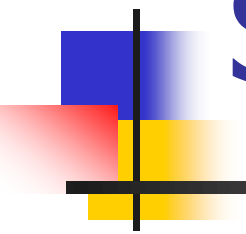


Sorting

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

Terminology

- List: a collection of records
  - ex) Student list of CS206 Data Structure
- Record: having one or more fields
  - ex) a student's data (name, age, sex, number …)
- Field: data
  - ex) name, age, …
- Key: distinguishing data
  - ex) name, number
- Sorting: making ordered list by the key
- Data type:

```
class Element
{
public
   int getKey() const { return key ;};
   void setKey(int k) { key = k;};
private:
   int key;
    // other fields
    .
    .
};
```

Motivation

- Sequential Search

```
template <class E, class K>
int SeqSearch (E *a, const int n, const K& k)
{ // Search a[1:n] from left to right. Return least i such that
  // the key of a[i] equals k. If there is no such i, return 0.
     int i;
     for(i=1; i<=n&&a[i]!=k; i++);
     if(i>n) return 0;
     return i;
}
```

Program 7.1: Sequential search

  - Analysis
    - The worst case time : O(n)
- Two ways of storage and search
  - Sequential
  - Nonsequential

Motivation (Cont.)

- Sorting
  - Two uses of sorting
    - As an aid in searching
    - As a means for matching entries in lists
      - (ex. Comparing two lists)
  - If the list is sorted, the searching time could be reduced.
    - From O(n) to $O(\log_2 n)$
      - ex) Binary Search : $O(\log_2 n)$

Example 1

- Directly comparing the two unsorted lists

```
void Verify1(Element *l1, Element *l2, const int n, const int m)
{ // Compare two unordered lists l1 and l2 of size n and m, respectively.
    bool *marked = new bool[m+1];
    fill(marked+1, marked+m+1, false);

    for(i=1; i<=n; i++)
    {
        int j = SeqSearch(l2, m, l1[i]);
        if(j==0) cout << l1[i] << " not in l2 " << endl;
        else
        {
            if(!Compare(l1[i], l2[j])
                cout << "Discrepancy in "  << l1[i] << endl;
            marked[j] = true; // mark l2[j] as being seen
         }
    }
    for(i=1; i<=m; i++)
        if(!marked[i]) cout << l2[i] << "not in l1." << endl;
    delete [] marked;
}
```

Program 7.2: Verifying two lists using a sequential search

Example 2

- Directly comparing the two sorted lists

```
void Verify2(Element *l1, Element *l2, const int n, const int m)
{ // Same task as Verify1. However, this time we fist sort l1, l2.
    Sort(l1, n);  // sort into increasing order of key
    Sort(l2, m);
    int i=1, j=1;
    while ( (i<=n) && (j<=m) )
        if (l1[i] < l 2[j]) {
            cout << l1[i] << " not in l2" << endl;
            i++;
        }
        else if (l1[i] > l2[j]) {
            cout << l2[j] << " not in l1" << endl;
            j++;
        }
        else  { // equal keys
            if (!Compare(l1[i], l2[j]))
                cout << "Discrepancy in " << l1[i] << endl;
            i++; j++;
        }
    if (i<=n) OutputRest(l1, i, n, 1);          // output records i through n of l1
    else if (j<=m) OutputRest(l2, j, m, 2);     // output records j through m of l2
}
```

Program 7.3: Fast verification of two lists

Example 1&2

- Time Complexity
  - Example 1
    - O(nm)
  - Example 2
    - $O(t_{sort}(n) + t_{sort}(m) + n + m)$

Another Example: Binary Search

- Input : sorted list
- Output : searched element

```
1 int BinarySearch (int *a, const int x, const int n)
2 // Search the sorted array a [0], ..., a [n-1] for x
3 {
4     for (int left = 0, right = n - 1; left <= right;) { // while more elements
5         int middle = (left + right)/2;
6         switch (compare (x, a[middle])) {
7             case '>': left = middle + 1; break; // x > a[middle]
8             case '<': right = middle - 1; break; // x < a[middle]
9             case '=': return middle; // x == a[middle]
10        } // end of switch
11    } // end of for
12    return -1; //  not found
13 } // end of BinarySearch
```

- Analysis : $O(\log_2 n)$

Sorting Problem

- **Input:** A sequence of n numbers $<a_1, a_2,\ldots,a_n>$
- **Output:** A reordering $<a'_1, a'_2,\ldots,a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- There are many sorting algorithms
  - Insertion sort
  - Merge sort
  - Quick sort

Insertion Sort Algorithm

- It uses an incremental approach!
- A sequence of n numbers A[1..n]
- It consists of n−1 passes
  - For pass j= 2 through n, it ensures that the elements in A[1..j] are in sorted order
  - Use the fact that the elements in A[1..j − 1] are already known to be in sorted order

Insertion Sort Algorithm

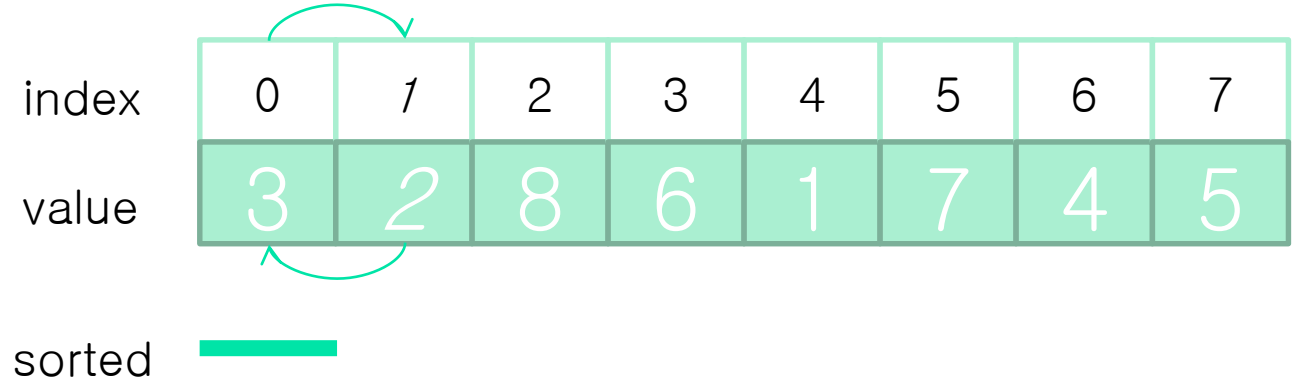
```
INSERTION-SORT(A)
1     for j = 2 to A.length
2         key = A[j]
3         // Insert A[j] into the sorted sequence A[1..j-1]
4         i = j - 1
5         while i>0 and A[i]>key
6               A[i+1] = A[i]
7               i = i-1
8         A[i+1] ← key
```

