Lecture 9: Algorithmic Complexity and Sorting

Read: Chpt. 9, Carrano.

Q: If we have several algorithms that can be used to solve a given problem, which algorithm should we use?

Q: If we need to use an ADT to support the implementation of an algorithm, which ADT should we use?

Q: If we need to use an ADT stack to support the implementation of an algorithm, how should we implement the stack class? Should we use array or linked implementation as discussed in class?

A: Pick an algorithm, ADT, and/or implementation that will allow us to obtain a solution to our problem as efficiently as possible.

Q: How do we measure the efficiency (complexity) of an algorithm or data structure?

Most Important Complexity Measures:
Time & memory (space) complexity.

Remark: We will concentrate on time complexity.

Measuring the Efficiency of Algorithms:

1. ***Experimental Profiling:*** Implement the algorithm and then measure the CPU time required in executing the algorithm with respect to a set of input data.
   (Quick and dirty approach): Used in low-level programming courses.

   Major problems:
   Highly machine/language/human/input dependent and results may be unreliable since we can't test our algorithm for all possible inputs.

2. ***Analytical Counting:*** Identify the most basic operation(s) that will dominate the execution of the algorithm and then count it. Algorithms will be compared based on the number of these basic operations used.

   Major problem:
   Mathematically involved; you must know how to count! ☺

Computing the Cost of an Algorithm:

Recall that an algorithm is a sequence of step-by-step instructions.

S_1;

S_2;

Algorithm A:

•

•

•

S_m;

Let C(n) be the total cost in executing an algorithm A with n inputs, and cost(S_i) be the cost in executing the statement S_i, $1 \le i \le m$.

Hence,

$$C(n) = \sum_{i=1}^{m} \cos t(S_i) .$$

Observe that C(n) is a function of n. Computational resource requirement increases as input size n increases!

Some Complications:

1. S_i *is a **conditional statement**:* if-then-else, case, switch, etc.

   cost(S_i) = cost in evaluating the condition + cost in evaluating one of the branches

2. S_i *is a **repetition (loop)**:* do-loop, while-loop, doWhile-loop, etc.

   cost(S_i) = (# times the loop condition is evaluated * cost in evaluating the loop condition) + (# times the loop is evaluated * cost in evaluating the body of the loop)

3. S_i *is a **recursive call**:* S_i *involves direct and indirect recursions*. May need to set up and solve a recurrence equation for cost(S_i).

Q: Let A be an algorithm for a given problem Π. What is the least, most, and average amount of computing resource required in order to execute A?

Some Important Complexity Measures:

Let D_n be the set of all possible inputs of P of size n,

C(I) be the amount of computing resource required to execute A with input I,

Pr(I) be the probability when I is the input to A,

R(n) be the complexity function of A when executed with any input of size n.

1. Best-Case Complexity:

$$R_b(n) = \min_{I \in Dn} C(I)$$

2. Worst-Case Complexity:

$$R_w(n) = \max_{I \in Dn} C(I)$$

3. Average-Case Complexity:

$$R_a(n) = \Sigma_{I \in Dn} Pr(I) * C(I)$$

Notations:

T(n) — time complexity

S(n) —space complexity

Remark: We will concentrate on time complexity only.

Model of Computation:

Assumptions:

1. Sequential computer.
2. All data require same amount of storage in memory.
3. Each datum in memory can be accessed in constant time.
4. Each basic computer operation can be executed in constant time.

Examples:

1. Traversing a linked list with n nodes:

```
Node *cur = head;                   // 1 assignment op
while (cur != NULL)                 // n+1 comparisons op
{   cout << cur→item << endl;       // n writes op
    cur = cur→next;                 // n assignments op
}
```

Assumption:

cost of assignment op $= C_1$,
cost of comparison op $= C_2$,
cost of write op $= C_3$.

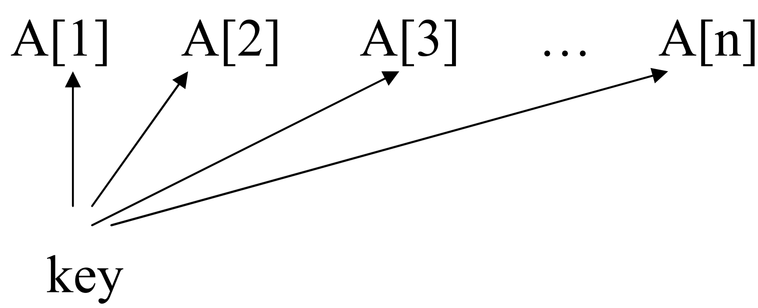
$$
\begin{aligned}
T(n) &= (n+1)C_1 + (n+1)C_2 + nC_3 \\
&= (C_1+C_2+C_3)n + (C_1+C_2) \\
&= K_1 n + K_2. \qquad (K_1 \text{ and } K_2 \text{ are constants.})
\end{aligned}
$$

2. Sequential Searching an Unordered Array:

Input: An array A[1..n] of distinct integers and an integer key.

Output: Return i, $1 \le i \le n$, if A[i] = key; else return 0.

Approach:



Sequential search algorithm:

Comparing key with A[1], A[2], …, A[i] successively until key = A[i] (return i) or key ≠ A[i], for all i, $1 \le i \le n$.

Algorithm:

```
Seq_Search(A: array, key: integer);
    i = 1;
    while i ≤ n and A[i] ≠ key do
        i = i + 1
    endwhile;
    if i ≤ n
        then  return(i)
        else  return(0)
    endif;
end Sequential_Search;
```

Implementation:

Given a partially filled array a[ ] of integers and an integer key (target). Returns the smallest index such that a[index] == target, if exists; otherwise, returns -1.

```
int search(const int a[ ], int number_used, int target)
{
   int index = 0;
   bool found = false;
   while ((!found) && (index < number_used))
      if (target == a[index])
         found = true;
      else
         index++;
   if (found)
      return index;
   else
      return -1;
}
```

Complexity Analysis:

Most Basic Operations:
    Comparisons between key and elements in A.

Simplified Approach:
    Count the # comparisons between the key and A[i].

Let's now count the number of comparisons between the key and A[i].

$T_b(n) = 1$

$T_w(n) = n$

$T_a(n) = (n+1)/2$

Q: What if A is sorted? How do you modify the above sequential search algorithm?
    Observe that we can terminate an unsuccessful search sooner in an ordered array than an unordered array!

3. Search an Ordered Array:

```
int bsearch(const int anArray[], int first, int last, int value)
// Use binary search to search an integer array from
// anArray[first] to anArray[last] for integer key.
// Precondition:  0 <= first, last <= size – 1, where size is
// the max size of array.
// If key is found, return array index; else return -1.
{
  int index;

  if (first > last)                    // base case; key not found
    index = -1;
  else
  {
    int mid = (first + last)/2;    // compute mid for dividing

    if (key == anArray[mid])      // key found
    {
      index = mid;
    }
    else if (key < anArray [mid])   // search left sub-array
          index = bsearch(anArray, first, mid–1, key);
        else                        // search right sub-array
          index = bsearch(anArray, mid + 1, last, key);
  }
  return index;
}  //    end bsearch
```

Let's count the #comparisons between the key and the elements in A again.

