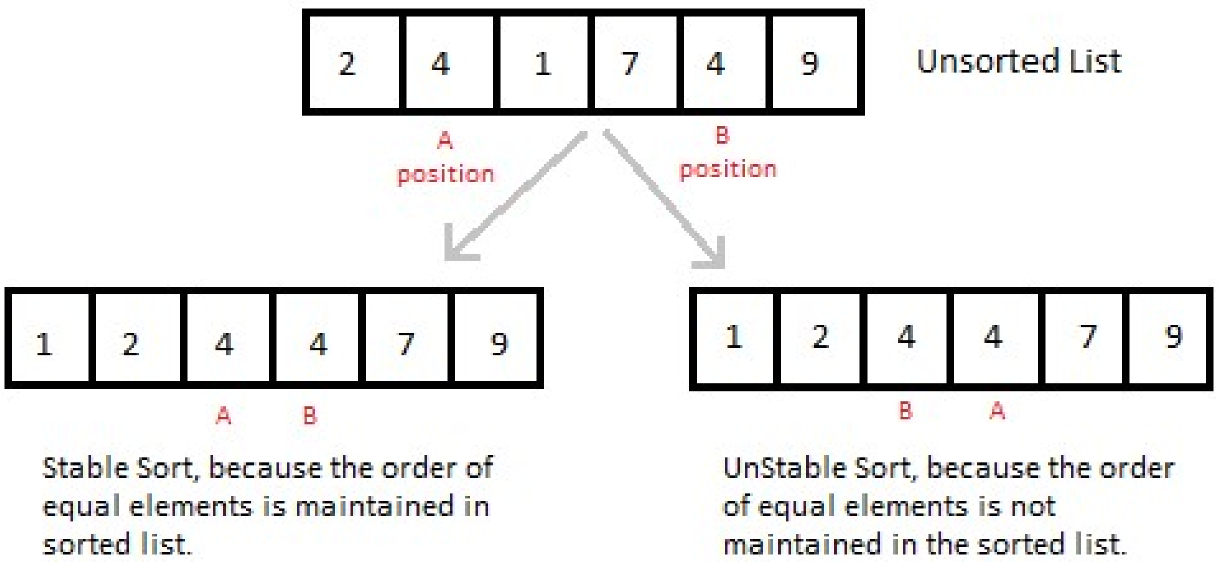
Sorting

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Sorting

- Rearrange n elements in a certain order.
  - 7, 3, 6, 2, 1 ➔ 1, 2, 3, 6, 7
- A sorting algorithm is *stable* if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.



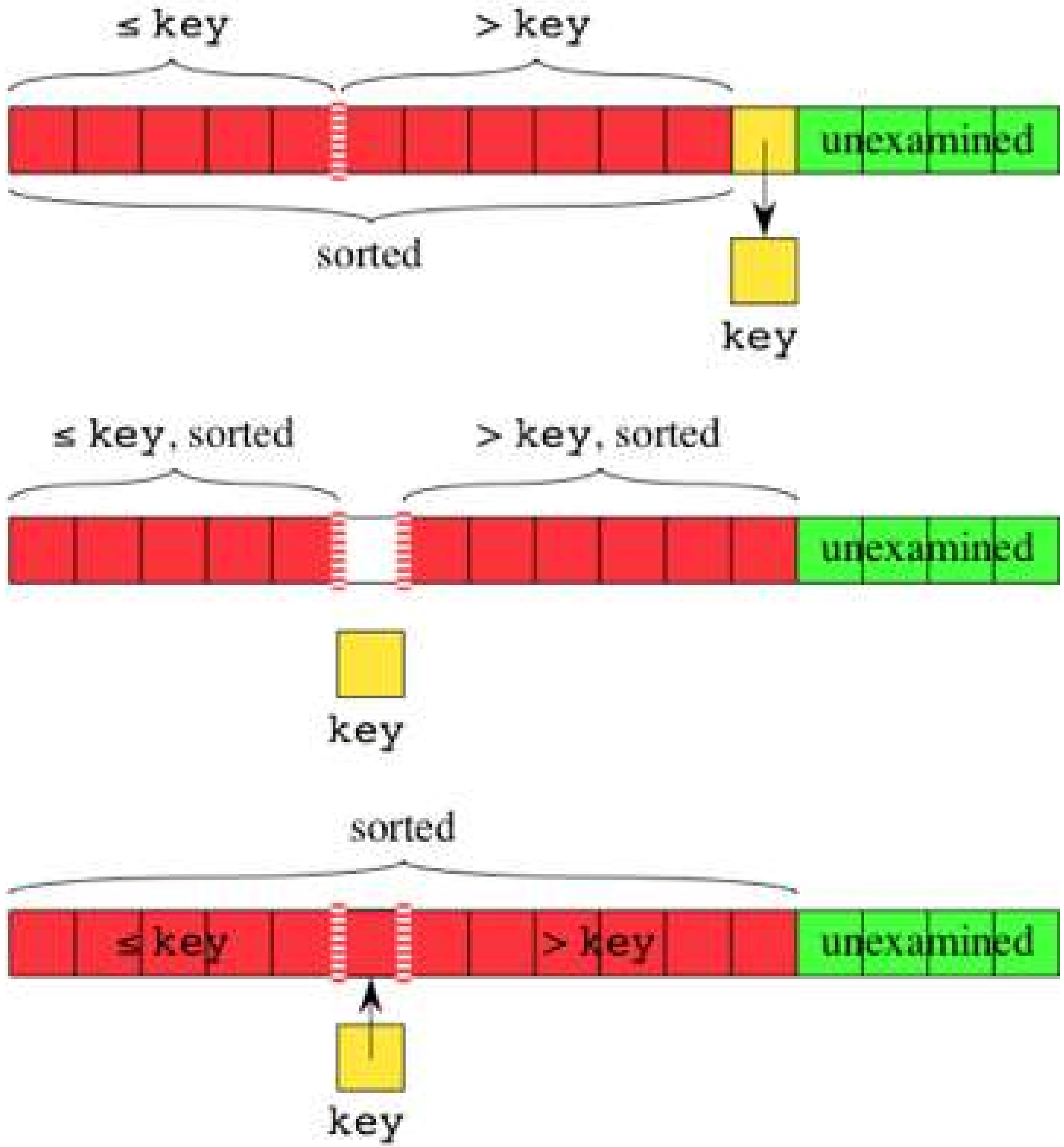
Sorting (cont.)