Insertion Sort

index	0	1	2	3	4	5	6	7
value	3	2	8	6	1	7	4	5

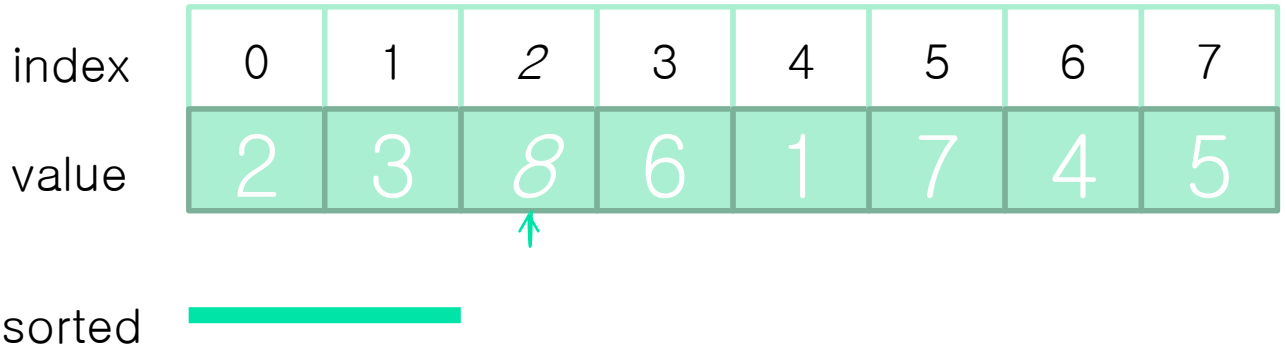
Insertion Sort

Iterator = 1



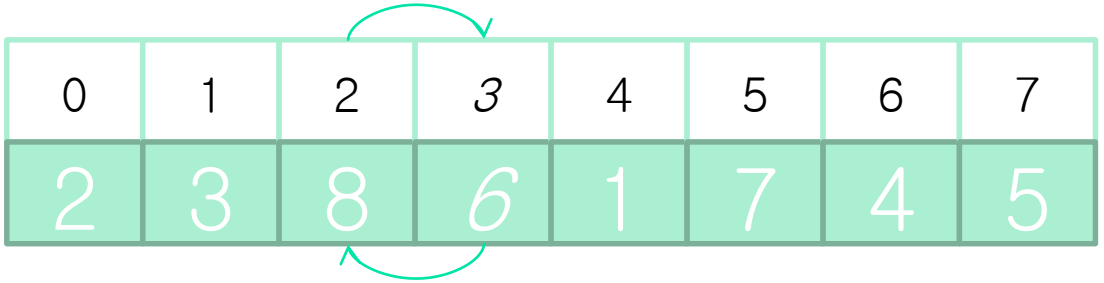
Insertion Sort

Iterator = 2



Insertion Sort

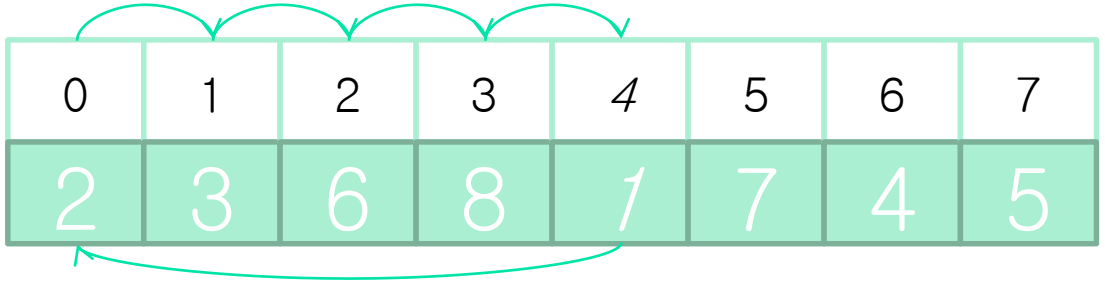
Iterator = 3



sorted

Insertion Sort

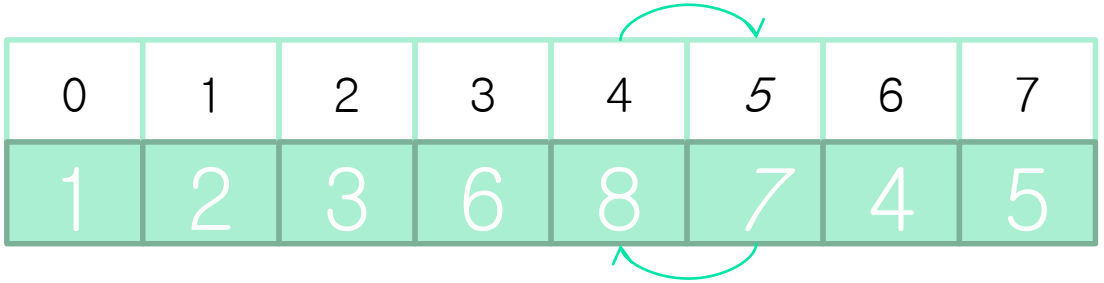
Iterator = 4



sorted

Insertion Sort

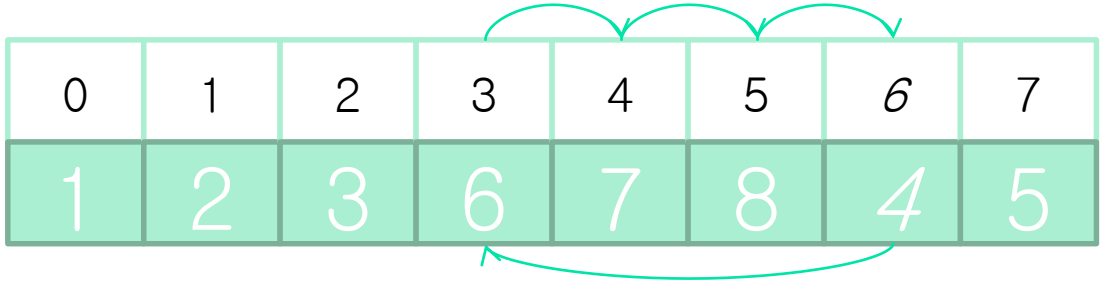
Iterator = 5



sorted

Insertion Sort

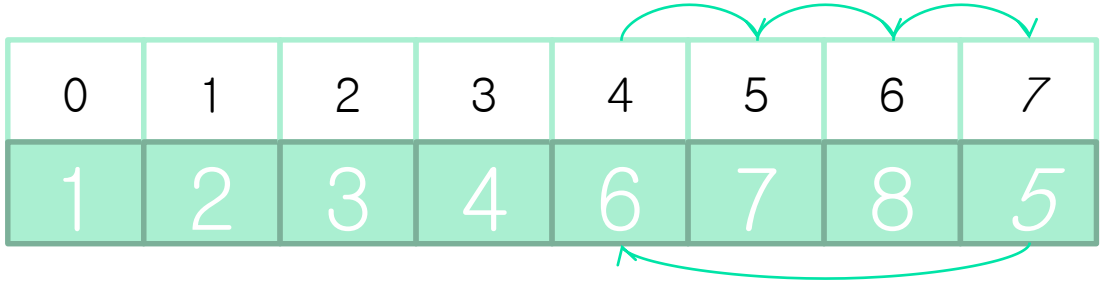
Iterator = 6



sorted

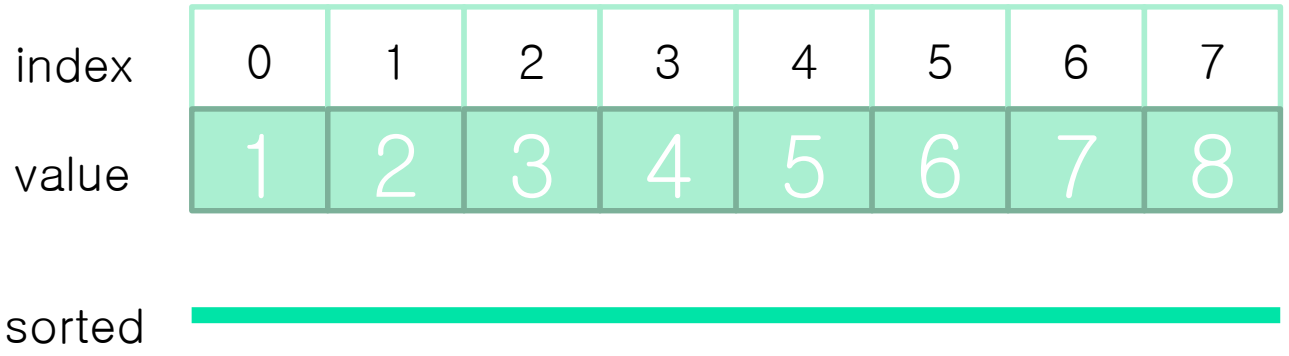
Insertion Sort

Iterator = 7



sorted

Insertion Sort



Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

p=4

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

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

p=4

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

