Indian Institute of Information Technology Design and Manufacturing,
Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe:
S.Dhanalakshmi

Sorting Algorithms

Objective: This module focuses on design and analysis of various sorting algorithms using paradigms such as Incremental Design and Divide and Conquer.

Sorting a list of items is an arrangement of items in ascending (descending) order. We shall discuss six different sorting algorithms and we begin our discussion with Bubble sort.

1 Bubble Sort

Bubble sort is a comparison based sorting algorithm wherein comparing adjacent elements is a primitive operation. In each pass, it compares the adjacent elements in the array and exchanges those that are not in order. Basically, each pass through the array places the next largest value in its proper place, hence the number of comparisons reduces by one at each pass. In other words, each item "bubbles"up to the location where it is supposed to be in the sorted sequence. This invariant is maintained by the algorithm in each pass and hence, bubble sort correctly outputs the sorted sequence. For an n-element array, the below pseudo code requires $n - i$ comparisons for the i^{th} iteration (Pass).

Origin: Initially, Bubble sort was referred to as "Sorting by exchange" in [1, 2] and further, it is referred to as "Exchange Sorting" in [3, 4]. The term "Bubble Sort " was first used by Iverson in 1962 [5].

Invariant: At the end of i^{th} iteration, the last i elements contain i largest elements. i.e. $a[n]$ contains the largest, $a[n-1]$ contains the second largest, and so on. At the end of n^{th} iteration, the array is sorted as it contains n largest elements.

Pseudo code: **Bubble Sort(Array** $a[\ ]$**)**

```
begin
  for i = 1 to n − 1
    for j = 1 to n − i
      if (a[j] > a[j + 1]) then
        Swap (a[j], a[j + 1])
end
```

Run-time Analysis

We shall analyze the run-time by considering the best case input and the worst case input. Interestingly, for bubble sort, irrespective of the nature of input, the number of passes to be made is $n - 1$. Further, the number of comparisons during i^{th} pass is $n - i$. By the end of every pass, at least one element is placed in its right position. In case of best case input, there is no swapping done and for every other input swapping may be required during each pass. Since the underlying model focuses on the number of comparisons (not on the number of swaps as it is less dominant operation), the number of comparisons is $n - 1 + n - 2 + \ldots + 2 + 1 = O(n^2)$ for all inputs.

Trace of Bubble Sort Algorithm

Input: $a[9] = \{54, 26, 93, 17, 77, 31, 44, 55, 20\}$

Pass 1:

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in its right position

Pass 2:

26	54	17	77	31	44	55	20	93	No Exchange
26	54	17	77	31	44	55	20	93	Exchange
26	17	54	77	31	44	55	20	93	No Exchange
26	17	54	77	31	44	55	20	93	Exchange
26	54	17	31	77	44	55	20	93	Exchange
26	54	17	31	44	77	55	20	93	Exchange
26	54	17	31	44	55	77	20	93	Exchange
26	54	17	31	44	55	20	77	93	No Exchange
26	54	17	31	44	55	20	77	93	77 in its right position

Pass 3:

26	54	17	31	44	55	20	77	93	No Exchange
26	54	17	31	44	55	20	77	93	Exchange
26	17	54	31	44	55	20	77	93	Exchange
26	17	31	54	44	55	20	77	93	Exchange
26	17	31	44	54	55	20	77	93	No Exchange
26	17	31	44	54	55	20	77	93	Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	55 in its right position

Pass 4:

26	17	31	44	54	20	55	77	93	Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	54 in its right position

Pass 5:

17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	44 in its right position

Pass 6:

17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	31 in its right position

Pass 7:

17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	26 in its right position

Pass 8:

In this pass, no exchange for all 8 comparisons. Since, the input array size is 9, the number of passes is 8. The algorithm terminates and the input array contains the sorted sequence.

2 Insertion Sort

This is a commonly used sorting algorithm with applications from arranging cards in a card game to arranging examination answer sheets based on students' roll number. Insertion sort follows incremental design wherein we construct a sorted sequence of size two, followed by a sorted sequence of size three, and so on. In this sorting, during i^{th} iteration, the first $(i-1)$ elements are sorted and i^{th} card is inserted to the correct place by performing linear search on the first $(i-1)$ elements. This algorithm performs well on smaller inputs and on inputs that are already sorted.
Origin: Insertion sort was mentioned by John Mauchly as early as 1946, in the first published discussion on computer sorting [6].

Pseudo code: **Insertion Sort**(a[ ]) Source: `CLRS`

```
begin
 for j = 2 to n
   key = a[j]
   i = j + 1
   while i > 0 and a[i] > key
         a[i + 1] = a[i]
         i = i − 1
 a[i + 1] =key
end
```

Run-time Analysis

We shall analyze the run time of insertion sort by considering its worst case and best case behavior. In the below table, for each line of the pseudo code, the cost (the number of times the line is executed) incurred in best and worst case inputs is given. The table presents precise estimate using step count analysis. Alternatively, the cost can also be obtained using recurrence relation.
Recurrence relation in worst case: $T(n) = T(n-1) + n - 1$, $T(2) = 1$, solving this using substitution method/recurrence tree method yields $T(n) = O(n^2)$. Recurrence relation in best case: $T(n) = T(n-1)+1$, $T(2) = 1$, and the solution is $T(n) = O(n)$.

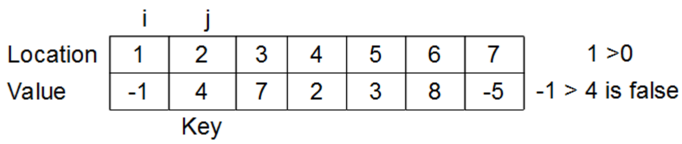
Table 1: Step Count Analysis

Pseudo code	**Worst Case Analysis** I/P: Descending Order	**Best Case Analysis** I/P: Ascending Order
for $j = 2$ to n	n	n
key $= a[j]$	$n-1$	$n-1$
$i = j+1$	$n-1$	$n-1$
while $i > 0$ and $a[i] > key$	$\sum_{j=2}^{n}(j) = \frac{n(n+1)}{2} - 1$	$\sum_{j=2}^{n}(1) = n-1$
$a[i+1] = a[i]$	$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$	0
$i = i-1$	$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$	0
$a[i+1] =$key	$n-1$	$n-1$
Total	$\frac{3n^2}{2} + \frac{7n}{2} - 4$ $= \theta(n^2)$	$5n-4$ $= \theta(n)$