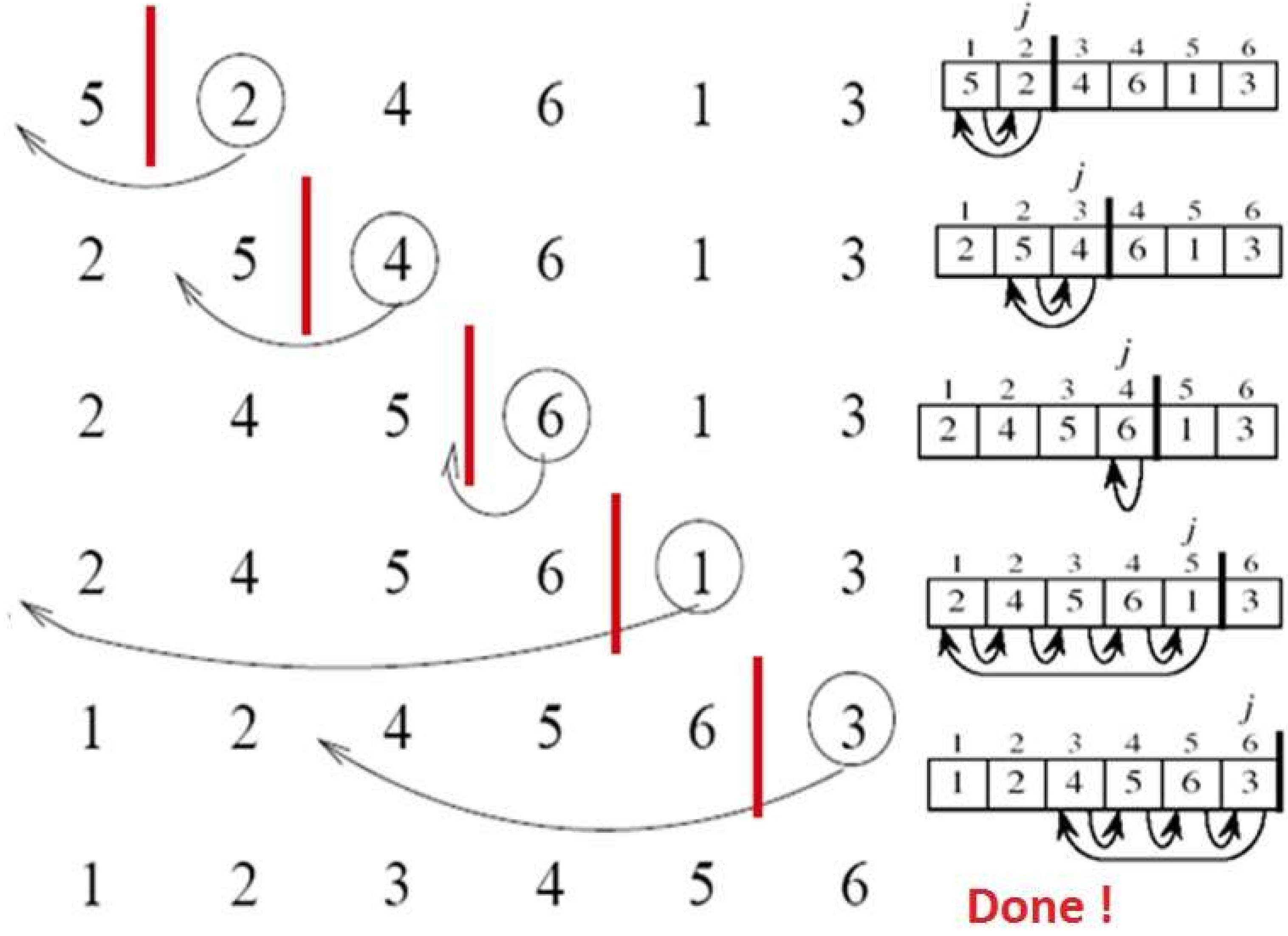
- A sorting algorithm is said to be *in-place* if
  - it updates the input sequence only through replacement or swapping of elements
  - it does not use auxiliary data structures but may require a small though non-constant extra space, usually O(log n), for its operation

Insertion Sort

- Insert a new record into a sorted sequence of i records in such a way that the resulting sequence of size i + 1 is also ordered
- Stable and in-place

i	[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


≤ key
> key
unexamined
sorted
key
≤ key, sorted
> key, sorted
unexamined
key
sorted
≤ key
> key
unexamined
key



j
Done !

```cpp
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);
  }
}

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.
// The use of a[0] enables us to simplify the while loop,
// avoiding a test for end of list (i.e., i < 1)
  a[0] = e;
  while (e < a[i])
  {
    a[i+1] = a[i];
    i--;
  }
  a[i+1] = e;
}
```

Time complexity: $O(n^2)$

Space complexity: $O(1)$

Quick Sort

- When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort
- On average, the algorithm takes $O(n \log n)$ comparisons to sort n items.
- In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare.
- In-place but not stable

Quick Sort (cont.)

- When n <= 1, the list is sorted.
- When n > 1, select a pivot element from out of the n elements.
- Partition the n elements into 3 segments left, middle and right.
  - All elements in the left segment are <= pivot.
  - The middle segment contains only the pivot element.
  - All elements in the right segment are >= pivot.
- Sort left and right segments recursively.
- Answer is sorted left segment, followed by middle segment followed by sorted right segment.

Example

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot.

equal values can go either way

<= 6 6 <=

2	5	4	1	3	6	7	9	10	11	8

Sort left and right segments recursively.

Choice of Pivot

- Pivot is leftmost element in list that is to be sorted.
  - When sorting a[6:20], use a[6] as the pivot.
  - Text implementation does this.

- Randomly select one of the elements to be sorted as the pivot.
  - When sorting a[6:20], generate a random number r in the range [6, 20]. Use a[r] as the pivot.