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

p=4
tmp=1

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

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

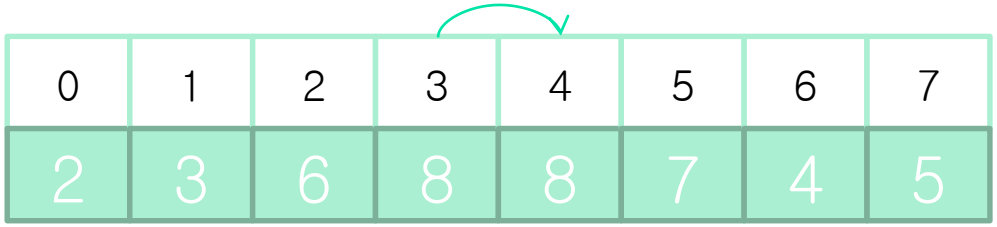
p=4
tmp=1
j=4

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

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

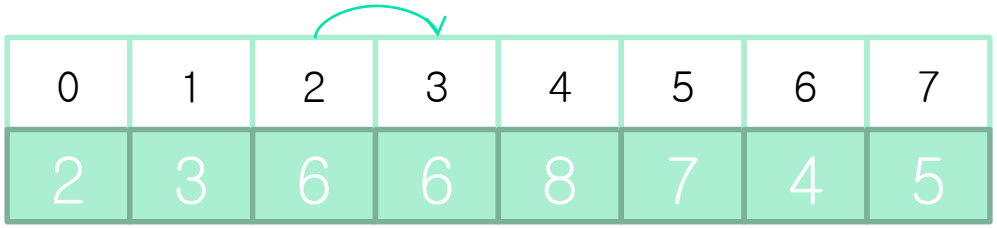
p=4
tmp=1
j=4



Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

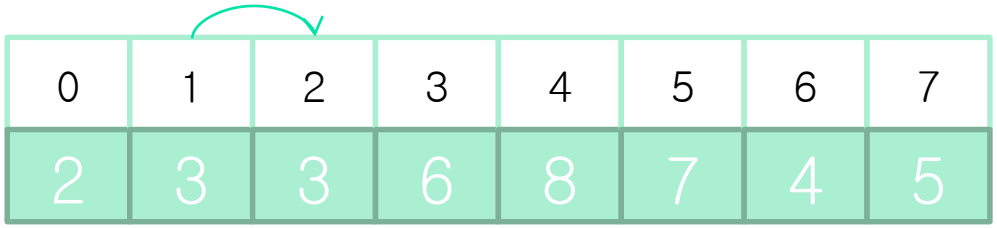
p=4
tmp=1
j=3



Insertion Sort

```
public static void insertionSort( int [] a)
1    {
2            int j;
3            for (int p = 1; p < a.length; p++)
4            {
5                    int tmp = a[p];
6                    for (j = p; j > 0 && tmp < a[j-1]; j--)
7                            a[j] = a[j-1];
8                    a[j] = tmp;
9            }
10   }
```

