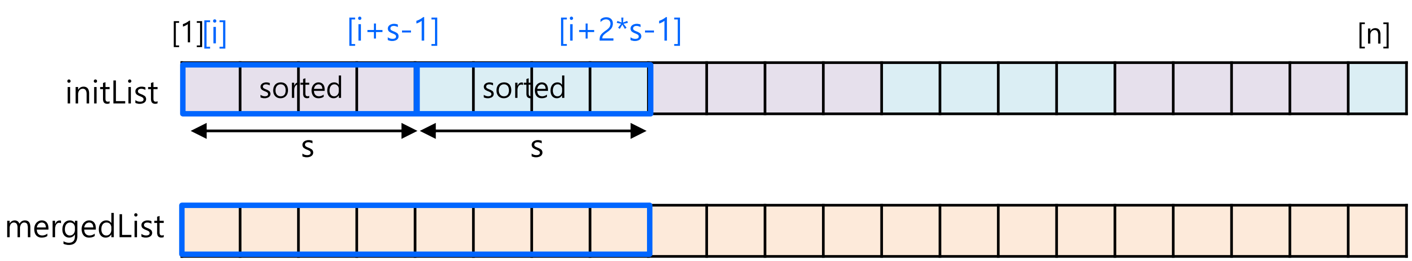
```
void mergePass(element initList[], element mergedList[],
                    int n, int s)
{/* perform one pass of the merge sort, merge adjacent
    pairs of sorted segments from initList[] into mergedList[],
    n is the number of elements in the list, s is
    the size of each sorted segment */
  int i,j;      i+2*s-1 <= n
  for (i = 1; i <= n - 2 * s + 1; i += 2 * s)
    merge(initList,mergedList,i,i + s - 1,i + 2 * s - 1);
(2) if (i + s - 1 < n)
    merge(initList,mergedList,i,i + s - 1,n);
(3) else
    for (j = i; j <= n; j++)
      mergedList[j] = initList[j];
}
```







Program 7.8: A merge pass

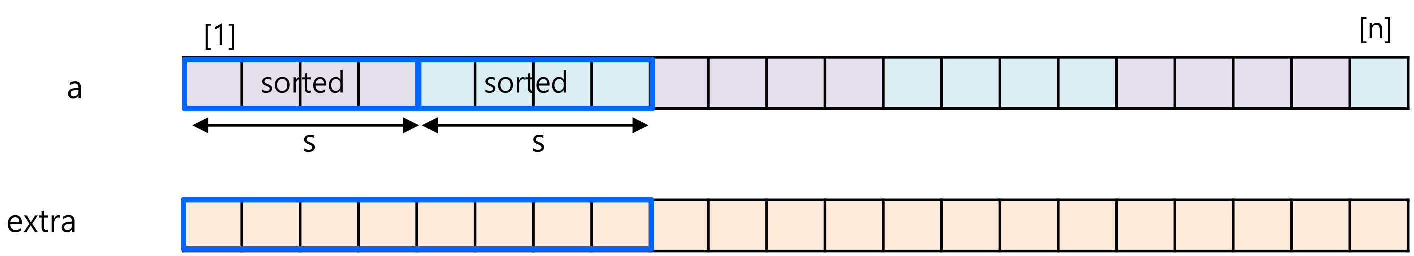


```
void mergeSort(element a[], int n)
{/* sort a[1:n] using the merge sort method */
   int s = 1; /* current segment size */
   element extra[MAX_SIZE];

   while (s < n) {
      mergePass(a, extra, n, s);
      s *= 2;
      mergePass(extra, a, n, s); //2*s 만큼 정렬됨
      s *= 2;
   }
}
```

s>n ?

