U O W

Sorting

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

What is Sorting

- The process of placing elements from a collection in some kind of order.
  - For example, a list of words could be sorted alphabetically or by length.
  - A list of cities could be sorted by population, by area, or by zip code.
  - The binary search is benefited from having a sorted list prior the search.
- Many sorting algorithms are existed
- Sorting a large number of items can take a substantial amount of computing resources.
- The efficiency of a sorting algorithm is related to the number of items being processed.
  - For small collections, a complex sorting method may be more trouble than it is worth. The overhead may be too high.
  - The efficient sorting algorithms show its advantage in the larger collection.

- Next, we will discuss several sorting techniques and compare them with respect to their running time.

Objectives

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort

Bubble Sort

- The bubble sort makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place.
- In essence, each item “bubbles” up to the location where it belongs.

Bubble Sort

Before sort

Index value	0	1	2	3	4	5	6	7	8	9	10
Values	88	34	77	44	26	82	39	28	20	9	65

First round of the bubble sort.

Index value	0	1	2	3	4	5	6	7	8	9	10
Values	34	77	44	26	82	39	28	20	9	65	88

Bubble Sort

Hash value	0	1	2	3	4	5	6	7	8	9	10
Start Second round	34	77	44	26	82	39	28	20	9	65	88
Not exchange	34	77	44	26	82	39	28	20	9	65	88
Exchange	34	44	77	26	82	39	28	20	9	65	88
Exchange	34	44	26	77	82	39	28	20	9	65	88
Not exchange	34	44	26	77	82	39	28	20	9	65	88
Exchange	34	44	26	77	39	82	28	20	9	65	88
Exchange	34	44	26	77	39	28	82	20	9	65	88
Exchange	34	44	26	77	39	28	20	82	9	65	88
Exchange	34	44	26	77	39	28	20	9	82	65	88
Exchange	34	44	26	77	39	28	20	9	65	82	88
End Second round	34	44	26	77	39	28	20	9	65	82	88

Bubble Sort

Hash value	0	1	2	3	4	5	6	7	8	9	10
3rd round- Not exchange	34	44	26	77	39	28	20	9	65	82	88
Exchange	34	26	44	77	39	28	20	9	65	82	88
Not Exchange	34	26	44	77	39	28	20	9	65	82	88
Exchange	34	26	44	39	77	28	20	9	65	82	88
Exchange	34	26	44	39	28	77	20	9	65	82	88
Exchange	34	26	44	39	28	20	77	9	65	82	88
Exchange	34	26	44	39	28	20	9	77	65	82	88
End-Exchange	34	26	44	39	28	20	9	65	77	82	88
4th round- exchange	26	34	44	39	28	20	9	65	77	82	88
Not exchange	26	34	44	39	28	20	9	65	77	82	88
Exchange	26	34	39	44	28	20	9	65	77	82	88
Exchange	26	34	39	28	44	20	9	65	77	82	88
Exchange	26	34	39	28	20	44	9	65	77	82	88
Exchange	26	34	39	28	20	9	44	65	77	82	88
End- Not exchange	26	34	39	28	20	9	44	65	77	82	88

Bubble Sort

Hash value	0	1	2	3	4	5	6	7	8	9	10
5th round- not change	26	34	39	28	20	9	44	65	77	82	88
Not exchange	26	34	39	28	20	9	44	65	77	82	88
Exchange	26	34	28	39	20	9	44	65	77	82	88
Exchange	26	34	28	20	39	9	44	65	77	82	88
Exchange	26	34	28	20	9	39	44	65	77	82	88
End-Not exchange	26	34	28	20	9	39	44	65	77	82	88
6th round- not change	26	34	28	20	9	39	44	65	77	82	88
Exchange	26	28	34	20	9	39	44	65	77	82	88
Exchange	26	28	20	34	9	39	44	65	77	82	88
Exchange	26	28	20	9	34	39	44	65	77	82	88
End-Not change	26	28	20	9	34	39	44	65	77	82	88

Bubble Sort

Hash value	0	1	2	3	4	5	6	7	8	9	10
7th round Not exchange	26	28	20	9	34	39	44	65	77	82	88
Exchange	26	20	28	9	34	39	44	65	77	82	88
Exchange	26	20	9	28	34	39	44	65	77	82	88
End- Not change	26	20	9	28	34	39	44	65	77	82	88
8th round-Exchange	20	26	9	28	34	39	44	65	77	82	88
Exchange	20	9	26	28	34	39	44	65	77	82	88
End-Not exchange	20	9	26	28	34	39	44	65	77	82	88
9th round- not change	9	20	26	28	34	39	44	65	77	82	88
End- Not exchange	9	20	26	28	34	39	44	65	77	82	88
Last round not change	9	20	26	28	34	39	44	65	77	82	88

Code Example of Bubble Sort

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Analysis of Bubble Sort Algorithm

- Regardless of how the items are arranged in the initial list.
- input size: n
  basic operation: comparison
  no.of comarision c(n): 1 + 2 + 3 + ...+ (n-1)
- Hence, the performance of this algorithm is O(n^2)

Bubble sort

Let's look at another example:

```
round 0 start [90, 100, 10, 20, 30, 40, 50, 60, 70, 80]
              [90, 100, 10, 20, 30, 40, 50, 60, 70, 80]
              [90, 10, 100, 20, 30, 40, 50, 60, 70, 80]
              [90, 10, 20, 100, 30, 40, 50, 60, 70, 80]
              [90, 10, 20, 30, 100, 40, 50, 60, 70, 80]
              [90, 10, 20, 30, 40, 100, 50, 60, 70, 80]
              [90, 10, 20, 30, 40, 50, 100, 60, 70, 80]
              [90, 10, 20, 30, 40, 50, 60, 100, 70, 80]
              [90, 10, 20, 30, 40, 50, 60, 70, 100, 80]
round 0 end   [90, 10, 20, 30, 40, 50, 60, 70, 80, 100]
```

```
round 1 start [90, 10, 20, 30, 40, 50, 60, 70, 80, 100]
              [10, 90, 20, 30, 40, 50, 60, 70, 80, 100]
              [10, 20, 90, 30, 40, 50, 60, 70, 80, 100]
              [10, 20, 30, 90, 40, 50, 60, 70, 80, 100]
              [10, 20, 30, 40, 90, 50, 60, 70, 80, 100]
              [10, 20, 30, 40, 50, 90, 60, 70, 80, 100]
              [10, 20, 30, 40, 50, 60, 90, 70, 80, 100]
              [10, 20, 30, 40, 50, 60, 70, 90, 80, 100]
round 1 end   [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Bubble sort

```
round 2 start [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
              [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
round 2 end   [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Notice that in `round 2`, NOT a single swap is needed. It means that the list has already been sorted. NO NEED TO GO ANY FURTHER TO `round 3`, `round 4`, `round 5`, ...

Bubble sort

Better algorithm:

pseudocode

```
n = length-of(intList)
FOR i from 0 to (n-2)
    swapped = false
    FOR j from 1 to (n-i-1)
        // compare adj items, swap if in wrong order
        IF intList[j-1] > intList[j]:
            swap intList[j-1] and intList[j]
            # remember that swap is needed
            swapped = true
        END IF
    END FOR
    BREAK IF swapped = false
END FOR
```

Bubble sort

Python implementation better algorithm

```
def bubbleSort(intList):
    n = len(intList)
    for i in range(0, n-1):
        swapped = False
        for j in range(1, n-i):
            # compare adj items, swap if in wrong order
            if intList[j-1] > intList[j]:
                # swap intList[j-1] and intList[j]
                temp = intList[j-1]
                intList[j-1] = intList[j]
                intList[j] = temp
                # remember that swap is needed
                swapped = True
        if not swapped:
            # swap is NOT needed, so list is SORTED
            break
```

