CSE221 Data Structures
Lecture 22: Sorting

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

December 1, 2021

Introduction

- Final exam is on Wednesday 15 December, 20:00–22:00.
- Assignment 4 will be posted tonight, due on Friday next week.
- Assignment 3 was graded by Seonghyeon Jue (`shjue@unist.ac.kr`). Grading script is now available.
- Today's lecture is on sorting algorithms.
- In Lecture 5, we already saw a sorting algorithm, insertion-sort, that runs in $O(n^2)$ time.
- Today, I will present to $O(n \log n)$-time sorting algorithms: Merge-sort and quicksort.
- Reference for this lecture: Textbook Chapter 11.1 and 11.2

Merge-Sort

85	24	63	45	17	31	96	50

85	24	63	45

17	31	96	50

24	45	63	85

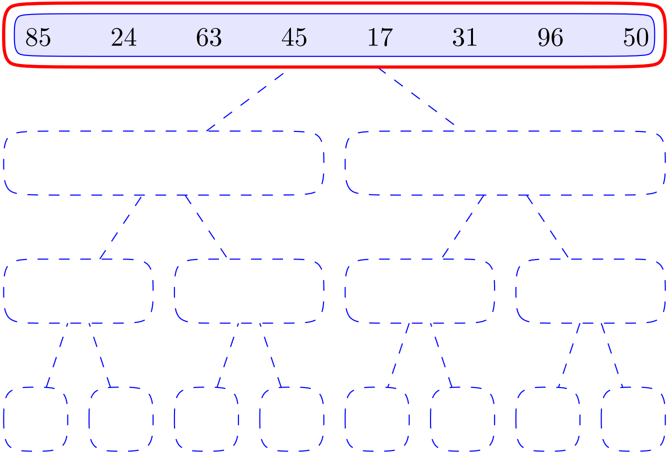
17	31	50	96

17	24	31	45	50	63	85	96

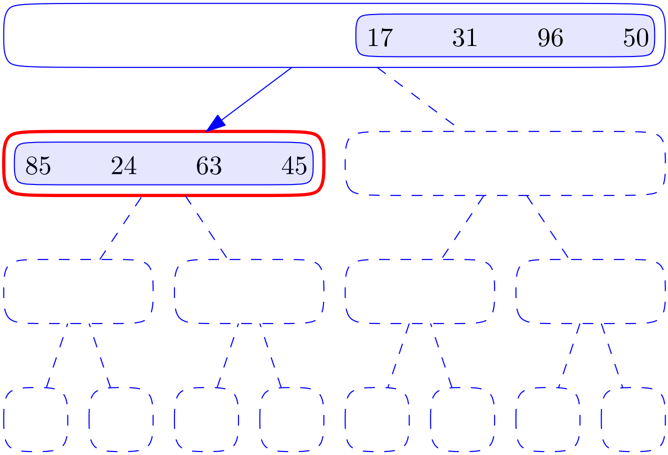
Sorting a sequence S by merge-sort

1. If S has zero or one element, return S. Otherwise, split S into two sequences S_1 and S_2 of size $n/2$.
2. Sort S_1 and S_2 recursively.
3. Merge S_1 with S_2.