Clearly,

$T_b(n) = 1.$

Q: How do we compute $T_w(n)$ for binary search?

Let T(n) be the maximum #comparisons between the key and the elements in A with n elements when binary search algorithm is applies. We have the following recurrence:

$$T(1) = 1,$$

$$T(n) = T(\frac{n}{2}) + 1, n > 1.$$

For simplicity, let's assume that $n = 2^k$, $k \geq 1$. Hence,

$$
\begin{aligned}
& T(n) \\
& = T(\frac{n}{2}) + 1 \\
& = T(\frac{n}{2^2}) + 2 \\
& = T(\frac{n}{2^3}) + 3 \\
& = \ldots \\
& = T(\frac{n}{2^k}) + k \\
& = 1 + \lg n.
\end{aligned}
$$

*This is called the **Method of Repeated Substitutions**.*

In the above examples, $T(n)$'s are represented by a simple mathematical expression. These are called the ***closed-form expression*** of $T(n)$.

Remark: *A closed-form expression of a complexity function is highly desirable since, if $T(n) = f(n)$, where $f(n)$ is an elementary function, $T(n)$ can be computed exactly by substituting n into $f(n)$.*

Q: What if such a closed-form expression of $T(n)$ can not be found (either doesn't exist or much too difficult to compute)?

Use approximation!

In order to provide a guarantee on how much computing resource is needed in executing A, we may want to find an elementary function $f(n)$ such that $T(n) \leq f(n)$ for all n.

We can simplify our computation even further by finding an elementary function $f(n)$ such that $T(n) \leq kf(n)$ for sufficiently large n. This leads us to the asymptotic notations of big-O.

Asymptotic Analysis of Algorithms:

Defn: A function f is an eventually positive function iff there exists a constant n_0 such that $f(n) > 0$ for all $n > n_0$.

Remark: Complexity function is a positive, or eventually positive, function.

Defn: Given a positive function f(n). Then $f(n) = O(g(n))$ iff there exist constants $k > 0$, $n_0 > 0$ such that $f(n) \le k(g(n))$, for all $n \ge n_0$.

Example: Prove that $n^2 + 5n - 268 = O(n^2)$.

Observe that

$$\begin{aligned}
&n^2 + 5n - 268 \\
&\le n^2 + 5n \\
&\le n^2 + 5n^2, n \ge 1 \\
&\le 6n^2, n \ge 1.
\end{aligned}$$

By choosing $k = 6$, $n_0 = 1$, we prove the assertion.

More Examples:

1. $2n^2 - 3n + 10 = O(n^3)$
2. $3\lg n! = O(n\lg n)$
3. $n^2 - 3n^{16} + 2^n = O(2^n)$
4. $n^2 - 36n\lg n - 1024 = O(n^2)$
5. $n^2 - 36n\lg n - 1024 \ne O(n)$
6. $2^{n+1} = O(2^n)$
7. $4^n \ne O(3^n)$

Defn: Given a positive function f(n). Then $f(n) = \Omega(g(n))$ iff there exist constants $k > 0$, $n_0 > 0$ such that $f(n) \geq k(g(n))$, for all $n \geq n_0$.

Theorem: $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.

Example: Prove that $n^2 + 5n - 268 = \Omega(n^2)$.
Observe that

$$\begin{aligned}
& n^2 + 5n - 268 \\
& \geq n^2 - 268, n \geq 1 \\
& = \frac{n^2}{2} + (\frac{n^2}{2} - 268), n \geq 1 \\
& \geq \frac{n^2}{2}, \text{ provided } \frac{n^2}{2} - 268 \geq 0, \text{ or } n \geq \sqrt{536}.
\end{aligned}$$

By choosing $k = \frac{1}{2}$, $n_0 > 24$, we prove the assertion.

More Examples:

1. $2n^2 - 3n + 10 \neq \Omega(n^3)$
2. $3\lg n! = \Omega(n\lg n)$
3. $n^2 - 3n^{16} + 2^n = \Omega(2^n)$
4. $n^2 - 36n\lg n - 1024 = \Omega(n^2)$
5. $n^2 - 36n\lg n - 1024 = \Omega(n)$
6. $2^{n+1} = \Omega(2^n)$
7. $4^n = \Omega(3^n)$

Defn: $f(n) = \Theta(g(n))$ iff there exist constants $k_1 > 0$, $k_2 > 0$, $n_0 > 0$ such that $k_2 g(n) \le f(n) \le k_1 g(n)$ $\forall$ $n \ge n_0$.

Theorem: The following statements are equivalence:

(1) $f(n) = \Theta(g(n))$.
(2) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
(3) $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Example: Since $n^2 + 5n - 268 = O(n^2)$ and $n^2 + 5n - 268 = \Omega(n^2)$, we have $n^2 + 5n - 268 = \Theta(n^2)$.

More Examples:

1. $2n^2 - 3n + 10 = \Theta(n^2)$.
2. $2n^2 - 3n + 10 \neq \Theta(n^3)$.
3. $3\lg n! = \Theta(n\lg n)$.
4. $n^2 - 3n^{16} + 2^n = \Theta(2^n)$.
5. $n^2 - 36n\lg n - 1024 = \Theta(n^2)$.
6. $n^2 - 36n\lg n - 1024 \neq \Theta(n)$.
7. $2^{n+1} = \Theta(2^n)$.
8. $4^n \neq \Theta(3^n)$.

Some Useful Function in Complexity Analysis:

f(n)	***Growth Rate***	***Algorithmic Performance***
n^n	Fastest	Worst
$n!$		
•		
•		
•		
3^n		
2^n	↑	↑
•		
•		
•		
$n^k, k \geq 2$		
n^2		
$n\lg n$		
n		
$\lg n$		
c	Slowest	Best

Analyzing the Performance of an Algorithm:

Try to compute a function f(n) such that T(n) = f(n).

If not possible, try T(n) = Θ(f(n)).

If not possible, try T(n) = O(f(n)).

Given two algorithms A_1 and A_2 with $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$. If f(n) = O(g(n)), then algorithm A_1 ***is potentially more efficient*** than algorithm A_2 for sufficiently large n.

Example: Consider algorithms A_1 and A_2 with complexity $T_1(n) = O(n^3)$ and $T_2(n) = O(n^{1000})$, you can not be 100% sure, that algorithm A_1 is more efficient than algorithm A_2 for sufficiently large n.

Proof. Consider $T_1(n) = n^3 = O(n^3)$ and $T_2(n) = n = O(n^{1000})$, clearly algorithm A_1 is **not** more efficient than algorithm A_2!

