12
Sorting
Chapter 12

Sorting

One of the most important computing studies.

Many algorithms rely on the assumption of "data is sorted."

Once we have a sorted data, we can do efficient searches (such as binary search).

There various sorting types (*):

- Comparison-based sorts
- Non-comparison sorts

(*) https://en.wikipedia.org/wiki/Sorting_algorithm

Sorting

Comparison-based Sort

Relies on the < (less than) operator. Items in the sequence should be able to be compared.

Relies on two mathematical properties:

- Irreflexivity (given a key k to compare, $k < k$ should evaluate to false)
- Transitivity (given three key k_1, k_2, and k_3 to compare, if $k_1 < k_2$ and $k_2 < k_3$, then $k_1 < k_3$)

Transivity property allows development of efficient sorting algorithms.

Sorting

Comparison-based Sort

Up to now, we have covered the following sorting algorithms in Array-based Squences and Priority Queues lectures:

- Insertion Sort
- Selection Sort
- Heap Sort

Sorting

Comparison-based Sort

In this lecture, we cover the following sorting algorithms:

- Merge Sort
- Quick Sort
- Bubble Sort

Merge Sort and Quick Sort alorithms use an algorithm design pattern called **divide-and-conquer**.

Sorting

Non-comparison Sort

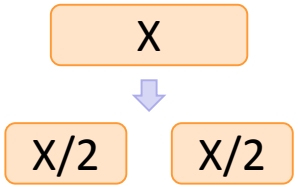
We will also cover the following non-comparison sorting algorithms:

- Bucket Sort
- Radix Sort

Divide-and-conquer Design Pattern

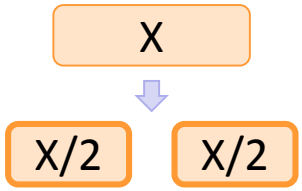
Comprises three steps:

Divide



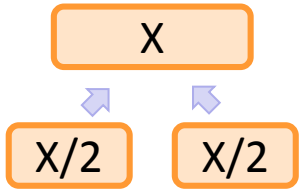
If the input size is below a threshold, return an immediate solution in constant time, otherwise, divide the input into multiple subsets.

Conquer



Recursively solve the subproblems over these subsets.

Combine



Merge the solutions to the subproblems into the final solution.

Divide-and-conquer Design Pattern

A sorting example (Sequence S has n items.):

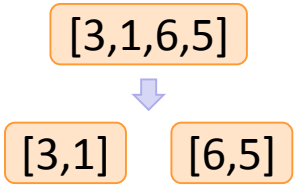
Divide

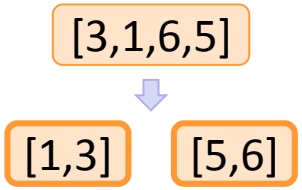
If S has zero or one item, return S immediately since it is already sorted. Otherwise, create two subsequences S1 and S2 as follows:

mid = len(S)//2

S1 = S[0, mid]

S2 = S[mid:len(S)]





Conquer

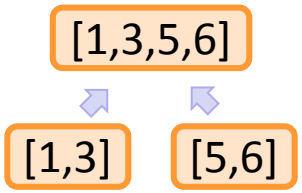
Recursively sort S1 and S2. At the end of this stage, S1 and S2 will be sorted.

Divide-and-conquer Design Pattern

A sorting example (Sequence S has n items.):

Combine

Merge the solutions for S1 and S2 so that problem becomes solved for S.



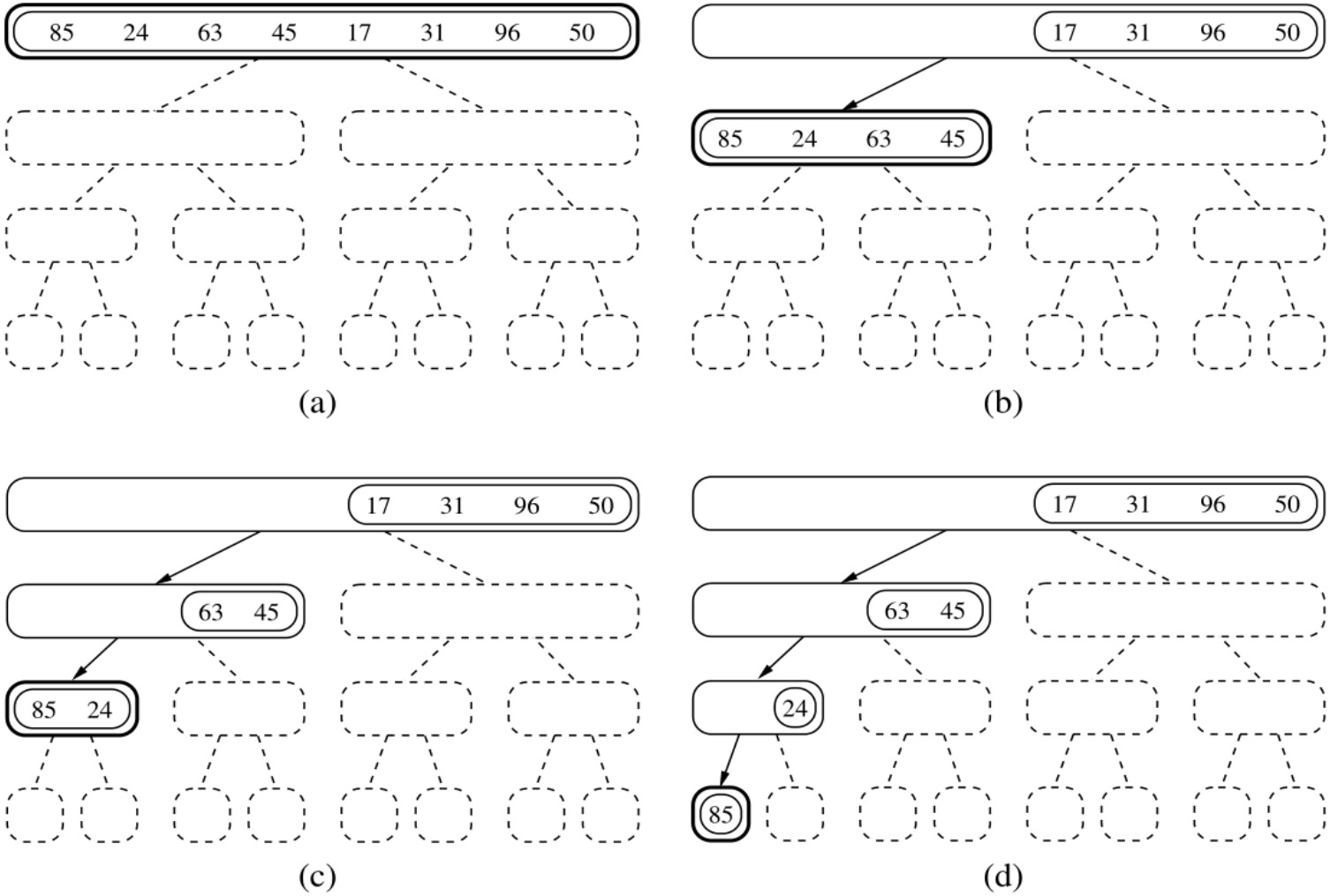
Merge Sort

Adopts divide-and-conquer pattern.