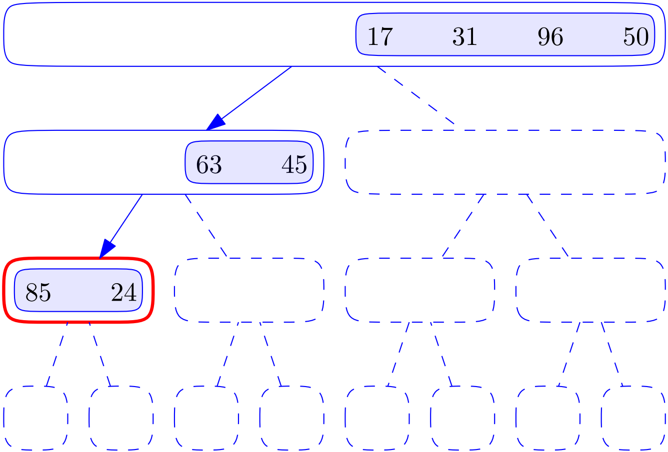
Merge-Sort



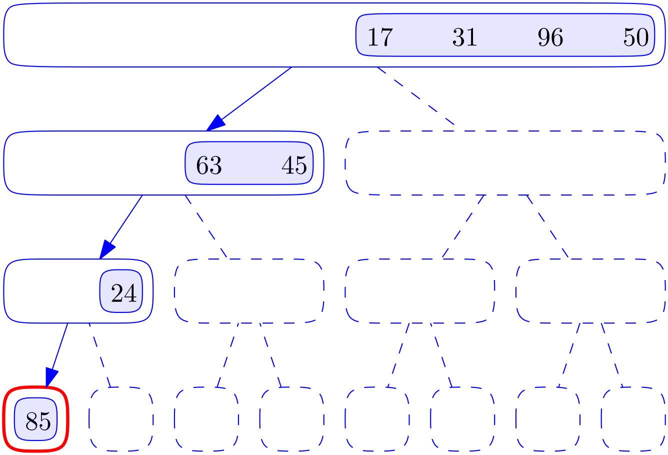
Merge-Sort



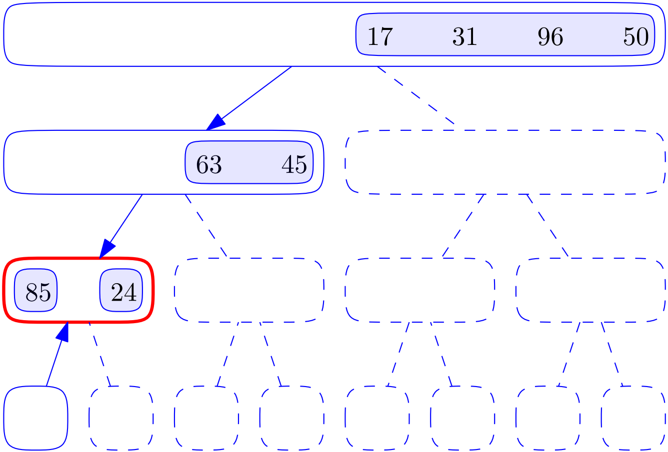
Merge-Sort



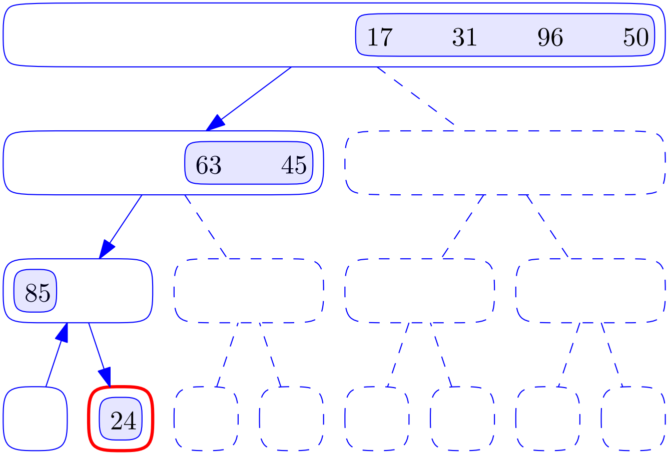
Merge-Sort



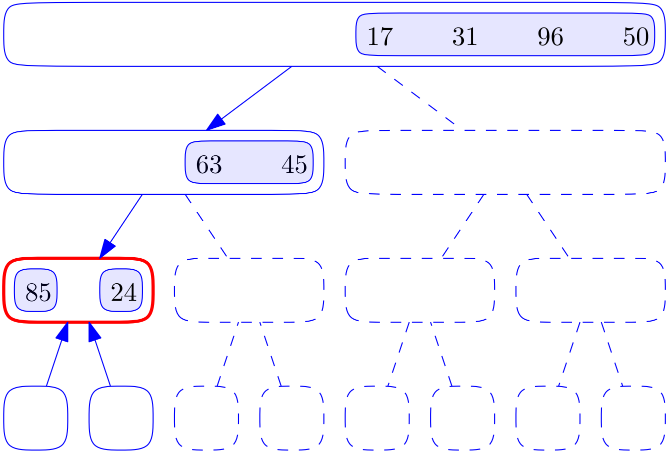
Merge-Sort



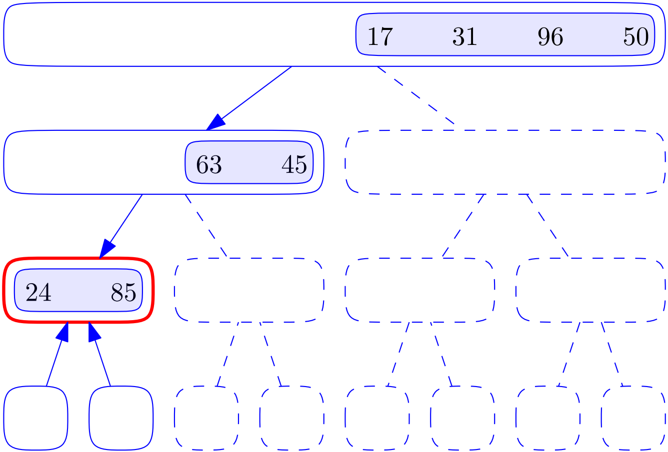
Merge-Sort



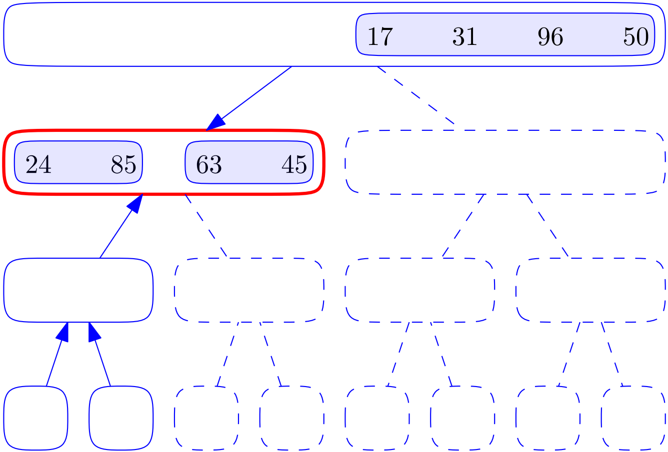
Merge-Sort



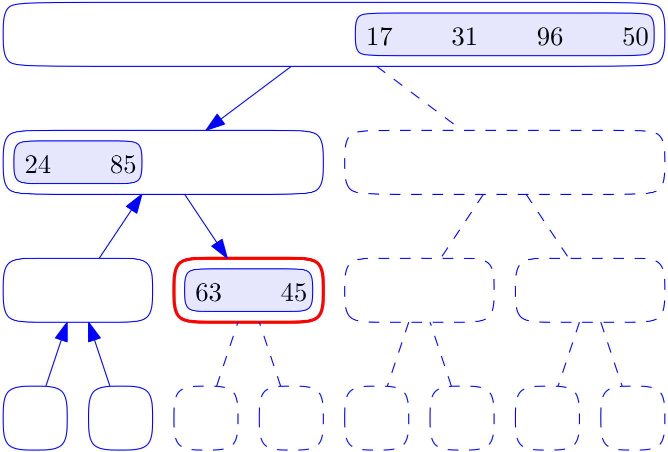
Merge-Sort



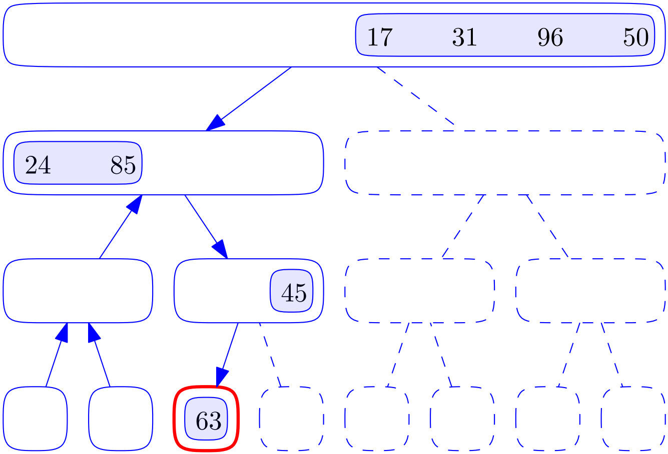
Merge-Sort



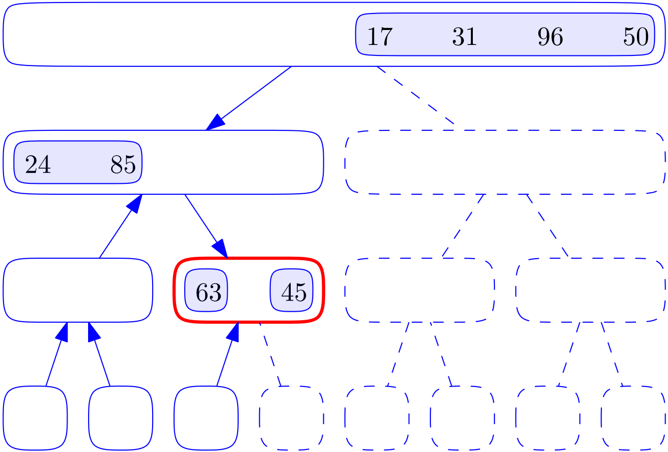
Merge-Sort



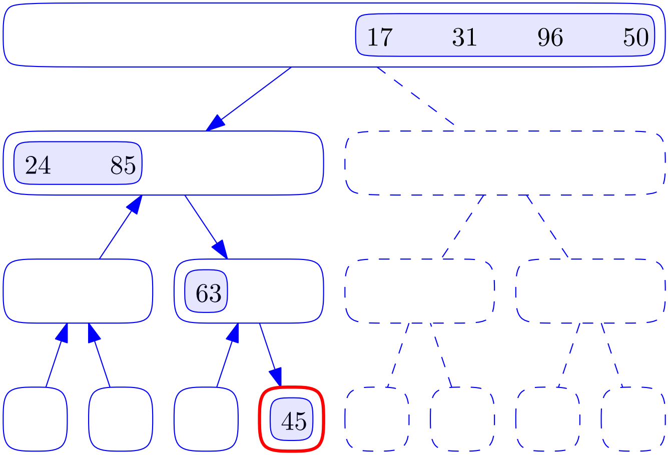