The best case: only first round required, $c(n) = n-1$

$O(n)$

The worst case: same as the previous one, $O(n^2)$

U O W

Selection Sort

CSIT-881 Python and Data Structures

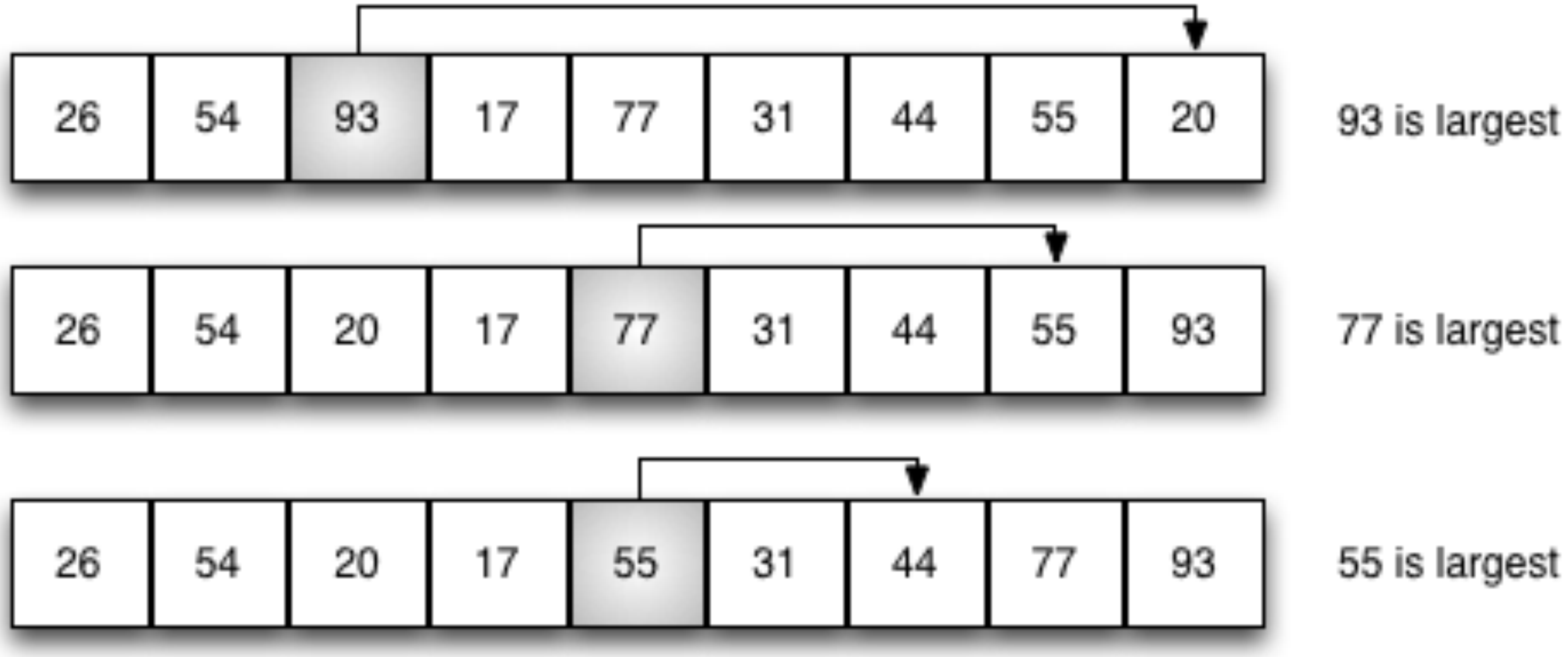
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Selection Sort

- The selection sort is making only one exchange for every pass through the list.
- The selection sort looks for the largest value as it makes a pass and then places it in the proper location.

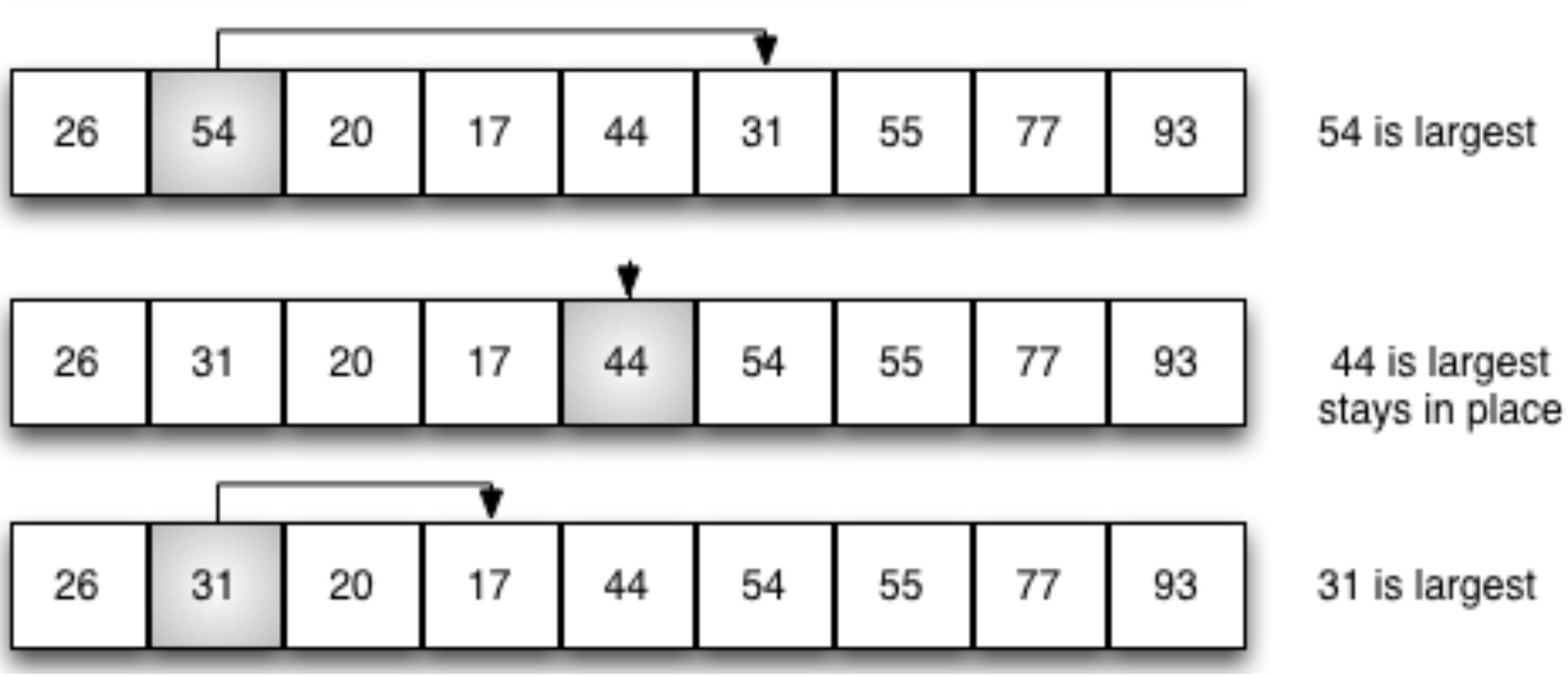
- For the bubble sort, the exchange is take place when it needs.
- After the first pass, the largest item is in the correct place.
- After the second pass, the next largest is in place.
- This process continues and requires n−1 passes to sort n items, since the final item must be in place after the (n−1) th pass.

