Chap 7. Sorting (3)

Contents

7.7 Sorting on Several Keys

- Sorting records with several keys $K^1, K^2, \ldots, K^r$
  - K^1: the most significant key
  - K^r: the least significant key
  - K^j_i : key K^j of record R_i

이름	수학	과학	국어	영어
김길동	98	60	50	70
박수동	92	80	60	85
홍길수	92	80	60	80
정인숙	70	98	95	95

- A list of records $R_1, \ldots, R_n$ is said to be *sorted* w.r.t. the keys $K^1, K^2, \ldots, K^r$ *iff* for every pair of records i and j, $i < j$ and $(K^1_i, K^2_i, \ldots, K^r_i) \le (K^1_j, K^2_j, \ldots, K^r_j)$.

- The r-tuple $(x_1, x_2, \ldots, x_r)$ is less than or equal to the r-tuple $(y_1, y_2, \ldots, y_r)$ *iff* either $x_i = y_i$, $1 \le i \le j$ and $x_{j+1} < y_{j+1}$ for some $j < r$ or $x_i = y_i$, $1 \le i \le r$.

$x_1, x_2, x_3, x_4, \ldots\ldots\ldots x_j, x_{j+1} \ldots\ldots\ldots, x_r$

$y_1, y_2, y_3, y_4, \ldots\ldots\ldots y_j, y_{j+1} \ldots\ldots\ldots, y_r$

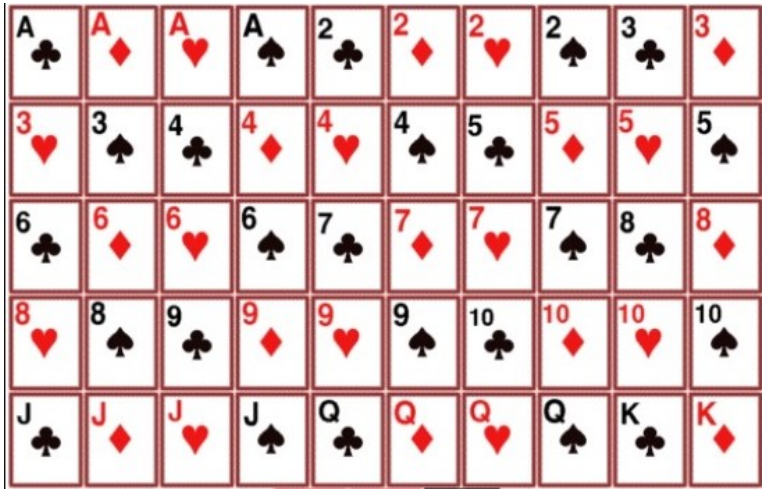
- Example : sorting a deck of cards
  - *A sort on two keys*, the suit and face values, with the following ordering relations:

K^1 [Suits]: ♣ < ♦ < ♥ < ♠

K^2 [Face values]: $2 < 3 < 4 \cdots < 10 < J < Q < K < A$

  - A sorted deck of cards :

2♣, . . ., A♣, . . ., 2♠, . . ., A♠



Two ways to sort on multiple keys

- MSD (Most Significant Digit) sort
  - Sort on the MSD key(K^1) → … → LSD key(K^r)
  - Original, unsorted list:
    - 170, 45, 65, 60, 802, 4, 2, 66

045, 065, 060, 004, 002, 066, 170, 802
004, 002, 045, 065, 060, 066, 170, 802
002, 004, 045, 060, 065, 066, 170, 802

  - Sorted list :
    - 2, 4, 45, 60, 65, 66, 170, 802

- LSD (Least Significant Digit) sort
  - Sort on the LSD key(K^r) → … → MSD key(K^1)
  - Original, unsorted list:
    - 170, 45, 65, 60, 802, 4, 2, 66

170, 060, 802, 002, 004, 045, 065, 066
802, 002, 004, 045, 060, 065, 066, 170
002, 004, 045, 060, 065, 066, 170, 802