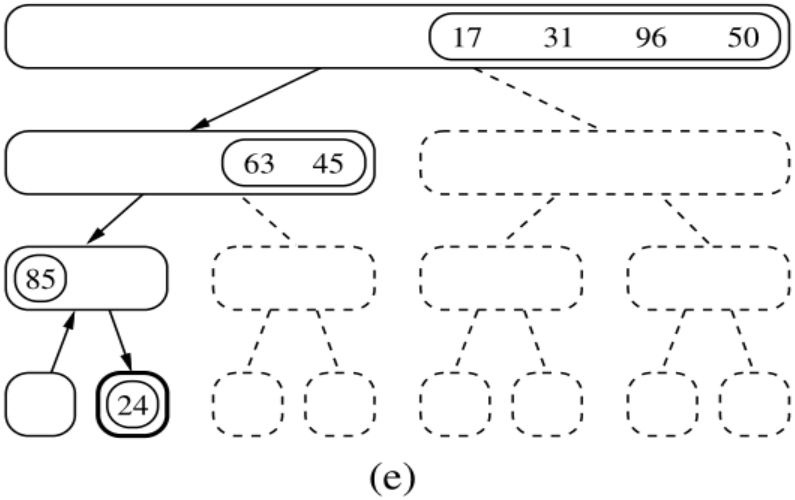
The problem is recursively divided into subproblems until a base condition is met: The subproblem size is less then 2 (i.e., the size of the sequence to sort is either 0 or 1).

When the base condition is met, the input sequence is returned immediately to the caller. Then the caller merges these two ordered lists such that the result is also an ordered list.

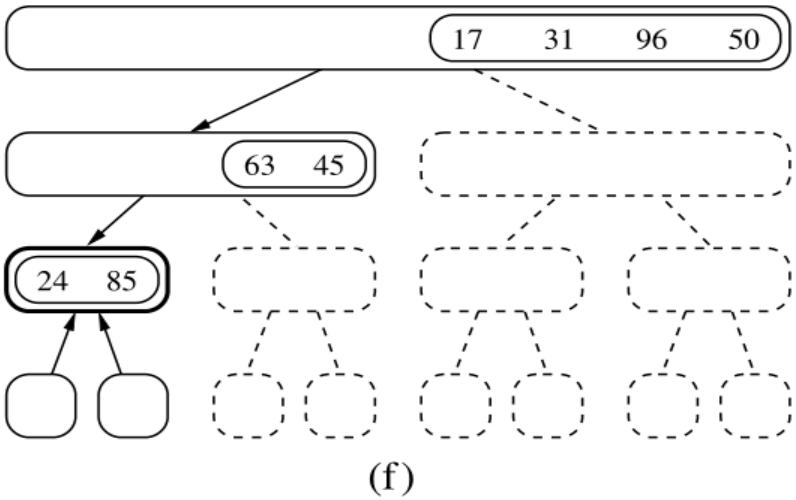
Merge Sort


85 24 63 45 17 31 96 50
(a)
17 31 96 50
85 24 63 45
(b)
17 31 96 50
63 45
85 24
(c)
17 31 96 50
63 45
24
85
(d)

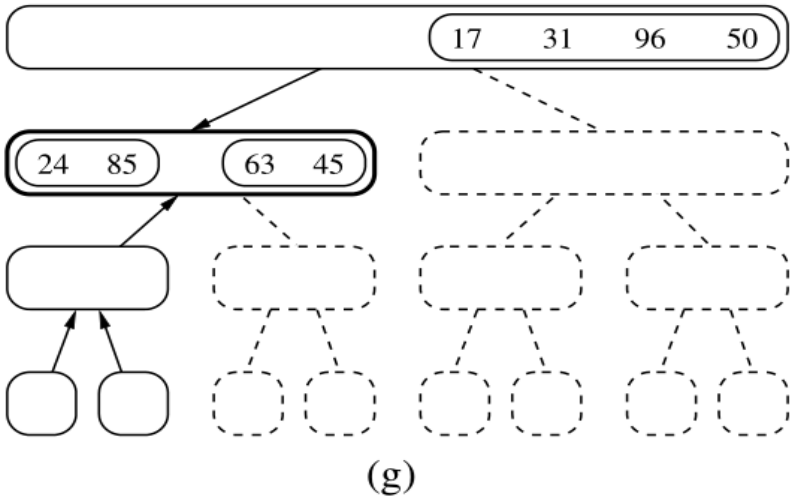
Merge Sort



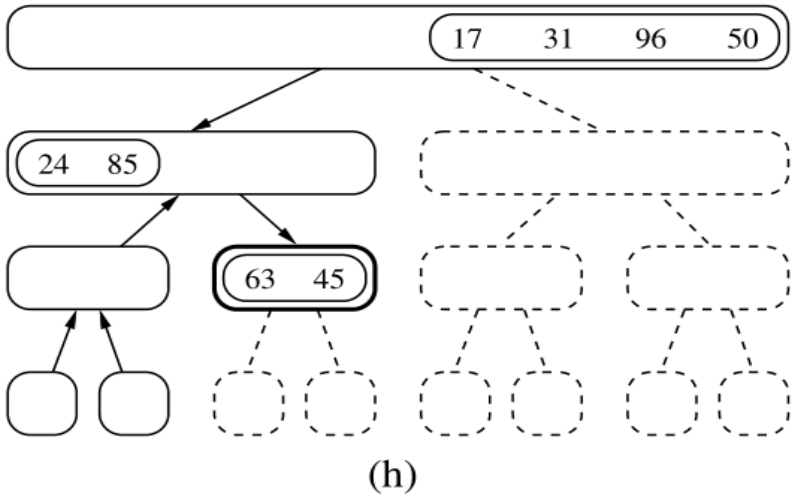
(e)



(f)

