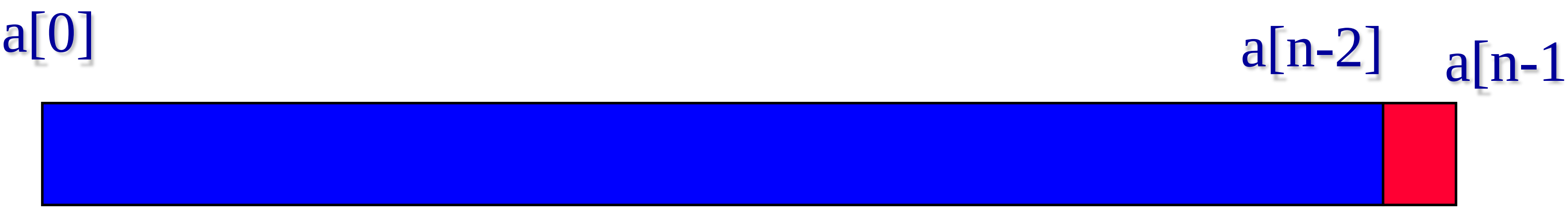
Sorting

- Rearrange n elements into ascending order.
- 7, 3, 6, 2, 1 ➔ 1, 2, 3, 6, 7

Insertion Sort



- n <= 1 ➔ already sorted. So, assume n > 1.
- a[0:n-2] is sorted recursively.
- a[n-1] is inserted into the sorted a[0:n-2].
- Complexity is $O(n^2)$.
- Usually implemented nonrecursively (see text).

Quick Sort

- 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.
- The middle segment contains only the pivot element.
- All elements in the left segment are <= pivot.
- 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.

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.

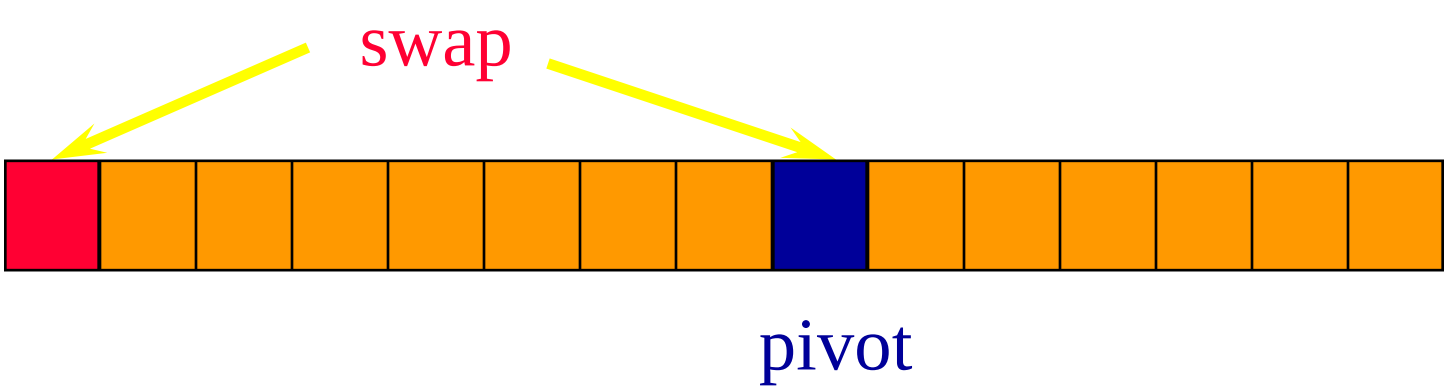
Choice Of Pivot

- 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

- 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 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 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

Complexity

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

  $t(n) = t(|left|) + t(|right|) + dn$,

  where d is a constant.
- $t(n)$ is maximum when either $|left| = 0$ or $|right| = 0$ following each partitioning.

Complexity

- This happens, for example, when the pivot is always the smallest element.
- For the worst-case time,

  $t(n) = t(n-1) + dn$, $n > 1$
- Use repeated substitution to get $t(n) = O(n^2)$.
- The best case arises when $|left|$ and $|right|$ are equal (or differ by 1) following each partitioning.

Complexity Of Quick Sort

- So the best-case complexity is O(n log n).
- Average complexity is also O(n log n).
- To help get partitions with almost equal size, change in-place swap rule to:
  - Find leftmost element (bigElement) >= pivot.
  - Find rightmost element (smallElement) <= pivot.
  - Swap bigElement and smallElement provided bigElement is to the left of smallElement.
- O(n) space is needed for the recursion stack. May be reduced to O(log n).