Merge-Sort



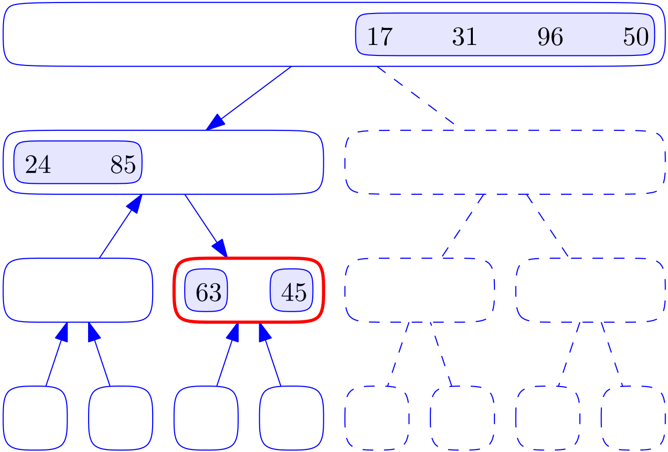
Merge-Sort



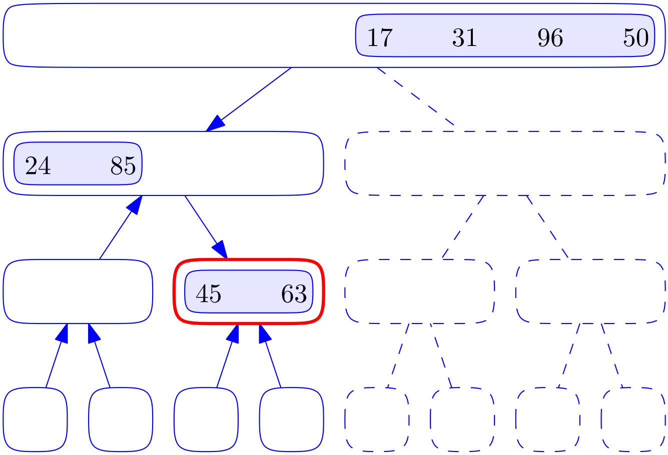
Merge-Sort



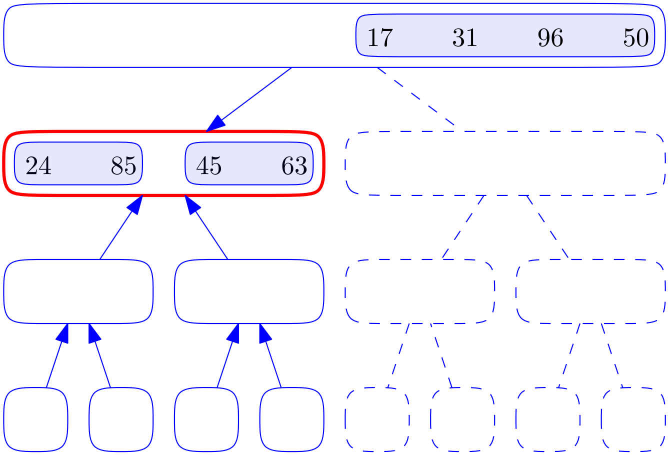
Merge-Sort



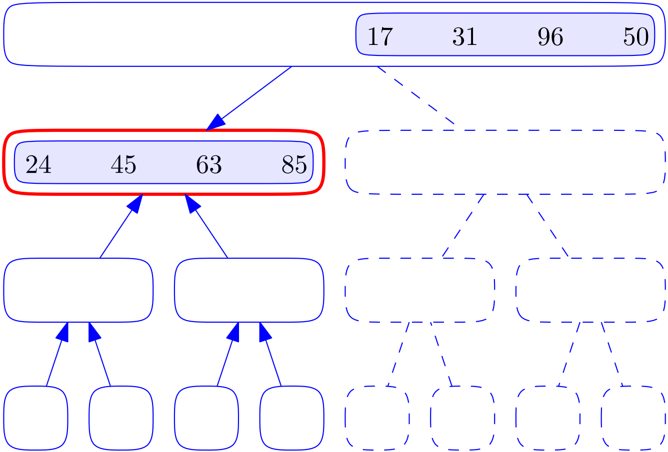
Merge-Sort



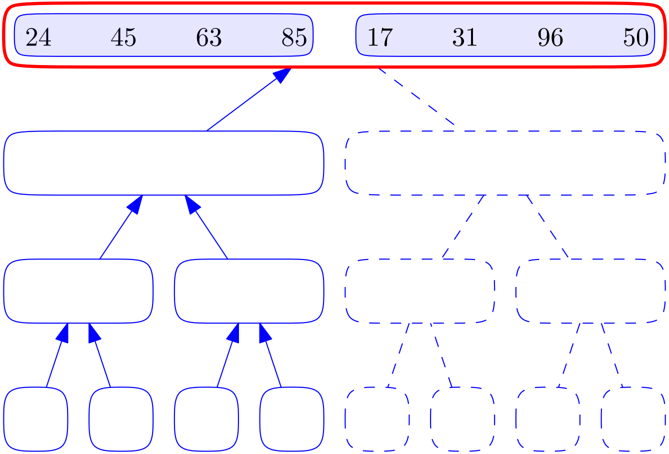
Merge-Sort



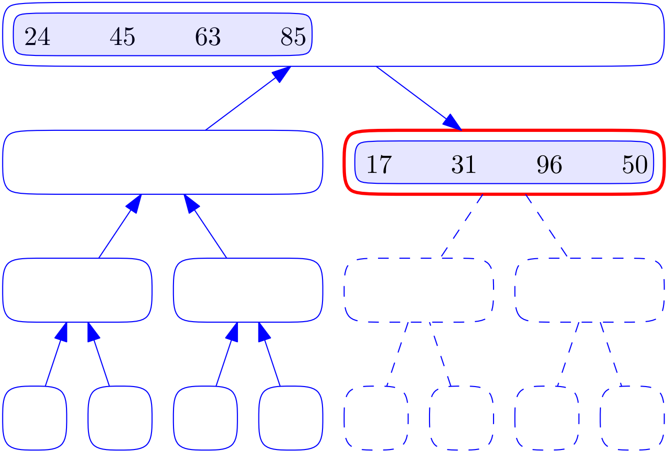
Merge-Sort



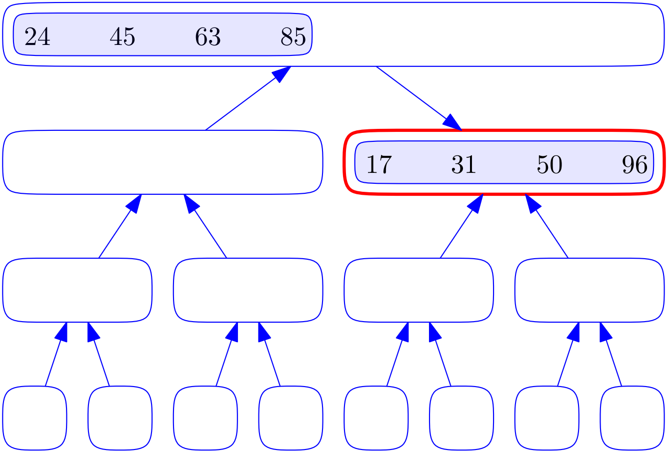
Merge-Sort



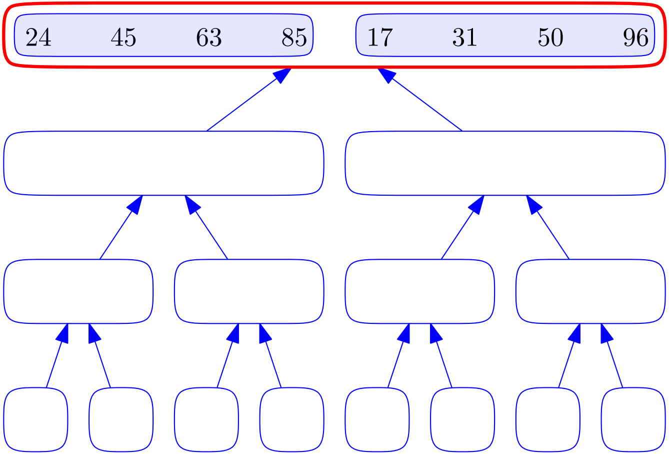
Merge-Sort



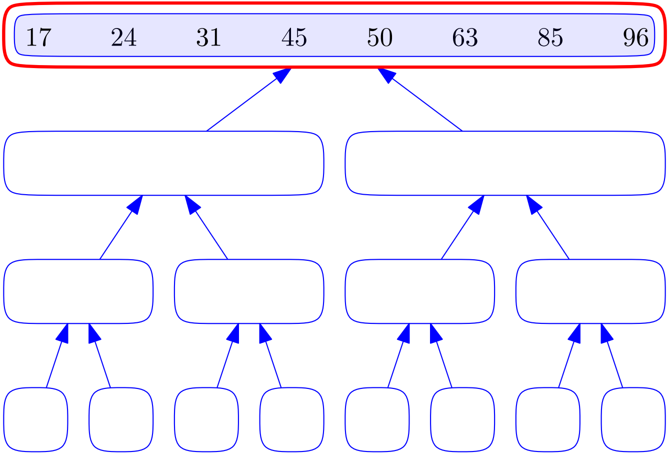
Merge-Sort



Merge-Sort



Merge-Sort