Selection Sort



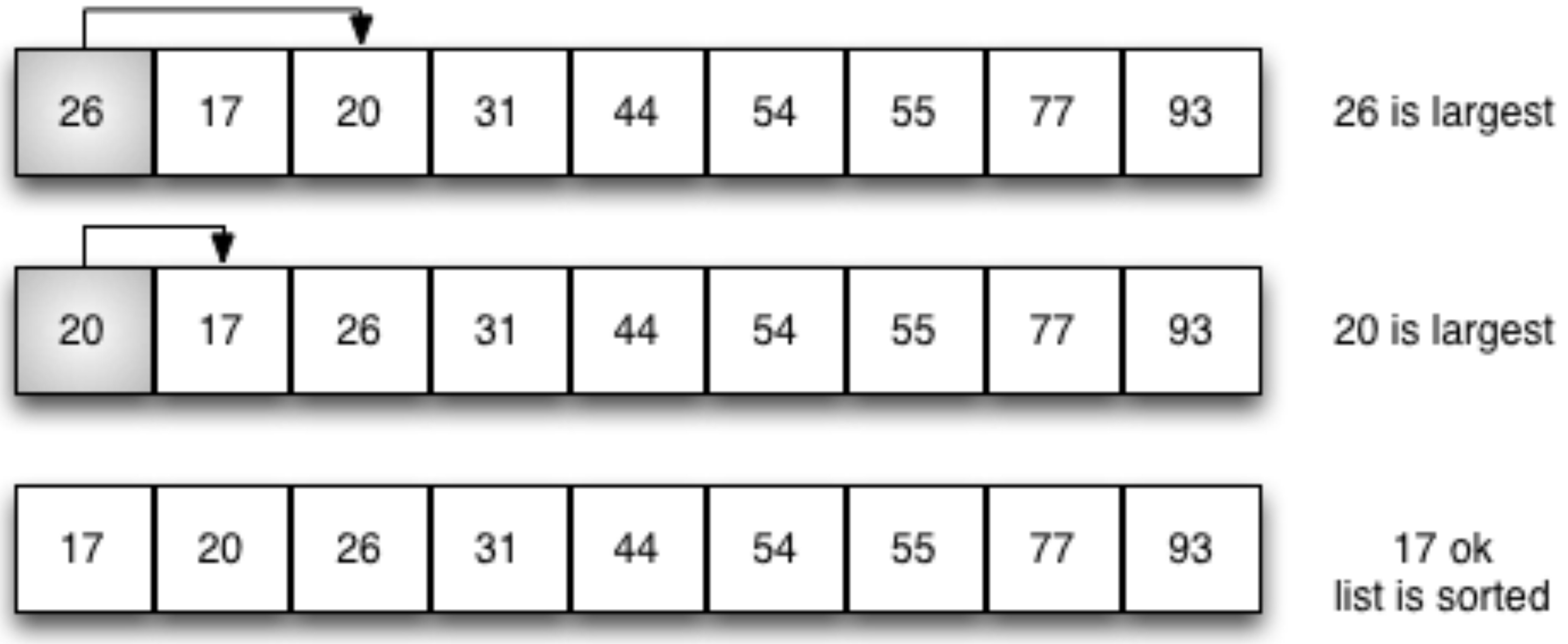
Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Selection Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Selection Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Code Example of Selection Sort

```
def selectionSort(alist):
   for fillslot in range(len(alist)-1,0,-1):
       positionOfMax=0
       for location in range(1,fillslot+1):
           if alist[location]>alist[positionOfMax]:
               positionOfMax = location

       temp = alist[fillslot]
       alist[fillslot] = alist[positionOfMax]
       alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

Analysis of Algorithm

- In fact, the computer performance of the selection sort algorithm in theory is O(n^2).
- no. of comparison operation c(n): 1 + 2 + 3 + ...+ (n-1)
- However, wrt. no.of swap operation

selection sort < bubble sort

U O W

Insertion Sort

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Insertion Sort

- The complexity of insertion sort is O(n^2) similar to the Selection sort and bubble sort.

- However, it works in a slightly different way than those two.

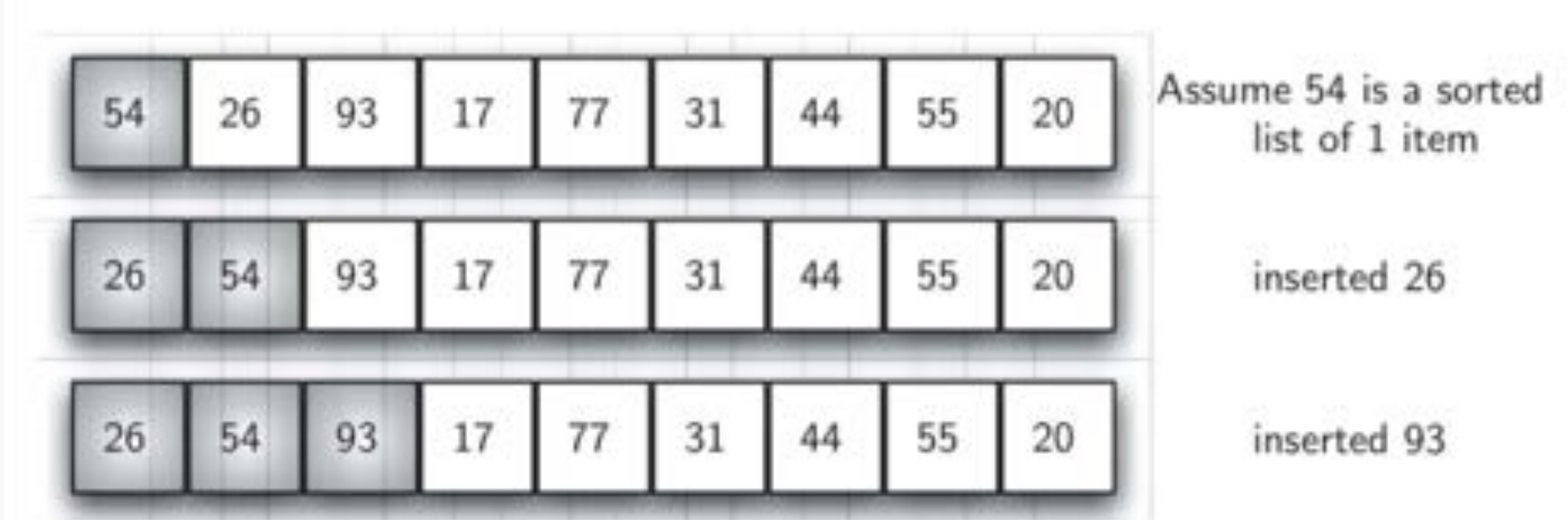