  - Sorted list :
    - 2, 4, 45, 60, 65, 66, 170, 802
  - simpler and less overhead than MSD sort
    - the subpiles do not have to be sorted independently

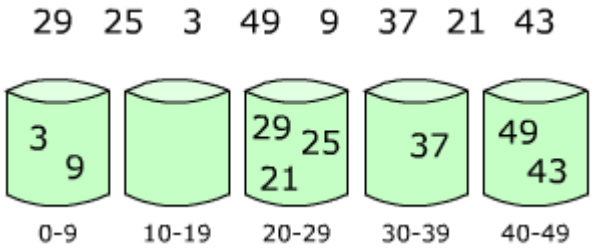
- The terms LSD and MSD specify only *the order in which the keys are to be sorted*.

- They don't specify how each key is to be sorted.
  - LSD and MSD can use *bin sort*
    - distribute records into bins and sort each bins

- LSD or MSD sort on *the records having one key*
  - interpret *the key* as consisting of *several subkeys*

Bin Sort[Bucket sort]

- Definition: Bin Sort is a way of sorting the keys to be sorted into a certain range and putting them in a bucket and sorting each bucket.



- For example, *if the keys are numeric*,
  - then *each decimal digit* may be regarded as *a subkey*.

- If the keys are in the range $0 \leq K \leq 999$,
  - we can use either the LSD or MSD sorts for three keys (K^1, K^2, K^3), where
    - K^1 is the digit in the hundredths place,
    - K^2 the digit in the tens place and
    - K^3 the digit in the units place.

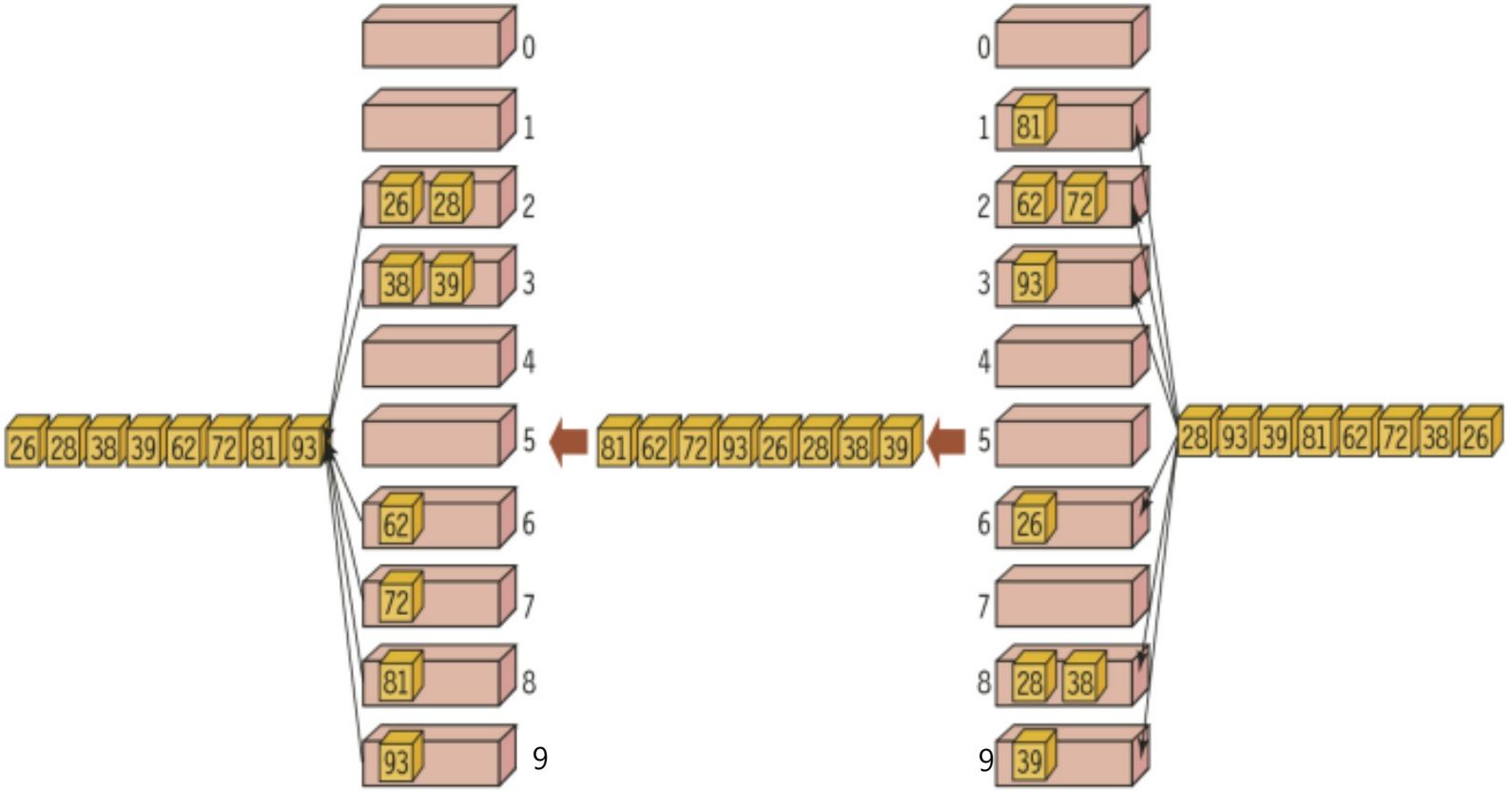
- Since $0 \leq K^i \leq 9$ for each key K^i,
  - the sort on each key can be carried out using *a bin sort with 10 bins*.