Remarks:

- This kind of conclusion can only be drawn with closed-form expression or big-Θ information.
- If $T_1(n) = \Theta(f(n))$, $T_2(n) = \Theta(g(n)$, and $f(n) = \Theta(g(n))$, then algorithms A_1 and A_2 are asymptotically equivalence.
- If $T_1(n) = \Theta(f(n))$, $T_2(n) = \Theta(g(n)$, and $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$, then algorithms A_1 is asymptotically more efficient than algorithm A_2.

Example: Let $T_1(n) = 2^{18}n^2$ and $T_2(n) = 2^n$. Since $2^{18}n^2 = O(2^n)$ and $2^n \neq O(f(2^{18}n^2)$, then $T_1(n)$ is asymptotically more efficient than $T_2(n)$.

Observe that when n is small, $T_2(n)$ is actually more efficient than $T_1(n)$.

Q: How do we find the smallest input size n_0 such that A_1 is faster than A_2, for all $n > n_0$.

Need to find smallest integer $n > 0$ such that $2^{18}n^2 \leq 2^n$.

$$\begin{aligned} 2^{18}n^2 &\leq 2^n \\ \lg 2^{18}n^2 &\leq \lg 2^n \\ \lg 2^{18} + \lg n^2 &\leq n \\ 18 + 2\lg n &\leq n \\ 0 &\leq n - 2\lg n - 18 \end{aligned}$$

Take $n = 2^4$, we have $2^4 - 2\lg 2^4 - 18 = -10$
Take $n = 2^5$, we have $2^5 - 2\lg 2^5 - 18 = 4$.

Hence, $2^4 < n_0 < 2^5$.

Apply binary search to the region $(2^4, 2^5)$ to find $n_0 = 28$.

Sorting:

Given a partially filled array A[0..size-1] of integers. Rearrange the elements in a[] such that
$A[0] \leq A[1] \leq A[2] \leq \ldots \leq A[size-1]$.

Classification of Sorting Algorithms:

1. Comparison-Based:
   Examples: Bubble sort, insertion sort, quick sort
2. Address Calculation:
   Example: Radix sort

Comparison-Based Sorting Algorithms:

Depends on outcome of comparisons among elements in A, algorithms can be modeled using comparison tree.

Dfn: A ***comparison tree*** is a tree used to model computation/algorithm based on comparisons such that

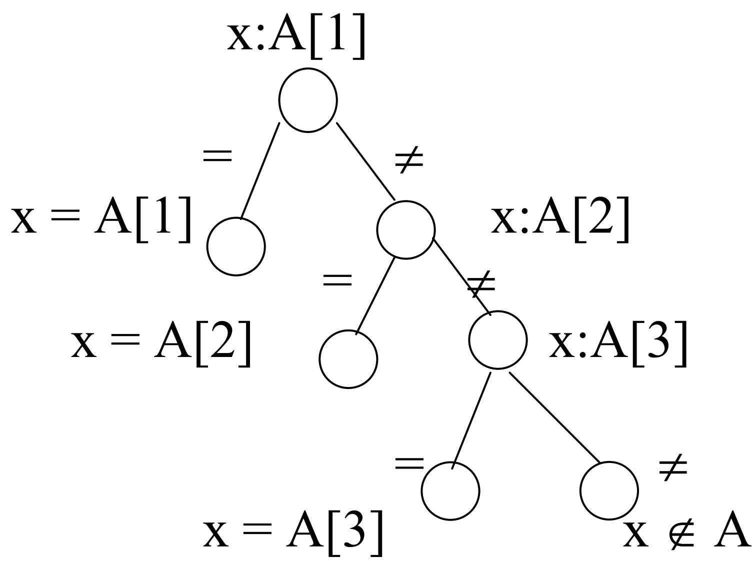
(1) Each non-leaf node represents a comparison, and

(2) Each leaf node represents the result of the computation.

Example: Modeling with Comparison Tree.

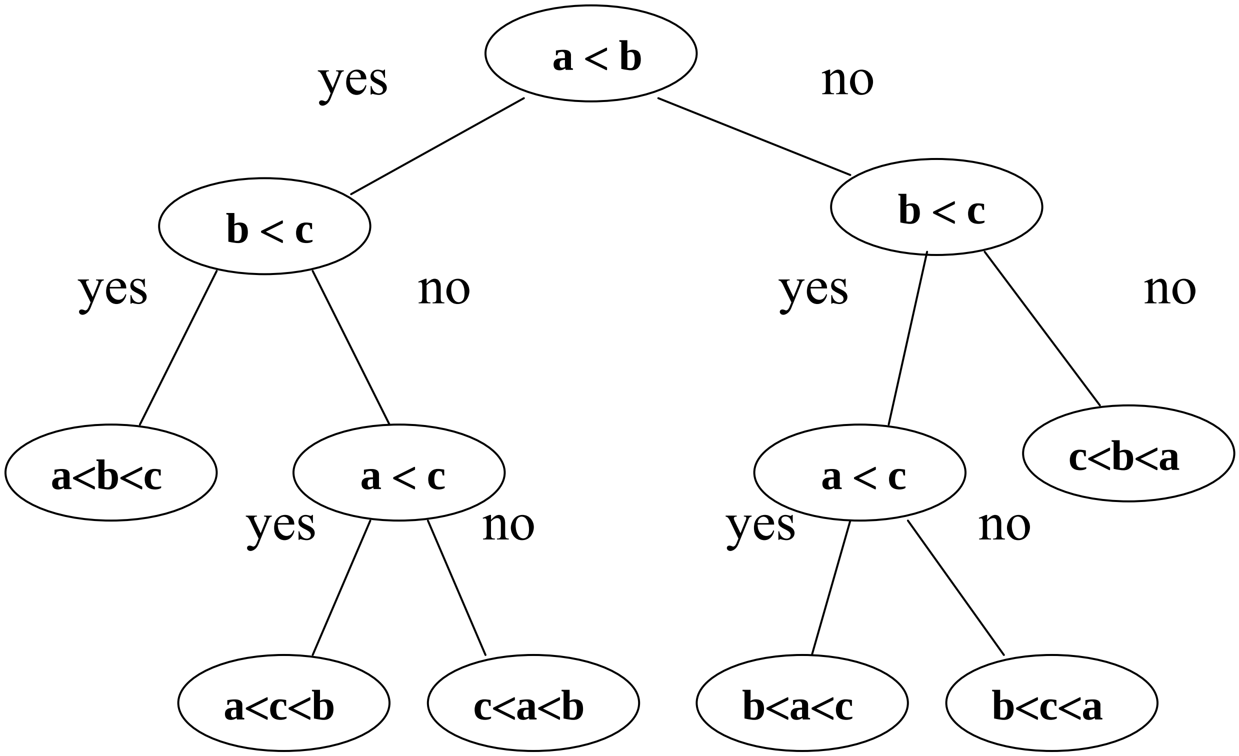
Given an array A[1..n] of distinct integers and a key x. Consider the sequential search algorithm in finding an index i such that A[i] = x if exists.