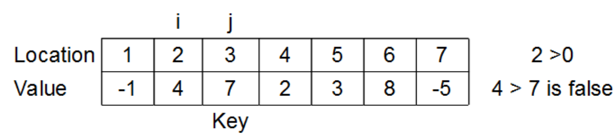
Trace of Insertion Sort Algorithm

As mentioned, at the end of i^{th} iteration, the first i elements are sorted. So, at the end of n^{th} iteration and for this example, at the end of seventh iteration the given seven elements are sorted.
Input: $a[7] = \{-1, 4, 7, 2, 3, 8, -5\}$

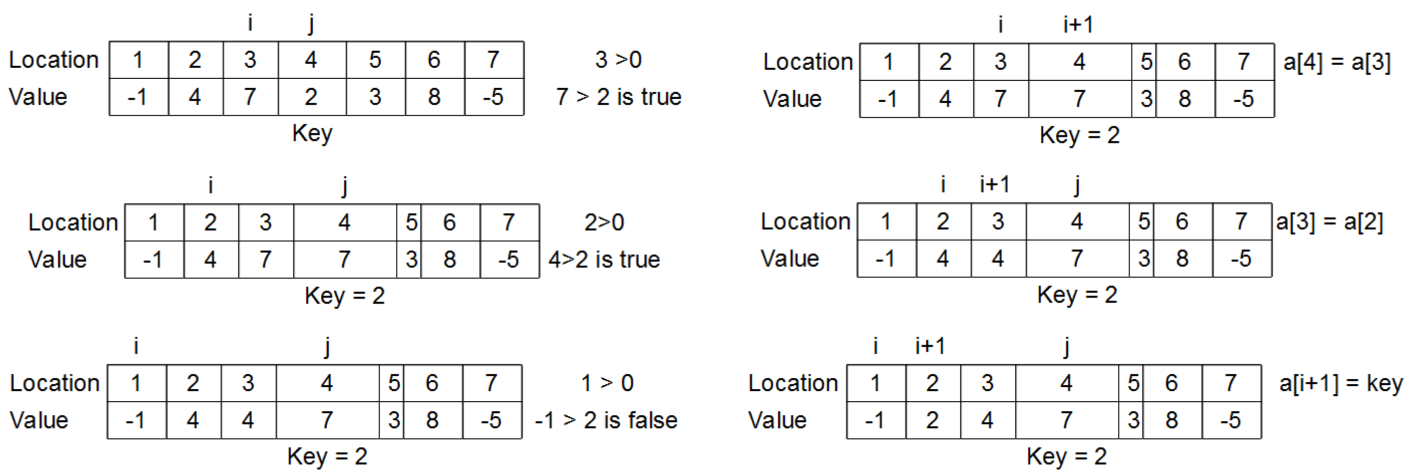
Iteration 1:



Iteration 2:



Iteration 3:



Iteration 4:

				i	j		
Location	1	2	3	4	5	6	7
Value	-1	2	4	7	3	8	-5

Key

4 > 0
7 > 3 is true

				i	j		
Location	1	2	3	4	5	6	7
Value	-1	2	4	7	7	8	-5

Key = 3

a[i+1]=a[i]

			i		j		
Location	1	2	3	4	5	6	7
Value	-1	2	4	7	7	8	-5

Key = 3

3 > 0
4 > 3 is true

			i		j		
Location	1	2	3	4	5	6	7
Value	-1	2	4	4	7	8	-5

Key = 3

a[i+1] = a[i]

		i			j		
Location	1	2	3	4	5	6	7
Value	-1	2	4	4	7	8	-5

Key = 3

2 > 0
2 > 3 is false

		i			j		
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	-5

Key = 3

a[i+1] = 3

Iteration 5:

					i	j	
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	-5

Key

5 > 0
7 > 8 is false

Iteration 6:

						i	j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	-5

Key

6 > 0
8 > -5 is true

						i	j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	8

Key = -5

a[7] = a[6]

					i		j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	8

Key = -5

5 > 0
7 > -5 is true

					i		j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	7	8

Key = -5

a[6] = a[5]

				i			j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	7	8	8

Key = -5

4 > 0
4 > -5 is true

				i			j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	4	7	8

Key = -5

a[5] = a[4]

			i				j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	4	7	8

Key = -5

3 > 0
3 > -5 is true

			i				j
Location	1	2	3	4	5	6	7
Value	-1	2	3	3	4	7	8

Key = -5

a[4] = a[3]

		i					j
Location	1	2	3	4	5	6	7
Value	-1	2	3	4	4	7	8

Key = -5

2 > 0
2 > -5 is true

		i					j
Location	1	2	3	4	5	6	7
Value	-1	2	2	3	4	7	8

Key = -5

a[3] = a[2]

	i						j	
Location	1	2	3	4	5	6	7	1 > 0
Value	-1	2	3	4	4	7	8	-1 > -5 is true
							Key = -5	

	i						j	
Location	1	2	3	4	5	6	7	a[2] = a[1]
Value	-1	-1	2	3	4	7	8	
							Key = -5	

i = 0							j	
Location	1	2	3	4	5	6	7	a[1] = key
Value	-5	-1	2	3	4	7	8	
							Key = -5	

3 Selection Sort

It is a natural sorting algorithm [1] in which we find minimum, second minimum, third minimum and so on and arrange them in increasing order. Like bubble sort, irrespective of the input, during i^{th} stage this algorithm incurs $(n-i)$ comparisons. Further, the algorithm does linear search to find i^{th} minimum.

Pseudocode: **Selection Sort**($a[\,]$)

```
1.begin
2.  for j = 1 to n − 1
3.    min = j
4.    for i = j + 1 to n
5.      if a[i] < a[min]
6.        min = i
7.    Swap(a[j], a[min])
8.end
```

Run-time Analysis

Note Line 5 of Selection Sort is executed for all inputs. During i^{th} iteration, the statement is executed $(n-i)$ times. Therefore, the total cost is $n-1+n-2+\ldots+1$, which is $O(n^2)$. Alternatively, the recurrence relation both in worst case and best case is $T(n) = T(n-1) + n - 1$, $T(2) = 1$. Thus, $T(n) = \theta(n^2)$.