Complexity Of Quick Sort

- 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 number of subdivisions exceeds some constant times $\log_2 n$.
  - Switch to insertion sort when segment size becomes small.

Merge Sort

- Partition the $n > 1$ elements into two smaller instances.
- First ceil(n/2) elements define one of the smaller instances; remaining floor(n/2) elements define the second smaller instance.
- 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 nonrecursively.

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)

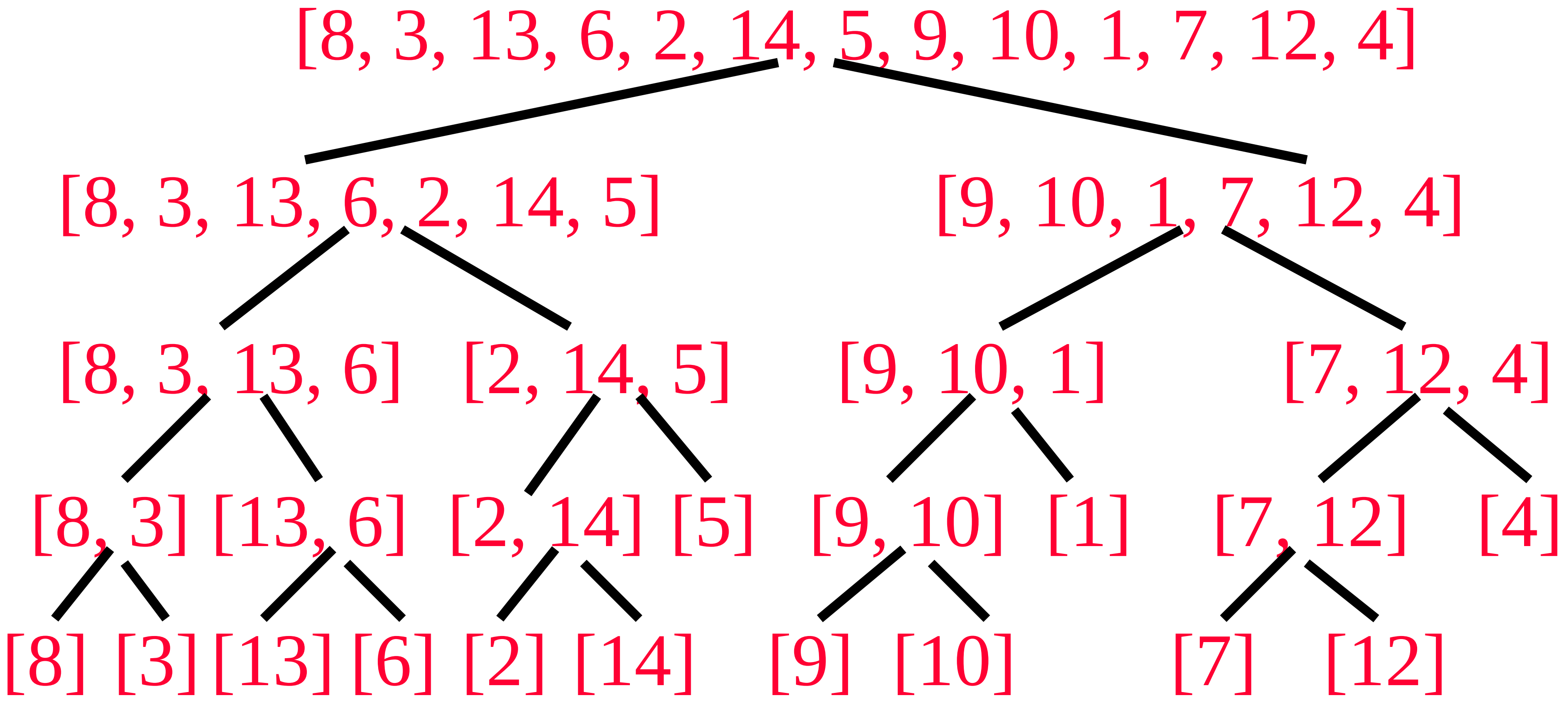
Merge Two Sorted Lists

- 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)