Take n = 3,



Q: Can you construct the comparison tree for sorting 3 integers?

Example: A comparison tree for sorting 3 distinct integers a, b, c.



Q: How good is a given comparison-based sorting algorithm?

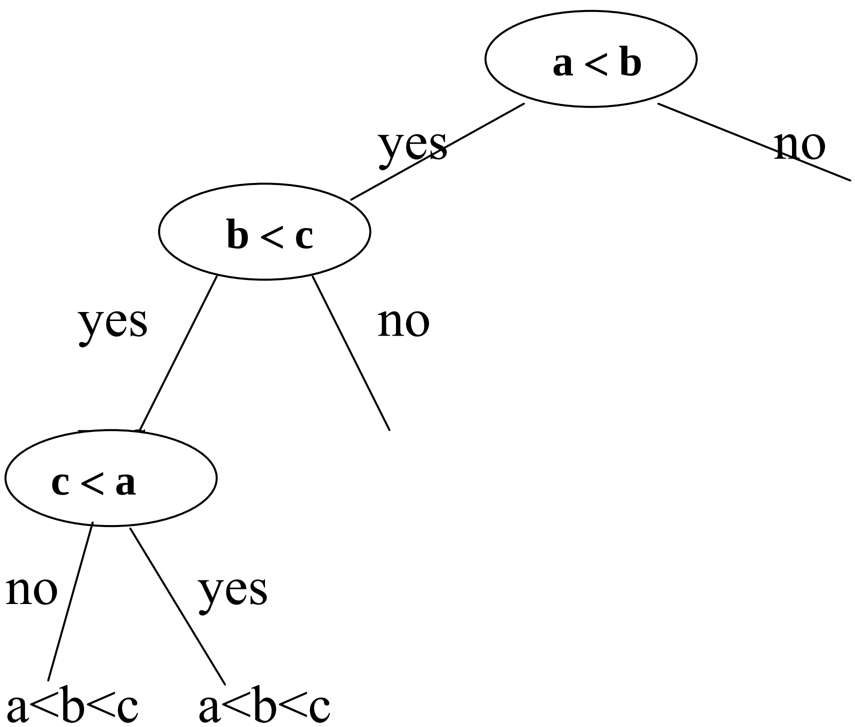
A: Observe that in executing a comparison-based algorithm, starting at the root, computation always terminates at a leaf. Hence, complexity depends on the height of the comparison tree and, a good sorting algorithm must have a corresponding comparison tree with small height.

Optimal sorting algorithm

⇔ Comparison tree with minimum height.

Non-optimal sorting algorithm may contain redundant operations (comparisons) in comparison tree.

Example: A comparison tree for sorting 3 distinct integers a, b, c with redundant comparisons.



Q: How good is a given comparison-based sorting algorithm.

Observations:

1. There are n! permutations in sorting n records.
2. Any comparison tree must contain ≥ n! leaves.
3. Any binary tree T with m leaves, m ≥ 1 must have height $\geq \lceil \log_2 m \rceil$.
4. Any comparison tree must have height $\geq \lceil \log_2 n! \rceil$.
5. Any comparison-based sorting algorithm must perform at least nlgn comparisons.
6. Any O(nlgn) comparison-based sorting algorithm is time-optimal.
7. There exists O(n) address calculation sorting algorithm (radix sort).

Some Simple but O(n^2) Sorting Algorithms:

Input: An array A[0..size-1] of records.

Output: Array A[0..size] in non-decreasing order.

1. Selection Sort:

Algorithm:

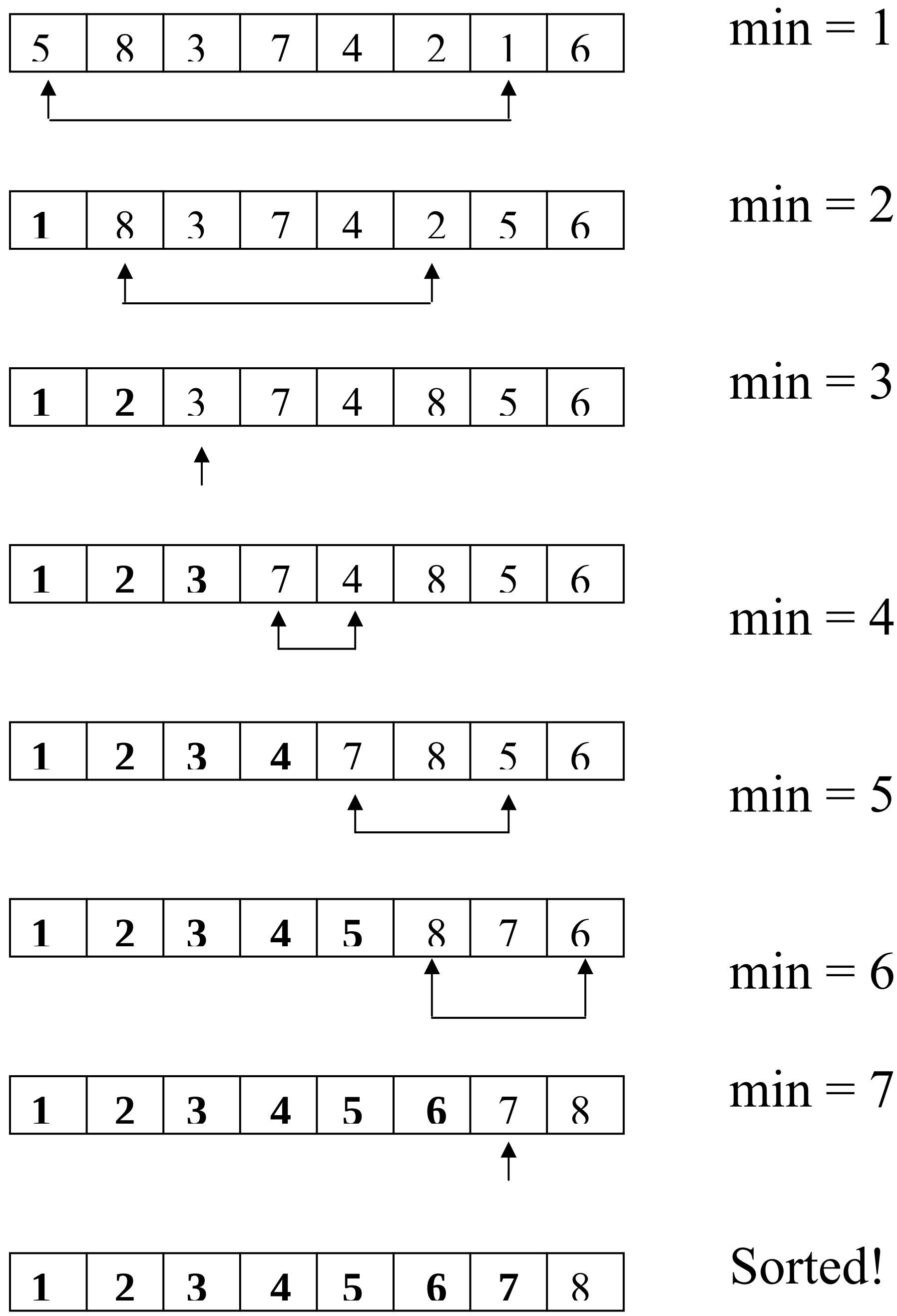
```
for index = 0 to size-2 do
    select min element among A[index], …, A[size-1];
    swap(A[index],min);
endfor;
```

Complexity:

$T_w(n) = T_a(n) = T_b(n) = O(n^2)$.