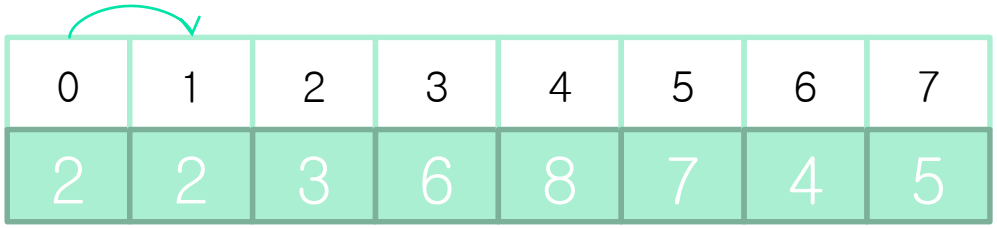
p=4
tmp=1
j=2



Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

p=4
tmp=1
j=1



Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

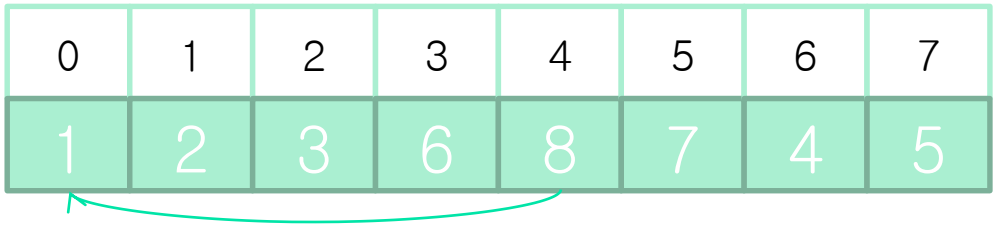
p=4
tmp=1
j=0

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

Insertion Sort

```
public static void insertionSort( int [] a)
1     {
2               int j;
3               for (int p = 1; p < a.length; p++)
4               {
5                         int tmp = a[p];
6                         for (j = p; j > 0 && tmp < a[j-1]; j--)
7                                   a[j] = a[j-1];
8                         a[j] = tmp;
9               }
10    }
```

p=4
tmp=1
j=0



Insertion Sort Algorithm

Original	34	8	64	51	32	21	Position Moved
After j =2	8	34	64	51	32	21	1
After j = 3	8	34	64	51	32	21	0
After j = 4	8	34	51	64	32	21	1
After j = 5	8	32	34	51	64	21	3
After j = 6	8	21	32	34	51	64	4

Insertion Sort Algorithm

- Time complexity
  - Worst case: $O(n^2)$
  - Best case: $O(n)$
  - Average case: $O(n^2)$

Sorting Terminology

- Record : $R_1, R_2, \ldots, R_n$
- List of records : $(R_1, R_2, \ldots, R_n)$
- Key value : K_i
- Ordering relation : $<$
- Transitive relation : $x < y$ and $y < z \Rightarrow x < z$
- Sorting Problem :
  - Finding a permutation σ such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$
  - The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$
- Stable sorting : σ_s
  - $K_{\sigma s(i)} \leq K_{\sigma s(i+1)}$, $1 \leq i \leq n-1$
  - If $i < j$ and $K_i == K_j$, 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 : $2_2, 2_1, 3, 6, 7, 8$

Insertion Sort

- Assume that ∃ a sorted list (R_1, R_2, …, R_i).
- Add one element e.
- Artificial record R_0 with key K_0 = MININT(the smallest number)

```
template <class T>
void Insert(const T& e, T *a, int i)
{ // Insert e into the ordered sequence a[1:i] such that the
  // resulting sequence 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<a[i])
     {
          a[i+1] = a[i];
          i--;
     }
     a[i+1] = e;
}
```

Program 7.4: Insertion into a sorted list

Insertion Sort (Cont.)

```
template <class T>
void InsertionSort(T *a, const int n)
{ // Sort a[1:n] into nondecreasing order.
     for (int j=2; j<=n, j++) {
          T temp = a[j];
          Insert(temp, a, j-1);
     }
}
```

Program 7.5: Insertion Sort

7.2 Insertion Sort Example

- Example 7.1

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**

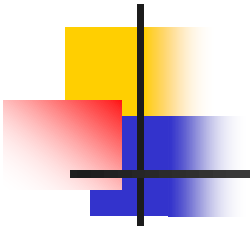
Insertion Sort Analysis

- Time Complexity Analysis
  - Insert(e, list, i) $\Rightarrow$ i+1 comparisons
  - Insertionsort(list, n) $\Rightarrow$ n-1 times

  $O(\sum_{i=1}^{n-1}(i+1)) = O(n^2)$
- Useful when the given list is partially ordered.
- Stable sorting method
- Useful for small size sorting ($n \leq 20$)

Divide and Conquer

- We solve a problem recursively by applying three steps at each level of the recursion.
  - Divide the problem into a number of subproblems that are smaller instances of the same problem
  - Conquer the subproblem by solving them recursively
    - If the problem sizes are small enough (i.e. we have gotten down to the base case), solve the subproblem in a straightforward manner
  - Combine the solutions to the subproblems into the solution for the original problem



Recurrences

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Recurrences give us a natural way to characterize the running times of divide-and-conquer algorithms.
- Thus, they go hand in hand with the divide-and-conquer paradigm.

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1
- If a recursive algorithm divide subproblems into unequal sizes, such as a 2/3-to-1/3 split and combine steps takes linear time, such an algorithm give rise to the recurrence

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1
- If a recursive algorithm divide subproblems into unequal sizes, such as a 2/3-to-1/3 split and combine steps takes linear time, such an algorithm give rise to the recurrence
  - $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \Theta(n)$

Recurrences

- If a recursive version of linear search linear search algorithm creates just one problem containing only one element fewer than the original problem, each recursive call would take constant time plus the time for the recursive calls it makes.
- Such an algorithm yields the recurrence
  - $T(n) = T(n-1) + \Theta(1)$

The methods for Solving Recurrences

- Brute-force method
- Substitution method
- Recursion tree method

Inequality Recurrences

- $T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Because such a recurrence states only an upper bound on T(n), we couch its solution using O-notation rather than Θ-notation
- $T(n) \geq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Because the recurrence gives only a lower bound on T(n), we use Ω-notation in its solution

Technicalities in Recurrences

- In practice, we neglect certain technical details
  - If we call MERGE-SORT on n elements, when n is odd, we end up with subproblems of size $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$.
  - Technically, the recurrence describing the worst-case running time of MERGE-SORT is
    - $T(1) = 1$

      $T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + n$ for n > 1
  - For convenience, we omit floors, ceilings and statements of the boundary conditions of recurrences and assume that T(n) is constant for small n.

Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

$$= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

...

