Lecture Notes on *Data Structures*

M1522.000900

Seoul National University

Fall 2022

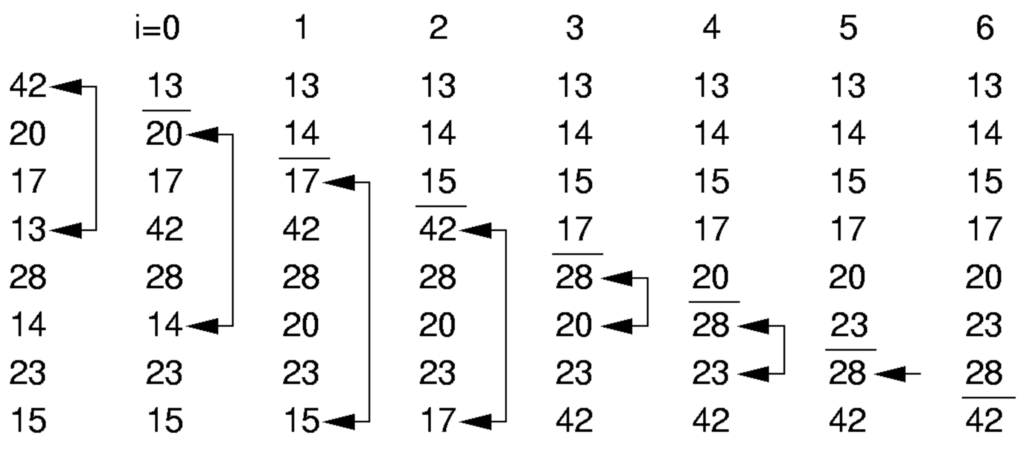
Part VIII

Sorting

Selection sort

Algorithm 1 (Selection sort)

```
for(i=0; i < N-1 ;i++) {
   find j such that A[j] == min{A[i] ..  A[N-1]};
   swap(A[j],A[i]);
}
```

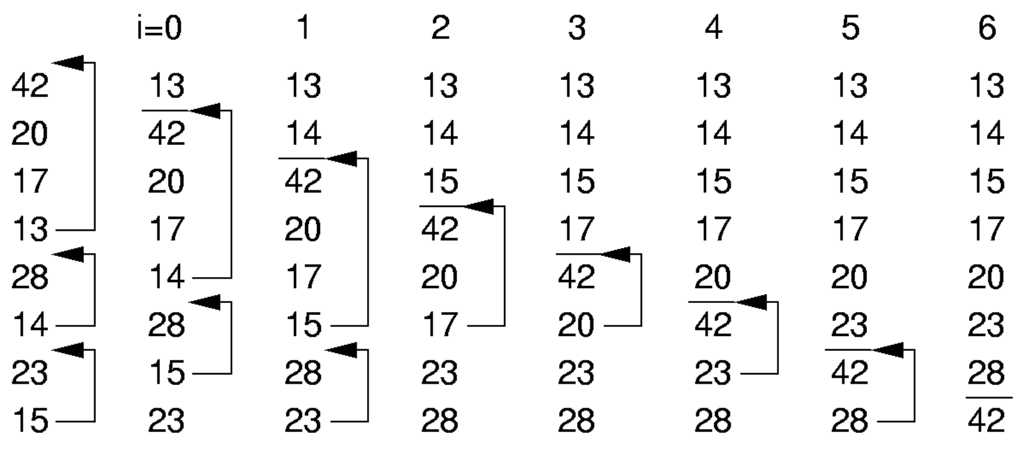


	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Bubblesort

Algorithm 2 (Naive Bubblesort)

```
for(i=0; i < N-1 ;i++)
   for(j=N-1; j > i ;j--)                        // bubbling up
      if (A[j-1] > A[j]) swap(A[j-1],A[j]);
// Inv:  A[i-1] stores i^th element after the i^th iteration.
```



	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	42	14	14	14	14	14	14
17	20	42	15	15	15	15	15
13	17	20	42	17	17	17	17
28	14	17	20	42	20	20	20
14	28	15	17	20	42	23	23
23	15	28	23	23	23	42	28
15	23	23	28	28	28	28	42

Improved Bubblesort

Algorithm 3 (Improved Bubblesort)

```
for(Bound=-1; Bound < N-1 ;) {
   temp = N-1;                  // track the most recent swap
   for(j=N-1; j > Bound+1 ;j--)                // bubbling up
      if (A[j-1] > A[j]) { swap(A[j-1],A[j]); temp = j-1; }
   Bound = temp;   // swap(A[Bound],A[Bound+1]) was the last
}
```

- Suppose a_{k-1} and a_k are the pair last swapped while bubbling up.
  - Then $a_0 \leq a_1 \leq \cdots \leq a_{k-1} \leq a_k$ and
  - $a_0, a_1, \cdots, a_{k-1}$ (but not a_k) are in the sorted positions.
- Therefore A[0 : Bound] are in their (final) sorted positions.