- It always maintains a sorted sub-list in the lower positions of the list.

- Each new item is then “inserted” back into the previous sub-list such that the sorted sub-list is one item larger.

- The figures in the following will show the insertion sorting process.

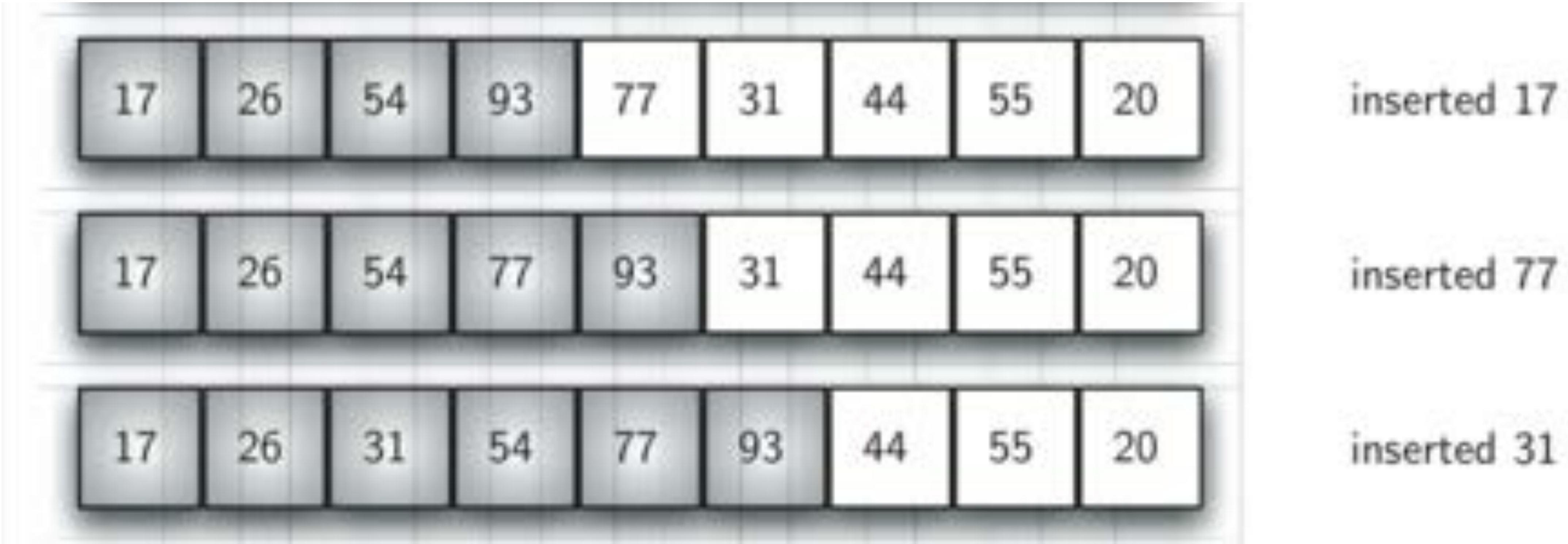
- The shaded items represent the ordered sub-lists as the algorithm makes each pass.

Insertion Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

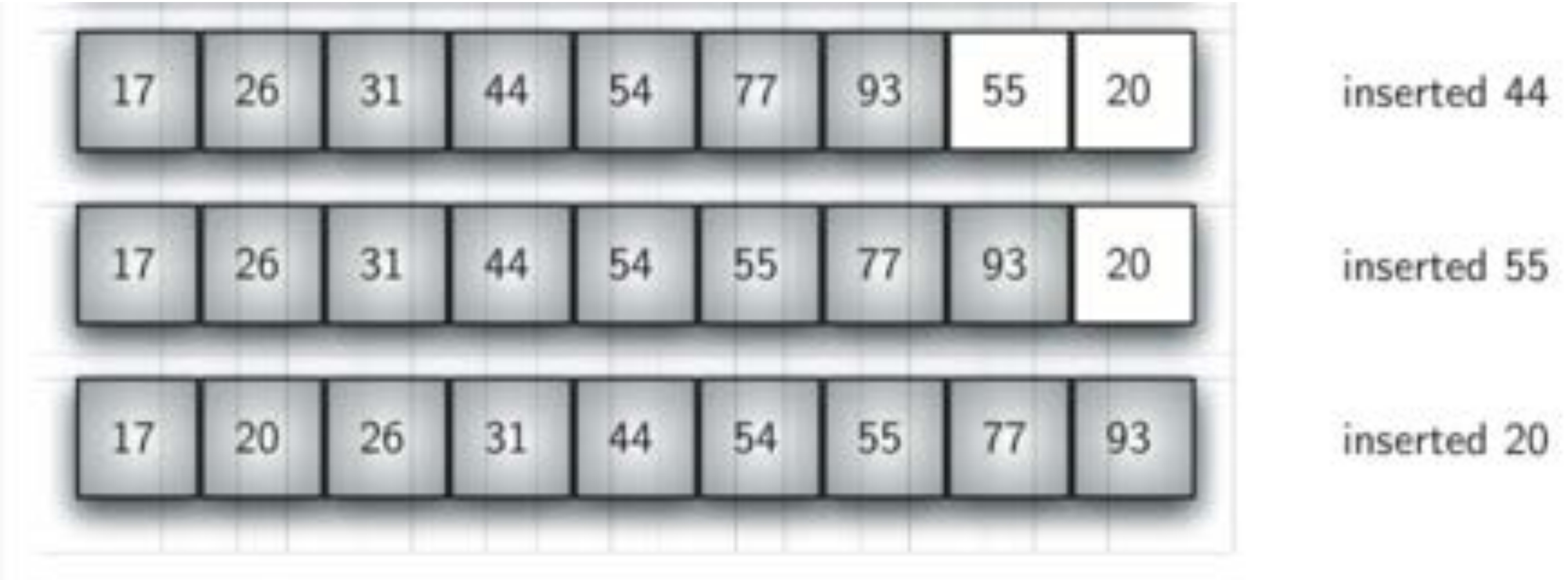
Insertion Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Insertion Sort



Code Example of Insertion Sort

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

Analysis of Algorithm

- **Hence, the performance of this algorithm is**

  the best case: $O(n)$

  the worst case: $O(n^2)$

U O W

Shell Sort

CSIT-881 Python and Data Structures

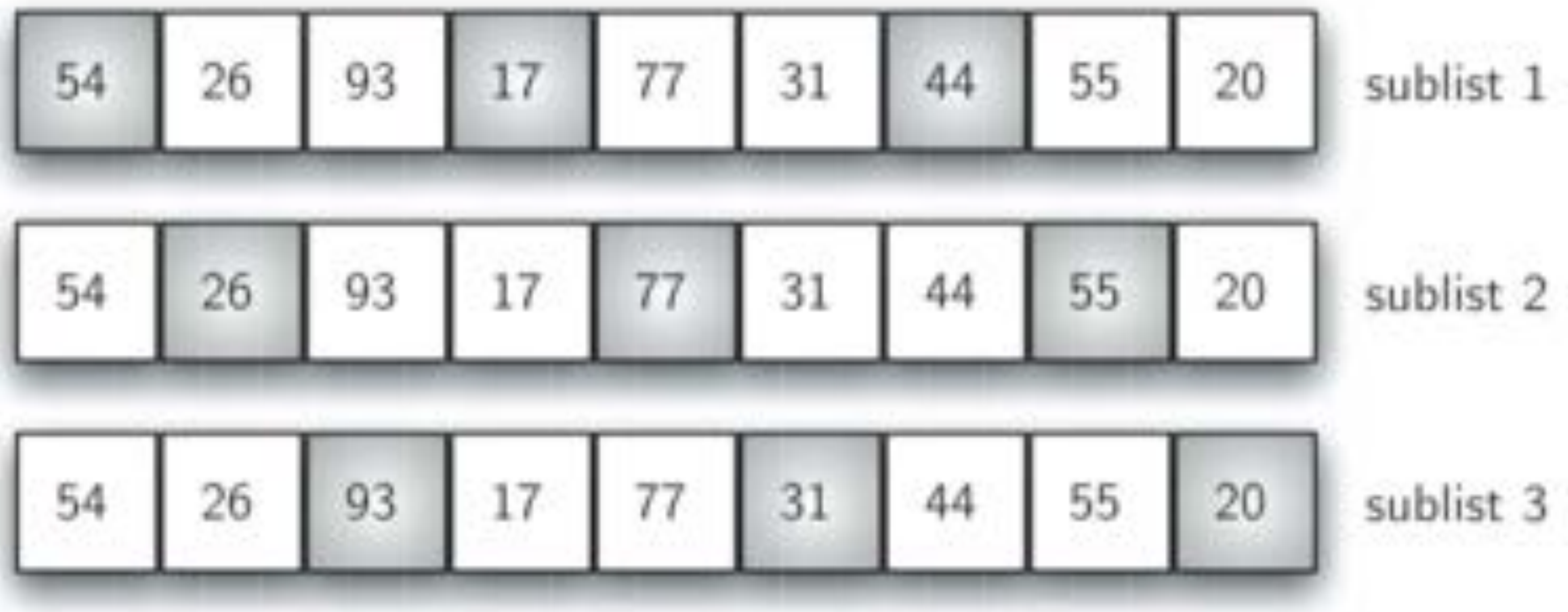
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Shell Sort

基于插入排序，将数列划分为几个字序列，对字序列再进行插入排序，基于gap划分，了解gap

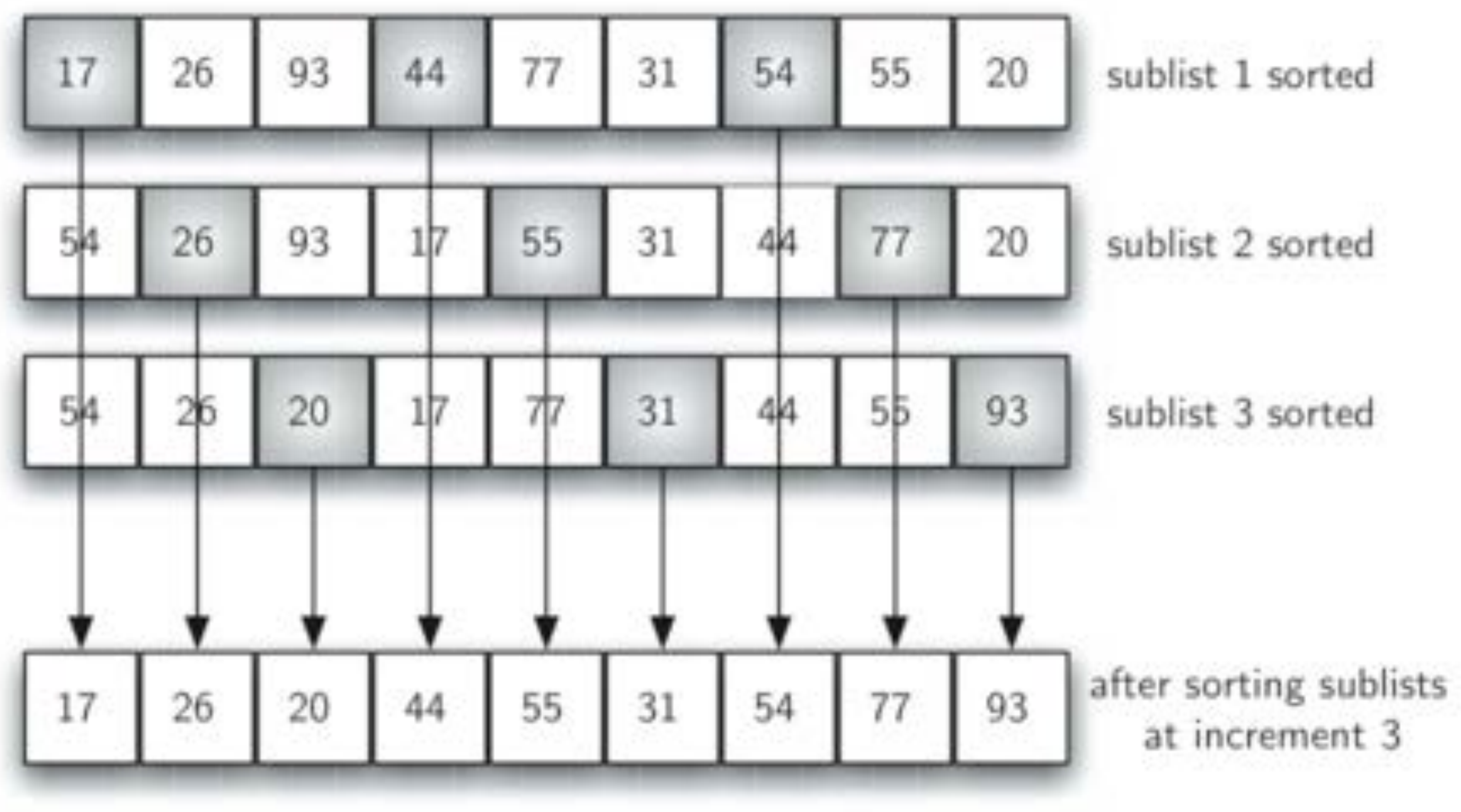
- The shell sort, sometimes called the “diminishing increment sort,”