Radix Sort

- Decompose the sort key using some radix *r*
  - *r*=10 → decimal decomposition
  - *r*=2 → binary decomposition

- In a radix-r sort, to sort $R_1, \ldots, R_n$
  - Decompose the record keys using a radix of *r*
  - each key has *d* digits in between 0 and *r*-1
  - Use *r* bins
  - *The records in each bin* is linked together into a chain → *linked queue*

LSD (Least Significant Digit) sort



```
#define BUCKETS 10
#define DIGITS 4
void radix_sort(int list[], int n)
{
int i, b, d, factor=1;
QueueType queues[BUCKETS];

   for(b=0;b<BUCKETS;b++) init(&queues[b]);          // initialize Queue
   for(d=0; d<DIGITS; d++){
      for(i=0;i<n;i++)                               // enqueue
         enqueue( &queues[(list[i]/factor)%10], list[i]);

      for(b=i=0;b<BUCKETS;b++)                       // dequeue
         while( !is_empty(&queues[b]) )
            list[i++] = dequeue(&queues[b]);
      factor *= 10;                                  // next digit
   }
}
```

- Example 7.8
  - To sort 10 numbers in the range [0,999] using r =10

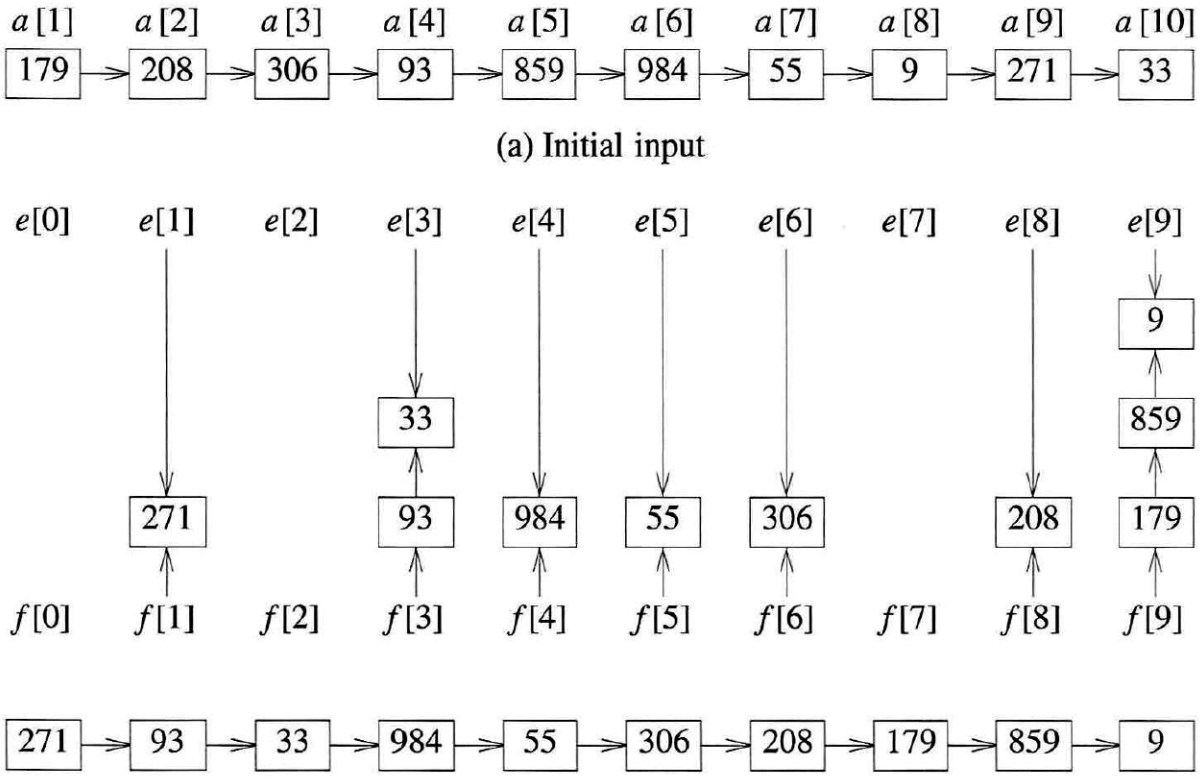
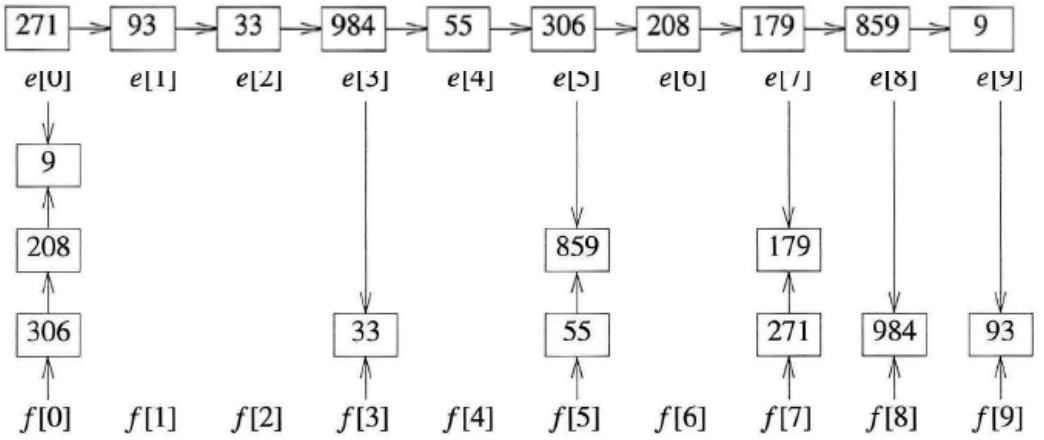
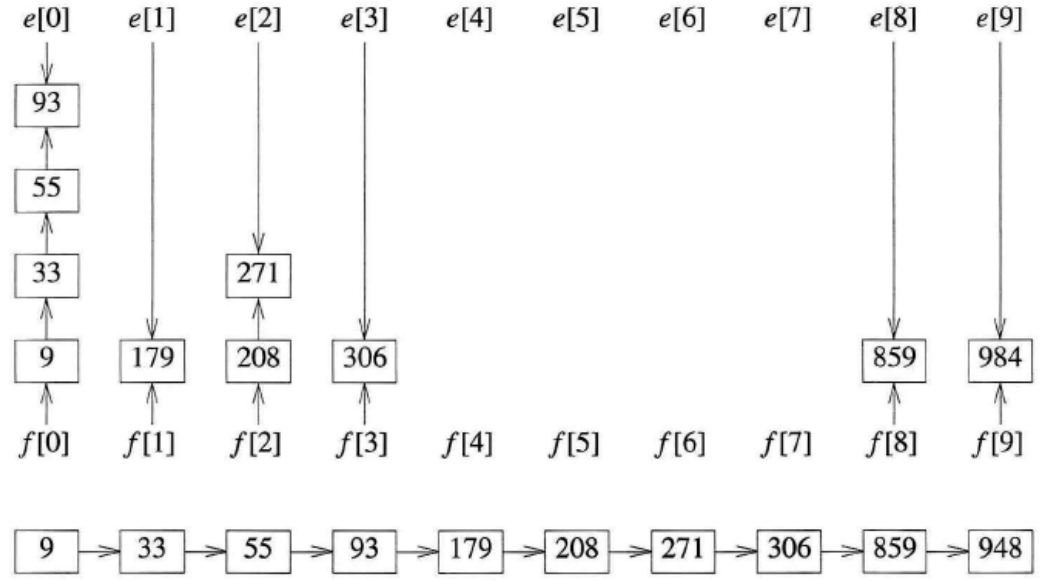