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$= n + n \lg n$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Another Brute-force Method

$$T(1) = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n \text{ for n > 1}$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

… …

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

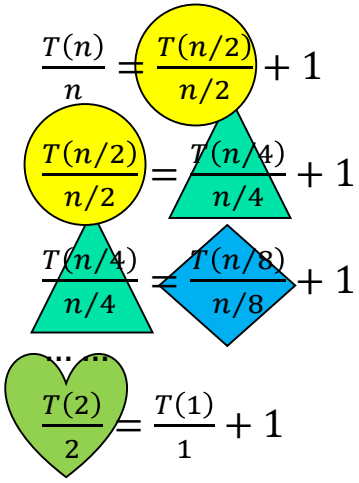
… …

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1



Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

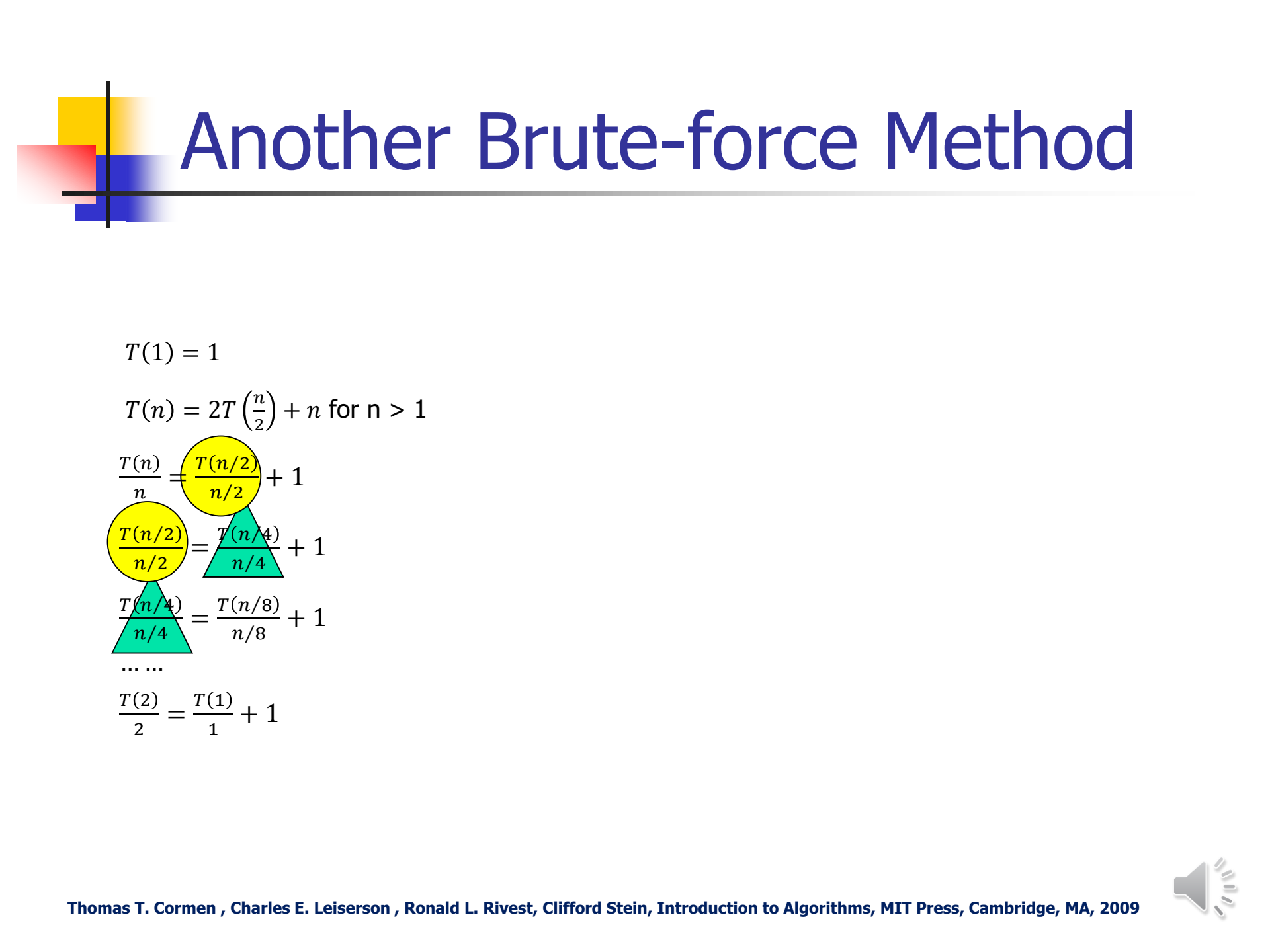
$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$

$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$

$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$

… …

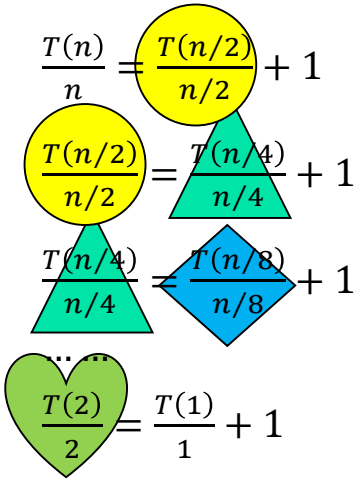
$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$



Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1



Divide by n

$\frac{T(n)}{n} = \frac{T(1)}{1} + \lg n$

Thus, $T(n) = n + n\log n = \Theta(n \lg n)$

Substitution Method

- Comprises two steps:
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works
- We can use the substitution method to establish either upper or lower bounds on a recurrence.

Making a Good Guess

- Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty
  - We might start with T(n) = $\Omega(n)$
  - We can prove T(n) = $O(n^2)$
  - Then, we can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically tight solution of T(n) = $\Theta(n \lg n)$

Substitution Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

- We guess that the solution is T(n) = O(n lg n).
- The substitution method requires us to prove that $T(n) \leq c\, n \lg n$ for an appropriate choice of the constant c > 0.
  - Base case: Examine later
  - Induction hypothesis: $T(m) \leq c\, m \lg m$ holds for all positive for $m < n$

Substitution Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

- We guess that the solution is T(n) = O(n lg n).
- The substitution method requires us to prove that $T(n) \leq c\, n \lg n$ for an appropriate choice of the constant c > 0.
  - Base case: Examine later
  - Induction hypothesis: $T(m) \leq c\, m \lg m$ holds for all positive for $m < n$
  - Induction step:

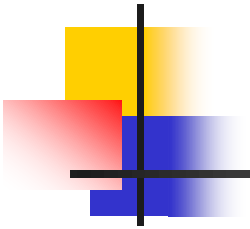
$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&\leq 2c\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + n \\
&= c\, n \lg\left(\frac{n}{2}\right) + n \\
&= c\, n \lg n - c\, n \lg 2 + n \\
&= c\, n \lg n - c\, n + n \\
&\leq c\, n\, lgn \quad \text{(for c} \geq 1\text{)}
\end{aligned}
$$