Remark: Selection sort is a ***priority queue sorting algorithm***. Performance can be improved if a “good” priority queue, such as heap, is used to maintain the records.

Example: Sort an array with keys 5, 8, 3, 7, 4, 2, 1, 6 using selection sort.



```
void sort(int A[ ], int number_used)
{
   int index_of_next_smallest;
   for (int index = 0; index < number_used - 1; index++)
   {
      index_of_next_smallest =
                index_of_smallest(A, index, number_used);
      swap_values(A[index], A[index_of_next_smallest]);
   }
}

void swap_values(int& v1, int& v2)
{  int temp;
   temp = v1;
   v1 = v2;
   v2 = temp;
}

int index_of_smallest(const int A[ ], int start_index,
int number_used)
{
   int min = A[start_index];
   index_of_min = start_index;
   for (int index = start_index + 1; index < number_used;
            index++)
      if (A[index] < min)
      {
         min = A[index];
         index_of_min = index;
```

```
        }
    return index_of_min;
}
```

2. Insertion Sort:

Algorithm:

```
for index = 1 to size-1 do
    insert A[index] into A[0], …, A[index-1];
endfor;
```

Complexity:

$T_w(n) = T_a(n) = O(n^2)$,

$T_b(n) = O(n)$.

Best-case instance:

The array A[] is sorted.

Worst-case instance:

The array A[] is sorted in reversed order.

Implementation:

H.W.

Example: Sort an array with keys 5, 8, 3, 7, 4, 2, 1, 6 using insertion sort.

5	**8**	3	7	4	2	1	6	insert 8
5	8	**3**	7	4	2	1	6	insert 3
3	5	8	**7**	4	2	1	6	insert 7
3	5	7	8	**4**	2	1	6	insert 4
3	4	5	7	8	**2**	1	6	insert 2
2	3	4	5	7	8	**1**	6	insert 1
1	2	3	4	5	7	8	**6**	insert 6
1	2	3	4	5	6	7	8	sorted

3. Bubble Sort:

Algorithm:

```
for i = 1 to size-1 do
   for index = 1 to size-i do
      if A[index] < A[index-1]
            swap(A[index], A[index-1]);
endfor;
```

Complexity:

$T_w(n) = T_a(n) = T_b(n) = O(n^2)$.

Remark:

Observe that if there is no swap during any iteration of bubble sort, the array A must be already sorted. Hence, we can terminate the execution of bubble sort whenever there is no swap in any iteration. By keeping track on the number of swaps, we can improve the best-case complexity of bubble sort to $T_b(n) = O(n)$.

Implementation:

H.W.

Example:

3	**8**	6	2	5
3	8	**6**	2	5
3	6	8	**2**	5
3	6	2	8	**5**
3	6	2	5	**8**
3	**6**	2	5	**8**
3	6	**2**	5	**8**
3	2	6	**5**	**8**
3	2	5	**6**	**8**
3	**2**	5	**6**	**8**
2	3	**5**	**6**	**8**
2	3	**5**	**6**	**8**
2	**3**	**5**	**6**	**8**
2	**3**	**5**	**6**	**8**

4. Divide-and-conquer Sorting Algorithms:

   Basic Idea:

   Given a linear data structure A with n records.
   Divide A into substructures S1 and S2.
   Sort S1 and S2 recursively.

   Q: Is the structure S1||S2 sorted?
   Not necessarily!

   Two cases:

   1. If (keys in S1 ≤ keys in S2), then S1||S2 is already sorted. This is **Quick Sort**.

   2. If no restriction on keys in S1 and S2, then we must merge the two sorted lists S1 and S2 together. This is **Merge Sort**.

4 (a): Quick Sort.

Initial condition: If $|S| = 1$, S is already sorted.

General case: If $|S| > 1$, divide S into two sub-arrays S1 and S2 using some pivot p in A such that S1 contains only those keys that are $< p$ and S2 contains only those keys that are $\geq p$.
Sort S1 and S2 recursively.

Given A[first..last]:

first **last**

| p |

pivotIndex

S1 = A[first..pivotIndex-1], every key in S1 < p
S2 = A[pivotIndex+1,last], every key in S2 ≥ p

Assume that we have a method ***partition(A,first,last,pivotIndex)*** that will return the position of the pivot p in the **sorted** array A.

```
// sort A[first..last] into non-decreasing order

void QuickSort(DataType A[], int first, int last)
{
   int pivotIndex;
   if (first < last)
   {  partition(A,first,last,pivotIndex);
      QuickSort(A,first,pivotIndex–1); // sort S1

      QuickSort(A,pivotIndex+1,last);  // sort S2
   }
}  // end QuickSort
```

Two Fundamental Operations:

Q: How do we select the pivot p in A?
How do we partition A into sub-arrays S1 and S2?

Selecting a Pivot for A:
Given an array A[first..last].
Some general methods in selecting a pivot:
1. Use first element A[first]
2. Use last element A[last]
3. Use middle element A[middle] with middle = (first+last)/2
4. Use a random key among elements in A
5. If |A| > 3, use the median of A[first], A[middle], and A[last]. This is called the median-of-three method.

Example: Given an array A[0..7] with keys 5, 8, 3, 7, 4, 2, 1, 6.
1. Using first item, x = 5.
2. Using last item, x = 6.
3. Using middle item, middle = (0+7)/2 = 3, x = 7.
4. Using a random key among items in A, every key can be used as pivot.
5. Using the median-of-three method, x is the median of {5, 6, 7}, x = 6.

Q: Which method should we use?

