Contents

Sorting on Several Keys

- Sorting records with several keys $K^1, K^2, \ldots, K^r$
  - for every pair of records *i* and *j*,
    $i < j$ and $(K_i^1, \ldots, K_i^r) \le (K^1_j, \ldots, K^r_j)$
    → A list of records $R_1, \ldots, R_n$, is said to be sorted with respect to the keys $K^1, K^2, \ldots, K^r$

Ex)

Sorting on Several Keys

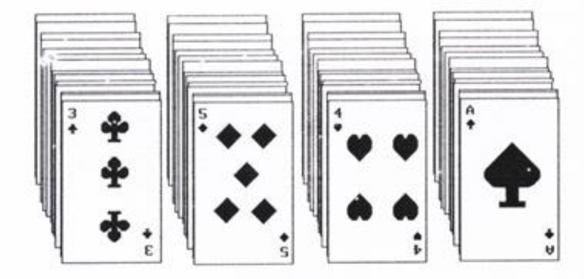
- Ex) Sorting a deck of cards
  - Sort on two keys, suit and face value

    K^1 [Suit]: ♣ < ♦ < ♥ < ♠

    K^2 [Face value]: 2 < 3 < 4 < … < 10 < J < Q < K < A

    key가 2개
  - Thus, a sorted deck of cards has the ordering:

    2♣, …, A♣, … , 2♠, … , A♠



Sorting on Several Keys

- Two approaches
  - MSD (most significant digit first) sort
    → Sort on K_1, then K_2, ... 키의 우선순위에 따라서 sorting
  - LSD (least significant digit first) sort
    → ...

- MSD first

  1) MSD sort (K_1)

  그림을 먼저 sorting

  e.g., 4 bins: ♣ ♦ ♥ ♠

  2) LSD sort (K_2)

  그 다음 숫자

  Result: 2♣, …, A♣, … , 2♠, … , A♠

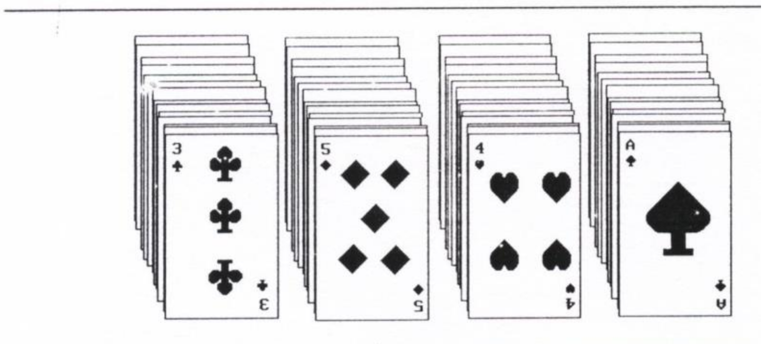


Figure 7.14: Arrangement of cards after first pass of an MSD sort

- LSD first MSD보다 더 간편함
  1) LSD sort (K_2)

     13 bins: 2, 3, 4, …, 10, J, Q, K, A

  2) MSD sort (K_1)
     - May not needed if we just classify these 13 piles into 4 separated piles
     - Simpler than the MSD one

Result:
2♣, …, A♣,
…
2♠, …, A♠

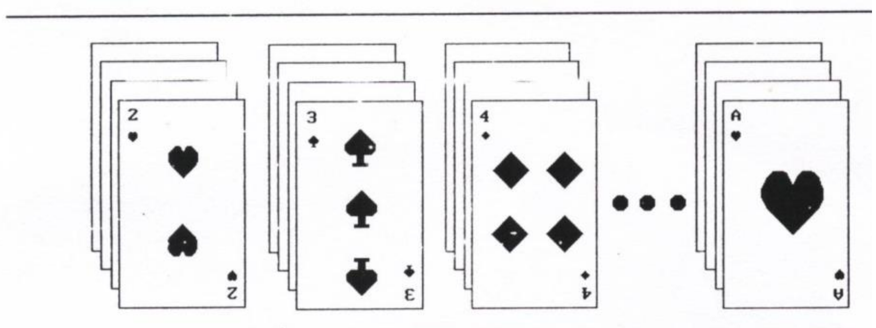


Figure 7.15: Arrangement of cards after first pass of LSD sort

- LSD or MSD sorting

  key가 한개여도 적용가능

  - can be used to sort even when the records have only one key

- Ex)
  Sorting 10 numbers in the range [0,999];