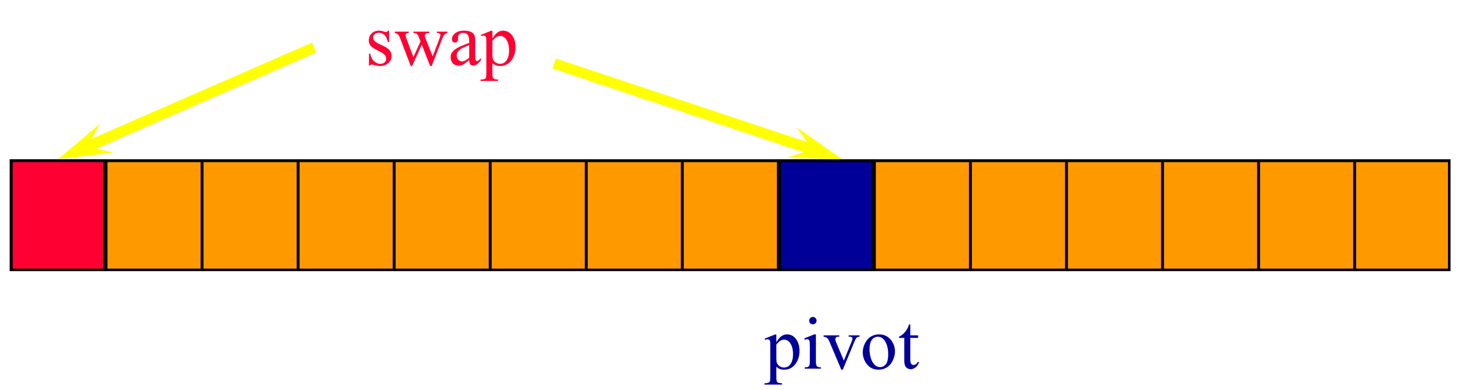
Choice of Pivot (cont.)

- Median-of-Three rule. From the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.
  - When sorting a[6:20], examine a[6], a[13] ((6+20)/2), and a[20]. Select the element with median (i.e., middle) key.
  - If a[6].key = 30, a[13].key = 2, and a[20].key = 10, a[20] becomes the pivot.
  - If a[6].key = 3, a[13].key = 2, and a[20].key = 10, a[6] becomes the pivot.

Choice of Pivot (cont.)

- If a[6].key = 30, a[13].key = 25, and a[20].key = 10, a[13] becomes the pivot.

- When the pivot is picked at random or when the median-of-three rule is used, we can use the quick sort code of the text provided we first swap the leftmost element and the chosen pivot.



Partitioning into Three Segments

- Sort a = [6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3].
- Leftmost element (6) is the pivot.
- When another array b is available:
  - Scan a from left to right (omit the pivot in this scan), placing elements <= pivot at the left end of b and the remaining elements at the right end of b.
  - The pivot is placed at the remaining position of the b.

Partitioning Example Using Additional Array

a

6	2	8	5	11	10	4	1	9	7	3

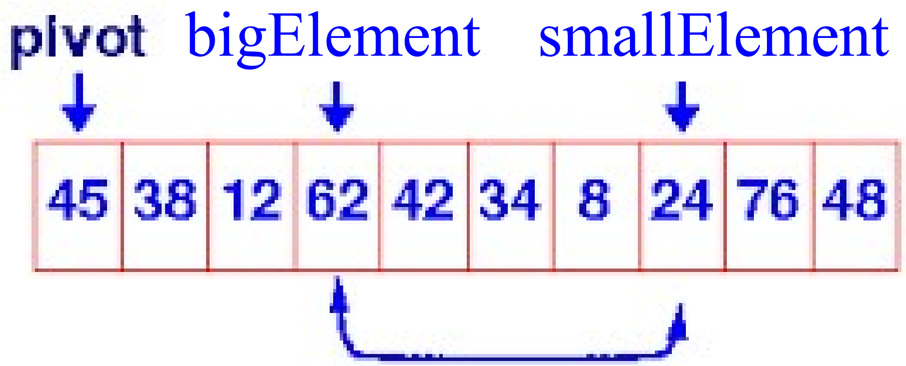
b

2	5	4	1	3	6	7	9	10	11	8

Sort left and right segments recursively.

In-Place Partitioning

- Find leftmost element (bigElement) > pivot.
- Find rightmost element (smallElement) < pivot.
- Swap bigElement and smallElement.
- Repeat.



In-Place Partitioning Example

a	6	2	8	5	11	10	4	1	9	7	3

a	6	2	3	5	11	10	4	1	9	7	8

a	6	2	3	5	1	10	4	11	9	7	8

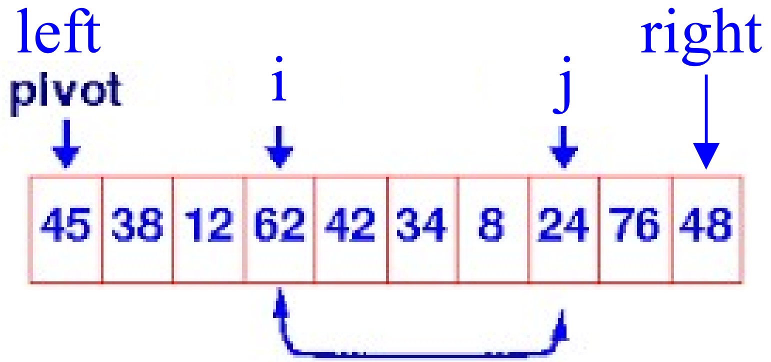
a	6	2	3	5	1	4	10	11	9	7	8

bigElement is not to left of smallElement, terminate process. Swap pivot and smallElement.