- It improves on the insertion sort by breaking the original list into a number of smaller sub lists
- each of which is sorted using an insertion sort.
- This unique way of splitting these sub lists is the key to the shell sort.
- The shell sort uses an increment i, sometimes called the gap, to create a sub list by choosing all items that are i items apart.
- In the following slide, An example list has nine items.
- If we use an increment of three, there are three sub lists, each of which can be sorted by an insertion sort.
- Although this list is not completely sorted, something very interesting has happened.
- By sorting the sub lists, we have moved the items closer to where they actually belong.

Shell Sort



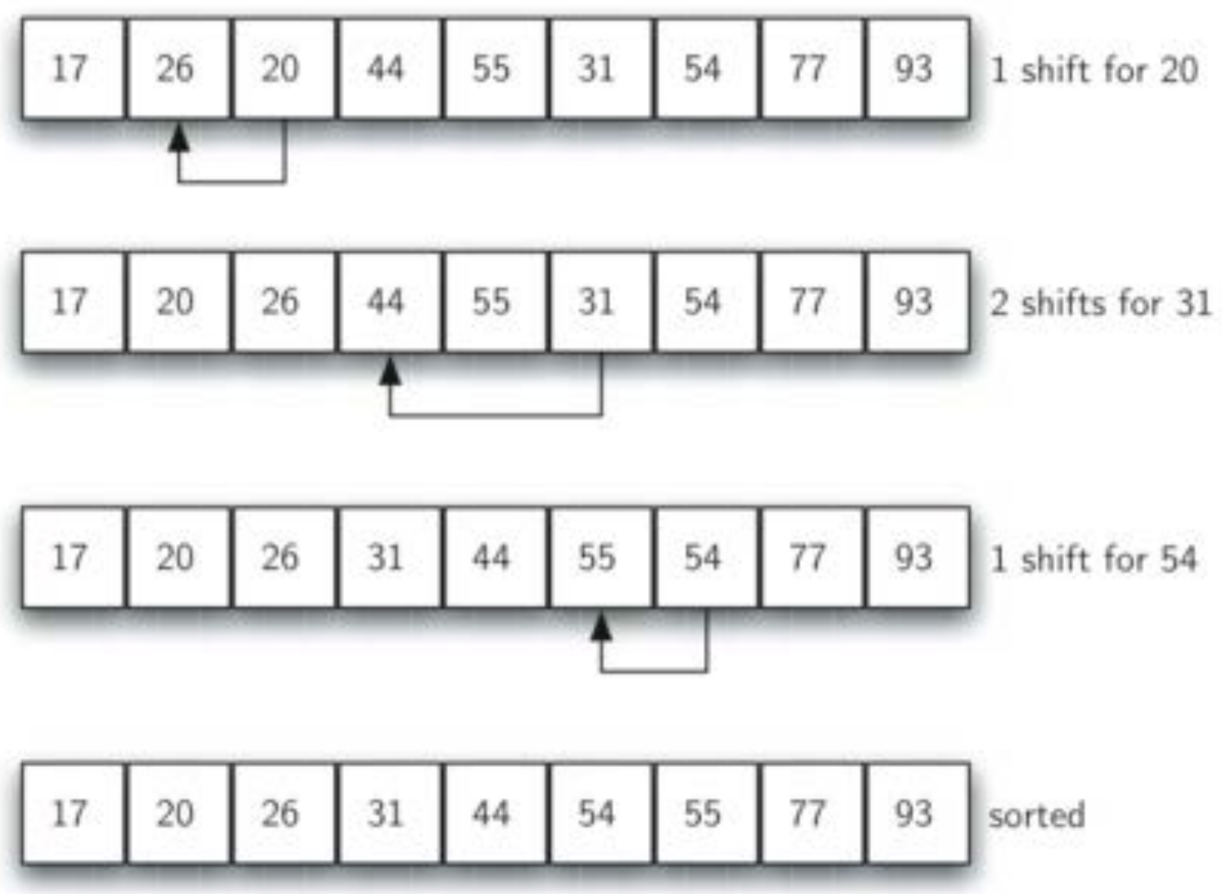
Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Shell Sort



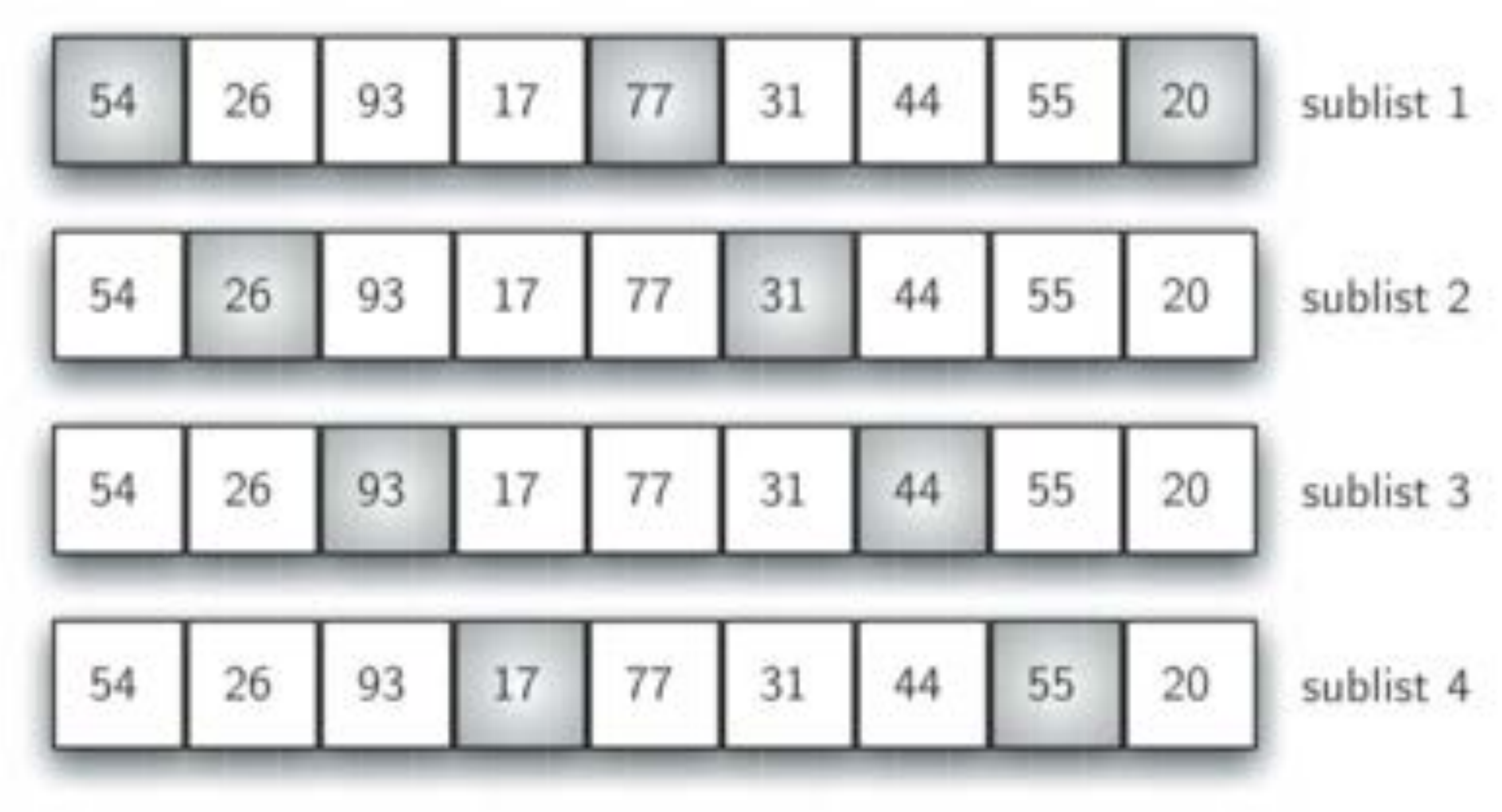
Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Shell Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Shell Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Code Example of Shell Sort

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                        "The list is",alist)

      sublistcount = sublistcount // 2
```