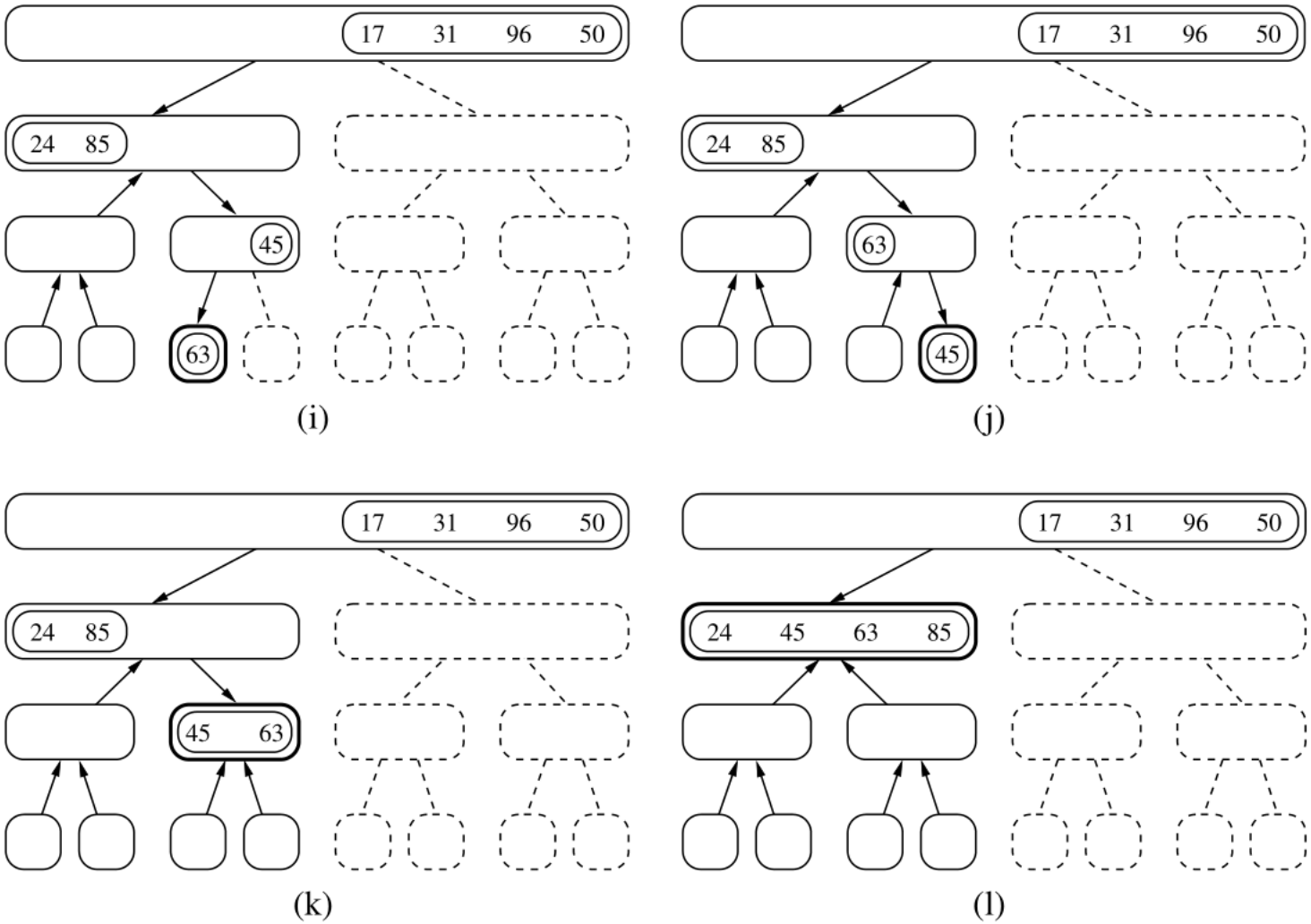


(g)

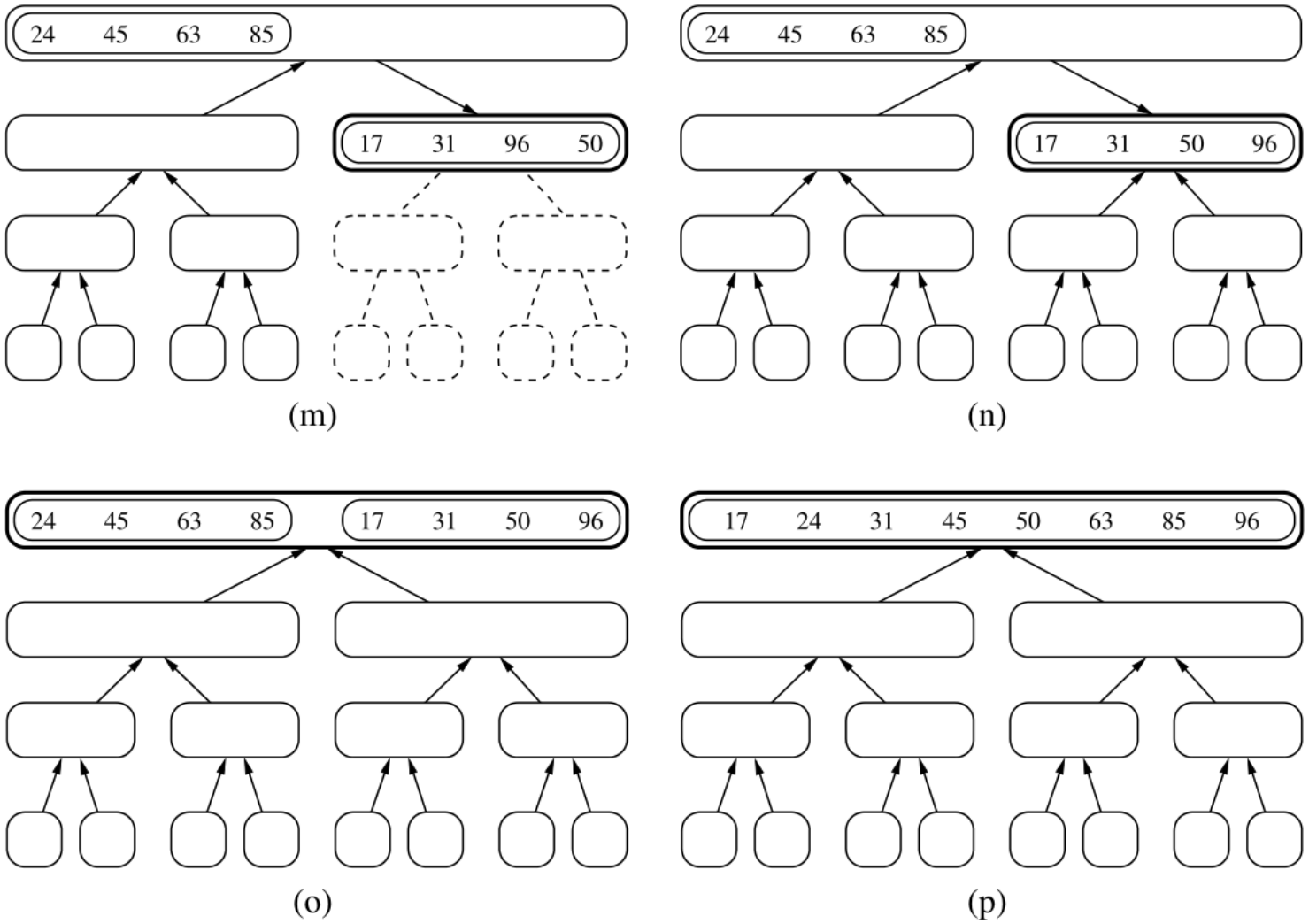


(h)