Bubbling Down

Algorithm 4 (Naive Bubblesort)

```
for(i=N; i > 1 ;i--)
   for(j=0; j < i-1 ;j++)                    // bubbling down
      if (A[j] > A[j+1]) swap(A[j],A[j+1]);
```

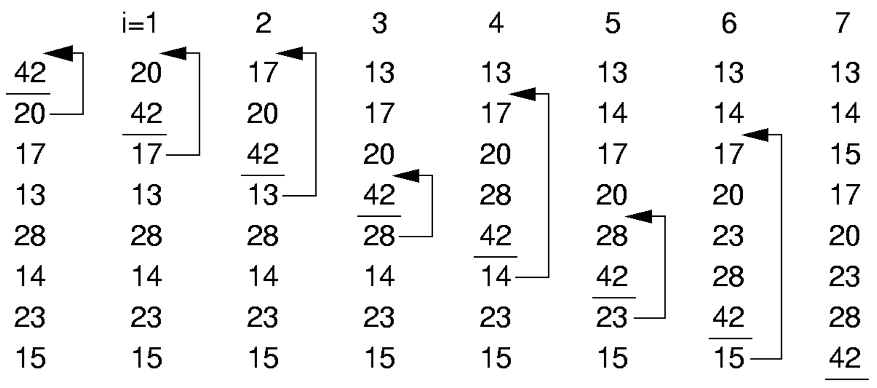
Algorithm 5 (Improved Bubblesort)

```
for(Bound=N; Bound > 0 ;) {
   temp = 0;                    // track the most recent swap
   for(j=0; j < Bound-1 ;j++)                // bubbling down
      if (A[j] > A[j+1]) { swap(A[j],A[j+1]); temp = j+1; }
   Bound = temp;  // A[Bound:N-1] are in the sorted position
}
```

Insertion sort

Algorithm 6 (Insertion sort)

```
for(i=1; i < N ;i++) {
   temp = A[i];                                  // Make a hole
   for(j=i-1; j >= 0 and A[j] > temp ;j--) // Scan left (up)
      A[j+1] = A[j];                           // Shift right (down)
   A[j+1] = temp;                              // Fill into the hole
}
```



	i=1	2	3	4	5	6	7
42	20	17	13	13	13	13	13
20	42	20	17	17	14	14	14
17	17	42	20	20	17	17	15
13	13	13	42	28	20	20	17
28	28	28	28	42	28	23	20
14	14	14	14	14	42	28	23
23	23	23	23	23	23	42	28
15	15	15	15	15	15	15	42

Selection sort vs. Bubblesort vs. Insertion sort

	Best case	Worst case
Selection	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Naive	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Improved	$\Theta(n)$ (for sorted input)	$\Theta(n^2)$ (for inversely sorted input)
Insertion	$\Theta(n)$ (for sorted input)	$\Theta(n^2)$ (for inversely sorted input)

- No more than $n-1$ iterations is necessary.
- Selection sort and Bubblesort (naive): A[0 : k-1] are in sorted positions after the k^{th} iteration.
- Bubblesort (improved): A[0 : Bound] are in sorted positions.
- Insertion sort: A[0 : k] are in *ascending* order after the k^{th} iteration.

Treesort

A set of n elements can be sorted by (1) building a BST, and then (2) performing an inorder traversal of the BST.

- What is the best and worst case running time of this approach?
- How much extra memory does it require?

Quicksort

The idea is similar to that of Treesort in the sense that elements smaller than the pivot (or the root of the BST) are moved to the left side of the pivot (or the left subtree of the root) and elements greater than the pivot are moved to the right side of the pivot (or the right subtree of the root).

- Unlike Treesort, however, the subtrees of the root (or pivot) are not BST. The same strategy need be applied recursively to the subtrees.
- Of course, Quicksort does it without explicitly building trees, but instead by swapping elements in the array. Quicksort is an *in-place* algorithm that only requires $\mathcal{O}(1)$ extra memory.
- Quicksort is a *divide-and-conquer* algorithm.

Algorithm 7 (Quicksort)

```
Quicksort(A,left,right)                    // input:  A[left : right]
   if (left >= right) return;
   pivot = A[left];                        // the first one as a pivot
   for(i=left,j=right+1; i < j ;) {
      while(i < right and A[++i] < pivot);
      while(j > left and A[--j] > pivot);
      // Inv:  (A[i] ≥ pivot or i = right)
      //             and (A[j] ≤ pivot or j = left)
      // Inv:  if (i < j) then A[i] ≥ pivot and A[j] ≤ pivot
      if (i < j) swap(A[i],A[j]);
   }
   // Inv:  left ≤ j and A[j] ≤ A[left] = pivot
   swap(A[left],A[j]);        // place the pivot between partitions
   // Inv:  A[left:j-1] ≤ A[j] ≤ A[j+1:right]
   Quicksort(A,left,j-1);
   Quicksort(A,j+1,right);
```

Example 1

Sort the following array with Quicksort algorithm.

72 6 57 88 85 42 83 73 48 60

Quicksort(A,0,9)	A[] = 72 6 57 88 85 42 83 73 48 60
	(i=3, j=9) → swap(A[i],A[j]) = swap(88,60)
	(i=4, j=8) → swap(A[i],A[j]) = swap(85,48)
	(i=6, j=5) → swap(pivot,A[j]) = swap(72,42)
Quicksort(A,0,4)	A[] = 42 6 57 60 48
	(i=2, j=1) → swap(pivot,A[j]) = swap(42,6)
Quicksort(A,0,0)	A[] = 6
Quicksort(A,2,4)	A[] = 57 60 48
	(i=3, j=4) → swap(A[i],A[j]) = swap(60,48)
	(i=4, j=3) → swap(pivot,A[j]) = swap(57,48)
Quicksort(A,2,2)	A[] = 48
Quicksort(A,4,4)	A[] = 60
Quicksort(A,6,9)	A[] = 83 73 85 88