Figure 7.9: Radix sort example (continued on next page)



(c) Second-pass queues and resulting chain



(d) Third-pass queues and resulting chain

Figure 7.9: Radix sort example

//bin 번호

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear										
front	0	0	0	0	0	0	0	0	0	0

//입력 인덱스

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	2	3	4	5	6	7	8	9	10	0
a	179	208	306	93	859	984	55	9	271	33

First : 1

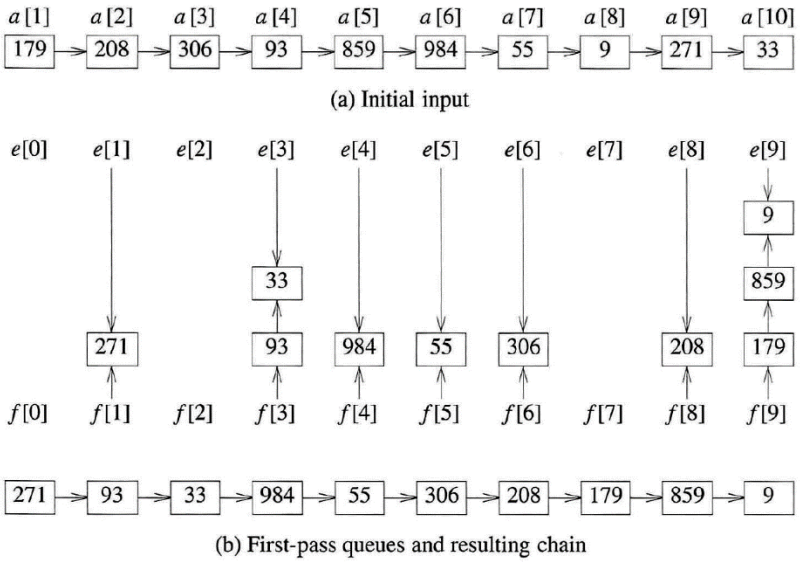


Figure 7.9: Radix sort example (continued on next page)

First Pass (Queue)

//bin 마지막 인덱스

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear		9		10	6	7	3		2	8
front	0	9	0	4	6	7	3	0	2	1

//bin 시작 인덱스

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	5	3	4	10	8	7	8	9	10	0
a	179	208	306	93	859	984	55	9	271	33

First : 9

//front 와 rear가 다를 때만 조정

First Pass (list) //빈 안에 내용을 합침

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear		9		10	6	7	3		2	8
front	0	9	0	4	6	7	3	0	2	1