Merge Sort


17 31 96 50
24 85
45
63
(i)
17 31 96 50
24 85
63
45
(j)
17 31 96 50
24 85
45 63
(k)
17 31 96 50
24 45 63 85
(l)

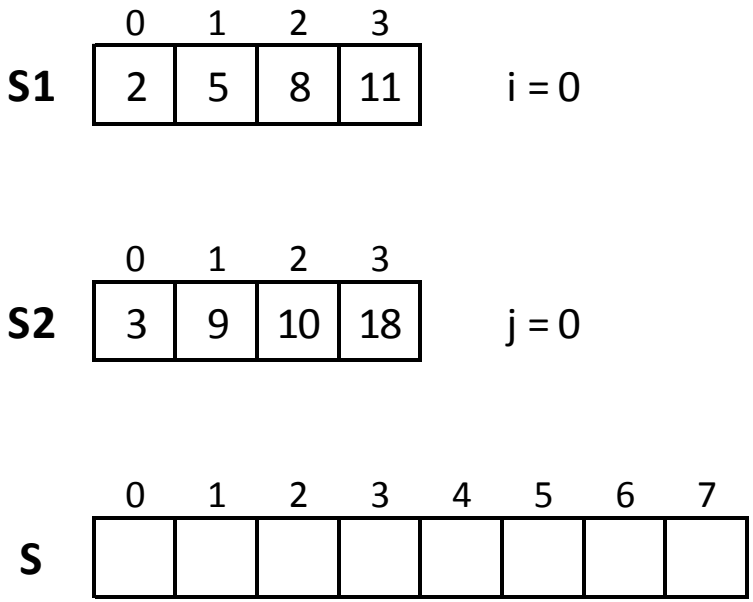
Merge Sort


24 45 63 85
17 31 96 50
(m)
24 45 63 85
17 31 50 96
(n)
24 45 63 85 17 31 50 96
(o)
17 24 31 45 50 63 85 96
(p)

Merge Sort

Merge Operation

Merge two ordered lists such that the merge result is also an ordered list.



Merge Sort

```
1  def merge(S1, S2, S):
2    """Merge two sorted Python lists S1 and S2 into properly sized list S."""
3    i = j = 0
4    while i + j < len(S):
5      if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6        S[i+j] = S1[i]                  # copy ith element of S1 as next item of S
7        i += 1
8      else:
9        S[i+j] = S2[j]                  # copy jth element of S2 as next item of S
10       j += 1
```

The complexity of merge operstion is O(len(S1) + len(S2)) = O(len(S)).

Merge Sort

```
1  def merge_sort(S):
2    """Sort the elements of Python list S using the merge-sort algorithm."""
3    n = len(S)
4    if n < 2:
5      return                          # list is already sorted
6    # divide
7    mid = n // 2
8    S1 = S[0:mid]                     # copy of first half
9    S2 = S[mid:n]                     # copy of second half
10   # conquer (with recursion)
11   merge_sort(S1)                    # sort copy of first half
12   merge_sort(S2)                    # sort copy of second half
13   # merge results
14   merge(S1, S2, S)                  # merge sorted halves back into S
```

Merge Sort

$T(0) = T(1) = 1$

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

Let's solve the recurrence equation.

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

$T(n) = 2^2T(n/4) + 4n$

$T(n) = 2^3T(n/8) + 6n$

...

$T(n) = 2^iT(n/2^i) + 2in$

Merge Sort

$T(n) = 2^iT(n/2^i) + 2in$

At some point during the execution, the subproblem size ($n/2^i$) will reach to 1 (the base case). At this moment i will be

$n/2^i = 1 \Rightarrow n = 2^i \Rightarrow \log n = i.$

So,

$T(n) = 2^{\log n} T(n/2^{\log n}) + 2*\log n*n$

$T(n) = n + 2n\log n$

For $g(n) = n\log n$, $k = 4$, and $n_0 = 2$, we can easily show that

T(n) is O(nlogn).

Quick Sort

Adopts the divide-and-conquer pattern (Assume sequence S is to be sorted.).

Divide

If S has less than 2 items, do nothing. Otherwise, pick a specific item x (it is called pivot) and remove all items from S and move them to sequences L, E, and G such that,

- All items less than x are in L,
- All items equal to x are in E, and
- All items greater than x are in G.