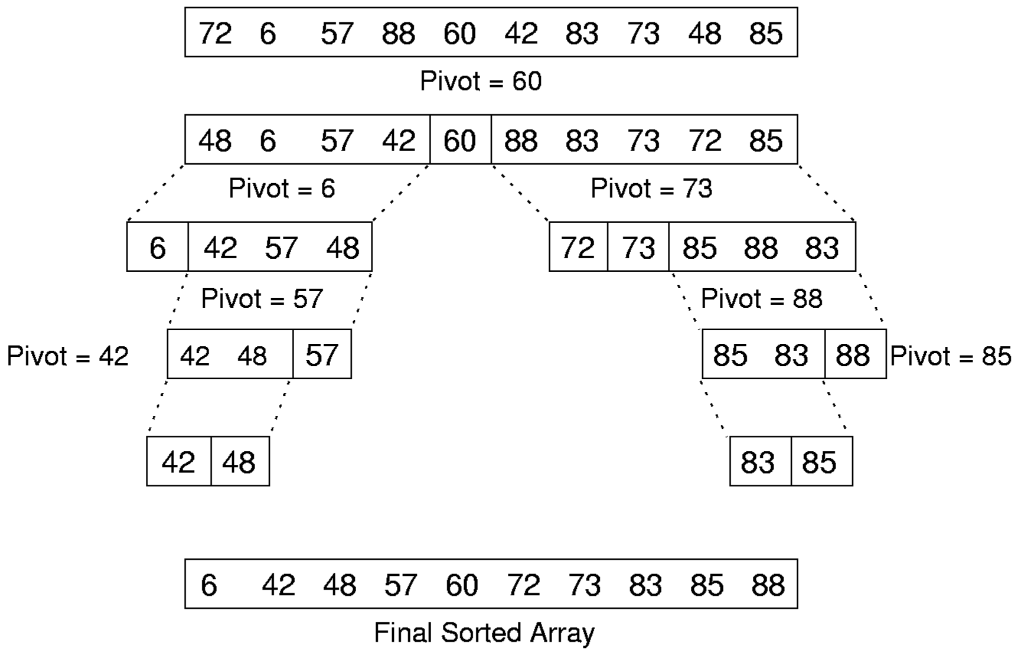
Algorithm 8 (Quicksort selecting the middle as a pivot)

```
Quicksort(A,left,right)                  // input:  A[left :  right]
   if (left >= right) return;
   pivotindex = (left+right)/2;        // the middle one as a pivot
   swap(A[left],A[pivotindex]);            // pivot in the leftmost
   pivot = A[left];
   ...
```

- Algorithm 7 will work slowly for a sorted array.
  (How slow in $\mathcal{O}()$ notation? A worst case for Quicksort?)
- The performance of Quicksort is sensitive to pivot selections.
- Algorithm 8 selects the middle element as a pivot and puts it in the leftmost place. The rest of the algorithm remains the same.
  (Will a sorted array be still a worst case for Quicksort?)

Problem 1

The diagram below illustrates how Quicksort partitions an array recursively. The middle element is picked as a pivot. If the middle element is swapped with an element at the rightmost position, what additional change do you need to make in Algorithm 8?



Quicksort: Analysis

Worst case: the pivot is the smallest (or largest) in every selection.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + cn \end{aligned}$$

By expanding,

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= \dots \\ &= T(1) + c\sum_{i=2}^{n} i \\ &\in \mathcal{O}(n^2). \end{aligned}$$

Best case: the pivot is the median in every selection.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 \times T(n/2) + cn \end{aligned}$$

By expanding,

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2 \times (2T(n/2^2) + c(n/2)) + cn = 2^2 T(n/2^2) + 2cn \\ &= 2^2 \times (2T(n/2^3) + c(n/2^2)) + 2cn = 2^3 T(n/2^3) + 3cn \\ &= \dots \\ &= 2^k T(n/2^k) + kcn \\ &= 2^{\log n} T(1) + \log n \times cn \qquad (2^{\log n} = n \text{ if } n = 2^k) \\ &\in \mathcal{O}(n \log n). \end{aligned}$$

Average case: the pivot is equally likely to be in any place.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1))+cn \end{aligned}$$

Since $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} T(n-i-1)$,

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + cn. \tag{1}$$

Multiplying (1) by n and $(n-1)$,

$$n \times T(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \tag{2}$$

$$(n-1) \times T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2 \tag{3}$$

Subtracting (3) from (2),

$$nT(n) = (n+1)T(n-1) + 2cn - c \tag{4}$$

Dividing (4) by $n(n+1)$,

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} - \frac{c}{n(n+1)}.$$

Ignore $\frac{c}{n(n+1)}$ and let $f(n) = \frac{T(n)}{n+1}$.

$$\begin{aligned} f(n) &= f(n-1) + \frac{2c}{n+1} \\ &= f(n-2) + \frac{2c}{n} + \frac{2c}{n+1} \\ &= f(1) + 2c\sum_{i=3}^{n+1}\frac{1}{i}. \end{aligned}$$

Therefore $T(n) \in \mathcal{O}(n\log n)$, because $\sum_{i=3}^{n+1}\frac{1}{i} \in \mathcal{O}(\log n)$.

Quicksort optimizations

- To choose a better pivot, take the median of three randomly chosen elements.
- Quicksort is slow when n is small. Use another sort algorithm (*e.g.*, Insertion) instead of calling Quicksort recursively.
- Using a stack, design an iterative version of Quicksort.

Mergesort

Mergesort is another *divide-and-conquer* algorithm.

1. An input array is divided to two sub-arrays of (almost) equal-length.
2. The sub-arrays are sorted recursively.
3. The sorted sub-arrays are merged into a single array.

Since an input array is divided to equal lengths, the running time is expected to be quite good.

What would the worst or best case be like for Mergesort?