Code Example of Shell Sort

```
def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

Analysis of Algorithm

- The performance of this algorithm is

$$O(n^{1.3\sim2})$$

U O W

Merge Sort

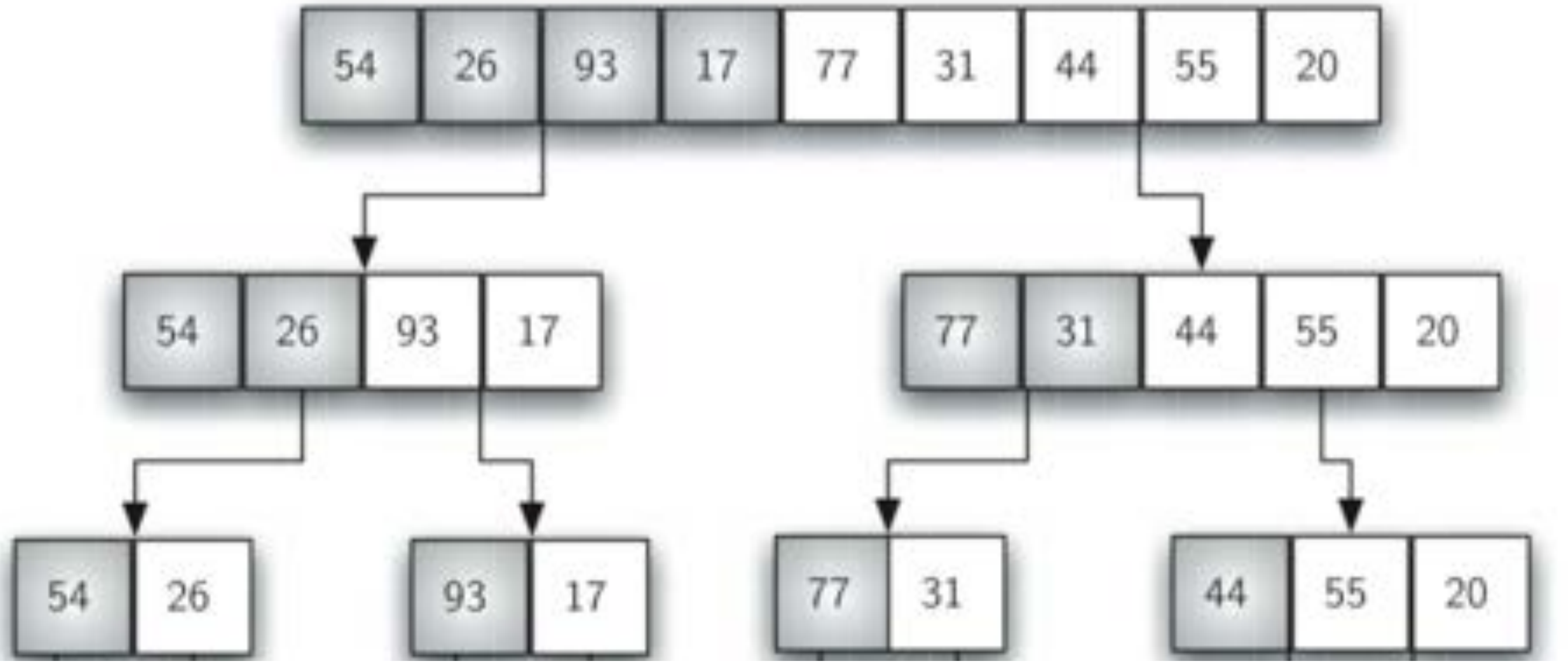
CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Merge Sort

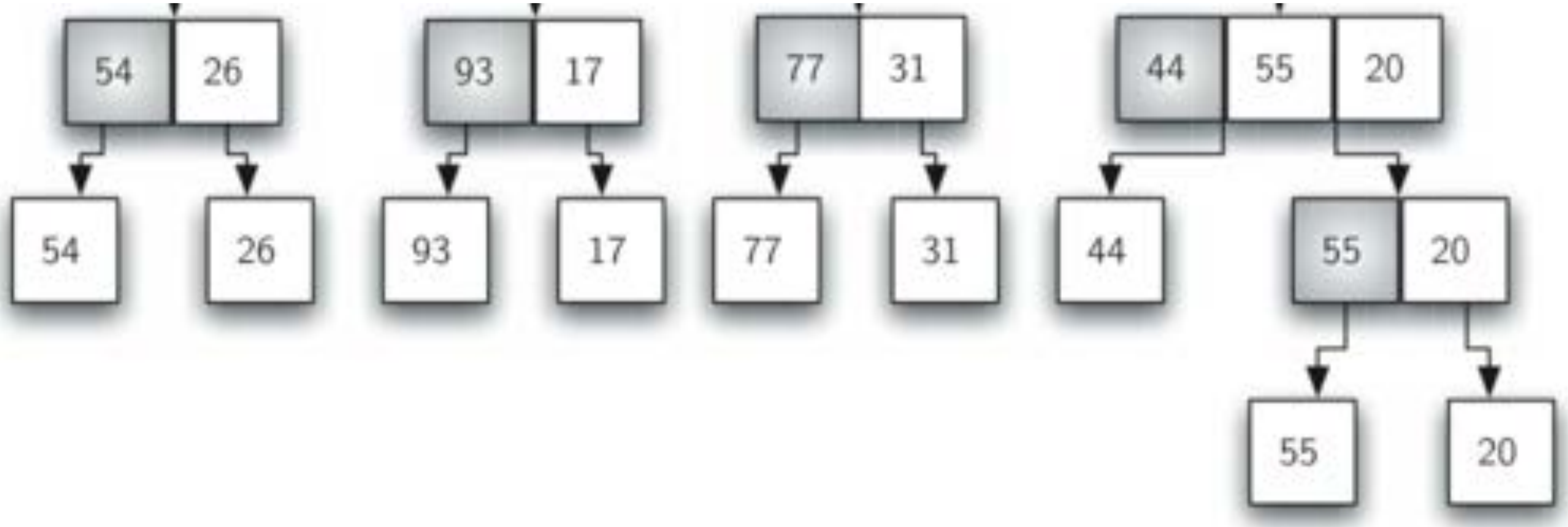
- We now turn our attention to using a divide and conquer strategy as a way to improve the performance of sorting algorithms.
- The first algorithm we will study is the merge sort.
- Merge sort is a recursive algorithm that continually splits a list in half.
- If the list is empty or has one item, it is sorted by definition (the base case).
- If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.
- Once the two halves are sorted, the fundamental operation, called a merge, is performed.
- Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

Merge Sort



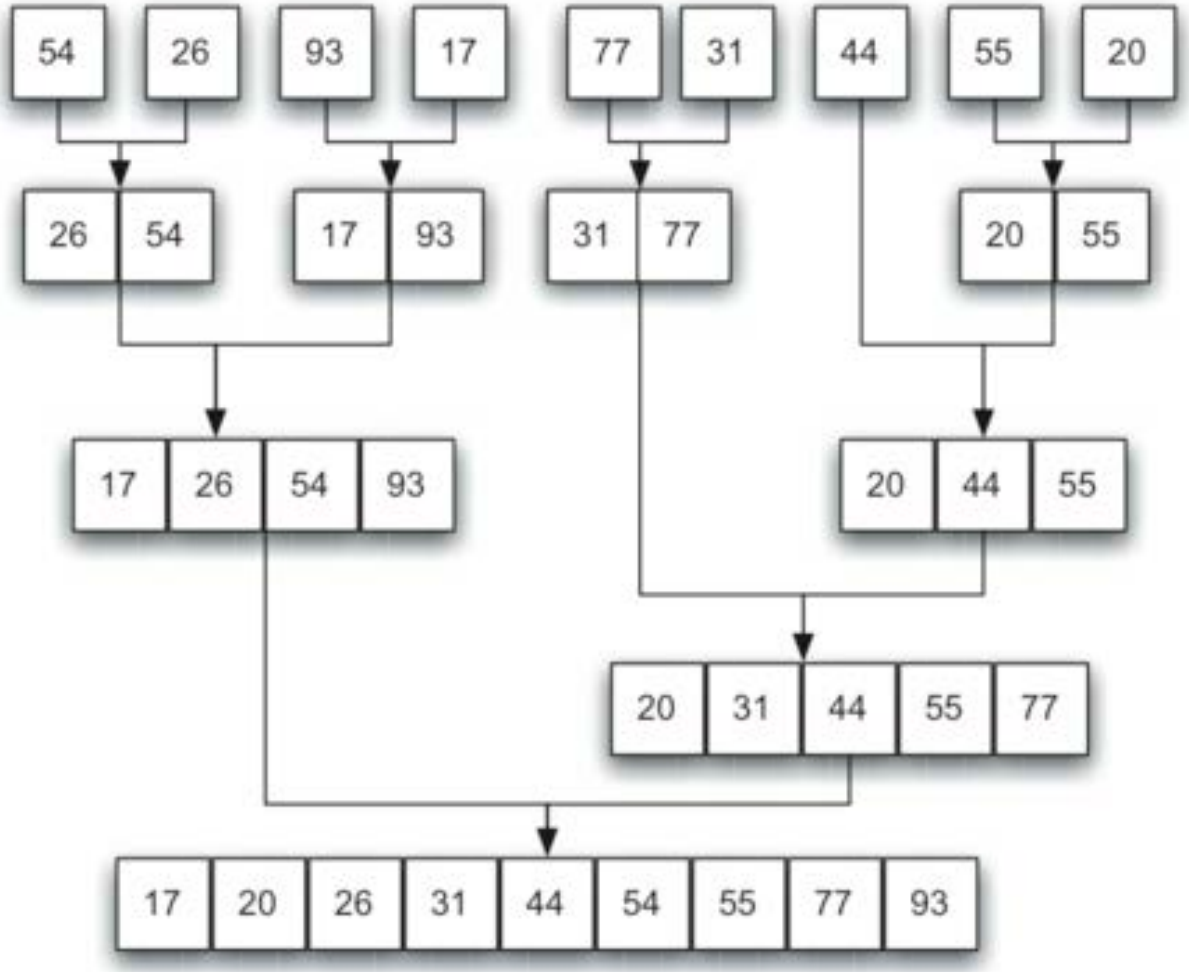
Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Merge Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Merge Sort



Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Recursion: the design of merge sort

- Recursion is a method of solving problems that involves breaking a problem down into smaller problems
- Recursive function is a function calling itself until it reaches some curtain conditions.
- Base case or termination case: is where the small problem can be easily solve. This is when the program end and start to return the result. For example, when splitting until left only one item per array.
- Recursive case: is where the problem is still not able to solve and need to continue breaking it down. For example, when splitting the array or merging the arrays.

Code Example of Merge Sort

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
```