Quick Sort

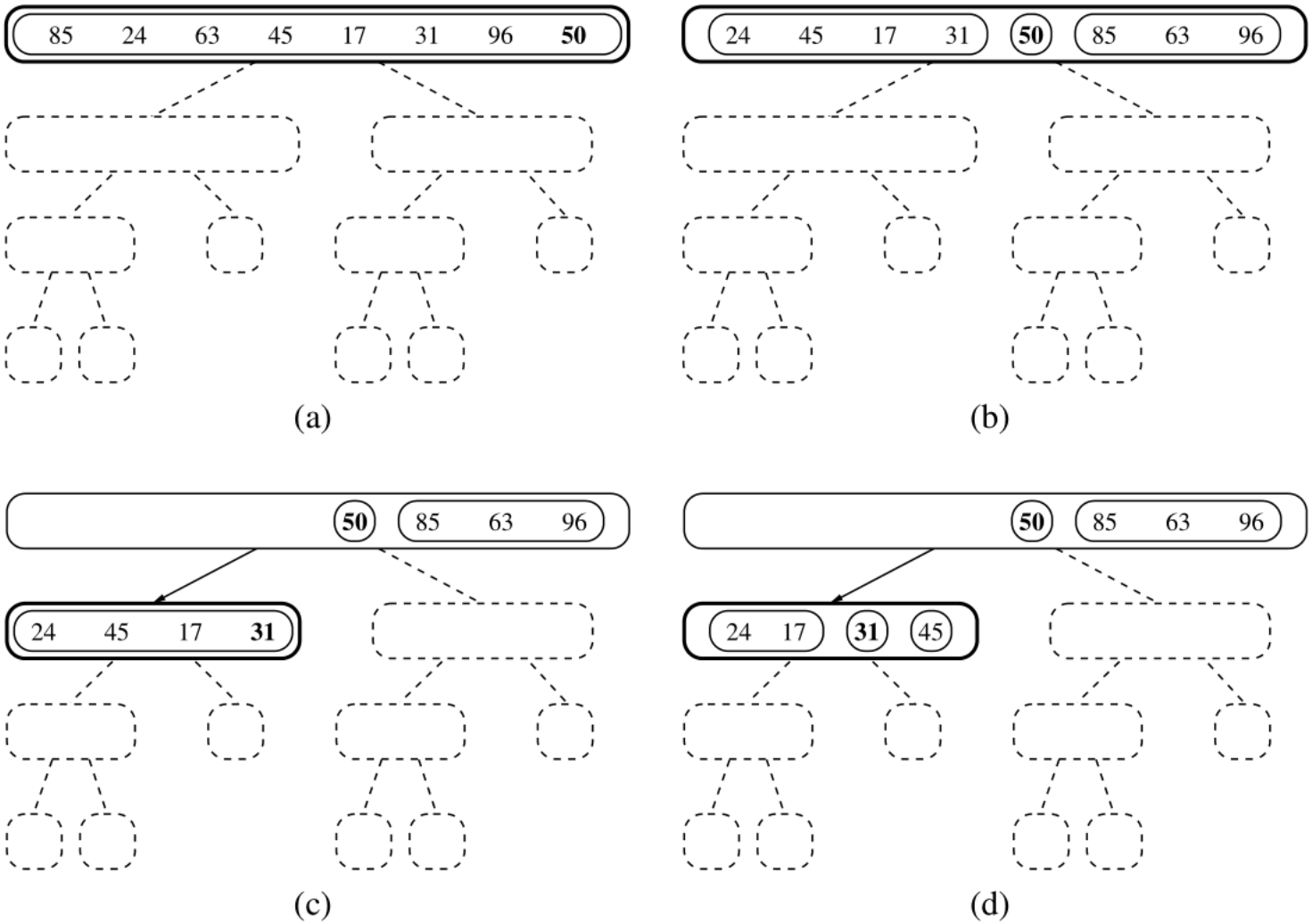
Conquer

Apply quick sort on L and G.

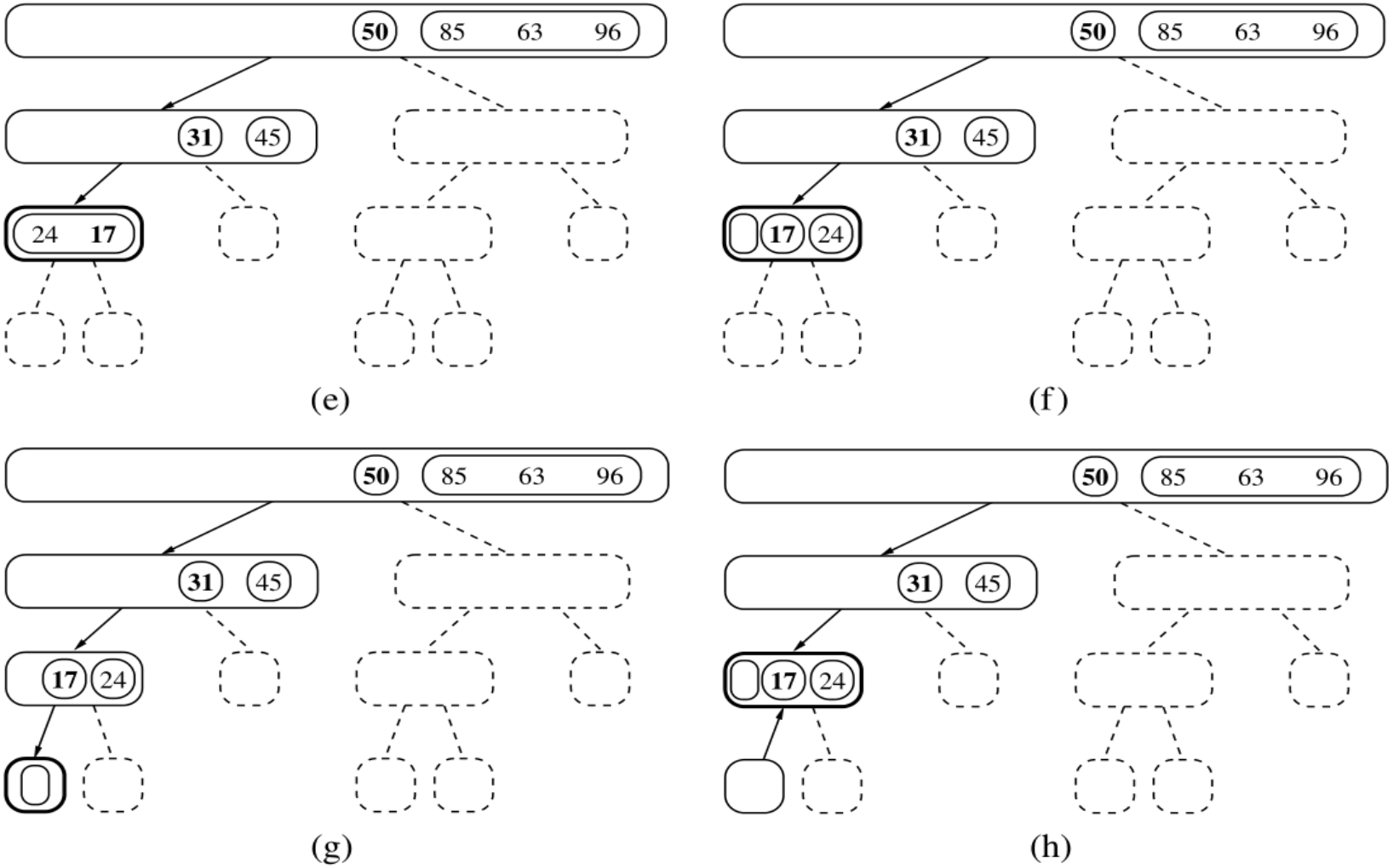
Combine

Put back the items into S in order by inserting items from L, then from E, and finally from G.