Algorithm 9 (Mergesort)

```
Marge(A,B)          // Both A and B are in sorted order
   for(i=j=k=0; i < |A| and j < |B| ;) {
      if (A[i] < B[j]) temp[k++] = A[i++];
      else temp[k++] = B[j++];
   }
   // Inv:  $i \geq |A|$ or $j \geq |B|$
   if (i < |A|)
      while(i < |A|) temp[k++] = A[i++];
   else
      while(j < |B|) temp[k++] = B[j++];
   return temp;

Mergesort(A)
   if (|A|==1) return A;
   B = Mergesort(A[0:N/2-1]);
   C = Mergesort(A[N/2:N-1]);
   return Merge(B,C);
```

Example 2

Sort the following array with Mergesort algorithm.

36 20 17 13 28 14 23 15

```
36   20   17   13   28   14   23   15

|20   36| |13   17| |14   28| |15   23|

|13   17   20   36| |14   15   23   28|

|13   14   15   17   20   23   28   36|
```

Mergesort: Analysis

Running time of Merge(A,B) is $\Theta(|A| + |B|) = \Theta(n)$.

Running time of Mergesort(A) is

$$T(1) = 1$$
$$T(n) = 2 \times T(n/2) + cn$$

This is identical to the best case of Quicksort. $T(n) \in \mathcal{O}(n \log n)$.

Mergesort is not an in-place algorithm, because the space overhead is $\Omega(n)$.

Heapsort

Treesort vs. Heapsort

- A heap is easier to build than a BST. (How exactly?)
- A heap is implemented using an array, and always balanced.

The idea of Heapsort is:

1. Build a min-heap from an input array.
2. Remove the minimum from the min-heap iteratively.

Algorithm 10 (Heapsort)

```
Heapsort(A)
   A = MinHeap(A,n);                 // n is the size of input.
   for(i=0; i < n ;i++)
      B[i] = removeMin(A,n-i); // n-i is the size of heap.
```

The running time is $cn + \sum_{i=1}^{n} \log i \in \mathcal{O}(n \log n)$, because

- It takes $\mathcal{O}(n)$ time to build a min-heap.
- It takes $\mathcal{O}(\log |H|)$ time to remove the minimum.

Algorithm 10 is not an in-place algorithm, because the space overhead is $\Theta(n)$.

Algorithm 11 (in-place Heapsort)

```
Heapsort(A)
   A = MaxHeap(A,n);                 // n is the size of input.
   for(i=0; i < n ;i++)
      A[n-i-1] = removeMax(A,n-i);
```

- After removing the minimum k times, Algorithm 10 leaves k unused spaces in the rear end of `A`.
- Algorithm 11 uses a max-heap in order to reuse the unused spaces, which grow from right to left.

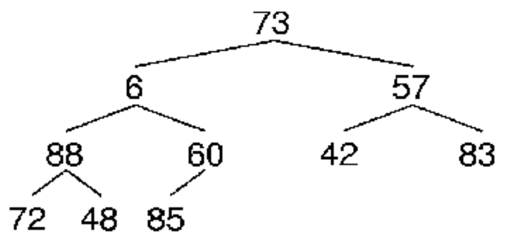
Example 3

Sort the following array with Heapsort algorithm.

73 6 57 88 60 42 83 72 48 85

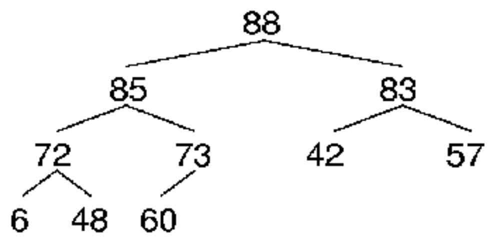
Original Numbers

73	6	57	88	60	42	83	72	48	85



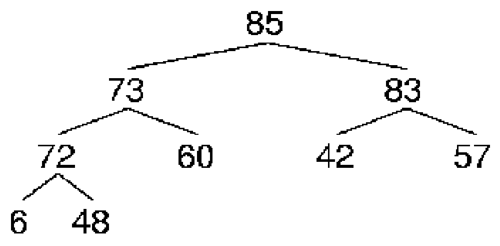
Build Heap

88	85	83	72	73	42	57	6	48	60



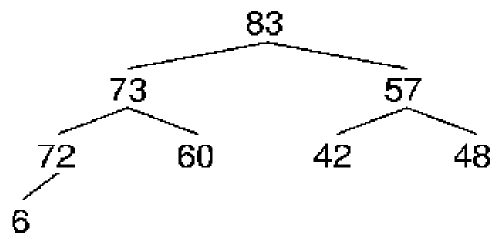
Remove 88

85	73	83	72	60	42	57	6	48	88



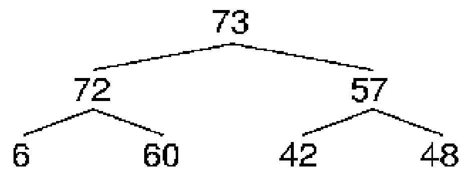
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Bucket sort

Example 4

A million people live in Tucson today. Sort them by the zipcode of their home address. Note that the zipcodes of Tucson residents differ only in the last two digits. What would be the sorting algorithm of your choice?

- Roughly speaking, 10,000 people share the same zipcode. Sorting these people is a waste of time.
- Applying Quicksort or any $\mathcal{O}(n \log n)$ algorithm will be overkill.
- We know that no one can sort faster than $\mathcal{O}(n \log n)$. But since we do not have to sort the 10,000 people, can we do better than that?