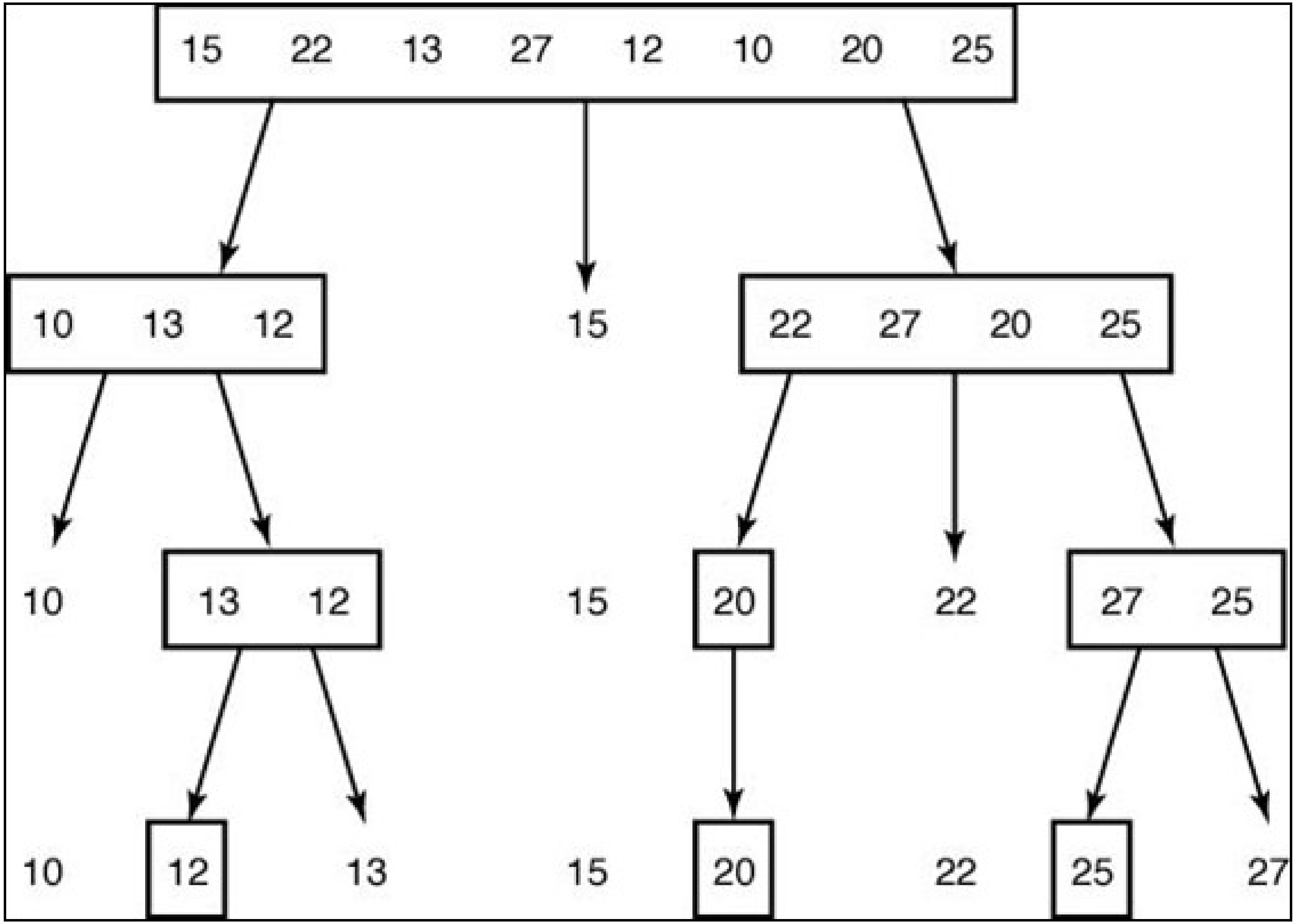
a	4	2	3	5	1	6	10	11	9	7	8

```
template <class T>
void QuickSort(T *a, const int left, const int right)
{
// Sort a[left:right] into nondecreasing order.
// a[left] is arbitrarily chosen as the pivot.
// Variables i and j are used to partition the subarray
// so that at any time a[m] <= pivot, m < i,
// and a[m] >= pivot, m > j.
// It is assumed that a[left] <= a[right + 1].
  if (left < right) {
    int i = left,
    j = right + 1,
    pivot = a[left];
    do {
      do i++; while (a[i] < pivot);
      do j--; while (a[j] > pivot);
      if (i < j) swap(a[i], a[j]);
    } while (i < j);
    swap(a[left], a[j]);

    QuickSort(a, left, j-1);
    QuickSort(a, j+1, right);
  }
}
```



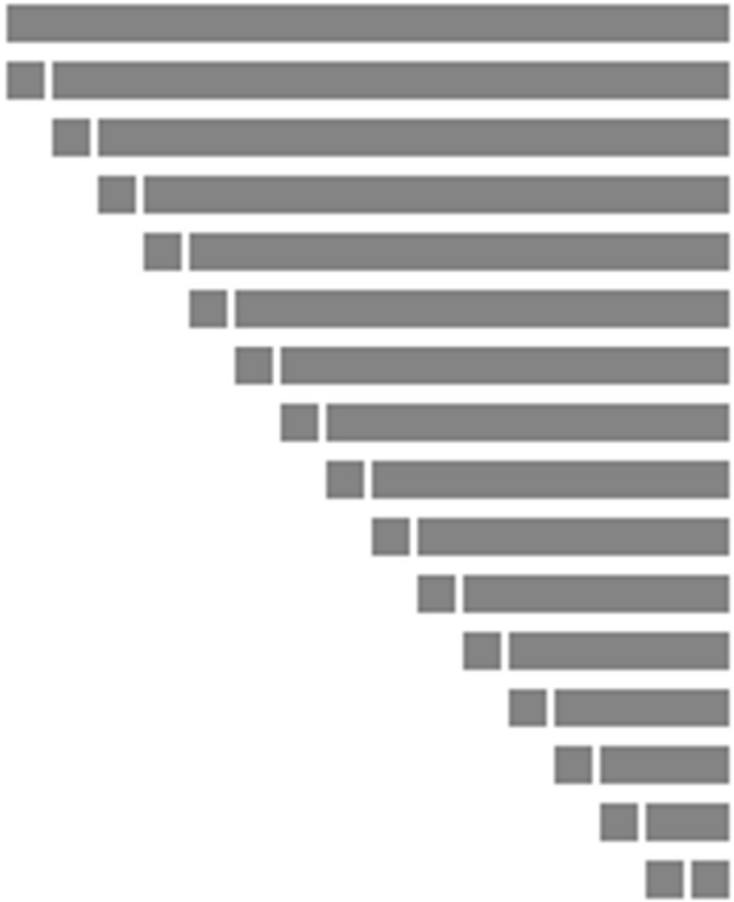
Quick Sort Example



Complexity

a) best case

b) average case



c) worst case

Time Complexity

- $O(n)$ time to partition an array of n elements.
- Let $T(n)$ be the time needed to sort n elements.
- $T(0) = T(1) = b$, where b is a constant.
- When $n > 1$,

  $T(n) = c*n + T(|left|) + T(|right|)$,

  where c is a constant.