$\lg\left(\frac{\beta}{\alpha}\right) = \lg(\beta) - \lg(\alpha)$

Substitution Method

- Base case revisited
  - $T(1) \leq c\,1\lg 1 = 0$ wrong!!
  - The base case of our induction proof fails to hold
- What should we do?
  - We can overcome this obstacle in proving an inductive hypothesis for a specific boundary condition with only a little more effort.
  - We are interested in asymptotic behavior.
  - Remove the difficult boundary condition from induction proof.
  - We do so by first observing that for n > 3, the recurrence does not depend directly on T(1).
  - Thus, we can replace T(1) by T(2) and T(3) as the base cases in the induction proof.
  - From the recurrence, we have T(2) $\leq$ c 2 lg 2 and T(3) $\leq$ c 3 lg 3.
  - Any choice of $c \geq 2$ suffices for the base cases of n=2 and n=3.
  - $T(n) \leq c\,n \log n$ for $c \geq 2$ and n ≥ 2

Binary Search



Binary Search

- Look in the middle
  - Case 1: If X is less than the item in the middle, look in the subarray to the left of the middle.
  - Case 2: If X is greater than the item in the middle, look in the subarray to the right of the middle.
  - Case 3: If X is equal to the item in the middle, we have a match.

Binary Search Algorithm

```
BINARY-SEARCH(A, low, high, X)
1.    if low == high
2.       if A[low] == X
3.          return low
4.       else
5.          return NOT_FOUND
6.    else
7.       mid = ( low + high ) / 2
8.       if A[mid] > X
9.          return BINARY-SEARCH(A, low, mid-1, X)
10.    else if A[mid] == X
11.         return mid
12.    else
13.         return BINARY-SEARCH(A, mid+1, high, X)
```

Worst Case Time Complexity

$T(1) = 1$

$T(n) = T\left(\frac{n}{2}\right) + 1$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$= T\left(\frac{n}{2^2}\right) + 2$$

$$= T\left(\frac{n}{2^3}\right) + 3$$

...

$$= T\left(\frac{n}{2^k}\right) + k$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n

$$= \Theta(\lg n)$$

Merge Sort

- **Divide**:

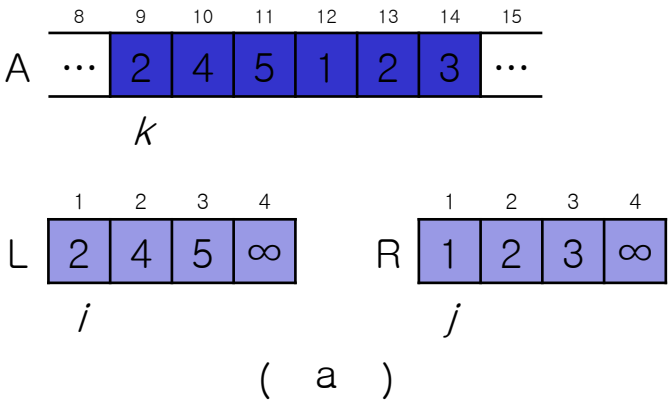
  If S has at least two elements, remove all the elements from S and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
- **Recurse**: Recursive sort sequences S_1 and S_2.
- **Conquer**: Merge the sorted sequences S_1 and S_2 into a unique sorted sequence S.

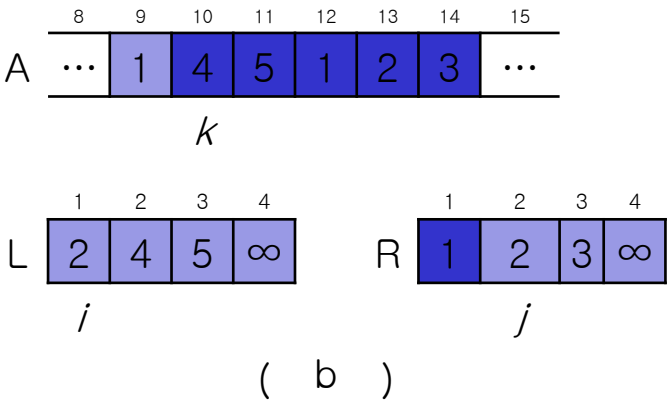
Merge(A,p,q,r)

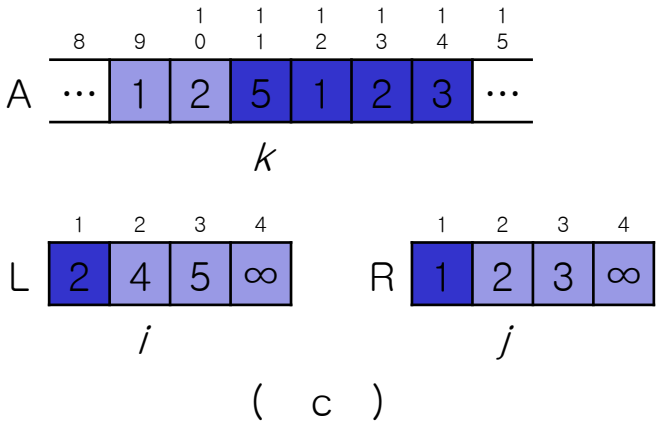
1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 create arrays $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$
4 for $i \leftarrow 1$ to n_1
5 do $L[i] \leftarrow A[p+i-1]$
6 for $j \leftarrow 1$ to n_2
7 do $R[j] \leftarrow A[q+j]$
8 $L[n_1+1] \leftarrow \infty$
9 $R[n_2+1] \leftarrow \infty$

Merge(A,p,q,r)

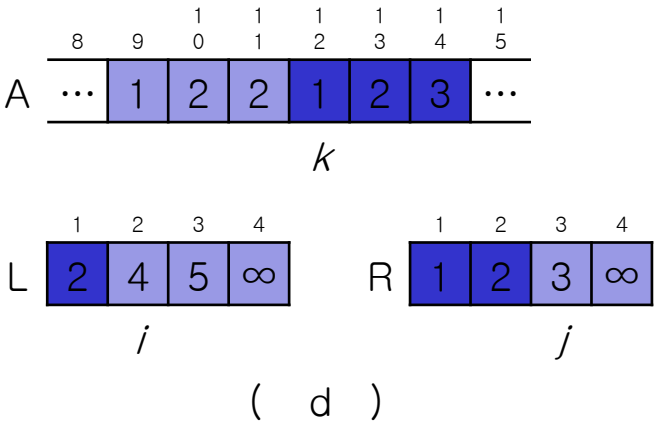
```
10 i ← 1
11 j ← 1
12 for k ← p to r
13         do if L[i] <= R[j]
14                   then A[k] ← L[i]
15                        i ← i + 1
16                   else A[k] ← R[j]
17                        j ← j + 1
```