Algorithm 12 (Bucket sort with Linked list)

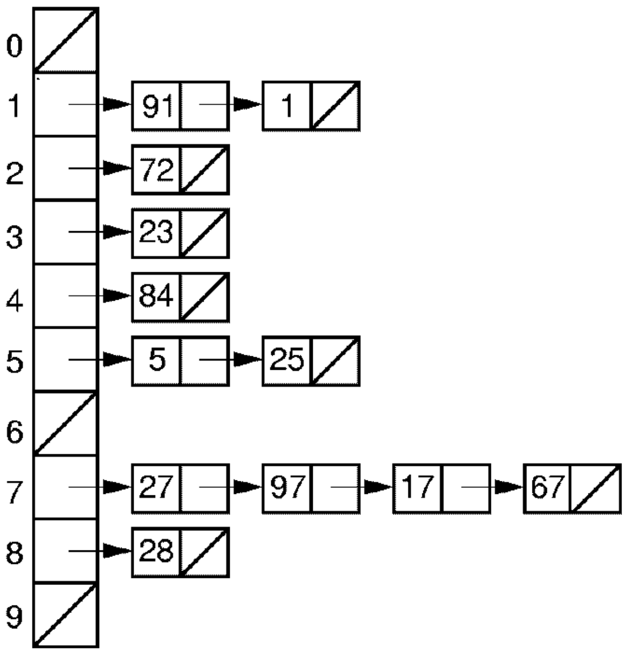
```
// Assume that $0 \le A[i] \le A_{max}$ for all $0 \le i < n$.
Create $A_{max}+1$ linked lists List[0:$A_{max}$];
// Let the value of A[i] determine its destination.
for(i=0; i < n ;i++) Append A[i] to List[A[i]];
for(i=j=0; j $\le A_{max}$ ;j++)
   foreach(x in List[j]) B[i++] = x;      // collect in temp
for(i=0; i < n ;i++) A[i] = B[i];         // copy back to A
```

- It works only when the domain of an input array is discrete.
- Both the running time and space overhead are $\mathcal{O}(n + A_{max})$.
  - The use of array `B[0:n-1]` can be avoided.
  - But the space overhead would still be the same.
- For example, it deteriorates to $\mathcal{O}(n^2)$ if A_{max} approaches to n^2.

Example 5

Sort the following array by the least significant digit.

27 91 1 97 17 23 84 28 72 5 67 25



☞ Can we sort the array without any linked list?

Algorithm 13 (Bucket sort with Arrays)

```
// Assume that $0 \le A[i] \le A_{max}$ for all $0 \le i < n$.
for(j=0; j $\le A_{max}$ ;j++) count[j] = 0;
for(i=0; i < n ;i++) count[A[i]]++;
// Inv:  count[j] = number of A elements whose value is j
for(j=1; j $\le A_{max}$ ;j++) count[j] += count[j-1];
// Inv:  count[j] = number of A elements whose value is $\le$ j
for(i=0; i < n ;i++) B[--count[A[i]]] = A[i];
  // group elements by values in separate partitions of B[]
for(i=0; i < n ;i++) A[i] = B[i];        // copy back to A[]
```

- Use $A_{max}+1$ counters instead of a directory of $A_{max}+1$ lists.
- The 3rd for-loop computes the cumulative sum of count[].
- Now, array B is divided to $A_{max}+1$ partitions, which are delimited by count[0:A_{max}] and will store A elements of the same value separately.
  - B[0:count[0]-1] stores A elements whose value is 0.
  - B[count[0]:count[1]-1] stores A elements whose value is 1.
  - ...

Radix sort

Example 6

Sort Tucson residents by the last 7 digits of their SSN.

- The cost of Bucket sort is $\mathcal{O}(n + A_{max})$ in time and space.
- What if your space budget is tight?
- If Bucket sort is applied to each *digit*, then A_{max} is reduced to 9.

```
for(k=0; k < 7 ;k++)
    apply Bucket sort to the $k^{th}$ least significant digit;
```

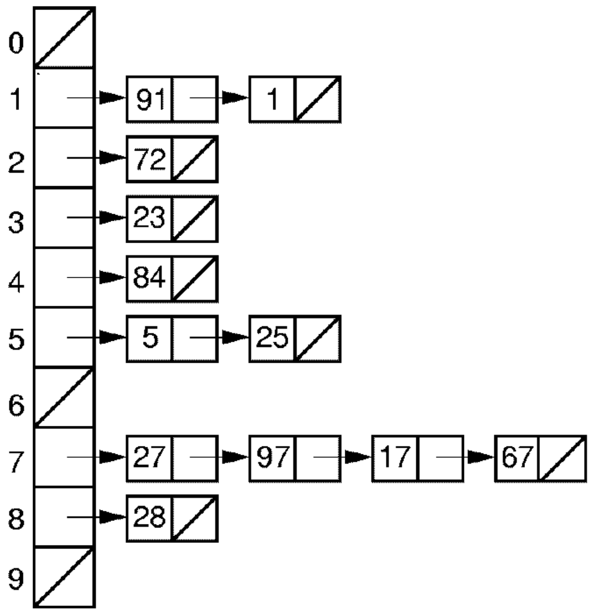
- Will this work? Why not in the most-to-least significant digits order?