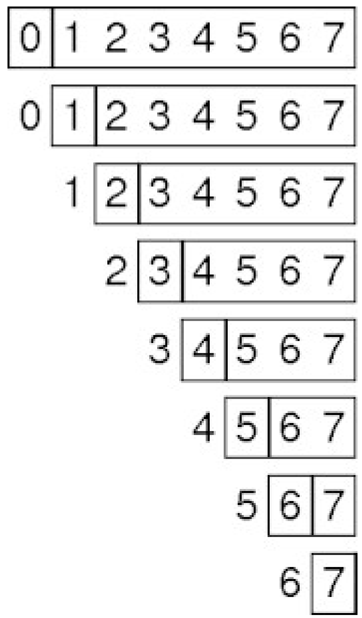
  – Hereafter, we will assume $c = 1$ for simplicity

Worst Case

- This happens when the pivot is always the smallest element.
  - |left segment| = 0
  - |middle segment| = 1
  - |right segment| = n - 1
- For the worst-case time,

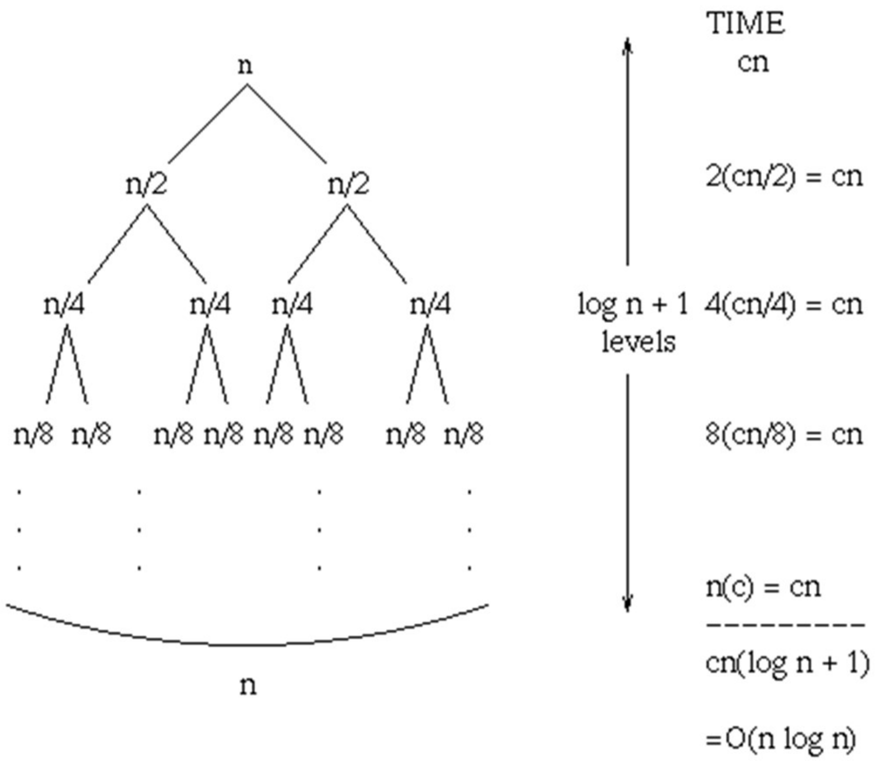
  $T(n) = n + T(n-1),\ n > 1$
- Use repeated substitution to get

  $T(n) = O(n^2)$



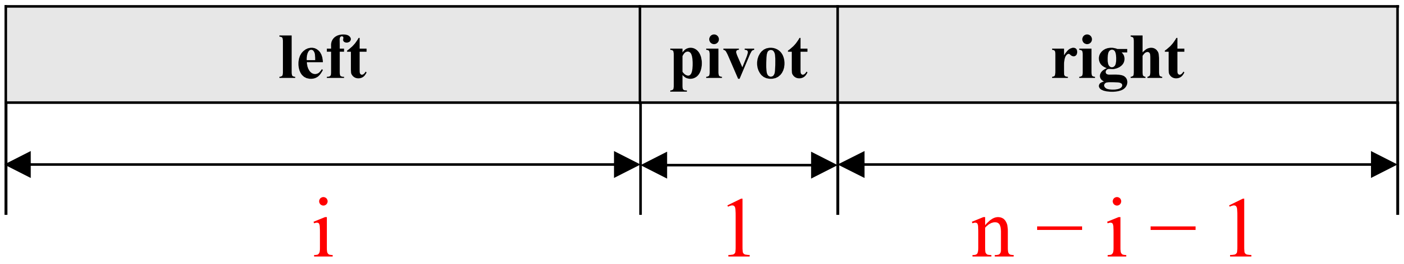
Best Case

- The best case arises when |left| and |right| are equal (or differ by 1) following each partitioning.
- $T(n) = n + 2\,T(n/2),\ n > 1$
- So the best-case complexity is $O(n \log_2 n)$.



Average Case

- Average complexity is also $O(n \log_2 n)$.
- When the input is a random permutation, the resulting segments of the partition have sizes i and $n - i - 1$, and i is uniform random from 0 to $n - 1$.



$$T(n) = n + T(i) + T(n - i - 1), n > 1$$

Average Case (cont.)