Code Example of Merge Sort

```
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] <= righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Analysis of Algorithm

- Time complexity of Merge Sort is O(n log n) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- The space complexity of merge sort is O(n).
- Hence, the performance of this algorithm is O($n \log n$)

Quick Sort

CSIT-881 Python and Data Structures

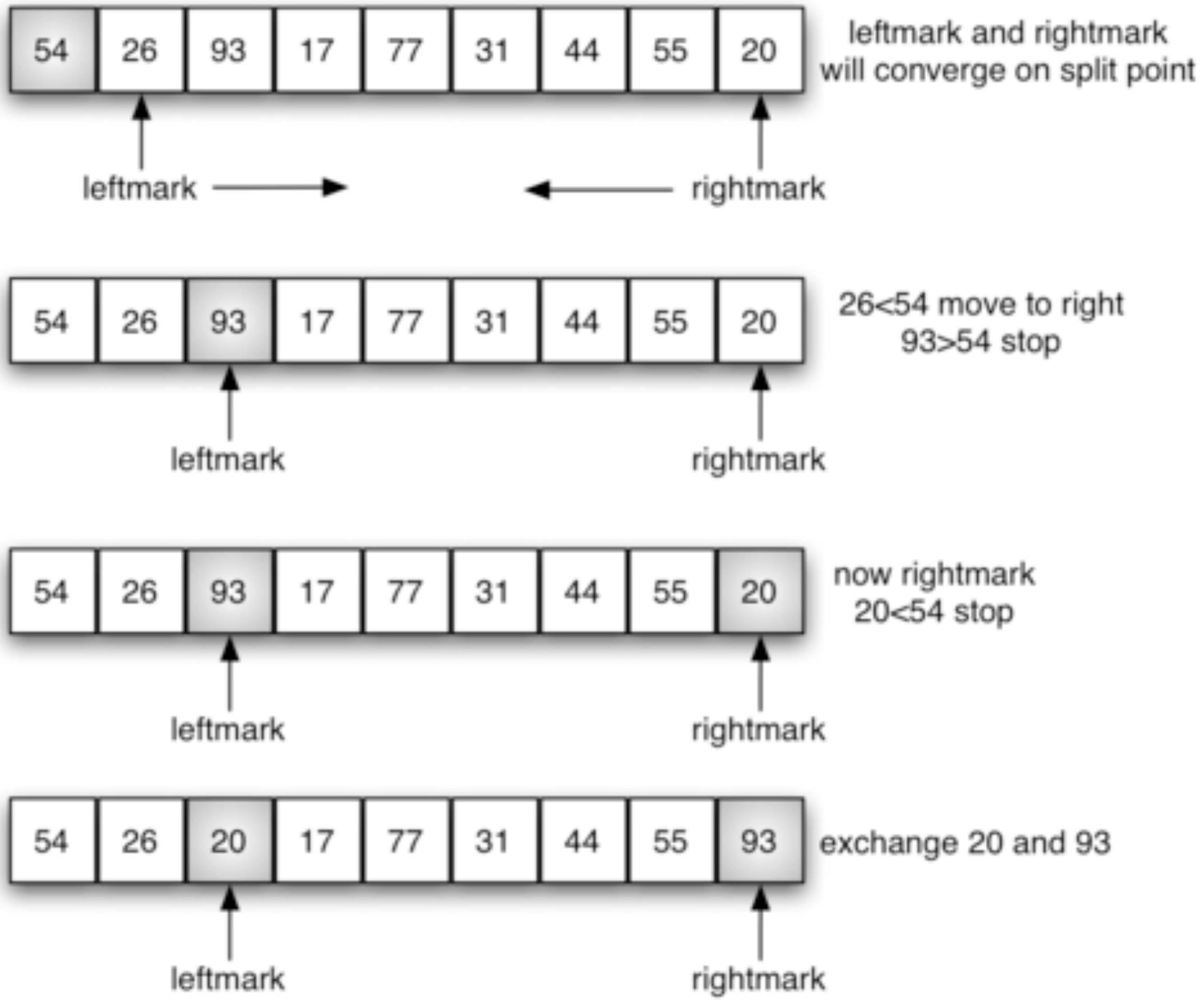
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Quick Sort

- The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.
- As a trade-off, however, it is possible that the list may not be divided in half.
- When this happens, we will see that performance is diminished.

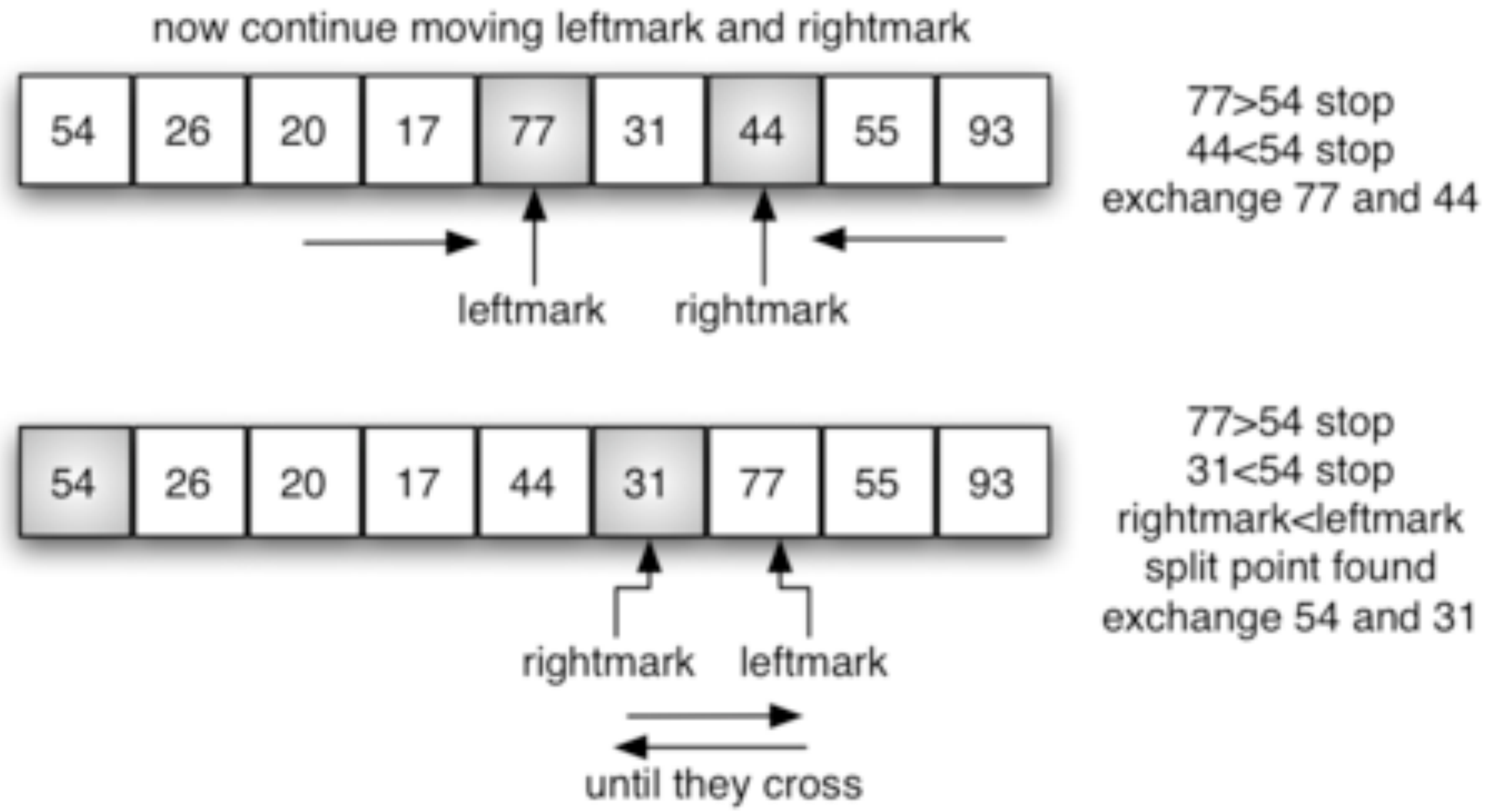
- A quick sort first selects a value, which is called the **pivot value.**
- Although there are many different ways to choose the pivot value, we will simply use the first item in the list.
- The role of the pivot value is to assist with splitting the list.
- The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Quick Sort

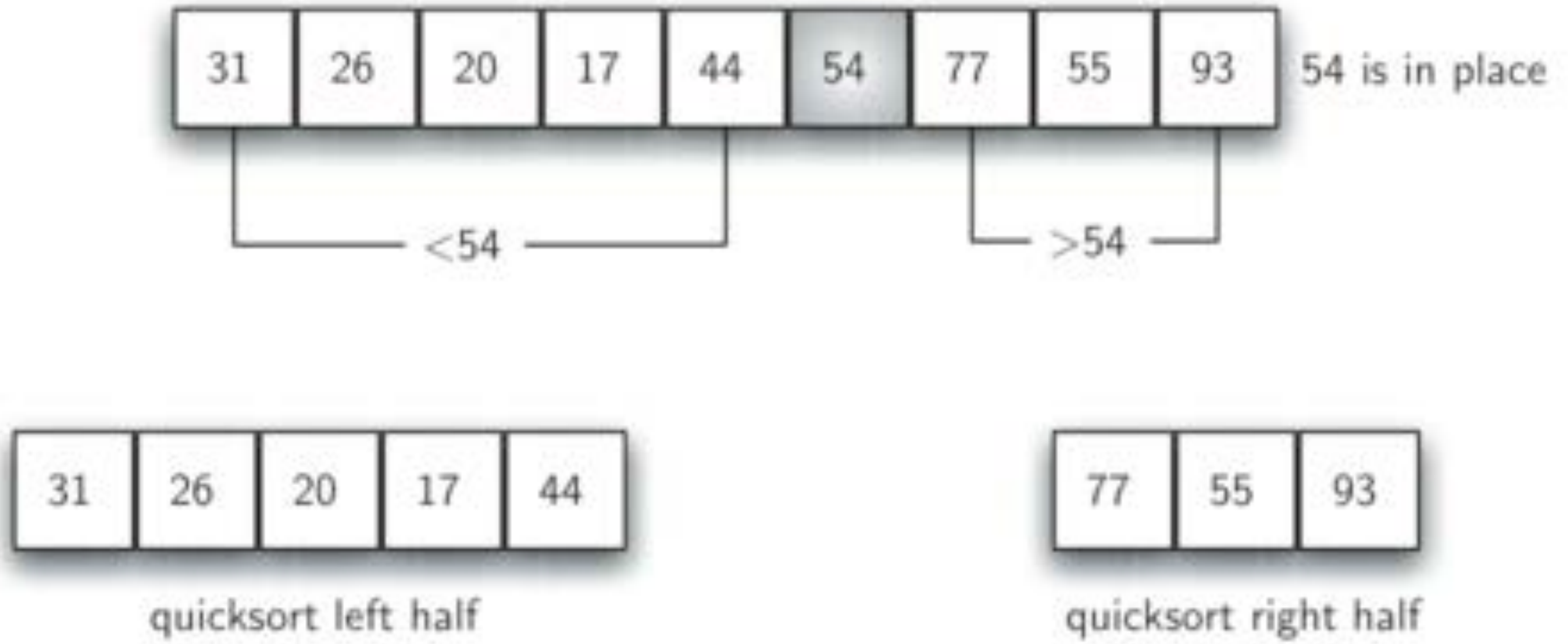


Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

Quick Sort


now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
leftmark
rightmark
77>54 stop
44<54 stop
exchange 77 and 44
54 26 20 17 44 31 77 55 93
rightmark
leftmark
until they cross
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31

Quick Sort



31
26
20
17
44
54
77
55
93
54 is in place
<54
>54
31
26
20
17
44
77
55
93
quicksort left half
quicksort right half

Code Example of Quick Sort

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

       splitpoint = partition(alist,first,last)

       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)
```