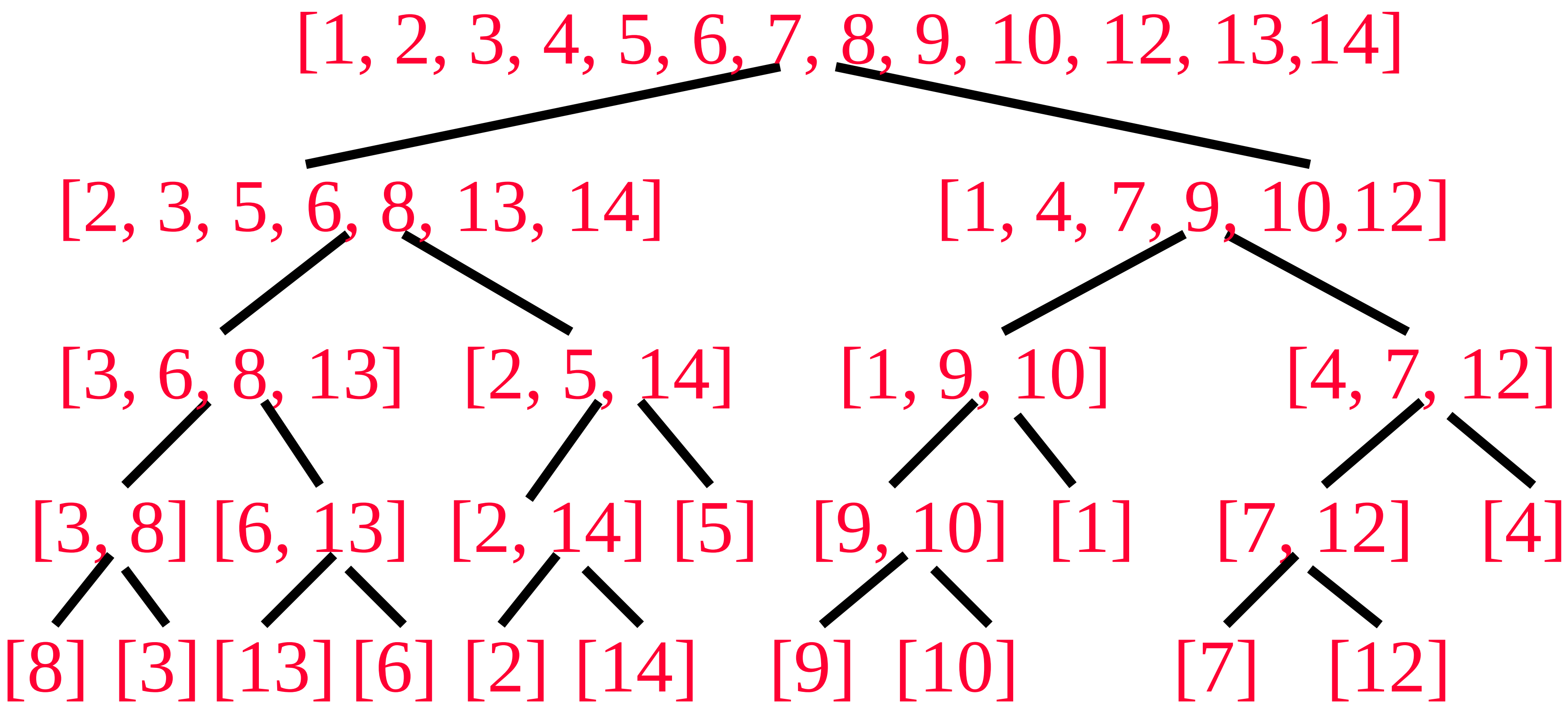
Merge Two Sorted Lists

- 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.

Merge Sort


[8, 3, 13, 6, 2, 14, 5, 9, 10, 1, 7, 12, 4]
[8, 3, 13, 6, 2, 14, 5]
[9, 10, 1, 7, 12, 4]
[8, 3, 13, 6]
[2, 14, 5]
[9, 10, 1]
[7, 12, 4]
[8, 3]
[13, 6]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Merge Sort


[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13,14]
[2, 3, 5, 6, 8, 13, 14]
[1, 4, 7, 9, 10,12]
[3, 6, 8, 13]
[2, 5, 14]
[1, 9, 10]
[4, 7, 12]
[3, 8]
[6, 13]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Time Complexity