Trace of Selection Sort Algorithm

Input: $a[7] = \{-1, 5, 3, 9, 12, 4, 8, 23, 15\}$

-1	5	3	9	12	4	8	23	15	First Min = -1 , swap(a[1],a[1])
-1	5	3	9	12	4	8	23	15	Second Min = 3, swap(a[3],a[2])
-1	3	5	9	12	4	8	23	15	Third Min = 4, swap(a[6],a[3])
-1	3	4	9	12	5	8	23	15	Fourth Min = 5, swap(a[6],a[4])
-1	3	4	5	12	9	8	23	15	Fifth Min = 8, swap(a[7],a[5])
-1	3	4	5	8	9	12	23	15	Sixth Min = 9, swap(a[6],a[6])
-1	3	4	5	8	9	12	23	15	Seventh Min = 12, swap(a[7],a[7])
-1	3	4	5	8	9	12	23	15	Eighth Min = 15, swap(a[9],a[8])
-1	3	4	5	8	9	12	15	23	**Sorted Array**

4 Merge Sort

Merge Sort is based on the paradigm divide and conquer which has divide and conquer (combine) phases. As part of divide phase which is a top-down approach, the input array is split into half, recursively, until the array size reduces to one. That is, given a problem of size n, break it into two sub problems of size $n/2$, again break each of this sub problems into two sub problems of size $n/4$, and so on till the sub problem size reduces to $n/2^k = 1$ for some integer k. (see *Figure 1(a)*). As part of conquer phase which is a bottom-up approach, we combine two sorted arrays of size one to get a sorted array of size two, and combine two sorted arrays of size two to get a sorted array of size 4, and in general, we combine two sorted arrays of size $n/2$ to get a sorted array of size n. Conquer phase happens when the recursion bottoms out and makes use of a black box which takes two sorted arrays and merges these two to produce one sorted array. In the example illustrated in *Figure 1(b)*, at the last level , the black box combines the sorted array $A = \{1\}$ and the sorted array $B = \{4\}$ and results in a sorted array $\{1, 4\}$. Similarly, it combines the array $\{-1\}$ and $\{8\}$ and results in $\{-1, 8\}$. This process continues till the root which eventually contain the sorted output for the given input.

Origin: Merge sort is one of the very first methods proposed for computer sorting (sorting numbers using computers) and it was suggested by John Von Neumann as early as 1945 [6]. A detailed discussion and analysis of merge sort was appeared in a report by Goldstine and Neumann as early as 1948 [7].

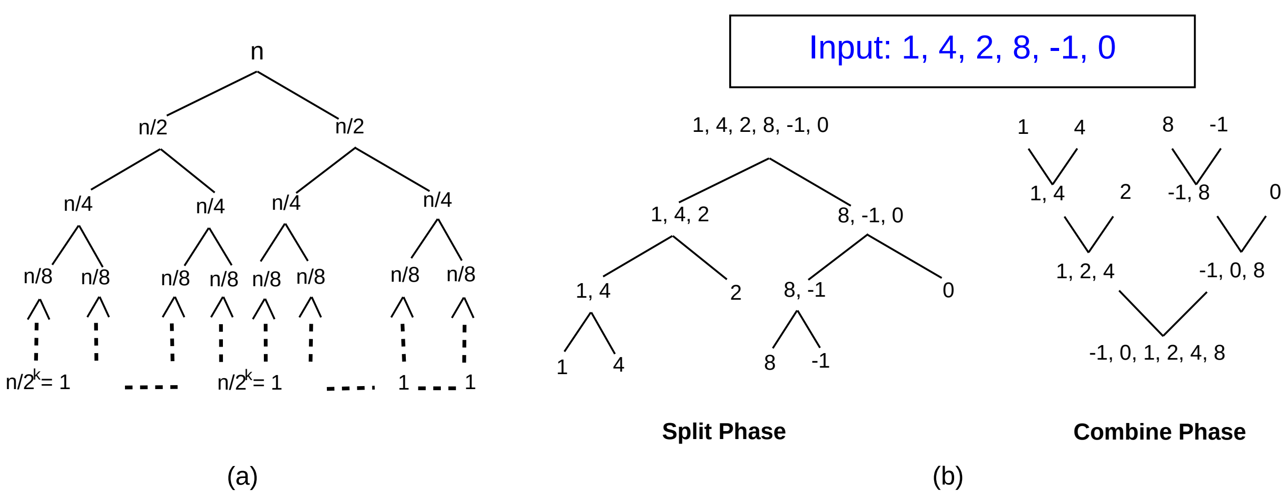


Figure 1: (a) Split phase of merge sort (top-down) (b) Conquer phase of merge sort (bottom-up)

Pseudocode: **Merge-Sort(A, p, r)** Source: CLRS

---

begin
if $p < r$
$q = \lfloor \frac{p+r}{2} \rfloor$
Merge-Sort(A, p, q)
Merge-Sort($A, q + 1, r$)
Merge(A, p, q, r)
end

Merge(A, p, q, r)

---

begin
$n_1 = q - p + 1$
$n_2 = r - q$
Create arrays: $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$
for $i = 1$ to n_1
$L[i] = A[p + i - 1]$
for $j = 1$ to n_2
$R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to r
if $L[i] \leq R[j]$
$A[k] = L[i]$
$i = i + 1$
else $A[k] = R[j]$
$j = j + 1$
end

Trace of Merge Sort Algorithm

We shall now trace the pseudo code using the following example.
The input array : $A = \{1, 4, 2, 8, -1, 0\}$; Start Index : $p = 1$; End Index : $r = 6$.
Initial call: **Merge-Sort**$(A, 1, 6)$
Since, $1 < 6, (p < r)$ is true and it executes all three statements inside `if`.

$q = \lfloor \frac{1+6}{2} \rfloor = 3$

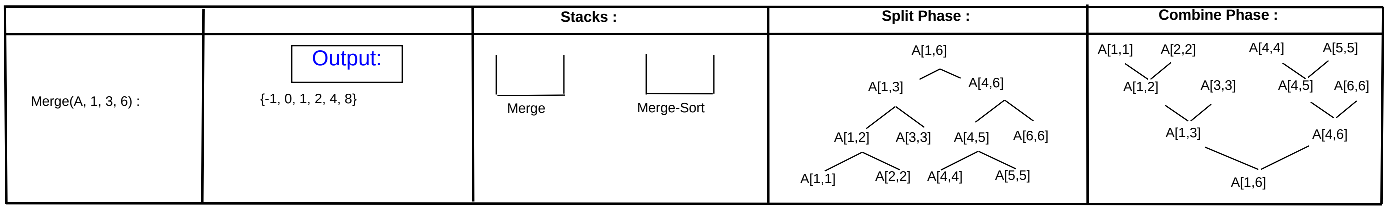
Thus, **Merge-Sort**$(A, 1, 3)$, **Merge-Sort**$(A, 4, 6)$, **Merge**$(A, 1, 3, 6)$ must be executed in order.
Note: Since **Merge-Sort**$(A, 1, 3)$ is a recursive call, when the recursion bottoms out with respect to this recursive call, the two statements following it will be executed. Hence, we need to remember the current value of p, q, r for which compiler makes use of system stack (activation record) where the return address and the current value of p, q, r are stored. Although system maintains just one stack, for clarity purpose, we show how stack is modified with respect to Merge-Sort() and Merge() separately.
When we call `Merge-Sort(A,1,3)`, since it is a recursive call, there will be a record containing `Merge-Sort(A,4,6)`, which will be taken care later, and there will also be record for merge containing `Merge(A,1,3,6)` .