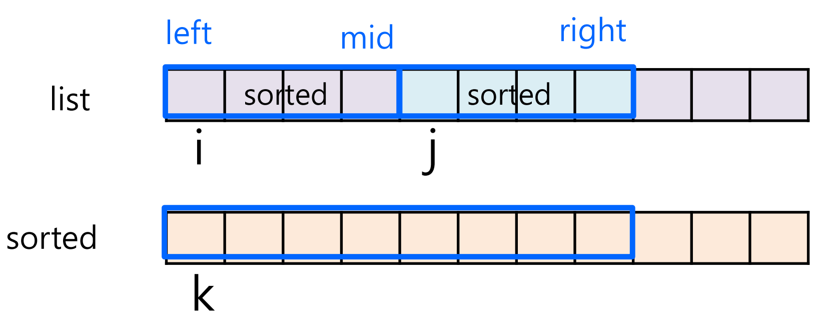
Program 7.9: Merge sort



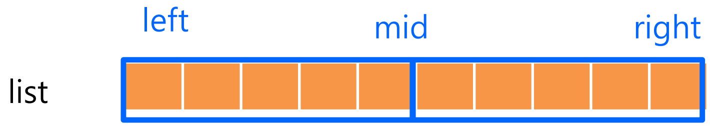
```
int sorted[MAX_SIZE];   // additional space

/* i left index of sorted list
   j right index of sorted list
   k index  */
void merge(int list[], int left, int mid, int right)
{
int i, j, k, l;
i=left;  j=mid+1;  k=left;

/* merge of sorted lists */
while(i<=mid && j<=right){
        if(list[i]<=list[j])
                sorted[k++] = list[i++];
        else
                sorted[k++] = list[j++];
}
if(i>mid)/* copy of remained elemnets */
        for(l=j; l<=right; l++)
                sorted[k++] = list[l];
else/* copy of remained elemnets */
        for(l=i; l<=mid; l++)
        sorted[k++] = list[l];
/* copy  sorted[] to  list[] */
for(l=left; l<=right; l++)
        list[l] = sorted[l];
}
```



```
void merge_sort(int list[], int left, int right)
{
int mid;
if(left<right){
      mid = (left+right)/2;
      merge_sort(list, left, mid);     /* sort partioned lists */
      merge_sort(list, mid+1, right); /* sort partioned lists */
      merge(list, left, mid, right);     /*merge */
      }
}
```



선택 C:\WINDOWS\system32\cmd.exe

```
<<<<<<<<<<<<< Input List >>>>>>>>>>>>>
8 3 13 6 2 14 5 9 10 1 7 12 4

<<<<< executing recursive merge sort >>>>>
call merge_sort(list, left=1, mid=7)
call merge_sort(list, left=1, mid=4)
call merge_sort(list, left=1, mid=2)
call merge_sort(list, left=1, mid=1)
call merge_sort(list, mid+1=2, right=2)
call merge(list, left=1, mid=1, right=2)
result : 3 8 13 6 2 14 5 9 10 1 7 12 4

call merge_sort(list, mid+1=3, right=4)
call merge_sort(list, left=3, mid=3)
call merge_sort(list, mid+1=4, right=4)
call merge(list, left=3, mid=3, right=4)
result : 3 8 6 13 2 14 5 9 10 1 7 12 4

call merge(list, left=1, mid=2, right=4)
result : 3 6 8 13 2 14 5 9 10 1 7 12 4

call merge_sort(list, mid+1=5, right=7)
call merge_sort(list, left=5, mid=6)
call merge_sort(list, left=5, mid=5)
call merge_sort(list, mid+1=6, right=6)
call merge(list, left=5, mid=5, right=6)
result : 3 6 8 13 2 14 5 9 10 1 7 12 4

call merge_sort(list, mid+1=7, right=7)
call merge(list, left=5, mid=6, right=7)
result : 3 6 8 13 2 5 14 9 10 1 7 12 4

call merge(list, left=1, mid=4, right=7)
result : 2 3 5 6 8 13 14 9 10 1 7 12 4
```

```
call merge_sort(list, mid+1=8, right=13)
call merge_sort(list, left=8, mid=10)
call merge_sort(list, left=8, mid=9)
call merge_sort(list, left=8, mid=8)
call merge_sort(list, mid+1=9, right=9)
call merge(list, left=8, mid=8, right=9)
result : 2 3 5 6 8 13 14 9 10 1 7 12 4

call merge_sort(list, mid+1=10, right=10)
call merge(list, left=8, mid=9, right=10)
result : 2 3 5 6 8 13 14 1 9 10 7 12 4

call merge_sort(list, mid+1=11, right=13)
call merge_sort(list, left=11, mid=12)
call merge_sort(list, left=11, mid=11)
call merge_sort(list, mid+1=12, right=12)
call merge(list, left=11, mid=11, right=12)
result : 2 3 5 6 8 13 14 1 9 10 7 12 4

call merge_sort(list, mid+1=13, right=13)
call merge(list, left=11, mid=12, right=13)
result : 2 3 5 6 8 13 14 1 9 10 4 7 12

call merge(list, left=8, mid=10, right=13)
result : 2 3 5 6 8 13 14 1 4 7 9 10 12

call merge(list, left=1, mid=7, right=13)
result : 1 2 3 4 5 6 7 8 9 10 12 13 14


<<<<<<<<<<<<< Sorted List >>>>>>>>>>>
1 2 3 4 5 6 7 8 9 10 12 13 14

계속하려면 아무 키나 누르십시오 . . .
```