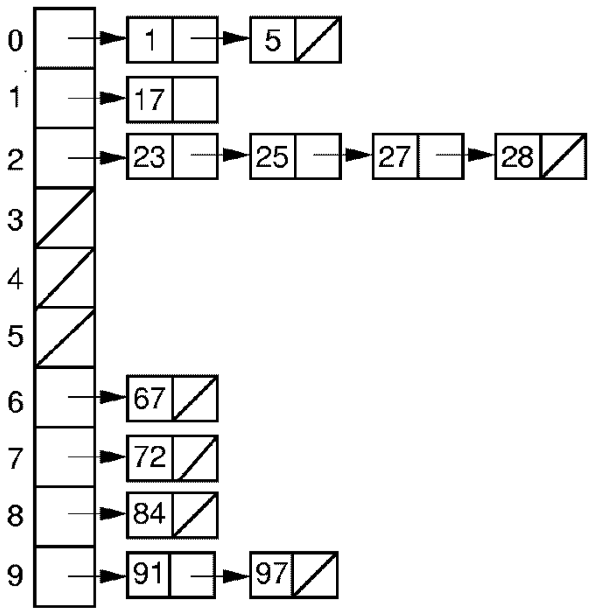
01 33 43 62 85 (sorted by ten digits)
01 62 33 43 85 (sorted by one digits)

Example 7

Sort the following array using Radix sort.

27 91 1 97 17 23 84 28 72 5 67 25



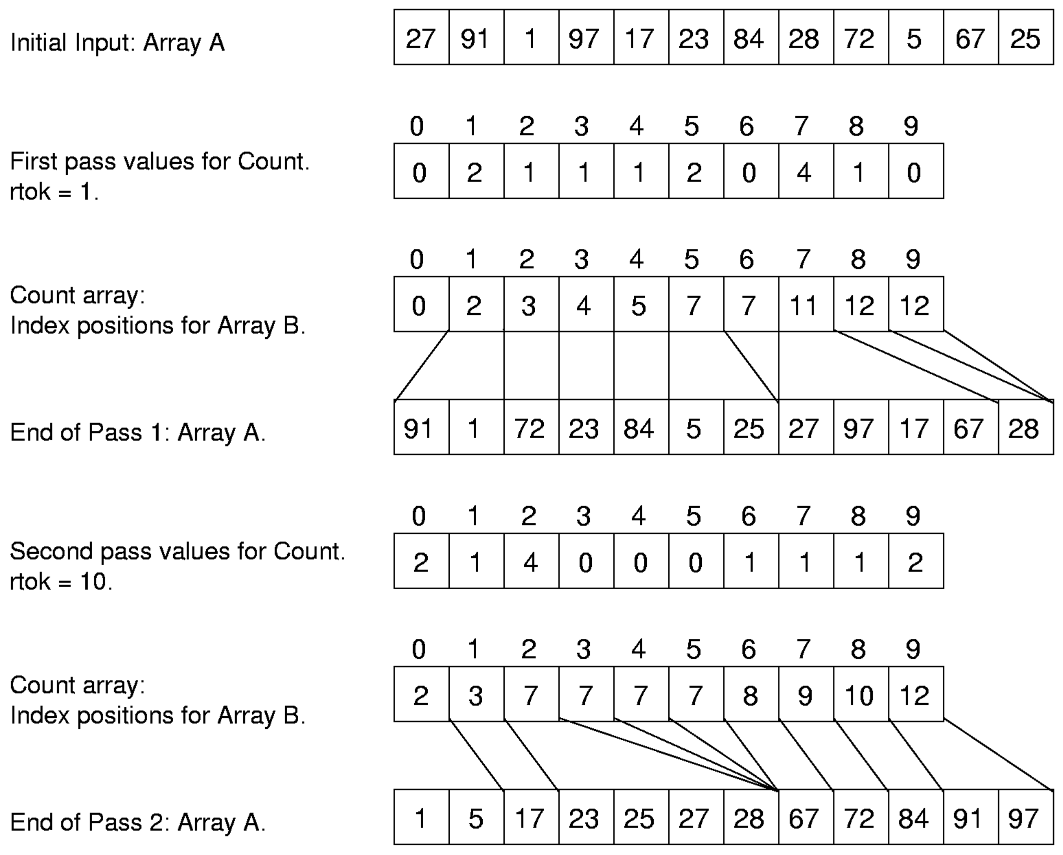


Algorithm 14 (Radix sort with Arrays)

```
for(k=0; k < d ;k++) {           // d is the number of digits
   for(j=0; j < r ;j++)         // r is 10 for decimal numbers
      count[j] = 0;
   for(j=0; j < n ;j++)        // $A_k$[j] = the $k^{th}$ digit of A[j]
      count[$A_k$[j]]++;
   for(j=1; j < r ;j++) count[j] += count[j-1];
   for(j=n-1; j $\geq$ 0 ;j--)
      B[--count[$A_k$[j]]] = A[j];            // collect backwards
   for(j=0; j < n ;j++) A[j] = B[j];                 // copy back
}
```

- For tied elements (with the same value in the k^{th} digit), sort by the k^{th} digit must preserve their relative order, which was determined by the $(k-1)^{th}$ digit. (For example, 23, 25, 27 and 28 in Example 7)

Illustration of Example 7:



Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count. rtok = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array: Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28

Second pass values for Count. rtok = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array: Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97

Remark 1

1. Radix sort (Algorithm 14) is a generalization of Bucket sort with Arrays (Algorithm 13).
2. After the 3rd inner for-loop, `count[j]` is the number of A elements whose k^{th} digit is $\leq$ j
3. Each time Bucket sort is applied, it maintains the relative order of keys that will be added to the same bucket (*i.e.*, the same k^{th} digit).
4. This is done by moving both `j` and `count[A`$_k$`[j]]` in the backward direction.
    - Does the Bucket sort algorithm do that too?

Radix sort: Analysis

- The running time of Radix sort is $\mathcal{O}(d \times (r+n))$. If both d and r are small, it is $\mathcal{O}(n)$.
- If all keys are unique and non-negative, at least $\log_r A_{max}$ digits are required to store them.

$$d = \lceil \log_r A_{max} \rceil \geq log_r n$$

  Since $d \in \Omega(\log n)$, the running time of Radix sort is $\Omega(n \log n)$.
- Is it $\mathcal{O}(n \log n)$ too? No, A_{max} could be much larger than n.