A: 1, 4, 2, 8, -1, 0

		Stacks :	**Split Phase :**	**Combine Phase :**
Merge-Sort(A, 1, 6) :	1 < 6 q = floor((1+6)/2) = 3	Merge: A[1,3] A[1,6] Merge-Sort: A[4,6]	A[1,6] A[1,3] A[4,6]	
Merge-Sort(A, 1, 3) :	1 < 3 q = floor((1+3)/2) = 2	Merge: A[1,2] A[1,3] A[1,6] Merge-Sort: A[3,3] A[4,6]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3]	
Merge-Sort(A, 1, 2) :	1 < 2 q = floor((1+2)/2) = 1	Merge: A[1,1] A[1,2] A[1,3] A[1,6] Merge-Sort: A[2,2] A[3,3] A[4,6]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[1,1] A[2,2]	
Merge-Sort(A, 1, 1) :	1 < 1 is false, Thus the recursion Bottoms out with respect to the recursive call Merge-Sort(A,1,1)	Pop(Merge) = A[1,1] = {1} Pop(Merge-Sort) = A[2,2]		A[1,1]

		Stacks :	Split Phase :	Combine Phase :
Merge-Sort(A,2,2)	2 < 2 is false, Thus the recursion Bottoms out with respect to the recursive call Merge-Sort(A,2,2) Merge = A[2,2] = {4}	A[1,2] A[1,3] A[1,6] Merge A[3,3] A[4,6] Merge-Sort Pop(Merge) = A[1,2]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[1,1] A[2,2]	A[1,1] A[2,2]
Merge(A, 1, 1, 2) :	{1, 4}	A[1,3] A[1,6] Merge A[3,3] A[4,6] Merge-Sort Pop(Merge-Sort) = A[3,3]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[1,1] A[2,2]	A[1,1] A[2,2] A[1,2]
Merge-Sort(A,3,3)	3 < 3 is false, Thus the recursion Bottoms out with respect to the recursive call Merge-Sort(A,3,3) Merge : A[3,3] = {2}	A[1,3] A[1,6] Merge A[4,6] Merge-Sort Pop(Merge) = A[1,3]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[1,1] A[2,2]	A[1,1] A[2,2] A[1,2] A[3,3]
Merge(A, 1, 2, 3) :	{1, 2, 4}	A[1,6] Merge A[4,6] Merge-Sort Pop(Merge-Sort) = A[4,6]	Same as above	A[1,1] A[2,2] A[1,2] A[3,3] A[1,3]

		Stacks :	Split Phase :	Combine Phase :
Merge-Sort(A, 4, 6) :	4 < 6 q = floor((4+6)/2) = 5	A[4,5] A[4,6] A[1,6] Merge A[6,6] Merge-Sort Pop(Merge) = A[4,5]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[4,5] A[6,6] A[1,1] A[2,2]	A[1,1] A[2,2] A[1,2] A[3,3] A[1,3]
Merge-Sort(A, 4, 5) :	4 < 5 q = floor((4+5)/2) = 4	A[4, 4] A[4,5] A[4,6] A[1,6] Merge A[5,5] A[6,6] Merge-Sort Pop(Merge-Sort) = A[5,5]	A[1,6] A[1,3] A[4,6] A[1,2] A[3,3] A[4,5] A[6,6] A[1,1] A[2,2] A[4,4] A[5,5]	A[1,1] A[2,2] A[4,4] A[5,5] A[1,2] A[3,3] A[1,3]
Merge-Sort(A,5,5)	5 < 5 is false, Thus the recursion Bottoms out with respect to the recursive call Merge-Sort(A,5,5) Merge : A[5,5] = {-1}	A[4, 4] A[4,5] A[4,6] A[1,6] Merge A[6,6] Merge-Sort Pop(Merge) = A[4,4] = {8} Pop(Merge) = A[4,5] = {-1, 8}	Same as above	A[1,1] A[2,2] A[4,4] A[5,5] A[1,2] A[3,3] A[4,5] A[1,3]
Pop(Merge-Sort) Merge-Sort(A,6, 6)	6 < 6 is false, Thus the recursion Bottoms out with respect to the recursive call Merge-Sort(A,6,6) Merge : A[6,6] = {0}	A[4,6] A[1,6] Merge Merge-Sort Pop(Merge) = A[4,6] = {-1,0,8}	Same as above	A[1,1] A[2,2] A[4,4] A[5,5] A[1,2] A[3,3] A[4,5] A[6,6] A[1,3] A[4,6]

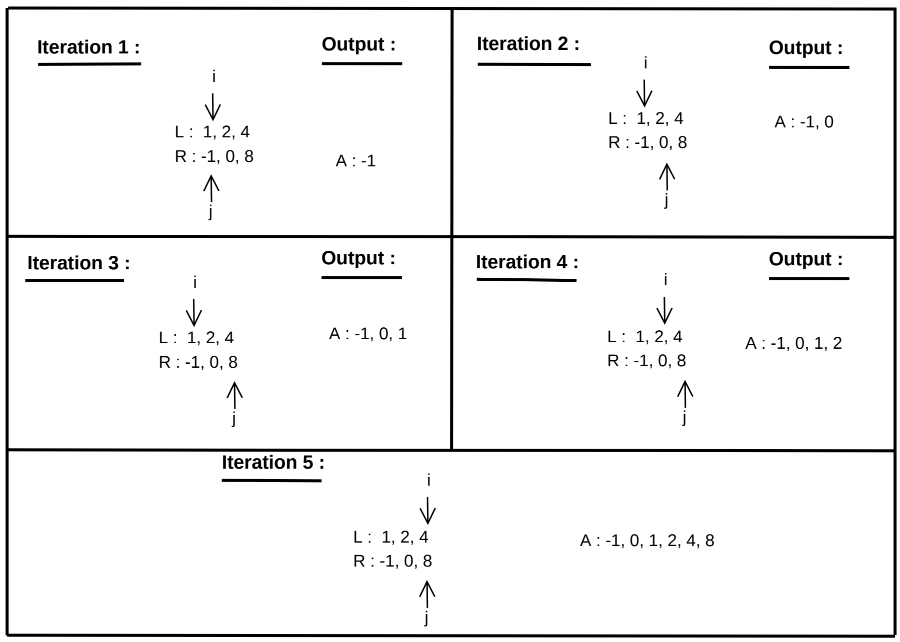


Here is an illustration for Merge:

Merge :

Input: Sorted Array L and sorted Array R such that | L | = m and | R | = s
Output: Sorted array A of size m + s, where A = L + R

Example: Input : L = { 1, 2, 4} and R = {-1, 0, 8}



For output in each iteration, Compare(i,j) and add the least to A

Run-time Analysis

Divide phase takes $O(1)$ time at each stage to create two sub problems. The run time of combine phase depends on the run time of `MERGE()` routine. In worst case, the number of comparisons to merge two sorted arrays of size m and n into one sorted array of size $m+n$ is $m+n-1$, this implies that at each stage merge() takes $O(n)$. Therefore, the total time taken by merge sort is given by the recurrence relation: $T(n) = 2T(n/2) + O(1) + O(n)$, and the solution is $\theta(n \log n)$.

Remark:

1. The sorted arrays $A = \{1, 3, 5, 7, 9\}$ and $B = \{2, 4, 6, 8, 10\}$ of size 5 each incur 9 comparisons by merge() routine to get a sorted sequence $\{1, 2, \ldots, 9\}$. In general, $m+n-1$ comparisons if $|A| = m$ and $|B| = n$.

2. There are $\log n + 1$ levels in the recurrence tree.
3. All leaves (sub problems of size one) are at level $\log n$ and there are $2^{\log n} = n$ leaves.

5 Quick Sort

Quick sort is another sorting algorithm that follows divide and conquer strategy. In this sorting, we pick a special element `pivot` and the given array is partitioned with respect to the pivot element x. i.e., elements that are smaller than x will be in one partition and the elements that are greater than x will be in another partition. This process is done recursively till sub problem size becomes one. The pivot element, in principle can be any element in the array, however, for our discussion we choose to work with the last element of the array.

Origin: The quick sort algorithm was developed in 1959 by Tony Hoare while he was a visiting student at Moscow State University. At that time, Hoare worked on a project on machine translation for National Physical Laboratory. As part of the translation process, he had to sort the words of Russian sentences prior to looking them up in a Russian-English dictionary which was already sorted in alphabetic order and stored in magnetic tape [8]. To fulfill this task he discovered Quick Sort and later published the code in 1961 [9].

Pseudocode: **Quick-Sort**(A, p, r) Source: CLRS

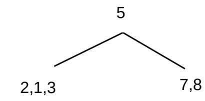
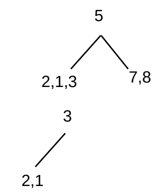
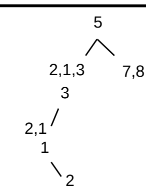
```
begin
  if p < r
    q = Partition(A, p, r)
    Quick-Sort(A, p, q − 1)
    Quick-Sort(A, q + 1, r)
end
```

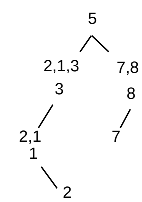
Partition(A, p, r)

```
begin
  x = A[r]
  i = p − 1
  for j = p to r − 1
    if A[j] ≤ x then
      i = i + 1
      Swap(A[i], A[j])
  Swap(A[j + 1], A[r])
  Return i + 1
end
```

Trace of Quick Sort Algorithm

A= { 2, 8, 7, 1, 3, 5} ; p = 1 ; r = 6

		Partition(A,p,r)	Stack :	Tree :
Quick-Sort(A, 1, 6) :	q = Partition(A,1,6) = 4	Partition(A,1,6) : x = A[6] = 5 ; i =0 j = 1 : 2 <= 5 so i = 1 and swap(A[1],A[1]) Thus, A = {2,8,7,1,3,5} j = 2 : 8 <= 5 is false j = 3 : 7 <= 5 is false j = 4 : 1 <= 5 so i = 2 and swap(A[2],A[4]) Thus, A = {2,1,7,8,3,5} j = 5 : 3 <= 5 so i = 3 and swap(A[3],A[5]) Thus, A = {2,1,3,8,7,5} swap(A[4],A[6]). Thus, A = {2,1,3,5,7,8} Return 4	A[5,6] Quick-Sort	
Quick-Sort(A, 1, 3) :	q = Partition(A,1,3) = 3	Partition(A,1,3) : x = A[3] = 3 ; i =0 j = 1 : 2 <= 3 so i = 1 and swap(A[1],A[1]) Thus, A = {2,1,3,5,7,8} j = 2 : 1 <= 3 so i = 2 and swap(A[2],A[2]) Thus, A = {2,1,3,5,7,8} swap(A[3],A[3]). Thus, A = {2,1,3,5,7,8} Return 3	A[5,6] Quick-Sort	
Quick-Sort(A, 1, 2) :	q = Partition(A,1,2) = 1	Partition(A,1,2) : x = A[2] = 1 ; i =0 j = 1 : 2 <= 1 is false swap(A[1],A[2]). Thus, A = {1,2,3,5,7,8} Return 1	Same as above Pop(Quick-Sort): Quick-Sort(A,5,6)	

		Partition(A,p,r)	Stack :	Tree :
Quick-Sort(A, 5, 6) :	q = Partition(A,5,6) = 6	Partition(A,5,6) : x = A[6] = 8 ; i =p-1 = 4 j = 5 : 7 <= 8 so i = 5 and swap(A[5],A[5]) Thus, A = {1,2,3,5,7,8} swap(A[6],A[6]). Thus, A = {1,2,3,5,7,8} Return 6	Quick-Sort	

Invariant

Note that at the beginning of iteration j of the quick sort the following invariant is true: the first i elements are less than or equal to the pivot x (window 1), the elements in indices $i+1,\ldots,j-1$ are greater than x (window 2), and we can not say anything about (whether the value is smaller/larger than the pivot) the elements in the indices $j,\ldots,n$. During iteration j, the value at $A[j]$ is compared with x. If $A[j] > x$ then window-2 increases by one. Otherwise, the first element of window-2 is swapped with x and the window-1 increases by one.