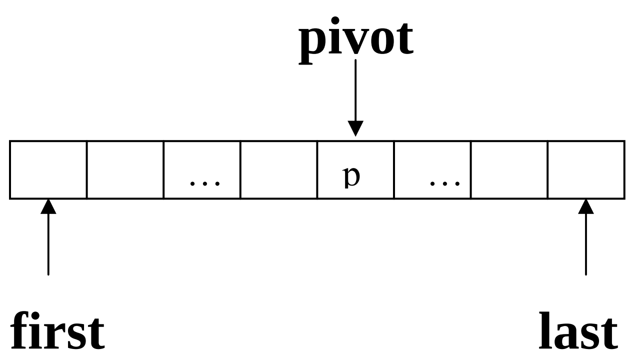
Characteristics of a "good" pivot:

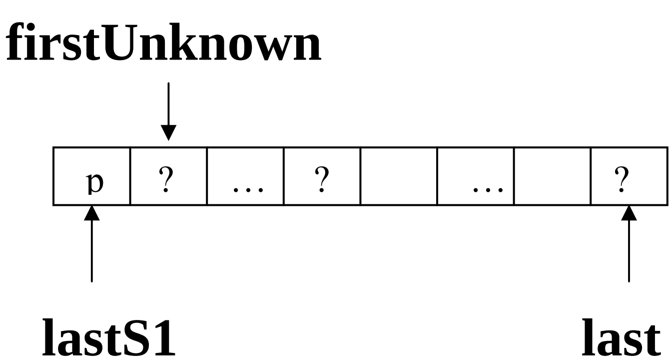
1. The pivot p can be computed in O(1) time.
2. A can be partitioned into S1 and S2 with "roughly" equal sizes.

Remark: Use median-of-three or median-of-five method.

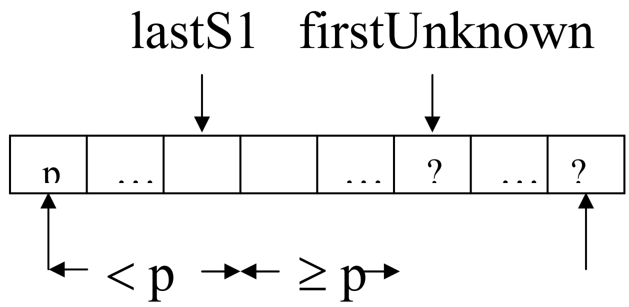
Partitioning A[first..last]:

Initial configuration:





lastS1: Pointing at last element in S1.
firstUnknown: Pointing at current item to be compared with pivot.
Initially, lastS1 = first; firstUnknown = first + 1;
General configuration:




```
if A[firstUnknown] < pivot    //elements out of order
{  lastS1 = lastS1 + 1; //find location to hold A[firstUnknown]
   swap(A[lastS1],A[firstUnknown]);
}
```

Let's consider using the middle key as the pivot.
Algorithm: partition(A,first,last)

```
   middle = (first+last)/2;
   pivot = A[middle];
   swap(A[middle],A[first];
   lastS1 = first;
   firstUnknown = first+1;
   for (; firstUnknown <= last; ++firstUnknown)
   {  if (A[firstUnknown] < pivot)
      {  ++lastS1;
         swap(A[firstUnknown],A[lastS1];
      }
   swap(A[first],A[lastS1]);
   pivotIndex = lastS1;
   } // endPartition
```

Complexity:

Worst-Case Complexity:
If Array a[] is in sorted order, we have

$$T(1) = 0,$$
$$T(n) = T(n-1) + (n-1),\ n > 1.$$

$\therefore T_w(n) = O(n^2)$,

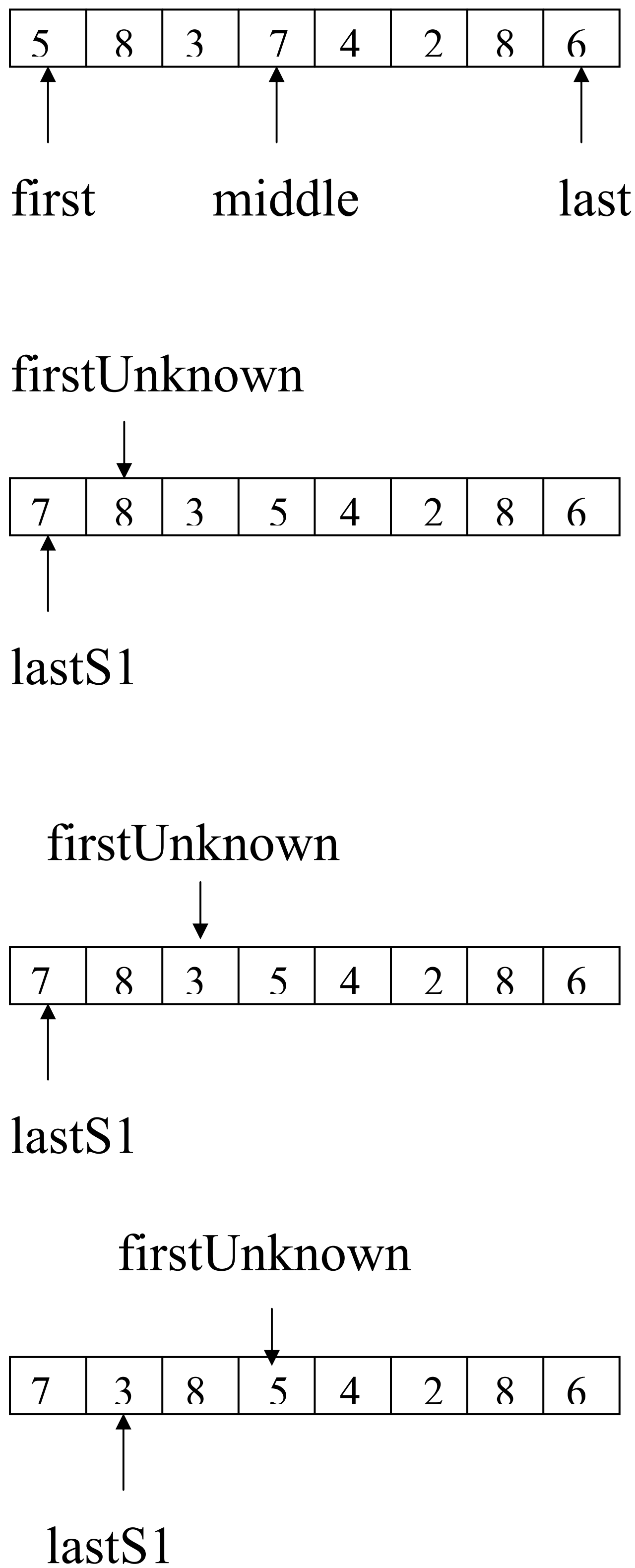
Average-Case Complexity:

$T_a(n) = O(n \lg n)$.