Shellsort

Donald Shell's idea of incremental sort (1959):

- Use a sequence of k increments $h_0, h_1, \ldots h_{k-1}$ such that

$$h_0 > h_1 > \ldots > h_{k-1} = 1.$$

- After an iteration with h_j, the input array A is h_j-sorted.

$$A[i] \leq A[i + h_j] \quad \forall\ 0 \leq i < n - h_j$$

Does it work?

- If A is h_{k-1}-sorted (*i.e.*, 1-sorted), then A is fully sorted.
- Any increments will work as long as the smallest increment is one.

Algorithm 15 (Donald Shell's Shellsort)

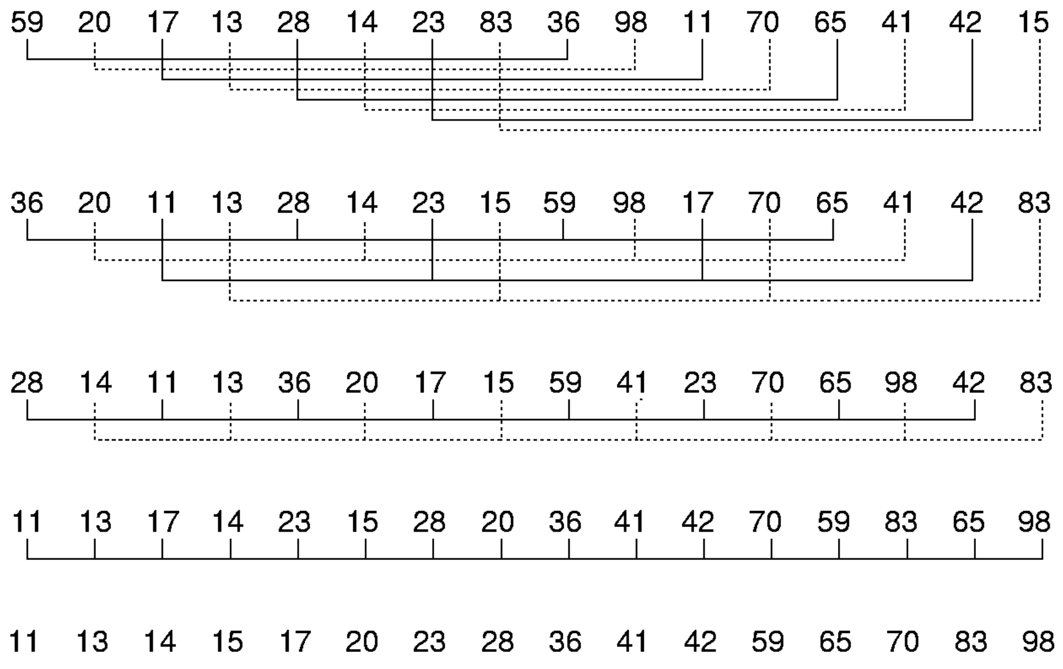
```
for(h=n/2; h > 0 ;h=h/2) {
   for(i=h; i < n ;i++) {                    // Insertion h-sort
      temp = A[i];
      for(j=i-h; j ≥ 0 and A[j] > temp ;j-=h)
         A[j+h] = A[j];
      A[j+h] = temp;
   }
}
```

- Donald Shell's increments $\{h_j\} = \{n/2, n/2^2, n/2^3, \ldots, 1\}$
- The inner for-loop for h_j-sort resembles Insertion sort.
- The inner for-loop is identical to Insertion sort when $h_j = 1$.

Example 8

Sort the following array with Donald Shell's Shellsort algorithm.

59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15



Shellsort: Analysis

Theorem 1

The running time of Shellsort with Donald Shell's increments is $\mathcal{O}(n^2)$.

Proof. In the h_j iteration, the input array is divided into h_j partitions of n/h_j elements each, and each partition is sorted separately by Insertion sort. The cost of an iteration is $\mathcal{O}(h_j \times (n/h_j)^2) = \mathcal{O}(n^2/h_j)$. For all iterations,

$$\mathcal{O}(\textstyle\sum_{j=0}^{k-1} \frac{n^2}{h_j}) = \mathcal{O}(n^2)$$

because

$$\sum_{j=0}^{k-1} \frac{1}{h_j} = (2 + 2^2 + 2^3 + \ldots + n)/n < 2.$$

$\square$

Shellsort: Hibbard's sequence of increments

Problem 2

Shellsort runs Insertion sort multiple times. It can never be better than Insertion sort, can it?

Thomas Hibbard suggested a sequence of numbers any pair of two consecutive numbers in which are relatively prime to each other (1963).

$$\{h_j\} = \{2^i - 1, 2^{i-1} - 1, \ldots, 15, 7, 3, 1\}$$

The running time of Shellsort with Hibbard's sequence is $\mathcal{O}(n^{3/2})$.

Remark 2

Shellsort with Hibbard's sequence was the first algorithm that broke through the $\mathcal{O}(n^2)$ barrier.

Empirical comparison of sorting algorithms

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

- Input: an array of random, sorted (Up), or inversely sorted (Down) integers.
- Insertion is the best for a sorted input; this Bubblesort is a naive one; Quicksort is the fastest and followed by Mergesort and then Heapsort.
- Radix/k denotes a radix sort with base 2^k.

Lower Bounds for Sorting

Problem 3

What is the complexity of a **sorting problem**?

We would like to analyze the cost of a sorting problem as opposed to a specific sorting algorithm (*e.g.*, quicksort, heapsort).