8 9 10 11 12 13 14 15
A ⋯ 2 4 5 1 2 3 ⋯
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j
(a)

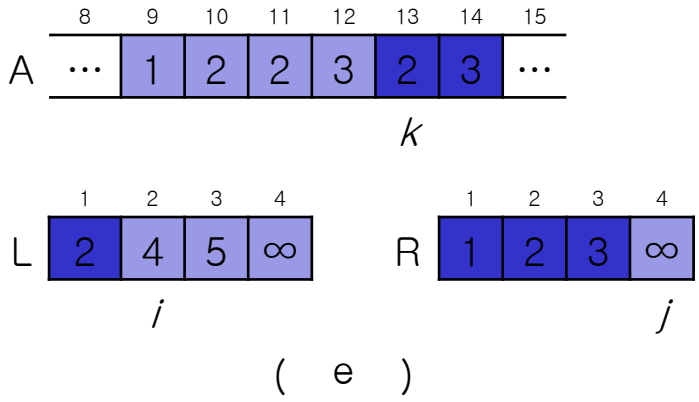

8 9 10 11 12 13 14 15
A ⋯ 1 4 5 1 2 3 ⋯
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j
(b)


8 9 10 11 12 13 14 15
A ··· 1 2 5 1 2 3 ···
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

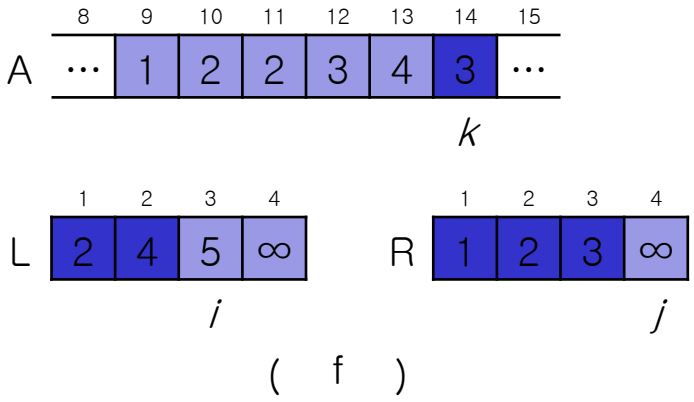
(c)


8 9 10 11 12 13 14 15
A ··· 1 2 2 1 2 3 ···
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

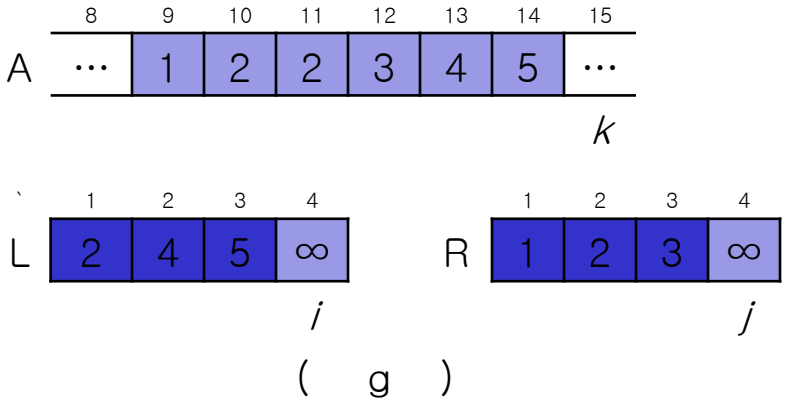
(d)


8 9 10 11 12 13 14 15
A ⋯ 1 2 2 3 2 3 ⋯
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(e)


8 9 10 11 12 13 14 15
A ⋯ 1 2 2 3 4 3 ⋯
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(f)



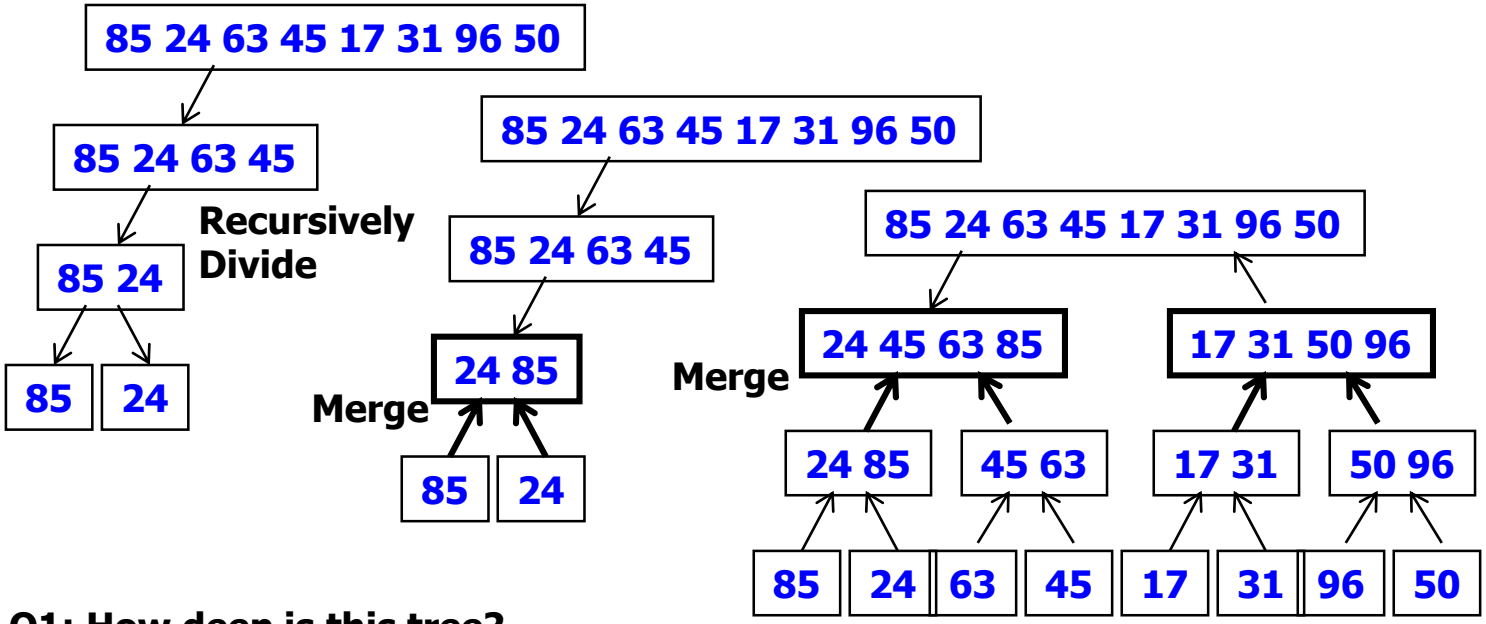
(g)

Merge Sort

```
MergeSort(A,p,r)
1       if p < r
2       then q = floor((p+r)/2)
3            MergeSort(A,p,q)
4            MergeSort(A,q+1,r)
5            Merge(A,p,q,r)
```

- Merge() is the procedure to merge two sorted lists.