  {179, 208, 306, 93, 859, 984, 55, 9, 271, 33}

  - Each decimal digit may be regarded as three subkeys (K^1, K^2, K^3)
  - Use LSD or MSD sorting for three keys

Radix Sort

- Decompose the sort key into digits using a radix *r*
  - Ex) {179, 208, 306, 93, 859, 984, 55, 9, 271, 33}
    → *r* =10 10진법정렬
    → Each key in the range 0 through *r* - 1
- In a Radix-*r* Sort, # of bins required is *r*
- To sort $R_1, \ldots, R_n$
  - The record keys are decomposed using a radix of *r*
  - The records in each bin is linked together into a chain
    - front[*i*] and rear[*i*], $0 <= i < r$
  - These chains will operate as queues

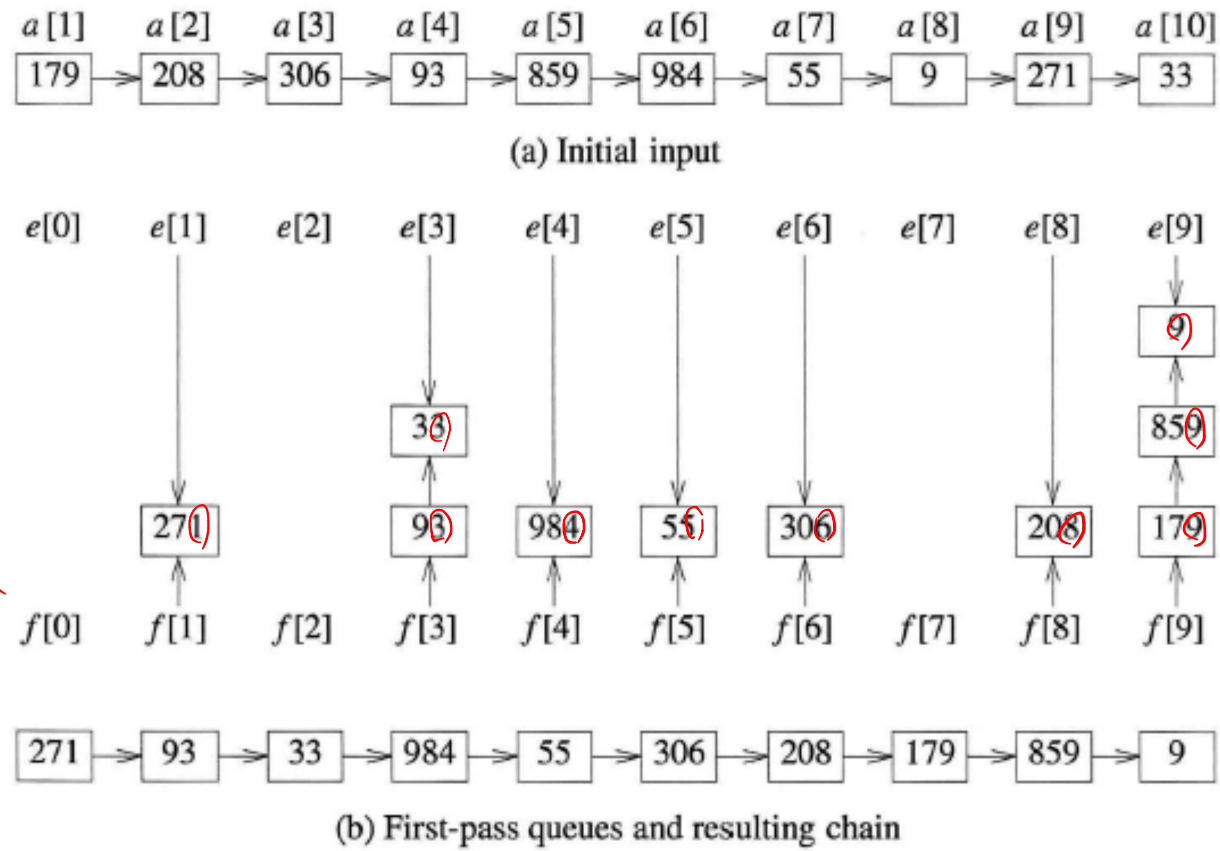
↱ 일의 자릿수부터 정렬

- Ex 7.8) LSD radix sort

rear

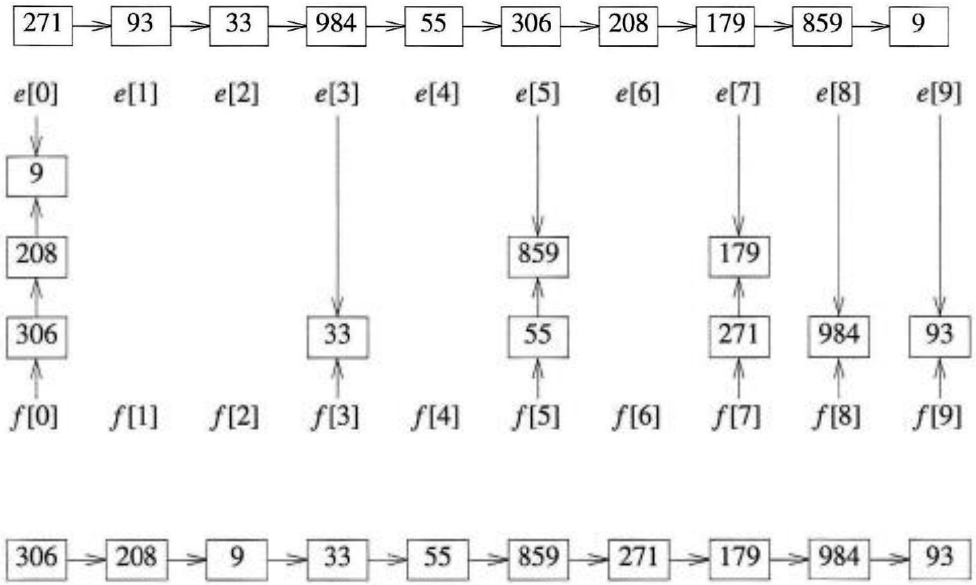
두개와 하나의 배열로 정렬

front



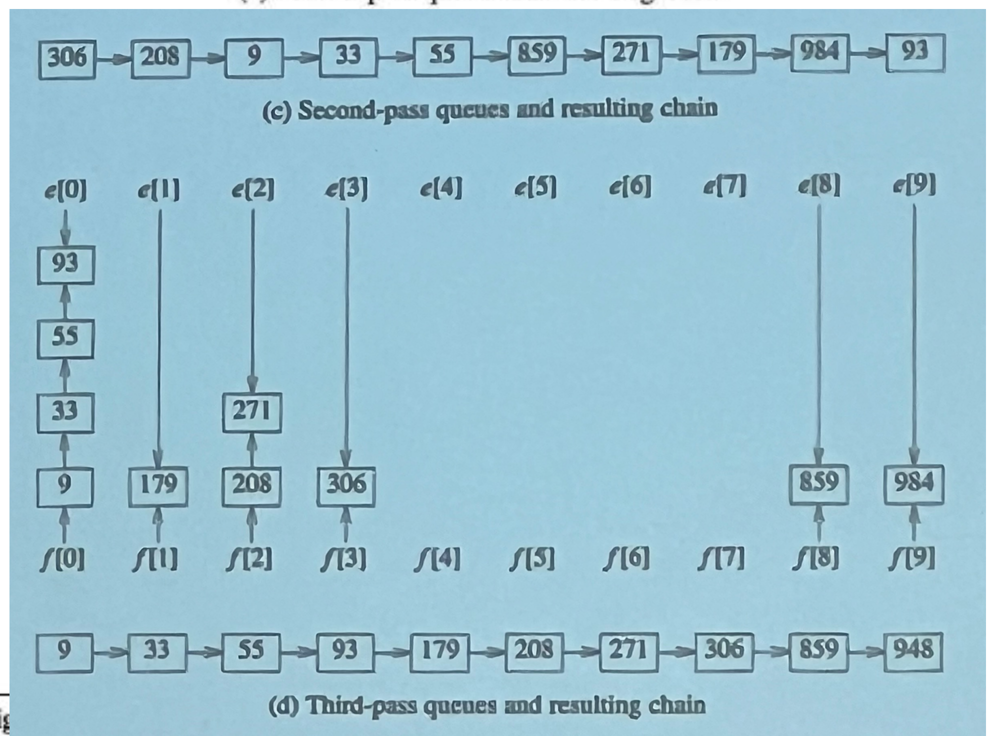
1의 자리수대로 정렬

Figure 7.9: Radix sort example (continued on next page)