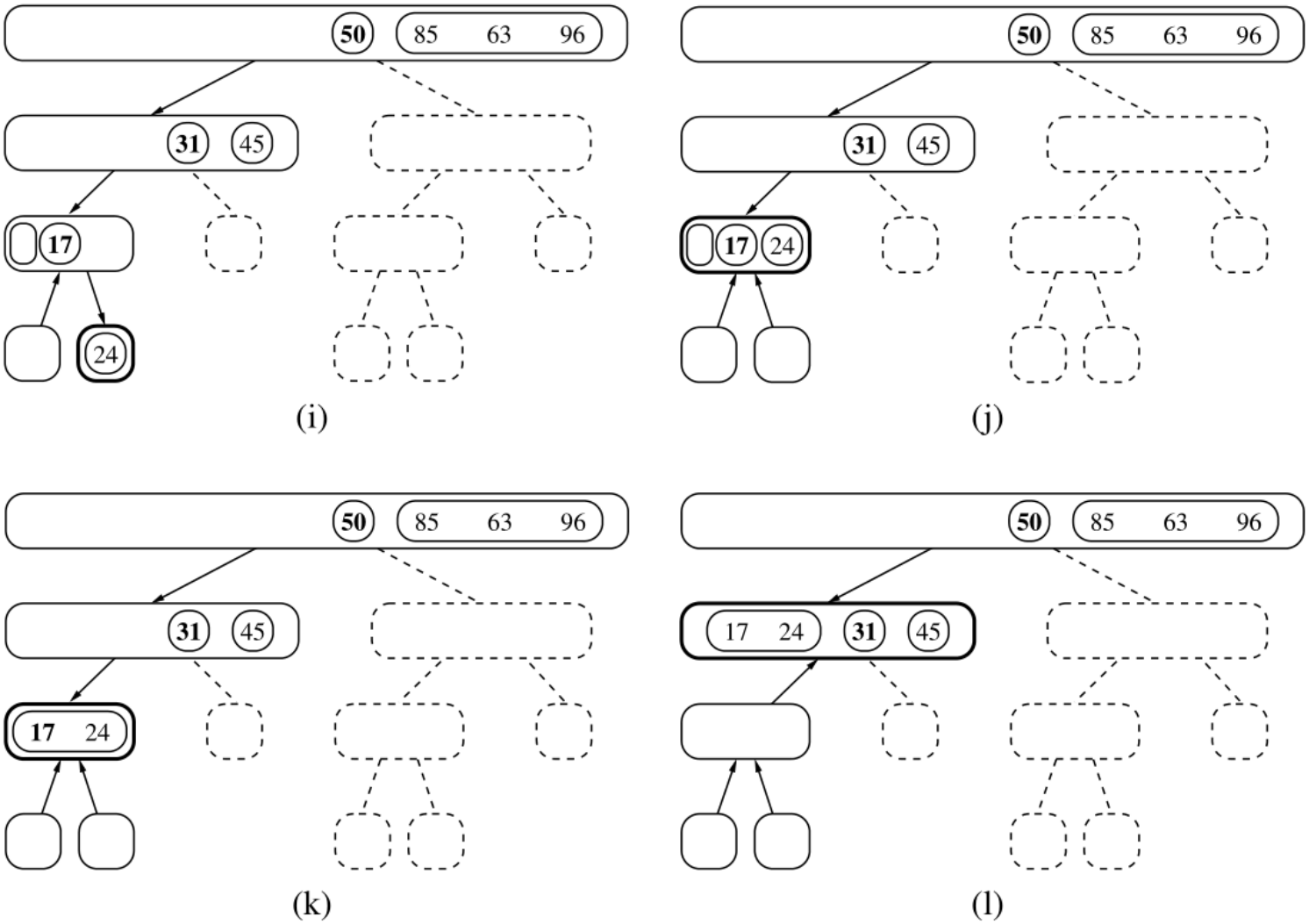
Quick Sort


85 24 63 45 17 31 96 50
(a)
24 45 17 31 50 85 63 96
(b)
50 85 63 96
24 45 17 31
(c)
50 85 63 96
24 17 31 45
(d)

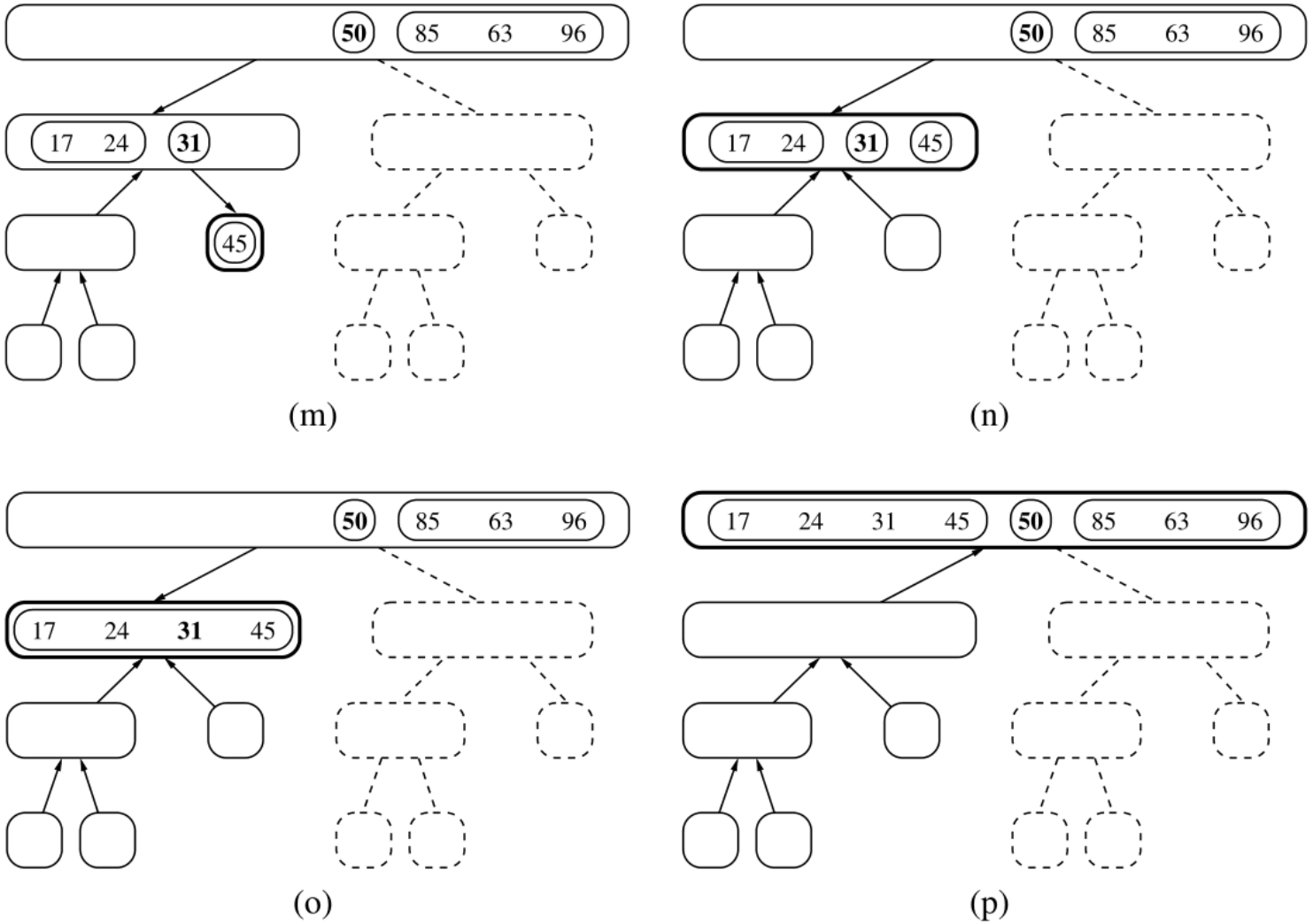
Quick Sort


50 85 63 96
31 45
24 17
(e)
50 85 63 96
31 45
17 24
(f)
50 85 63 96
31 45
17 24
(g)
50 85 63 96
31 45
17 24
(h)