- The average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

  $T_{avg}(n) = n + \frac{1}{n}\sum_{i=0}^{n-1} (T_{avg}(i) + Tavg(n - i - 1))$

- Since i may take on any of 0, …, n – 1

  $T_{avg}(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} T_{avg}(i)$

- Solving the recurrence gives

  $T_{avg}(n) = 2n \ln n \approx 1.39n \log_2 n$

Space Complexity

- Extra space for the recursion stack
  – Each recursive call will create a stack frame on the call stack, which takes up space
- Worst case: O(n) space
  – Optimized version: O(log n) space
- Best and average case: O(log n) space

In Practice

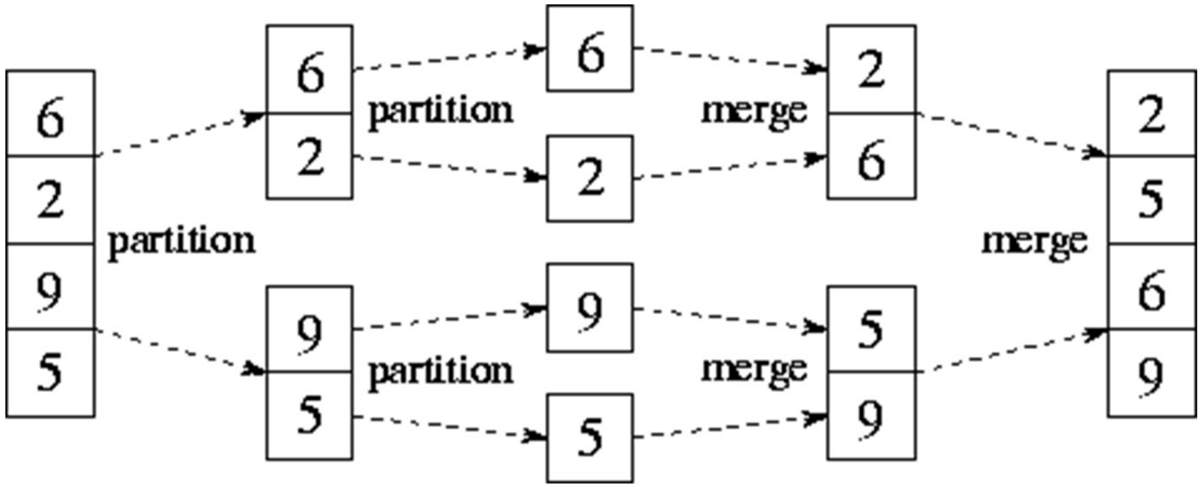
- To improve performance, stop recursion when segment size is <= 15 (say) and sort these small segments using insertion sort.

C++ STL sort Function

- Quick sort.
  - Switch to heap sort when the recursion goes too deep
  - Switch to insertion sort when segment size becomes small.

Merge Sort

- Recursively splits the unsorted list into sublists until sublist size is 1, then merges those sublists to produce a sorted list



Merge Sort (cont.)

- Partition the $n > 1$ elements into two smaller instances.
- Each of the two smaller instances is sorted recursively.
- The sorted smaller instances are combined using a process called merge.
- Complexity is $O(n \log n)$.
- Usually implemented non-recursively.
- Stable but not in-place

Merge Two Sorted Lists

- A = (2, 5, 6)
  B = (1, 3, 8, 9, 10)
  C = ()
- Compare smallest elements of A and B and merge smaller into C.
- A = (2, 5, 6)
  B = (3, 8, 9, 10)
  C = (1)

Merge Two Sorted Lists

- A = (5, 6)
  B = (3, 8, 9, 10)
  C = (1, 2)
- A = (5, 6)
  B = (8, 9, 10)
  C = (1, 2, 3)
- A = (6)
  B = (8, 9, 10)
  C = (1, 2, 3, 5)

Merge Two Sorted Lists

- A = ()

  B = (8, 9, 10)

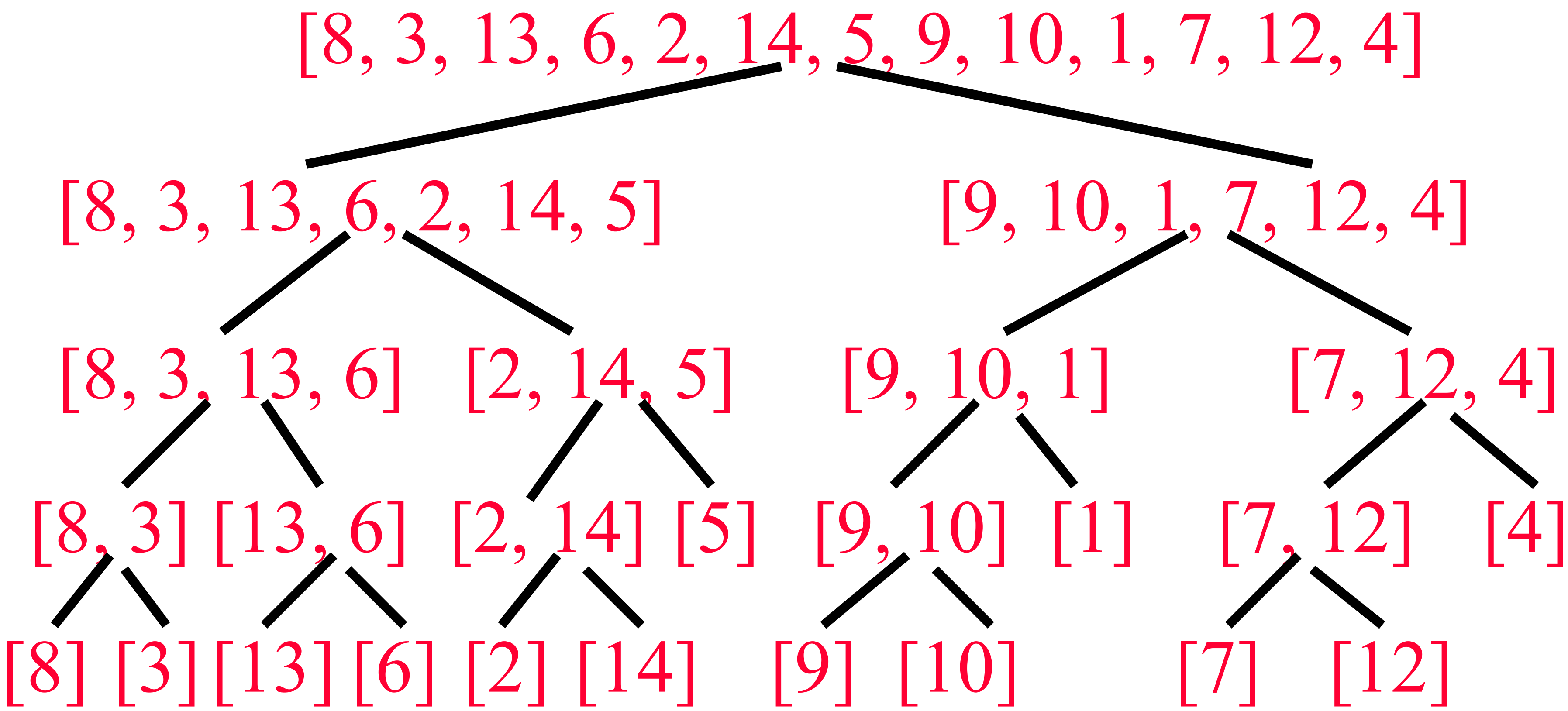
  C = (1, 2, 3, 5, 6)
- When one of A and B becomes empty, append the other list to C.
- O(1) time needed to move an element into C.
- Total time is O(n + m), where n and m are, respectively, the number of elements initially in A and B.