Run-time Analysis

The recursion for quick sort depends the size of recursive sub problem generated at each stage of the recursion. Since the pivot can take any value, the size of a sub problem can take any value in the range $[0..n-1]$. i.e., $T(n) = T(k) + T(n-k-1) + O(n), T(2) = 1$ where k is the size of the window-1 at the end of n iterations. The size of the other recursive problem is $n-k-1$ (total elements minus pivot and window-1).

In **best case**, each subproblem is balanced with equal size or nearly good split; $T(n) = T(n/2) + T(n/2 - 1) + O(n)$ or $T(n) = T(n/2-2) + T(n/2+1) + O(n)$. For unbalanced split the recurrence looks like: $T(n) = T(n/3-1) + T(2n/3) + O(n)$ or $T(n) = T(n/6) + T(5n/6-1) + O(n)$ or in general $T(n) = T(\alpha \cdot n) + T((1-\alpha) \cdot n) + O(n)$, $\alpha < 1$. Note that the $O(n)$ component in $T(n)$ is the cost of partition routine. For all the above recurrence, using recurrence tree, one can show that $T(n) = O(n \log n)$. Therefore, the best case input of quick sort takes $O(n \log n)$ to sort an array of n elements.
In **worst case**, the size of one recursive problem is zero or a very small non-zero constant and the other sub problem size is nearly n. i.e., $T(n) = T(n-1) + O(n)$ or $T(n) = T(n-4) + T(3) + O(n)$ or in general $T(n) = T(n-l) + O(1) + O(n)$ where l is a fixed integer. Clearly, using substitution or recurrence tree method, we get $T(n) = \theta(n^2)$. An example for worst case input is any array in ascending or descending order.

Remark:
1. If input is in increasing order, then it becomes a best case input for insertion sort whereas it becomes the worst case input for quick sort.
2. An example for best case input of quick sort is $\{1, 3, 2, 5, 7, 6, 4\}$, every time the pivot element divides the array into two equal halves. This is true because the pivot at each iteration is the median.
3. One can get a best case input by taking a labelled balanced binary search tree T (nodes are filled with suitable values respecting BST property) and running post-order traversal on T. The resulting post-order sequence is an example input for best case.
4. Any increasing or decreasing order sequence is an example input for worst case.
5. Although, run-time of quick sort is $\theta(n^2)$ in worst case, for an arbitrary input the run-time is $O(n \log n)$. Due to this reason, quick sort is a candidate sorting algorithm in practice.
6. Quick sort can be made to run in $O(n \log n)$ for all inputs (i.e. worst case time: $O(n \log n)$) if median element is chosen as a pivot at each iteration and median can be found in $O(n)$. i.e., $T(n) = 2T(n/2) + O(n) + O(n)$. The first $O(n)$ is for finding median in linear time and the second $O(n)$ is for the pivot subroutine. Further, since median is the pivot, each iteration of quick sort yields two sub problems of equal size.

6 Heap Sort

We shall discuss yet another sorting algorithm using the data structure `Heaps`. A complete binary tree is a binary tree in which each internal node has exactly two children and all leaves are at the same level. A nearly complete binary tree with l levels is a complete binary tree till $(l-1)$ levels and at level l, the nodes are filled from left to right. A nearly complete binary tree is also known as a `Heap`. An array representation of heap fills elements from $A[1]$ with root of the tree at $A[1]$ and its left child at $A[2]$ and the right child at $A[3]$. In general, if the node is at $A[i]$, then its left child is at $A[2i]$ and the right child is at $A[2i+1]$.

We define two special heaps, namely, Max-heap and Min-heap. Max-heap is a heap with the property that for any node, the value at the node is at least the value of its children and for min-heap the value of a node is less than or equal to its children. We shall now describe an approach that will construct max-heap from a given array of n-elements.

Top-down approach: In this approach, given an array of n-elements, we construct max-heap itera-