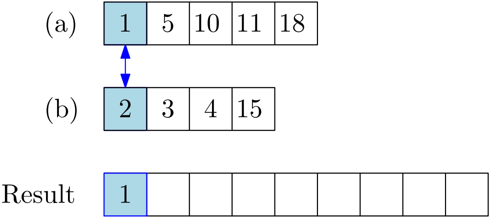
Merging two Sorted Sequences

(a)	1	5	10	11	18

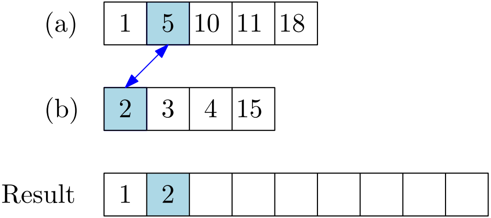
(b)	2	3	4	15

Result									

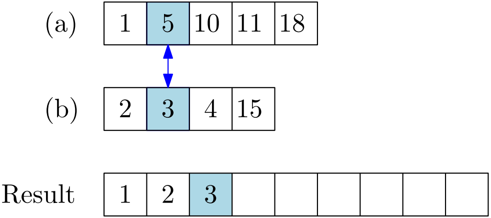
Merging two Sorted Sequences



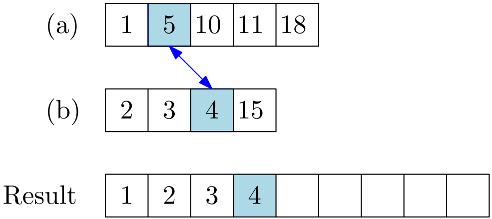
Merging two Sorted Sequences



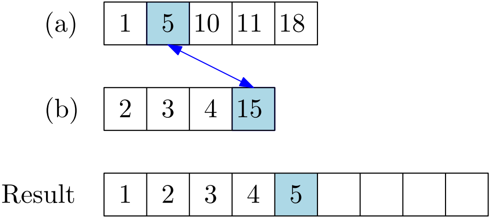
Merging two Sorted Sequences



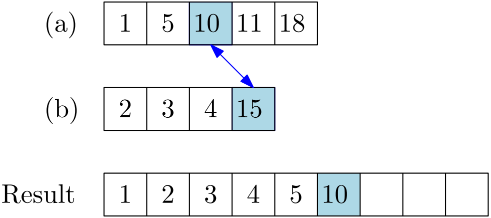
Merging two Sorted Sequences



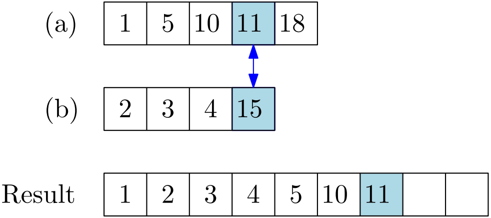
Merging two Sorted Sequences



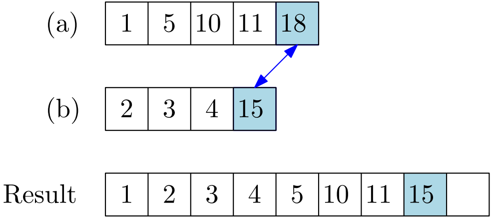
Merging two Sorted Sequences