Recursive Merge Sort

procedure $mergesort(L = a_1, a_2, \ldots, a_n)$
if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2, \ldots, a_m$
    $L_2 := a_{m+1}, a_{m+2}, \ldots, a_n$
    $L := merge(mergesort(L_1), mergesort(L_2))$
{L is now sorted into elements in increasing order}

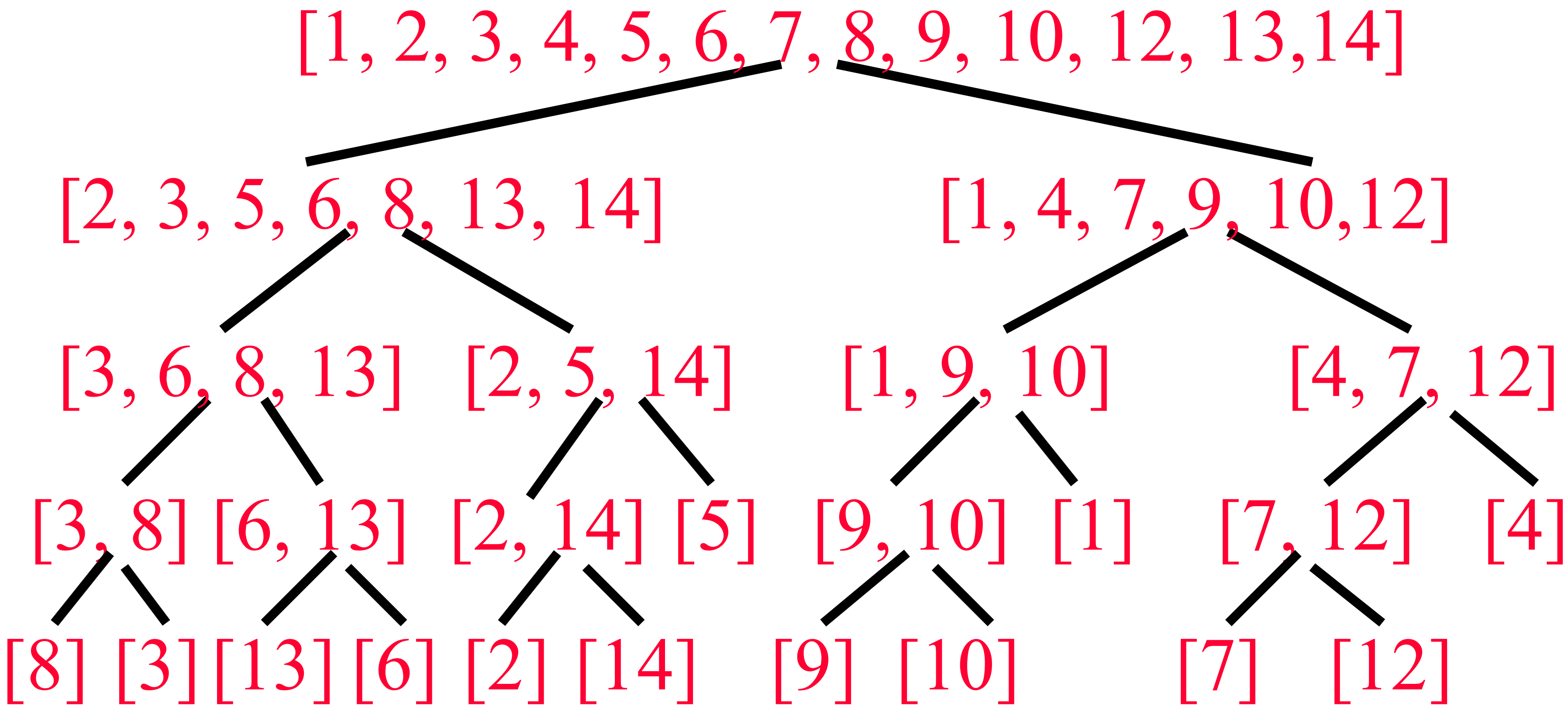
Downward Pass

- Downward pass over the recursion tree.
  - Divide large instances into small ones.