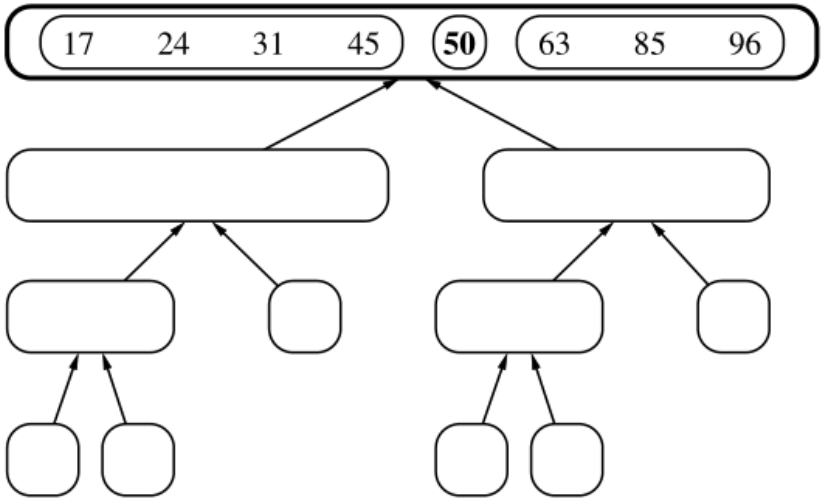
Quick Sort


50 85 63 96
31 45
17
24
(i)
50 85 63 96
31 45
17 24
(j)
50 85 63 96
31 45
17 24
(k)
50 85 63 96
17 24 31 45
(l)

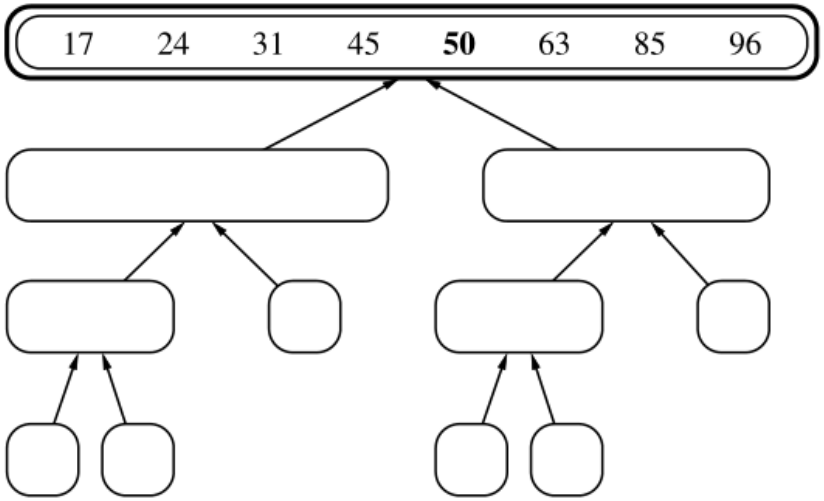
Quick Sort


50 85 63 96
17 24 31
45
(m)
50 85 63 96
17 24 31 45
(n)
50 85 63 96
17 24 31 45
(o)
17 24 31 45 50 85 63 96
(p)

Quick Sort



(q)



(r)

Quick Sort

```
1   def quick_sort(S):
3     n = len(S)
4     if n < 2:
5       return
7     p = S.first( )
8     L = LinkedQueue( )
9     E = LinkedQueue( )
10    G = LinkedQueue( )
11    while not S.is_empty( ):
12      if S.first( ) < p:
13        L.enqueue(S.dequeue( ))
14      elif p < S.first( ):
15        G.enqueue(S.dequeue( ))
16      else:
17        E.enqueue(S.dequeue( ))
19    quick_sort(L)
20    quick_sort(G)
21    # concatenate results
22    while not L.is_empty( ):
23      S.enqueue(L.dequeue( ))
24    while not E.is_empty( ):
25      S.enqueue(E.dequeue( ))
26    while not G.is_empty( ):
27      S.enqueue(G.dequeue( ))
```

Quick Sort Complexity

Worst-case

S is order (either in ascending or descending order, does not matter), and the pivot is selected as the last item in S, there are n items in S, and items are distinct.

In that case, (assuming the sequence is ordered in ascending order) each division would create L, E, and G subsequences whose sizes n-1, 1, and 0, respectively.

Following iterations would bear the same unbalanced results and in total,

(n-1) + (n-2) + ... + 1 comparison operations would be performed.

Similarly, merge operation would cost 2+3+...+n operations.

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

Quick Sort Complexity

Average- and Best-cases

Quick sort is O(nlogn) in the best and average cases.

The pivot could be selected randomly (this is called randomized quick sort) or other techniques such as median-of-three could be used. Due to these reasons, it is highly likely that quick sort will work in average time.

Median-of-three

Take the median of the values in the front, middle, and the tail of the array.

Bubble Sort

A bad and inefficient sorting algorithm working in $O(n^2)$.

Somehow traditionally took its spot in data structures and algorithms courses for decades.

The only advantage seems to be its implementation simplicity.

Bubble Sort

How it works:

For each consecutive pair of items (a, b) in the array, a comparison is made.

- If a > b, then a swap operation between these two items is performed so that they are positioned in the array as (b,a).
- Otherwise, no action is performed.

There will be n-1 comparisons from start to the end. This operation of n-1 comparisons is called a scan.

Scan operation is performed iteratively until there is no need for any swaps.

Bubble Sort

scan 1	**5**	**1**	4	2	8	=>	**1**	**5**	4	2	8	swap 5 > 1	
	1	**5**	**4**	2	8	=>	1	**4**	**5**	2	8	swap 5 > 4	
	1	4	**5**	**2**	8	=>	1	4	**2**	**5**	8	swap 5 > 2	
	1	4	2	**5**	**8**	=>	1	4	2	**5**	**8**	no action	swapped = true
scan 2	**1**	**4**	2	5	8	=>	**1**	**4**	2	5	8	no action	
	1	**4**	**2**	5	8	=>	1	**2**	**4**	5	8	swap 4 > 2	
	1	2	**4**	**5**	8	=>	1	2	**4**	**5**	8	no action	
	1	2	4	**5**	**8**	=>	1	2	4	**5**	**8**	no action	swapped = true
scan 3	**1**	**2**	4	5	8	=>	**1**	**2**	4	5	8	no action	
	1	**2**	**4**	5	8	=>	1	**2**	**4**	5	8	no action	
	1	2	**4**	**5**	8	=>	1	2	**4**	**5**	8	no action	
	1	2	4	**5**	**8**	=>	1	2	4	**5**	**8**	no action	swapped = false

https://en.wikipedia.org/wiki/Bubble_sort

Bubble Sort

```
def bubble_sort(S):
    swapped = True
    n = len(S)
    while swapped:
        swapped = False
        for i in range(n-1):
            if S[i] > S[i+1]:
                S[i], S[i+1] = S[i+1], S[i]
                swapped = True
```

Wors-case and average case is $O(n^2)$.