(c) Second-pass queues and resulting chain

10의 자리로 정렬



100의 자리로 정렬

Fi

가장 큰 수의 자릿수개수 10진법 ⇒ r=10

```
int radixSort(element a[], int link[], int d, int r, int n)
{/* sort a[1:n]) using a d-digit radix-r sort, digit(a[i],j,r)
    returns the jth radix-r digit (from the left) of a[i]'s key
    each digit is in the range is [0,r); sorting within a digit
    is done using a bin sort */
```

digit(a[1], 2, 10);

array[0] [1] [2]

0	1	2
1	**7**	**9**

cnt=3

↓

0	1	9 2
0	0	9

10^{cnt}

179 → 100으로 나눈 몫 1, 100으로 나눈 나머지 79

79 → 10으로 〃 7, 10으로 9

9 → 1로 9, 1 0

3 10 10

```
int radixSort(element a[], int link[], int d, int r, int n)  {
      int front[r], rear[r];
      int i, bin, current, first, last;
      first = 1;
      for(i = 1; i < n; i++) link[i] = i+1;
      link[n] = 0;
      for(i=d-1; i >=0; i--)       {
            for(bin = 0; bin < r; bin++) front[bin] = 0;
            for(current = first; current; current = link[current])
            {/* put records into queues/bins */
                  bin = digit(a[current], i, r);  9
                  if(front[bin] == 0)   front[bin] = current;
                  else  link[ rear[bin] ] = current;
                  rear[bin] = current;
            }
            /* find first nonempty queue/bin */
            for (bin= 0; !front[bin]; bin++);
            first= front[bin]; last= rear[bin];
           /* concatenate remaining queues */
            for(bin++; bin < r; bin++)
                  if(front[bin]){ link[last]=front[bin]; last = rear[bin];}
            link[last] = 0;
      }
      return first;
}
```

Current=8

d=3, r=10

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
r										
f										

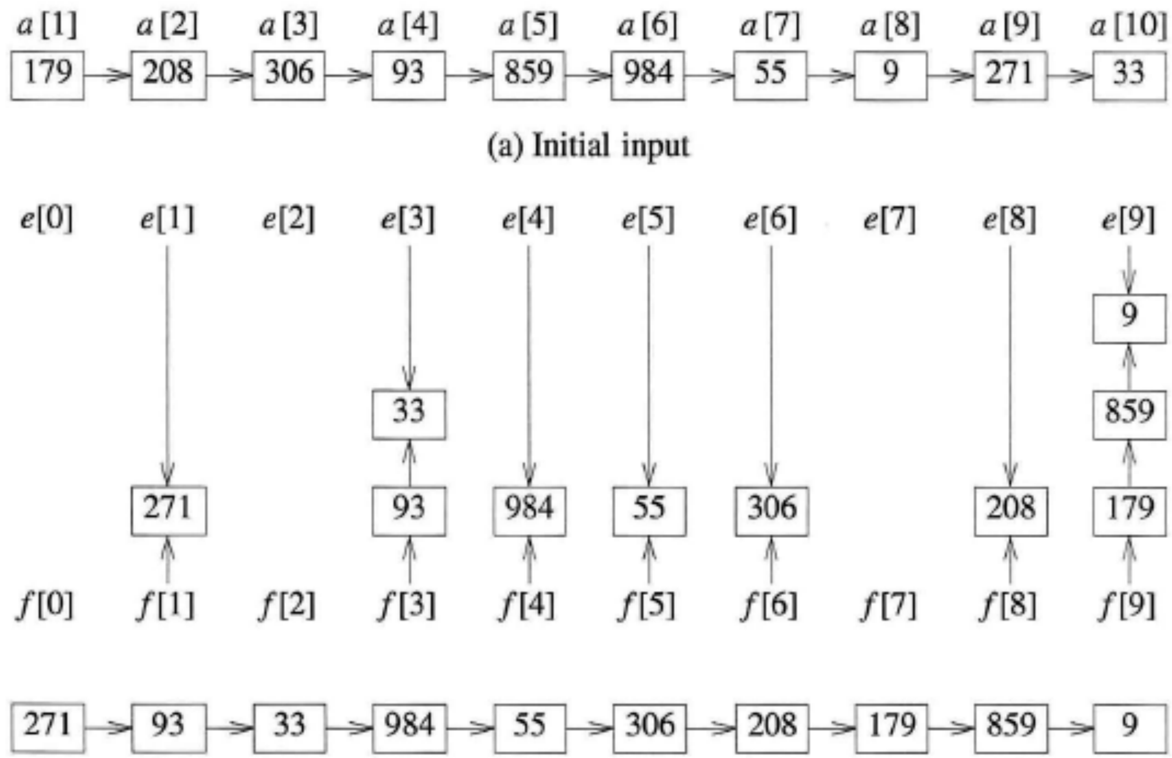
[1]