Merging two Sorted Sequences



Merging two Sorted Sequences



Merging two Sorted Sequences

(a)	1	5	10	11	18
(b)	2	3	4	15	

Result	1	2	3	4	5	10	11	15	18

Merging two Sorted Arrays

```
procedure MERGE(S_1, S_2, S)
    i ← j ← 0
    while i < S_1.size() and j < S_2.size() do
        if S_1[i] ⩽ S_2[j] then
            S.insertBack(S_1[i])          ▷ copy the ith elements of S_1
            i ← i + 1
        else
            S.insertBack(S_2[j])          ▷ copy the jth elements of S_2
            j ← j + 1
    while i < S_1.size() do               ▷ copy the remaining elements of S_1
        S.insertBack(S_1[i])
        i ← i + 1
    while j < S_2.size() do               ▷ copy the remaining elements of S_2
        S.insertBack(S_2[j])
        j ← j + 1
```

Merging two Sorted Sequences

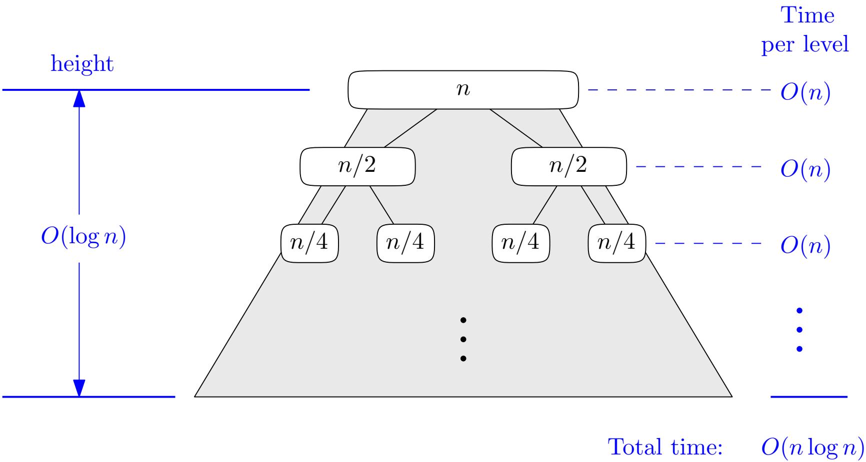
- Analysis: Let n_1 and n_2 be the sizes of S_1 and S_2, respectively.
- At each iteration of a loop, either i or j is incremented.
- So there are $n_1 + n_2$ iterations in total.
- So the running time is $O(n_1 + n_2)$.

Proposition

Two sorted sequences can be merged in linear time.