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

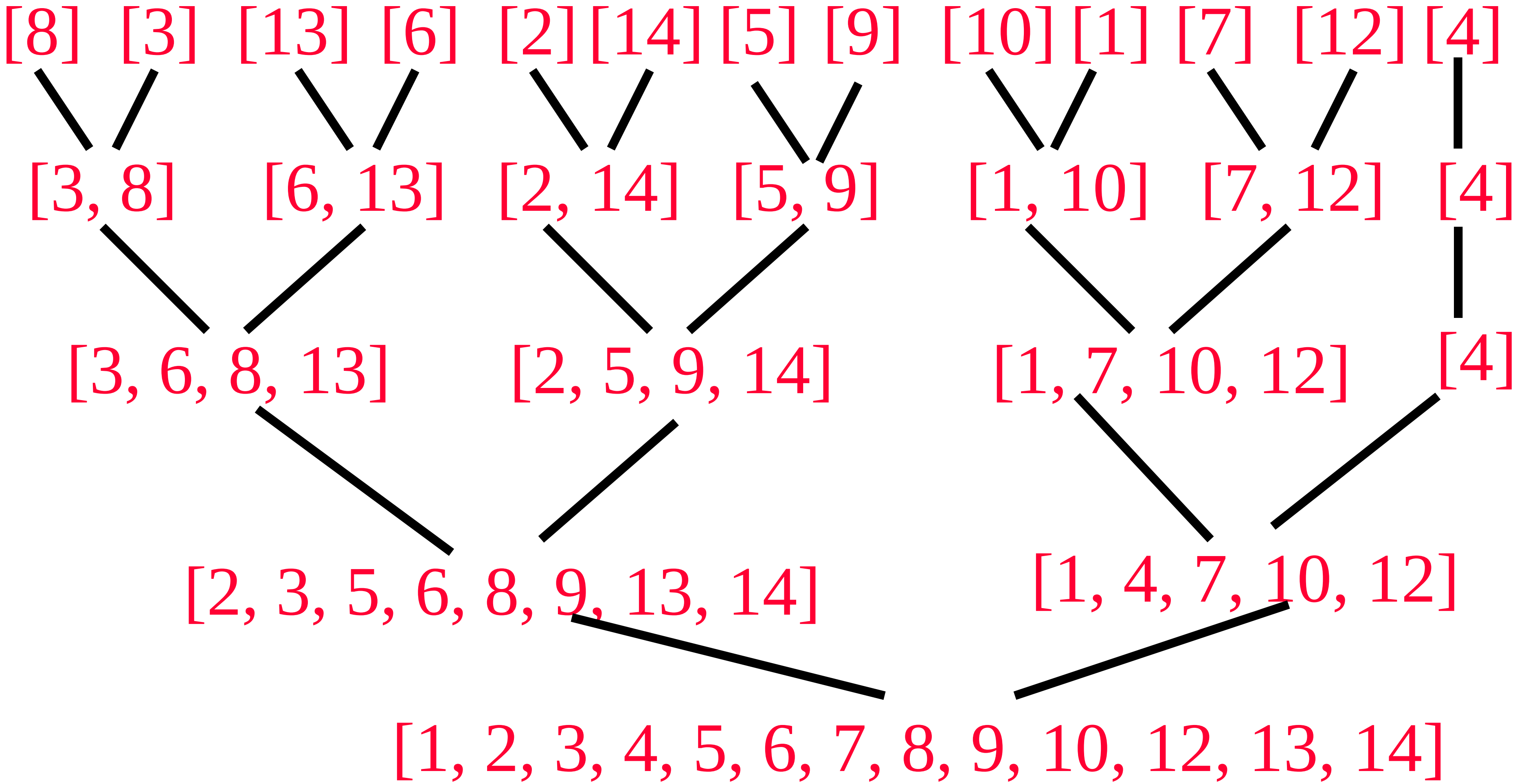
  $t(n) = t(\text{ceil}(n/2)) + t(\text{floor}(n/2)) + dn$,

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

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



Complexity

- Sorted segment size is 1, 2, 4, 8, …
- Number of merge passes is ceil($\log_2 n$).
- Each merge pass takes O(n) time.
- Total time is O(n log n).
- Need O(n) additional space for the merge.
- Merge sort is slower than insertion sort when n <= 15 (approximately). So define a small instance to be an instance with n <= 15.
- Sort small instances using insertion sort.
- Start with segment size = 15.

Natural Merge Sort

- Initial sorted segments are the naturally ocurring sorted segments in the input.
- Input = [8, 9, 10, 2, 5, 7, 9, 11, 13, 15, 6, 12, 14].
- Initial segments are:

  [8, 9, 10] [2, 5, 7, 9, 11, 13, 15] [6, 12, 14]
- 2 (instead of 4) merge passes suffice.
- Segment boundaries have a[i] > a[i+1].

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.