Non-comparison Sorting Algorithms

Up to now, we have covered comparison-based sorting algorithms.

The best computational complexity for comparison-based sorting seems to be O(nlogn).

From theoretical point of view, for any comparison-based sorting algorithm, the lower bound for sorting is Ω(nlogn).

Can we do any better?

- If more information about the sequence (to be sorted) is provided, it is possible.
    - If the range of numbers to be sorted is known, if sorting involves only characters, etc.

Bucket Sort

Let S be a sequence of n entries (i.e., (k,v) pairs) with distinct keys ($k_i \neq k_j$ where $i \neq j$), and it is known that k is an integer between [0, N-1] where N is an integer s.t. N >= 2.

Let B be an array of buckets, i.e., objects of a sequence type or a linear data structure such as queue or stack (We will use a queue. So B[k] will be a queue instance).

The problem is to sort the entries of S by their keys in ascending order.

Example Input: (3, 'A'), (1, 'B'), (2, 'S'), (8, 'F')

Example Output: (1, 'B'), (2, 'S'), (3, 'A') , (8, 'F')

Bucket Sort

Pseudo Code for Bucket Sort

S is a sequence of (k,v) tuples where k is key and v is value, and keys are distinct and known to be in range [0, N-1].

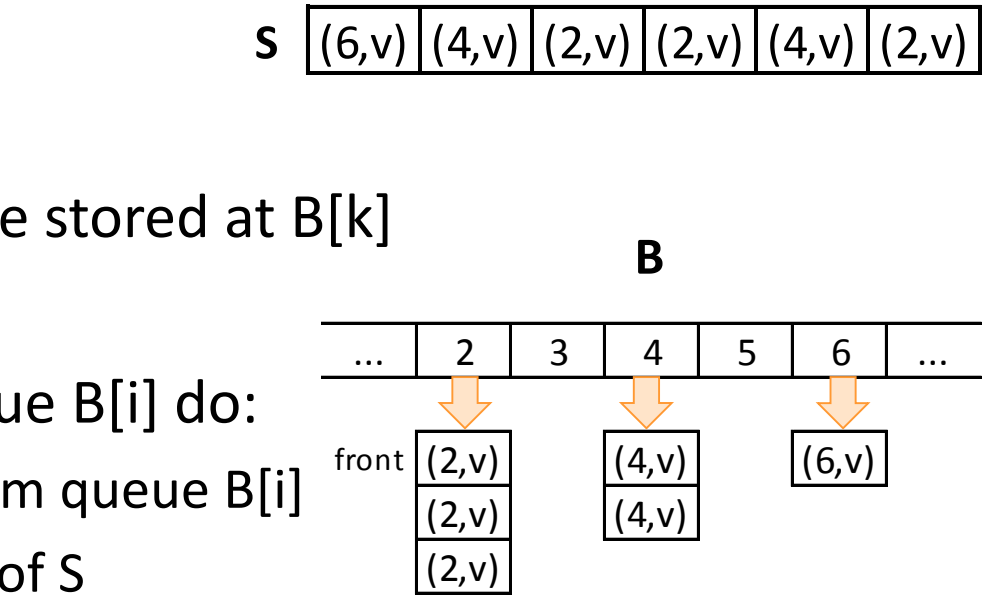
Let B be an array of N queues, whose each cell is initially empty.

for each entry (k,v) in S do:

- Remove (k,v) from S
- Enqueue (k,v) to the queue stored at B[k]

for i in 0 ... N-1 do:

- for each entry (k,v) in queue B[i] do:
  - Remove entry (k,v) from queue B[i]
  - Insert (k,v) to the end of S



Bucket Sort

Complexity

O(n+N). Why?

N determines bucket sort's complexity. If N is O(n), then the bucket sort is O(n), if N is O(nlogn), then the bucket sort is O(nlogn).

Recall that [0, N-1] is the range where keys can be in. So if the range of the keys bounded by the number of items, bucket sorting is O(n).

Bucket Sort

Bucket sort is stable.

A sorting algorithm is said to be stable when two entries having the same key values are in the same order before and after sorting operation.

Example

Input Array: (4,A), (2,J), (2,K), (1,F)

Output Array: (1,F), (2,J), (2,K), (4,A)

One of the possible outputs if the sorting is not stable:

(1,F) , (2,K), (2,J), (4,A)

Radix Sort

Bucket sorting algorithm operates over a single key. There might be cases where we can have multiple keys.

Radix sort is a sorting algorithm where bucket sorting is applied multiple times on a sequence of entries with multiple keys.

Multiple key example: $((k_1, k_2, k_3), v)$

For a sequence of entries with d keys (d-tuples), the bucket sorting is applied starting from $(d-1)^{st}$ key down until to the 0^{th} key.

Radix Sort

Example

Sequence to be sorted with two keys (k_0 and k_1)

S = (3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)

First sort by k_1:

S_1 = (1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)

Then sort by k_2:

$S_{1,0}$ = (1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)

Complexity

O(d(n+N)).

Comparison of Sorting Algorithms

Insertion Sort

Complexity $O(n^2)$

Pros

Works great with almost sorted sequences.

A great choice for small sequences (n less then 50, 100, or 200-ish).

Simple to program.

Cons

Bad for sorting large sequences.

Comparison of Sorting Algorithms

Heap Sort

Complexity O(nlogn)

Pros

Can be executed in place --- low memory footprint.

Good for small- and medium-sized problems.

Cons

For large sequences, not as good as quick sort and merge sort.

Does not support stable sorting.

Comparison of Sorting Algorithms

Quick Sort

Complexity $O(n^2)$

Pros

Expected time is O(nlogn) and experimentally shown that it is better than heap sort and merge sort.

Cons

O(nlogn) is highly likely, but not guaranteed.

Does not support stable sort.

Info

It was the default sorting algorithm for many famous programming languages like C and Java, and Unix environments.

Comparison of Sorting Algorithms

Merge Sort

Complexity O(nlogn)

Pros

It's worst-case complexity is O(nlogn).

Easy to be processed by the computing system --- merge operations performed with blocks, no references to other parts of the memory.

Cons

Hard to make it run inplace --- memory footprint is larger than that of heap sort or quick sort.

Info
A mixture of merge and insertion sort is used in Python's `list` class.

Comparison of Sorting Algorithms

Bucket Sort and Radix Sort

Complexity O(n+N) / O(d(n+N))

Pros

Very effective if (a) keys are in a small range, (b) when sorting strings, or (c) d-tuple keys from a discrete range --- it can run in close-to-linear time, which is significantly below O(nlogn).

Cons

If above conditions are not met, it becomes useless. Particularly if N goes way beyond n, complexity might be worse then even $O(n^2)$.