Merge Sort Tree



Q1: How deep is this tree?

Q2: How much memory is needed for merge sort?

Merge Sort Analysis

Recurrence equation :

$$T(1) = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$T(n) = n\log n + n = O(n\log n)$$

Merge Sort

- Merging two half arrays S_1, S_2 into a full array S requires three pointers, one for S_1, another for S_2, and the other for S.
- The formal analysis result coincides with the intuitive count of the big-Oh, namely, the area taken by the merge sort tree.
- The amount of memory needed for merge sort
  - An extra array

Merge Sort

- Merging
  - Merge two sorted lists to a single sorted list.

    $(initList_{\ell}, \ldots, initList_{m})$ $(initList_{m+1}, \ldots, initList_{n})$

    $\Rightarrow (mergeList_{\ell}, \ldots, mergeList_{n})$

  - Analysis

    $O(n-\ell+1) \Rightarrow O(n)$

  - Example

    (3, 4, 8, 9) (5, 7, 10, 11)

    $\Rightarrow$ (3, 4, 5, 7, 8, 9, 10, 11)

  - Stable sorting

Merge

```
template <class T>
void Merge(T *initList, T *mergedList, const int l, const int m, const int n)
{ // initList [l :m] and initList[m+1:n] are sorted lists. They are merged to obtain
  // the sorted list mergedList [l :n]
     for (int i1 = l, iResult = l, i2 = m+1; // i1, i2, and iResult are list positions
             i1 <= m && i2 <= n; // neither input list if exhausted
             iResult++)
          if (initList[i1] <= initList[i2])
          {
               mergedList[iResult] = initList[i1];
               i1++;
           }
          else
          {
               mergedList[iResult] = initList[i2];
               i2++;
           }
      // copy remaining records, if any, of first list
     copy(initList+i1, initList+m+1, mergedList+iResult);
     // copy remaining records, if any, of second list
     copy(initList+i2, initList+n+1, mergedList+iResult);
}
```

Program 7.7: Merging two sorted lists

Iterative Merge Sort

- We assume that the input is n sorted lists. But each list is of length 1.
- These lists are merged by pairs to obtain n/2 lists.

```
template <class T>
void MergePass(T* initList, T* resultList, const int n, const int s)
{ // Adjacent pairs of sublists of size s are merged from
  // initList to resultList. n is the number of records in initList.

    for (int i=1; // i is first position in first of the sublists being merged
            i <= n-2*s+1; // enough elements for two sublists of length s?
            i += 2*s)
        Merge(initList, resultList, i, i + s - 1, i + 2 * s - 1);

    // merge remaining list of size < 2*s
    if( (i+s-1) < n) Merge(initList, resultList, i, i + s - 1, n);
    else copy(initList + i, initList + n + 1, resultList + i);
}
```

Program 7.8: Merge pass

Iterative Merge Sort

```
template <class T>
void MergeSort(T *a, const int n)
{ // Sort a[1:n] into nondecreasing order.

    T* tempList = new T[n+1];
    // l is the length of the sublist currently being merged
    for (int  l=1; l<n; l *= 2)
    {
        MergePass(a, tempList, n, l);
        l *= 2;
        MergePass(tempList, a, n, l); // interchange role of a and tempList
    }
    delete [] tempList;
}
```

Program 7.9: Merge Sort

Iterative Merge Sort example

- Example 7.5
  - input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

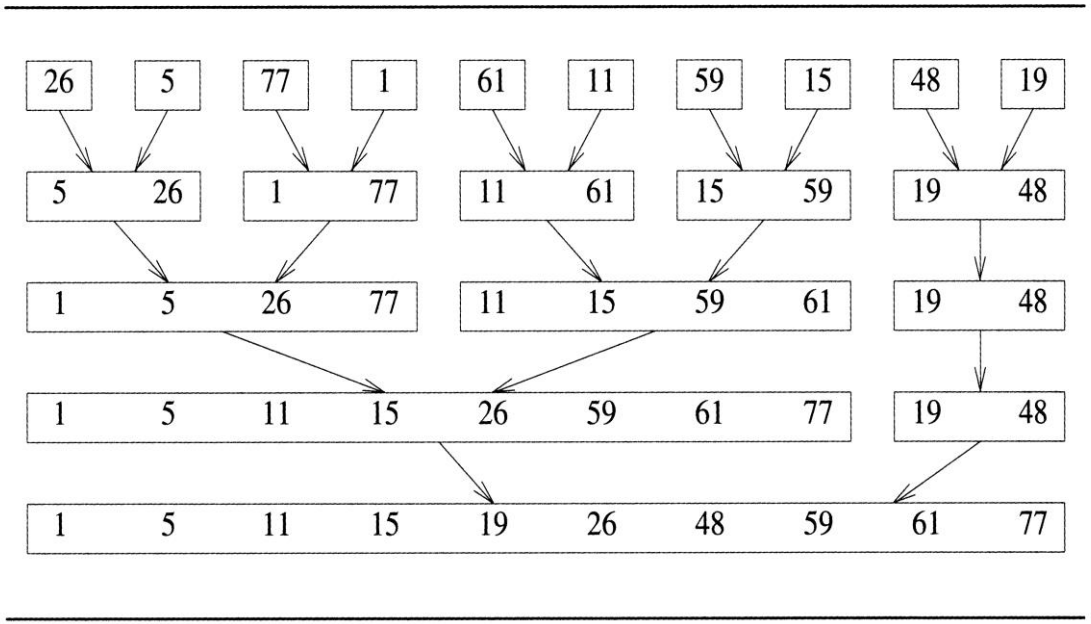


Figure 7.4: Merge tree

Iterative Merge Sort analysis

- Analysis :
  - merge-pass : O(n)
  - number of merge passes : $O(\log_2 n)$

    $\Rightarrow O(n\log_2 n)$
- Stable sorting