- The upperbound of the fastest *known* algorithm is $\mathcal{O}(n \log n)$ in the worst case.
  $\therefore$ The upperbound of a sorting problem is $\mathcal{O}(n \log n)$.
- To prove that the lowerbound of a sorting problem is $\Omega(n \log n)$, we need to show *any (even not yet invented)* algorithm will take $\Omega(n \log n)$ time in the worst case. In other words, we need to show that there exists no algorithm faster than $\Omega(n \log n)$ in the worst case.

Types of sorting algorithms

Comparison sort:

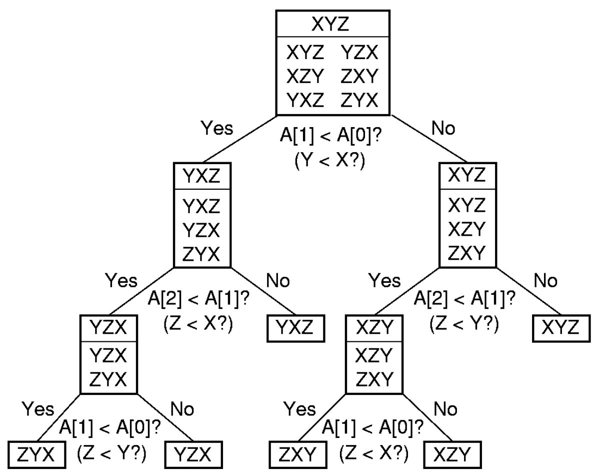
- Only comparisons between elements are used to gain order information about an input array.
- Examples are Insertion, Bubble, Selection, Quicksort, Mergesort, Heapsort.

No-comparison sort:

- Instead of direct comparisons between elements, the value of a specific digit determines the rank of an element in the sorted order.
- Examples are Bucket sort and Radix sort.

Decision Tree for a Comparison Sort

We should know the worst case running time of any comparison sort algorithm. But the trouble is we cannot tell which array is the worst case for all sort algorithms. The decision tree below shows all possible scenarios of *Insertion Sort* applied to an input array of three elements.



- Each box represents the current sequence of elements and all possible sorted orders.
- Each leaf node represents a sorted order produced by Insertion Sort.
- The path length indicates the number of comparisons required.
- The shape of the tree is determined by a chosen sort algorithm.
- *How tall* can the decision tree be?

Theorem 2

The running time of any comparison sort algorithm is $\Omega(n \log n)$ in the worst case.

Proof. Without loss of generality, assume all n elements in the input array are distinct. To determine the worst case running time (measured in the number of comparisons) of a comparison sort algorithm, we need to consider all possible cases of the input array.

The number of distinct input arrays (or permutations) of size n is $n!$, and so is the number of leaf nodes in the decision tree. Since a binary tree of height h has no more than 2^h leaf nodes, the height of a decision tree with $n!$ leaf nodes is no less than $\log(n!)$. From $\log(n!) \geq \frac{n}{2} \log n$, the height is $\Omega(n \log n)$.

$\therefore$ The worst case running time of any comparison sort is $\Omega(n \log n)$. $\square$

External Sort

- Problem: An input data set is too large to fit in main memory. Assume data are stored on a disk drive.
- Portions of the data must be brought into main memory, sorted in memory, and returned back to disk.
- Secondary memory (or disk) is divided into equal-sized blocks (typically 512B sectors or 4KB pages).
- Blocks are a unit of data transfer between memory and disk.
- Primary goal is to minimize I/O operations. The cost of in-memory sort is often ignored.
- An external sort (based on a whole new idea) is desired for minimizing disk accesses.

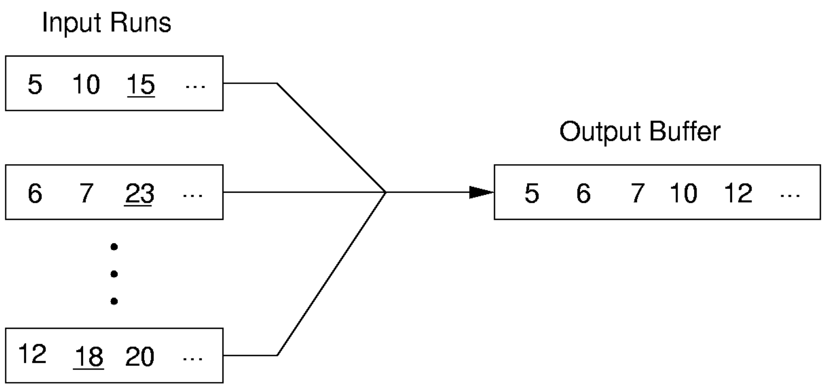
Two-Pass External Mergesort

Pass One:

- Read a segment of an input file to memory and sort internally.
- Write the sorted run back to disk (in separate files for runs).

Pass Two:

- Read a block from each of the sorted runs to memory.
- Perform a multi-way merge for them into an output buffer.
- Write back to disk if the output buffer becomes full.



Analysis of External Mergesort

Memory requirement:

- Pass One: A run should be fit in memory, *i.e.*, $M \geq |run|$. This leads to $\#(runs) \geq |File|/M$.
- Pass Two: A buffer (or memory) block is required for each input run, and another buffer block for output. That is, $M \geq 1 + \#(runs)$.

Memory requirement of the two-pass external Mergesort is

$$M \geq 1 + \#(runs) \geq 1 + |File|/M > |File|/M.$$

$\therefore$ An input file can be sorted in two passes, if $M > \sqrt{|File|}$.

Generally, the number of passes is $\log_M |File|$.

In practice, the number of passes is very small, and the I/O cost of external mergesort is linear.