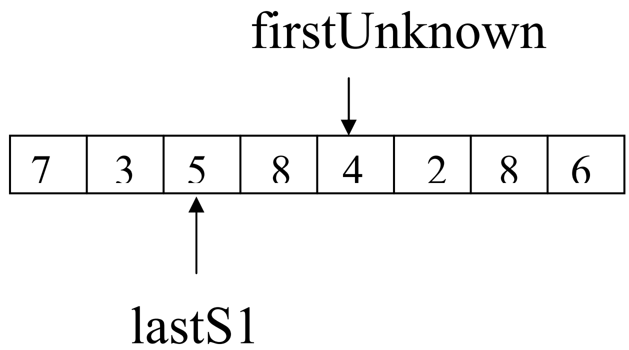
Remark: "Balancing" the sizes of the subproblems in DAC algorithm is very critical!

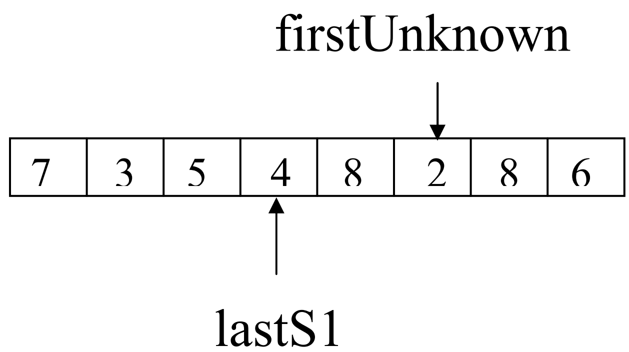
Example:

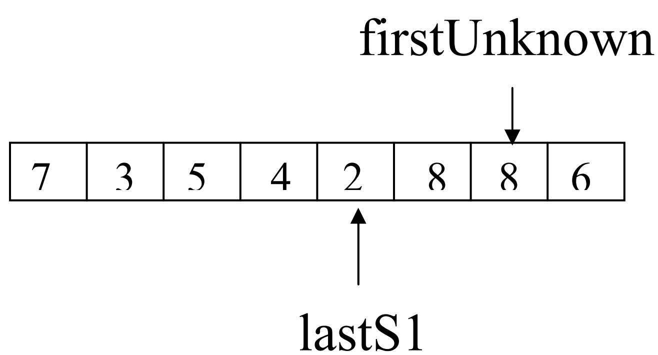
p = 7

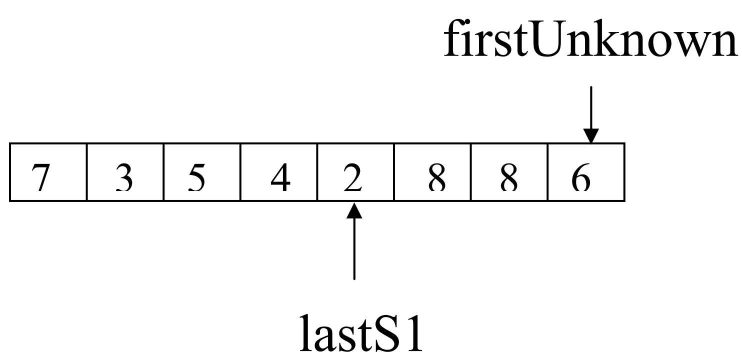


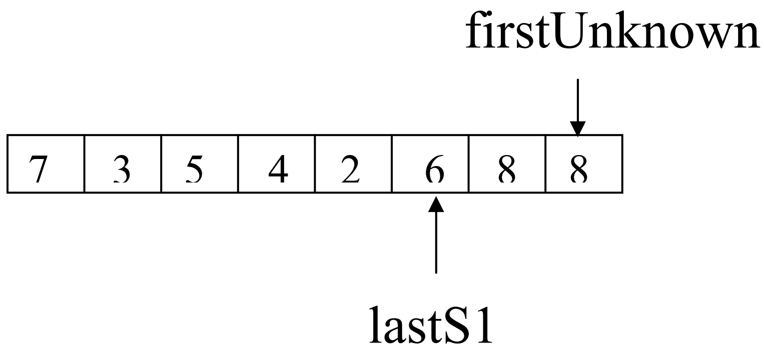
5 8 3 7 4 2 8 6
first
middle
last
firstUnknown
7 8 3 5 4 2 8 6
lastS1
firstUnknown
7 8 3 5 4 2 8 6
lastS1
firstUnknown
7 3 8 5 4 2 8 6
lastS1

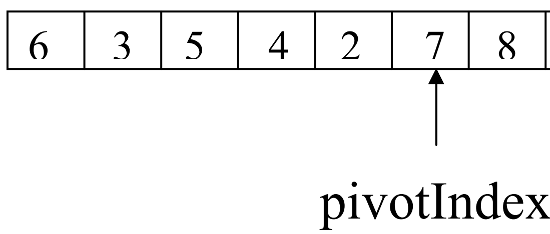

firstUnknown
7 3 5 8 4 2 8 6
lastS1


firstUnknown
7 3 5 4 8 2 8 6
lastS1


firstUnknown
7 3 5 4 2 8 8 6
lastS1


firstUnknown
7 3 5 4 2 8 8 6
lastS1





Hence,
pivotIndex = 5,
S1 = A[0..4],
S2 = A[6..7].

Remark: To improve upon (local) performance, if |A| < 10, use insertion sort.

4 (b): Merge Sort.

Algorithm:

```
mergeSort(A,first,last)
{
   if (first < last)
   {
      mid = (first + last)/2;
      mergeSort(A, first, mid);
      mergeSort(A, mid+1, last);
      merge(A, first, mid, last)
   }
}  // end mergeSort
```

Q: How do we merge the two sorted lists together?

visualize the two sorted lists to be stored in two separate stacks S_1 and S_2;

compare the top elements of the two stacks and pop the smaller one to another list structure L until one of the stacks is empty;