Code Example of Quick Sort

```
def partition(alist,first,last):
•       pivotvalue = alist[first]
•
        leftmark = first+1
•       rightmark = last
•
        done = False
•       while not done:
•
            while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
•               leftmark = leftmark + 1
•
            while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
•               rightmark = rightmark -1
•
            if rightmark < leftmark:
•               done = True
•           else:
•               temp = alist[leftmark]
•               alist[leftmark] = alist[rightmark]
•               alist[rightmark] = temp
```

Code Example of Quick Sort

```
    # after the first loop end
    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp


    return rightmark

alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

Analysis of Algorithm

- Best case and average case of this algorithm is n log n.
- However, the performance of this algorithm in the worse case is O(n^2)

U O W

The End

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Important Recurrence Types

Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size *n* and a smaller size *n* – 1.
Example: n!
The recurrence equation for investigating the time efficiency of such algorithms typically has the form
$C(n) = C(n\text{-}1) + f(n)$

Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size *n* into several smaller instances of size *n/b*, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.
Example: binary search.
The recurrence equation for investigating the time efficiency of such algorithms typically has the form
$C(n) = aC(n/b) + f\,(n)$

Decrease-by-one Recurrences

One (constant) operation reduces problem size by one.

$C(n) = C(n\text{-}1) + c \qquad C(1) = d$

Solution:

$C(n) = (n\text{-}1)c + d$ <u>*linear*</u>

A pass through input reduces problem size by one.

$C(n) = C(n\text{-}1) + cn \qquad C(1) = d$

Solution:

$C(n) = [n(n+1)/2 - 1]\, c + d$ <u>*quadratic*</u>

Decrease-by-a-constant-factor recurrences – The Master Theorem

$C(n) = aC(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k>=0$

1. $a < b^k$ $\quad C(n) \in \Theta(n^k)$
2. $a = b^k$ $\quad C(n) \in \Theta(n^k \log n\ )$
3. $a > b^k$ $\quad C(n) \in \Theta(n^{\log_b a})$

Examples:

$C(n) = C(n/2) + 1$ $\quad \Theta(\log n)$

$C(n) = 2C(n/2) + n$ $\quad \Theta(n \log n)$

$C(n) = 3C(n/2) + n$ $\quad \Theta(n^{\log_2 3})$