- In the pseudocode above, we assumed that S_1 and S_2 are recorded in arrays. It also works if they are stored in linked lists, and still in linear time. (See textbook.)

Analysis



- We just showed that the running time of merge-sort is $O(n \log n)$ using the *recursion tree* method.

C++ Implementation

- We present below a C++ implementation of merge-sort.
- It uses a *comparator* class. See Lecture 15.

```
template <typename E, typename C>
void mergeSort(list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;
  int n = S.size();
  if (n <= 1)
    return;                                  // already sorted
  list<E> S1, S2;
  Itor p = S.begin();
  for (int i = 0; i < n/2; i++)
    S1.push back(*p++);                 // copy first half to S1
  for (int i = n/2; i < n; i++)
    S2.push back(*p++);                // copy second half to S2
  S.clear();
  mergeSort(S1, less);                 // recur on first half
  mergeSort(S2, less);                 // recur on second half
  merge(S1, S2, S, less);              // merge S1 and S2 into S
}
```

```
template <typename E, typename C>
void merge(list<E>& S1, list<E>& S2, list<E>& S,
           const C& less) {
  typedef typename list<E>::iterator Itor;
  Itor p1 = S1.begin();
  Itor p2 = S2.begin();
  while(p1 != S1.end() && p2 != S2.end()) {
    if(less(*p1, *p2))
      S.push back(*p1++);
    else
      S.push back(*p2++);
  }
  while(p1 != S1.end())
    S.push back(*p1++);
  while(p2 != S2.end())
  S.push back(*p2++);
}
```

Divide-and-Conquer

- Merge-sort is an example of a *divide-and-conquer* algorithm. It is a general approach to algorithm design.

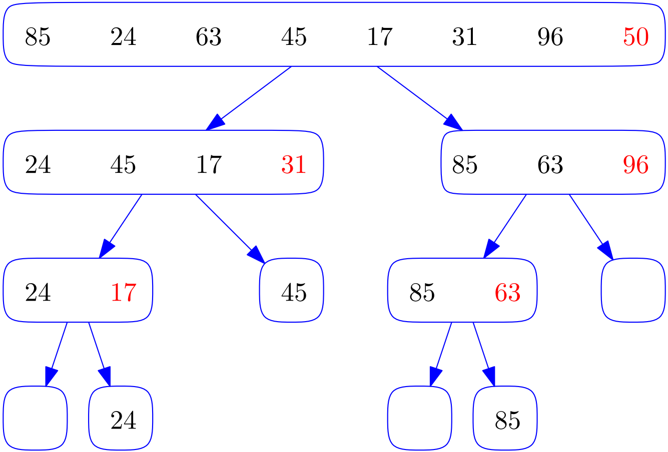
Divide-and-Conquer

The divide-and-conquer approach consists of three steps:

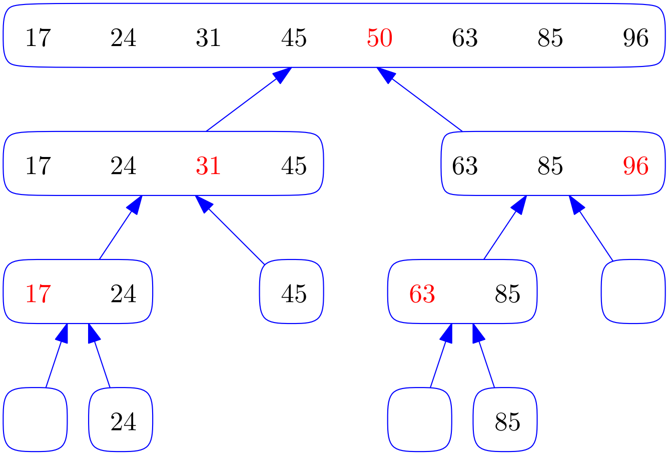
- **Divide:** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution obtained. Otherwise, divide the input data into two or more disjoint subsets.
- **Recur:** Recursively solve the subproblems associated with the subsets.
- **Conquer:** Take the solutions to the subproblems and "merge" them into a solution to the original problem.

- We now give another example: Quicksort.

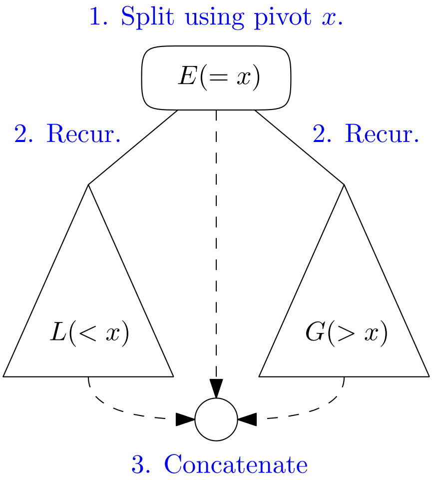
Quicksort



Quicksort



Quicksort



Quicksort

- *Quicksort* sorts a sequence S as follows:

1. **Divide**: If $|S| \geqslant 2$, choose an element $x \in S$, called the *pivot*. Usually x is the last element in S. Remove all the elements from S and put them in three sequences:
   - L, storing the elements in S less than x.
   - E, storing the elements in S equal to x.
   - G, storing the elements in S greater than x.
2. **Recur**: Recursively sort sequences L and G.
3. **Conquer**: Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

- Next slide presents pseudocode for an input sequence implemented as an array or a linked list.

```
procedure QuickSort(S)
    if S.size() ⩽ 1 then return
    p ← S.back().element()                                    ▷ the pivot
    L, E, G ← empty list-based sequences
    while !S.empty() do              ▷ scan S backwards and split in L, E, G
        if S.back().element()< p then
            L.insertBack(S.eraseBack())
        else if S.back().element()= p then
            E.insertBack(S.eraseBack())
        else
            G.insertBack(S.eraseBack())
    QuickSort(L)                                   ▷ recur on elements < p
    QuickSort(G)                                   ▷ recur on elements > p
    while !L.empty() do S.insertBack(L.eraseFront())
    while !E.empty() do S.insertBack(E.eraseFront())
    while !G.empty() do S.insertBack(G.eraseFront())
    return
```

Analysis

- Let $T(n)$ denote the running time of Quicksort. Let s_i denote the total size of the nodes at depth i in the recursion tree.
- We have $s_i \leqslant n - i$ for all i, because the pivots at level i disappear at level $i + 1$.
- So the total time spent at level i is at most $Cs_i \leqslant C(n-1)$ for some constant C.
- It follows that

$$
\begin{aligned}
T(n) &\leqslant C(n + (n-1) + \cdots + 2 + 1) \\
&= C\frac{n(n+1)}{2} = O(n^2)
\end{aligned}
$$

- So Quicksort is quadratic in the *worst case*.

Analysis

- The analysis in previous slide is in the *worst case*, when there is one pivot chosen at each level of the recursion tree.
- It means that the pivot x is always the largest element of the array.
- It could happen in practice if the input array is already sorted.

Proposition

The worst-case running time of Quicksort is $\Theta(n^2)$.

Analysis

- What about the best case?
- Suppose that the pivot x always falls in the middle of the current subsequence.
- Then the running time satisfies

$$T(n) = 2T(n/2) + O(n).$$

- It solves to $T(n) = O(n \log n)$ because it is the same as merge-sort.

Proposition

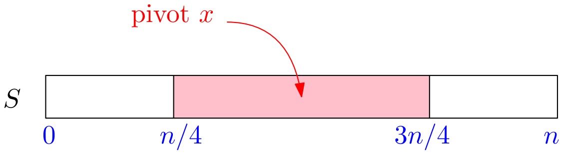
The best-case running time of Quicksort is $\Theta(n^2)$.

Randomized Quicksort

- Now suppose that we always pick x *at random* in the current sequence. What is the probability that x is always the largest element in all the arrays on which we recurse? Answer:

$$\frac{1}{n} \times \frac{1}{n-1} \times \cdots \times \frac{1}{2} = \frac{1}{n!}$$

- So the worst case is extremely unlikely to happen.
- What about the average case?
- In half of the cases, the pivot will split S into two sequences of sizes in $[n/4, 3n/4]$. We say that it is a *good* pivot.



Randomized Quicksort

- With probability $1/2$, the pivot is good and the size of the subsequences we recurse on shrinks by a factor at least $4/3$.
- So intuitively, the depth of the recursion tree is $O(\log n)$.
- I will not prove it in this course. See textbook, or CSE331: Introduction to Algorithms.

Proposition

The expected running time of randomized Quicksort is $O(n \log n)$.

- This is an *average-case* analysis. Here the average is over the random choices of the algorithm: Even on worst-case input, randomized Quicksort is expected to run in $O(n \log n)$ time.
- It can also be shown that it is $O(n \log n)$ with high probability.
- This is why Quicksort is very fast in practice.

Implementation

- Our implementation of merge-sort requires to create, in addition to the input sequence S, three additional sequences L, E, and G.
- The sum of their sizes is n.
- So merge-sort uses $O(n)$ space in addition to the size of the input.
- On the other hand, it is possible to implement Quicksort in such a way that it uses only $O(1)$ space in addition to the input array.
- We say that such an implementation of Quicksort is *in-place*.
- This is one reason why Quicksort is very efficient in practice: it uses very little space.
- Next slide gives the C code of in-place Quicksort.
- More detailed explanation are given in the textbook, or CSE331: Introduction to Algorithms.

C Code (source: *The C programming language* by Kernighan & Ritchie)

```
void qsort(int v[], int left, int right){
   int i, last;
   void swap(int v[], int i, int j);
   if (left >= right)
      return;
   swap(v, left, (left + right)/2);
   last = left;
   for (i = left + 1; i <= right; i++)
      if (v[i] < v[left])
         swap(v, ++last, i);
   swap(v, left, last);
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}
void swap(int v[], int i, int j){
   int temp; temp = v[i]; v[i] = v[j]; v[j] = temp;
}
```