3	**1**	**4**	**5**	**0**	link
9	**271**	**859**	**179**	**59**	a

first 2

last= 5

1st pass:
271→ 9→ 859 →179→ 59



(a) Initial input

(b) First-pass queues and resulting chain

Figure 7.9: Radix sort example (continued on next page)

```
int radixSort(element a[], int link[], int d, int r, int n)  {
    int front[r], rear[r];
    int i, bin, current, first, last;
    first = 1;
    for(i = 1; i < n; i++) link[i] = i+1;
    link[n] = 0;
    for(i=d-1; i >=0; i--)      {
        for(bin = 0; bin < r; bin++) front[bin] = 0;
        for(current = first; current; current = link[current])
        {/* put records into queues/bins */
            bin = digit(a[current], i, r);
            if(front[bin] == 0)   front[bin] = current;
            else  link[ rear[bin] ] = current;
            rear[bin] = current;
        }
        /* find first nonempty queue/bin */
        for (bin= 0; !front[bin]; bin++);
        first= front[bin]; last= rear[bin];
        /* concatenate remaining queues */
        for(bin++; bin < r; bin++)
            if(front[bin]){ link[last]=front[bin]; last = rear[bin];}
        link[last] = 0;
    }
    return first;
}
```

digit(a[i], 1, 10)

d=3, r=10

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
r								/		
f								/		

[1]

3	1	4	5	0	link
9	271	859	179	59	a

first = 2

271→ 9→ 859 →179→ 59

2nd pass:

7.9 SUMMARY

- No one method is best under all circumstances
  - Some are good for small n, others for large n
- Insertion sort
  - Good when the list is partially ordered
  - Best sorting method for small n
- Merge sort
  - same
  - Has the best worst case behavior
  - But requires more storage than heap sort
- Quick sort
  - Best average behavior
  - But worst case: $O(n^2)$

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

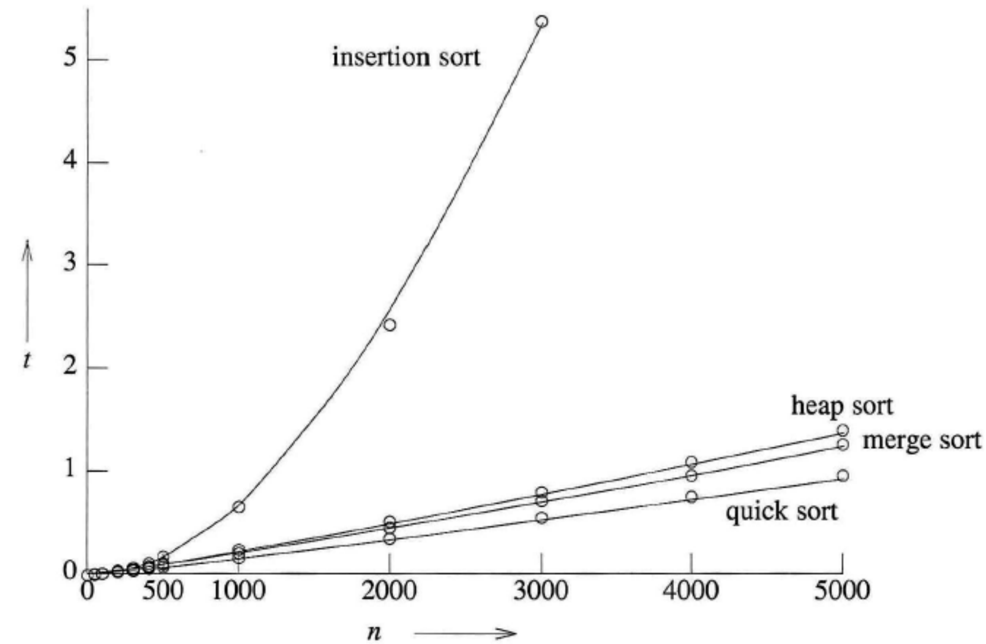


Figure 7.18: Plot of average times (milliseconds)

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

Contents