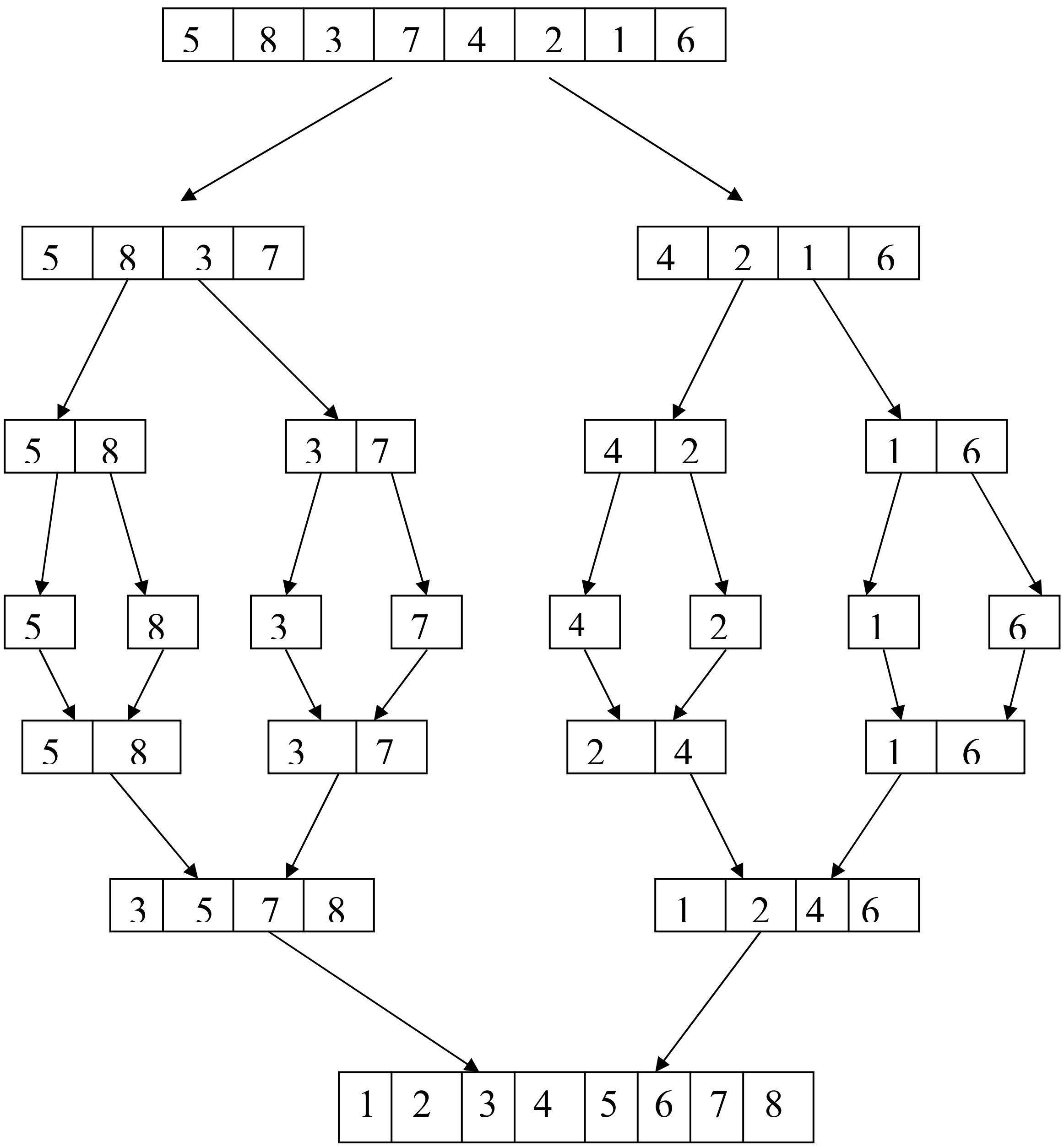
pop, until empty, the remaining non-empty stack to L;

Complexity:

$T_w(n) = 2T_w(n/2) + (n-1) = O(n^2)$.

$T_a(n) = O(n\lg n)$.

Example: Sort an array with keys 5, 8, 3, 7, 4, 2, 1, 6 using merge sort.



5. Address Calculation Sorting Algorithms.

To improve upon the O(nlgn) complexity in sorting, we need to use a different approach! Observe that the executions of all previous sorting algorithms are all based on comparisons among keys to be sorted; no prior knowledge of keys is assumed.

Q: What if we have some knowledge of the keys?

Let $x_1, x_2, \ldots, x_n$ be a set of n items to be sorted according to their keys $k_1, k_2, \ldots, k_n$, respectively.

Assume that for all i, $1 \le k_i \le m$, for some fixed constant integer m.

5 (a): Distribution (Bucket) Sort.

initialize m initially empty buckets (queues), B[1], …, B[m];
for i = 1 to n do
  insert k_i to B[k_i]
endfor;
concatenate all the buckets; // B[1] || B[2] | … || B[m]

Complexity:

Initialize m buckets: $\Theta(m)$
Distributing n items: $\Theta(n)$
Concatenate m buckets: $\Theta(m)$
$T(n) = \Theta(m) + \Theta(n)$.

If $n \geq m$, then $T(n) = \Theta(n)$.

Q: What if $n << m$?

If we take $m = n^k$, $k \geq 2$, $T(n) = \Theta(n^k)$, which is worse than all our previous sorting algorithm!

Remark: You do not want to use distribution sort to sort 100 items with integer keys between 1 and $10^{1,000,000}$.

Remedy: Use radix sort.
Radix sort is a special kind of distribution sort algorithms that can allow us to sort a set of integer keys in the form $a_t a_{t-1} \ldots a_0$ in a given radix (base) m.

5(b): Radix Sort.
Basic Idea:

Let the set of keys be in the form $(a_t a_{t-1} \ldots a_0)_m$. Use m buckets but iterate the distributive sort algorithm k times, each time using a digit of the key $(a_0, a_1, \ldots, a_t)$ as the key for sorting.

Example: Consider sorting {361, 27, 840, 13, 25, 30, 79, 156}. Observe that

m = base = 10,

n = # items = 8,

$x_i = (a_2a_1a_0)_{10}$, $a_i \in \{0, 1, \ldots, 9\}$,

$x_i \leq 10^3 - 1$, implying that k = 3; hence, 3 iterations.

Use 10 buckets and iterates 3 times, according to a_0, a_1, and then a_2.

	Iteration (a_0)	**Iteration (a_1)**	**Iteration (a_2)**
Bucket 0	840, 30		13, 25, 27, 30, 79
Bucket 1	361	13	156
Bucket 2		25, 27	
Bucket 3	13	30	361
Bucket 4		840	
Bucket 5	25	156	
Bucket 6	156	361	
Bucket 7	27	79	
Bucket 8			840
Bucket 9	79		

Concatenated items after 1st iteration:

840, 30, 361, 13, 25, 156, 27, 79

Concatenated items after 2nd iteration:

13, 25, 27, 30, 840, 156, 361, 79

Sorted after 3rd iteration!

Summary:
Worst-Case Complexity of Sorting Algorithms:

	In sorted order	**In reversed sorted order**
Bubble sort	$O(n)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$
Merge sort	$O(nlgn)$	$O(nlgn)$
Quick sort	$O(n^2)$	$O(n^2)$
Radix sort	$O(n)$	$O(n)$

Sorted order = Non-decreasing order.

Complexity of Sorting Algorithms for Random Data:

	$T_b(n)$	**$T_w(n)$**	**$T_a(n)$**
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$
Quick sort	$O(nlgn)$	$O(n^2)$	$O(nlgn)$
Radix sort	$O(n)$	$O(n)$	$O(n)$

Stability of Sorting Algorithms:

A stable sorting algorithm is a sorting algorithm that preserves the relative ordering of items with the same values. Hence, if two items x and y are having the same value with x preceding y in an input list, then x will remain preceding y in the sorted list.

Sorting Algorithm	**Stability**
Bubble sort	Yes
Insertion sort	Yes
Selection sort	Yes
Heap sort	No
Merge sort	Maybe
Quick sort	Maybe
Radix sort	Yes