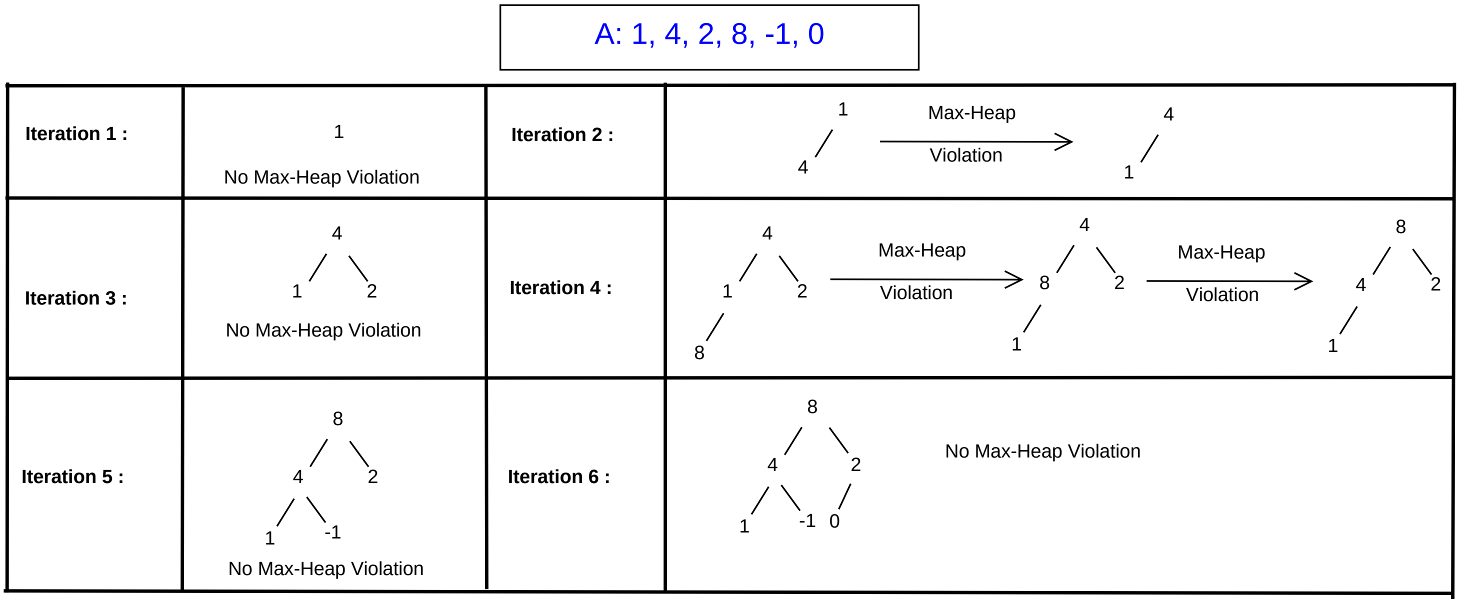


Figure 2: Top Down Approach: Construction of Max-Heap

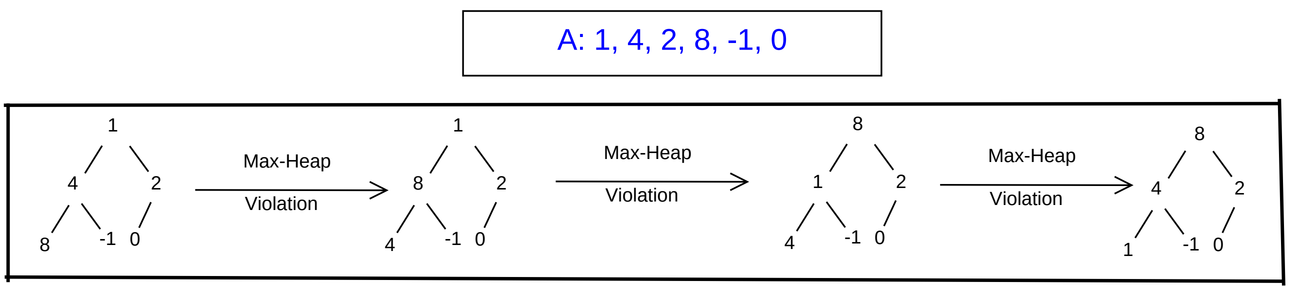


Figure 3: Bottom up Approach: Construction of Max-Heap

tively. Clearly $A[1]$ is a max-heap. We add $A[2]$ to already constructed max-heap and set right the heap if there is a max-heap violation. In general, during i^{th} iteration, $A[i]$ is added to the max-heap already constructed on $A[1..i-1]$. While adding $A[i]$ to the max-heap, we check whether the max-heap property is violated or not. If it violates, then $A[i]$ is swapped with its parent and if any further violation due to swapping, then its parent is swapped with its grand parent and so on. i.e., the new element inserted moves up the tree till it finds a suitable position and in the worst case it becomes the root. An illustration is given in Figure 2.

Bottom-up approach: In bottom-up approach, the given array itself is seen as a heap and in bottom-up fashion, the heap is converted into a max-heap. i.e., check for max-heap violation is done from the last level i to root (level 1). An illustration is given in Figure 3.

Pseudocode: **Heap-Sort(A)** Source: CLRS Bottom-up Construction

```
begin
Build-Max-Heap(A)
  for i = A.length down to 2
    Swap(A[1], A[i])
    A.heapsize = A.heapsize -1
    Max-Heapify(A, 1)
end
```

Build-Max-Heap(A)

```
begin
  A.heapsize = A.length
  for i = ⌊A.length/2⌋ down to 1
  Max-Heapify(A, i)
end
```

Max-Heapify(A, i)

```
begin
  l = Left(i)
  r = Right(i)
  if l ≤ A.heapsize and A[l] > A[i]
  largest = l
    else largest = i
  if r ≤ A.heapsize and A[r] > A[largest]
    largest = r
  if largest ≠ i
  Swap(A[i], A[largest])
   Max-Heapify(A, largest)
end
```

Run-time Analysis: Top-down Approach A trivial analysis tells us the cost for constructing a max-heap is $n \times O(h)$, where, h is the height of a heap. We shall now analyze the height of a heap on n nodes. Since it is complete upto the height $h-1$ and we know that there are at least one node and at most 2^h nodes in level h, the lower bound for the number of nodes n in a tree is $n \geq 2^0 + 2^1 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1$. Therefore, $n \geq 2^h$, $h = O(\log n)$.
Similarly, the upper bound is $n \leq 2^0 + 2^1 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$. $h \geq \log \frac{n+1}{2}, h = \Omega(\log n)$.
Thus, the cost for constructing a max-heap in bottom-up approach is $O(nh) = O(n \log n)$.

Run-time Analysis: Bottom-up Approach There are $\lceil \frac{n}{2} \rceil$ elements at the last level and all are trivially max-heaps, so there is no cost at each node at this level. At last but one level, there are $\lceil \frac{n}{4} \rceil$ nodes and in the worst case at each node we need 2 comparisons to set right max-heap property. At last but second level, there are $\lceil \frac{n}{8} \rceil$ nodes and in the worst case at each node we need 4 comparisons to set right max-heap property. i.e., in the worst case, 2 comparisons for the first call to max-heapify() and max-heapify() will be recursively called one more time for which we need additional 2 comparisons. At last but i level, there are $\lceil \frac{n}{2^{i+1}} \rceil$ nodes and in the worst case at each node we need $O(i)$ comparisons to set right max-heap property. There will be i calls to max-heapify and each call incurs 2 comparisons. Therefore, the overall cost is $\sum_{i=0}^{\log n} \lceil \frac{n}{2^{i+1}} \rceil \cdot O(i)$.

$$= O(\sum_{i=0}^{\log n} \lceil \frac{n}{2^{i+1}} \rceil \cdot i).$$
$$= O(\sum_{i=0}^{\log n} \lceil \frac{n}{2^{i}} \rceil \cdot i).$$
$$= O(n \cdot \sum_{i=0}^{\log n} \frac{i}{2^{i}}).$$

Note that $\sum_{i=0}^{\log n} \frac{i}{2^{i}}$ is at most 2. Therefore, the total cost for build-max-heap() in bottom-up fashion is $O(n)$. It is important to highlight that the leaves in any heap are present in locations $A[\lfloor \frac{n}{2} \rfloor + 1 \ . \ . \ n]$ and due to this elements in $A[\lfloor \frac{n}{2} \rfloor + 1 \ . \ . \ n]$ are trivial max-heaps. Further, due to this reason, the for loop in build-max-heap() is run from $A[\lfloor \frac{n}{2} \rfloor \ . \ . \ 1]$.

Origin: Heap sort was invented by J. W. J. Williams in 1964. This was also the birth of the heap, presented already by Williams as a useful data structure in its own right [10].

Application: Heap Sort To get a sorting sequence on the given input array, follow the steps: (i) Construct a max-heap (ii) Extract root (maximum element) and place it in $A[n]$. (iii) Place $A[n-1]$ at the root. (iv) Call max-heapify with respect to the root. Repeat the above steps till $A[2]$. Cost for Step (iv) in the worst case is $O(h)$, therefore, the run-time of heap sort is $O(n \log n)$.