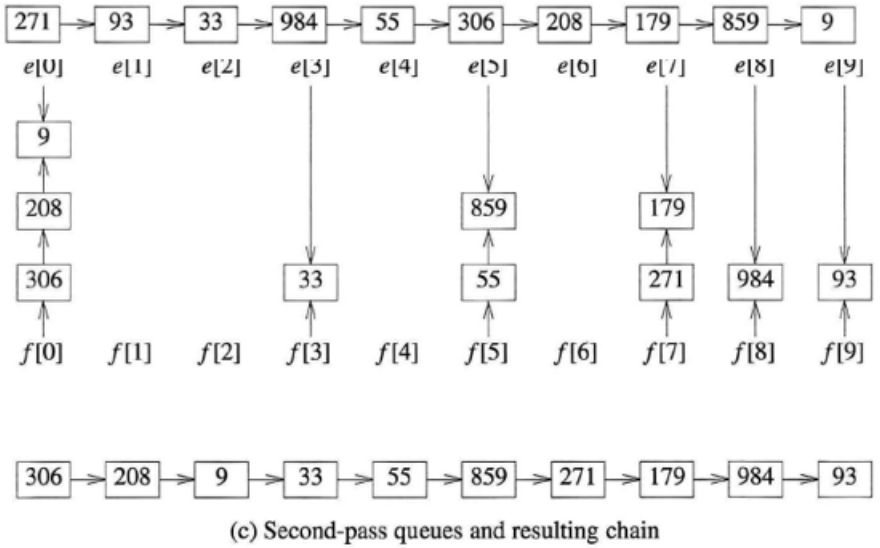
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	5	1	2	10	8	7	3	0	4	6
a	179	208	306	93	859	984	55	9	271	33

First : 9

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear										
front	0	0	0	0	0	0	0	0	0	0

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	5	1	2	10	8	7	3	0	4	6
a	179	208	306	93	859	984	55	9	271	33

First : 9



(c) Second-pass queues and resulting chain

Second Pass (Queue)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear	8	9		10	6	5	3	1	6	4
front	3	0	0	10	0	7	0	9	6	4

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	5	8	2	10	8	7	5	9	1	0
a	179	208	306	93	859	984	55	9	271	33

First : 3

Second Pass (list)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear	8	9		10	6	5	3	1	6	4
front	3	0	0	10	0	7	0	9	6	4

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	6	8	2	0	9	4	5	10	1	7
a	179	208	306	93	859	984	55	9	271	33

First : 3

```
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 */
   int front[r], rear[r]; /* queue front and rear pointers */
   int i, bin, current, first, last;
   /* create initial chain of records starting at first */
   first = 1;
   for (i = 1; i < n; i++) link[i] = i + 1;
   link[n] = 0;

   for (i = d-1; i >= 0; i--)      //d:3, r:10
   {/* sort on digit i */
      /* initialize bins to empty queues */
      for (bin = 0; bin < r; bin++) front[bin] = 0;

                                                  //bin에 분리
      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 */ //bin을 연결
      for (bin++; bin < r; bin++)
         if (front[bin])
            {link[last] = front[bin]; last = rear[bin];}
      link[last] = 0;
   }
   return first;
}
```

Program 7.14: LSD radix sort

d : 3, r : 10, n : 10

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
rear										
front	0	0	0	0	0	0	0	0	0	0

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
link	2	3	4	5	6	7	8	9	10	0
a	179	208	306	93	859	984	55	9	271	33

first → 179 → 208 → 306 → 93 → 859 → 984 → 55 → 9 → 271 → 33

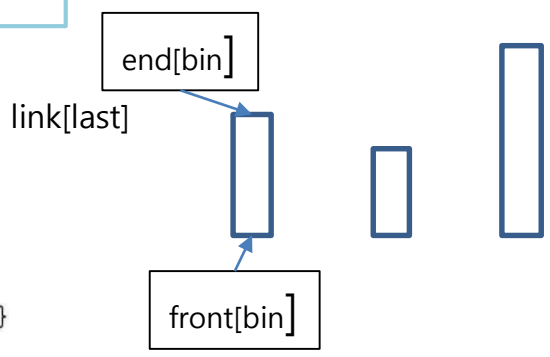
current

※ digit(a[1], 2, 10);

digit i : 0 1 2

1 7 9 a[i]

bin : 9



	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	5	3	4	10	8	7	8	9	10	0
	179	208	306	93	859	984	55	9	271	33