Upward Pass

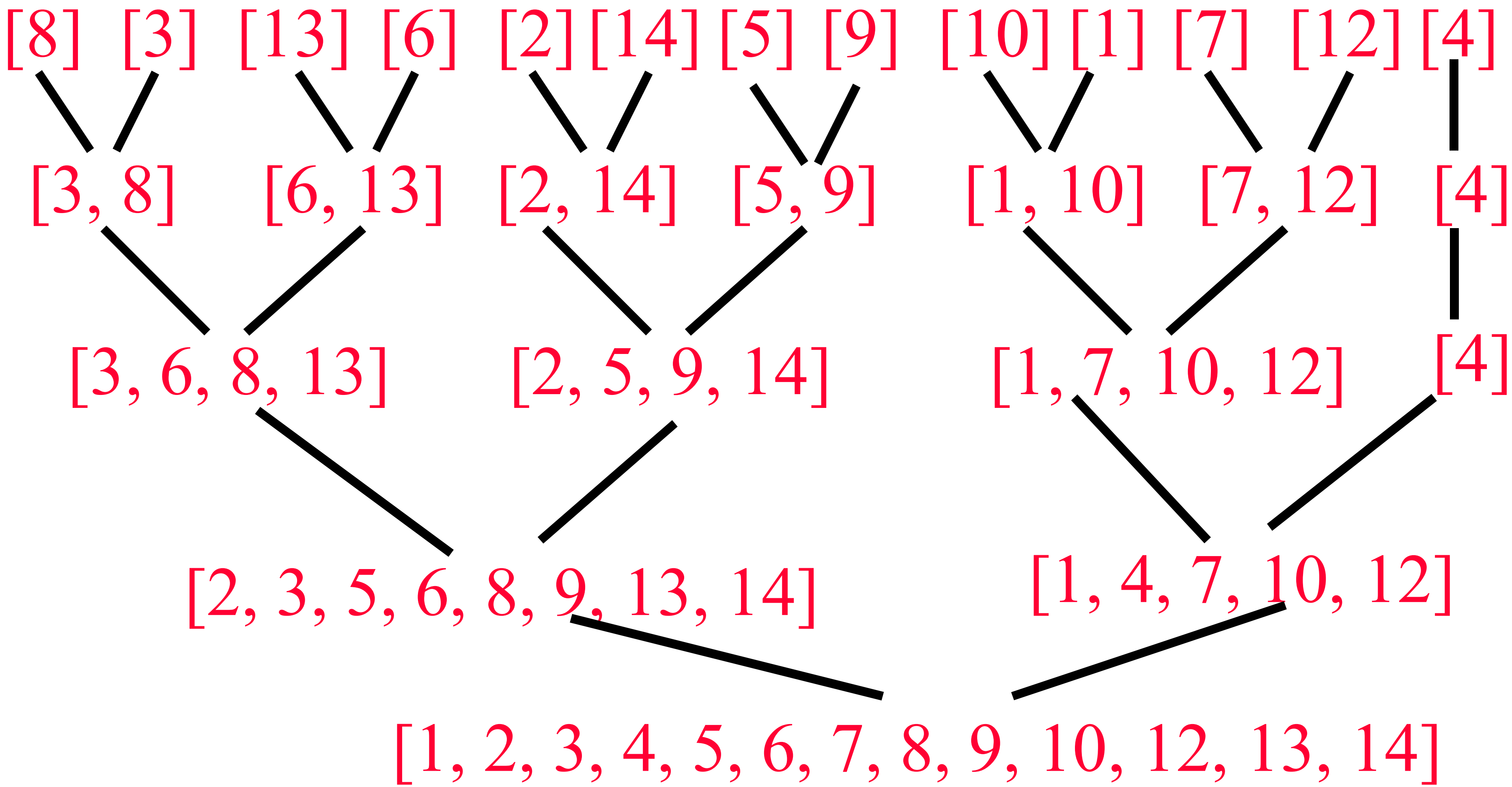
- Upward pass over the recursion tree.
  - Merge pairs of sorted lists.



Nonrecursive Version

- Eliminate downward pass.
- Start with sorted lists of size 1 and do pairwise merging of these sorted lists as in the upward pass.

Nonrecursive Merge Sort



Time Complexity

- Let $T(n)$ be the time required to sort n elements.
- $T(0) = T(1) = b$, where b is a constant.
- When $n > 1$,

  $T(n) = 2*T(n/2) + c*n$,

  where c is a constant.
- To solve the recurrence, assume n is a power of 2 and use repeated substitution.
- $T(n) = O(n \log_2 n)$.

C++ STL stable_sort Function

- Merge sort is stable (relative order of elements with equal keys is not changed).
- Quick sort is not stable.
- STL's stable_sort is a merge sort that switches to insertion sort when segment size is small.

```cpp
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].
// i1, i2, and iResult are list positions.
  for (int i1 = l, iResult = l, i2 = m+1;
        i1 <= m && i2 <= n; // neither input list is exhausted.
        iResult++)
    if (initList[i1] <= initList[i2])
    {
      mergedList[iResult] = initList[i1];
      i1++;
    }
    else
    {
      mergedList[iResult] = initList[i2];
      i2++;
    }
  // copy remaining records, if any, of the first list.
  copy(initList+i1, initList+m+1, mergedList+iResult);

  // copy remaining records, if any, of the second list.
  copy(initList+i2, initList+n+1, mergedList+iResult);
}
```

Program 7.8:Merge pass

```
=======================================
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);
}
=======================================
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;
}
```

```
template <class T>
int rMergeSort(T* a, int* link, const int left, const int right)
{
// a[left:right] is to be sorted. link[i] is initially 0 for all i.
// rMergeSort returns the index of the first element in the sorted chain.
   if (left >= right) return left;
   int mid = (left + right) / 2;
   return ListMerge(a, link,
                    rMergeSort(a, link, left, mid), // sort left half
                    rMergeSort(a, link, mid + 1, right)); // sort right half
}
template <class T>
int ListMerge(T* a, int* link, const int start1, const int start2)
{
// The sorted chains beginning at start1 and start2, respectively, are merged.
// link[0] is used as a temporary header.  Return start of merged chain.
  int iResult = 0; // last record of result chain
  for (int i1 = start1, i2 = start2; i1 && i2; )
    if (a[i1] <= a[i2]) {
      link[iResult] = i1;
      iResult = i1; i1 = link[i1];
    }
    else {
      link[iResult] = i2;
      iResult = i2; i2 = link[i2];
    }

  // attach remaining records to result chain
  if (i1 == 0) link[iResult] = i2;
  else link[iResult] = i1;
  return link[0];
}
```