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

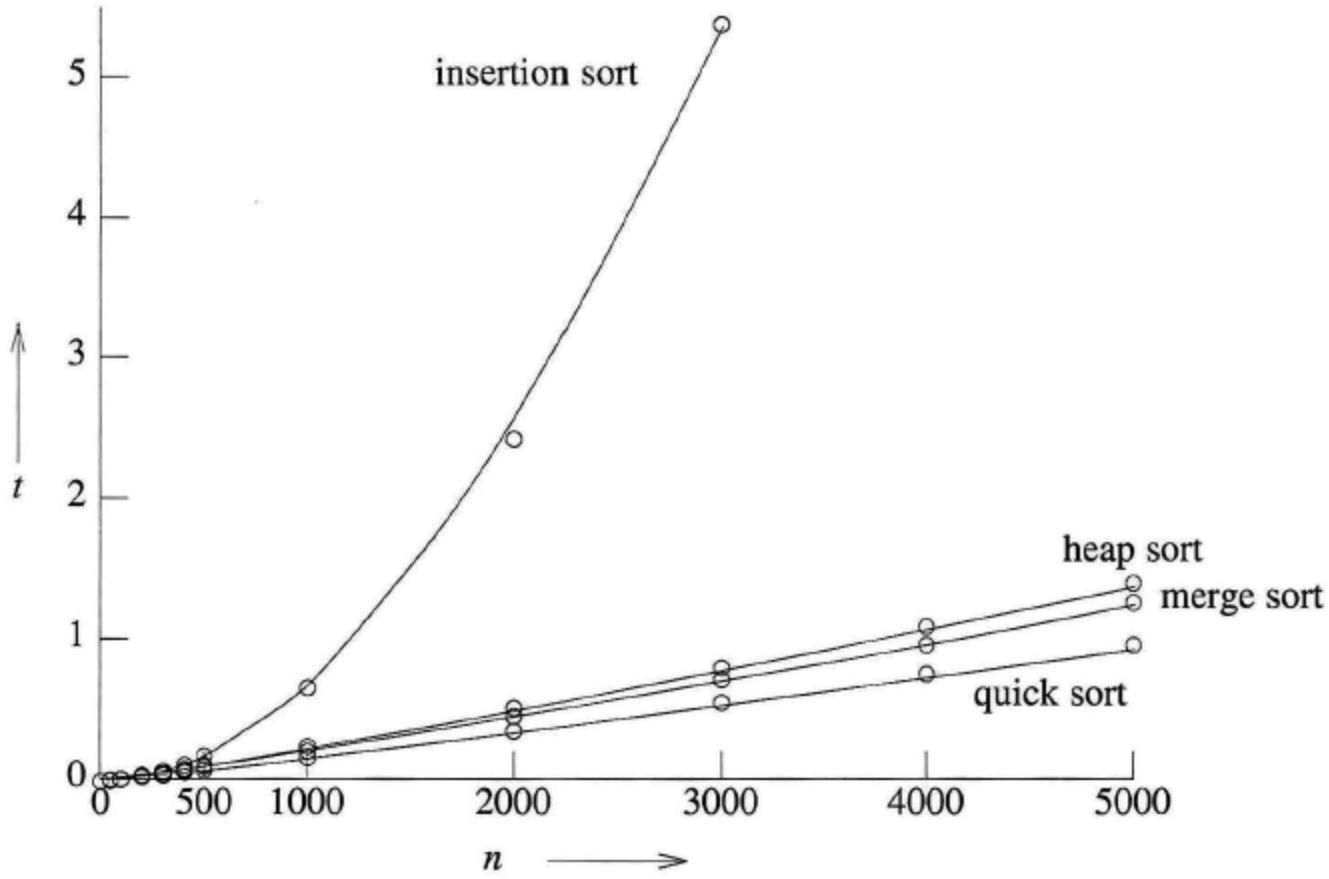


Figure 7.18: Plot of average times (milliseconds)