Aliter: One can also obtain the cost of step (iv) [max-heapify subroutine] using the following recurrence. Note that when max-heapify() is called with respect to the root with array size being n, based on the value of left child/right child, the next recursive call on max-heapify() will be to the left subtree of the root/right subtree of the root. If the original tree is balanced, then the size of subtree in either case is $\frac{n}{2}$. If the original tree in which max-heapify was called is unbalanced, i.e., the last level of the tree is half-full, then the left subtree has at most $\frac{2n}{3}$ nodes and the right subtree has $\frac{n}{3}$ nodes. Therefore, in the worst case, the size of the recursive sub-problem for max-heapify() is at most $\frac{2n}{3}$. If $T(n)$ is the cost of max-heapify() on n-node heap, then $T(n) = T(\frac{2n}{3}) + 2$. The constant '2' in the recurrence is for 2 comparisons done at each level. Therefore, the cost of max-heapify() (use master theorem) is $T(n) = O(\log n)$.

Overview: In this article, we have discussed 6 different sorting algorithms with a trace, a proof of correctness, and the run-time analysis. Worst case analysis of bubble, selection, insertion, and quick sort reveals that, for any input the run-time is $O(n^2)$ and for heap and merge sort, it is $O(n \log n)$ in worst case. It is now natural to ask; are there sorting algorithms for which the run-time is $o(n \log n)$. For example, does there exist a sorting algorithm with run-time $O(n \log \log n)$ or $O(n)$. In an attempt to answer this question, we shall shift our focus and analyze the inherent complexity of the problem. i.e.,

- What would be the least number of comparisons required by any sorting algorithm in worst case.

Assuming there are 10 algorithms for a given problem (say sorting), to answer the above question, compute the worst case time of each algorithm and take the minimum. However, in general, we do not know how many algorithms exist for a problem. Therefore, one should analyze inherent complexity of the problem without bringing any algorithm into the picture and such analysis are known as lower bound analysis in the literature. To begin with, we shall discuss the lower bound analysis of `Problem:1 sequential search` and `Problem:2 search in a sorted array`. Subsequently, we present a lower bound analysis for `Problem:3 sorting problem`.

Claim: `Input:` Arbitrary Array A, an element x. `Question:` Is $x \in A$. Any algorithm for sequential search in worst case incurs $\Omega(n)$, n: the size of the input array A.

Proof: To correctly say that the element x to be searched is in A or not. For any algorithm, x must be compared with every $A[i]$, $1 \leq i \leq n$ at some iteration of the algorithm. Since the comparison is $\Omega(1)$ effort,

any algorithm for sequential search in worst case incurs $n \cdot \Omega(1) = \Omega(n)$.
Remark:
1. Linear search is an example algorithm for sequential search in which x is searched in A in linear fashion.
2. One can also pick n distinct random elements from A and search whether x is present in A or not.

Claim: `Input:` Sorted Array A, an element x. `Question:` Is $x \in A$. Any algorithm for search in a sorted array in the worst case incurs $\Omega(\log n)$, n: the size of the input array A.

Proof: Consider a binary tree in which each node corresponds to a comparison done between x and $A[i]$ for some i. We shall view the binary tree as follows; root node represents the comparison between x and $A[i]$ for some i. Based on the comparison, if $x < A[i]$, then the next level search is done on $A[1..(i-1)]$, otherwise search is done on $A[(i+1)..n]$. We naturally get a binary tree modelling the computations performed at each level. The structure of the binary tree is based on the underlying algorithm and the input. However, for any algorithm and input, a complete binary tree acts as a decision tree (a tree capturing all comparisons) to capture the lower bound analysis. For every other binary tree, the number of comparisons is at least the number of comparisons taken by the complete binary tree. Therefore, the least number of comparisons required by any algorithm to search in a sorted array is precisely the height of the complete binary tree which is $O(\log n)$.

Remark:
1. Binary search is an example algorithm for searching an element in a sorted array wherein; at level 1, node 1 represents the comparison between x and $A[\frac{n}{2}]$, node 2 represents the comparison between x and $A[\frac{n}{4}]$ and node 3 represents the comparison between x and $A[\frac{3n}{4}]$ and so on. Therefore, the recurrence is $T(n) = T(n/2) + 1$, and its solution is $O(\log n)$
2. Ternary search is another example algorithm where x is compared with $A[\frac{n}{3}]$ and $A[\frac{2n}{3}]$ at each iteration on the recursive subproblem. Therefore, $T(n) = T(n/3) + 2$. Solving using master theorem, we get, $T(n) = O(\log n)$

Claim: Any algorithm for sorting problem under comparison model incurs $\Omega(n \log n)$ in worst case.

Proof: Note that our analysis is based on the assumption that the underlying algorithm arranges the elements based on the primitive operation `comparison between two elements`. It may be possible to get $o(n \log n)$ or $O(n)$ for the sorting problem if the underlying model is different from comparison model. For example, counting and radix sort arranges the elements in the input sequence without comparing a pair of elements and hence these two algorithms do not fall into our discussion. Moreover, counting and radix sort run in $O(n)$ under some assumptions.

We shall now analyze the lower bound analysis of sorting using a decision tree. A decision tree is a binary tree where each node represents a comparison between a pair of elements $(A[i], A[j])$ for some i and j. For example, a decision tree for the input size 3 is given in Figure 4. Note that $(A[i], A[j])$ must be compared at some iteration of any sorting algorithm and the exact iteration number varies from algorithm to algorithm. Decision tree precisely presents all possible comparisons that can happen in an array of n elements. The exact sequence of comparisons depends on the input. Note that for the input $< a_1, \ldots, a_n >$ there are $n!$ leaves as each of the permutations of $< a_1, \ldots, a_n >$ appears as a leaf node. Based on the nature of $< a_1, \ldots, a_n >$, any algorithm will try to explore one of the paths from the root to the leaf node that contains the sorted sequence of $< a_1, \ldots, a_n >$. We are interested in what would be the least number of comparisons in worst case taken by any input $< a_1, \ldots, a_n >$ to reach the right leaf. It is now clear that the number of comparisons required by any sorting algorithm for any input (in worst case) is precisely the height of the decision tree or the length of the longest path from the root to a leaf node.

Clearly, the number of nodes in a decision tree for the input of size n is $2n! - 1$, ($n! - 1$ internal nodes and $n!$ leaves). We know that at height h, there are 2^h nodes. Since there are $n!$ leaves, $2^h \geq n!$. Further,

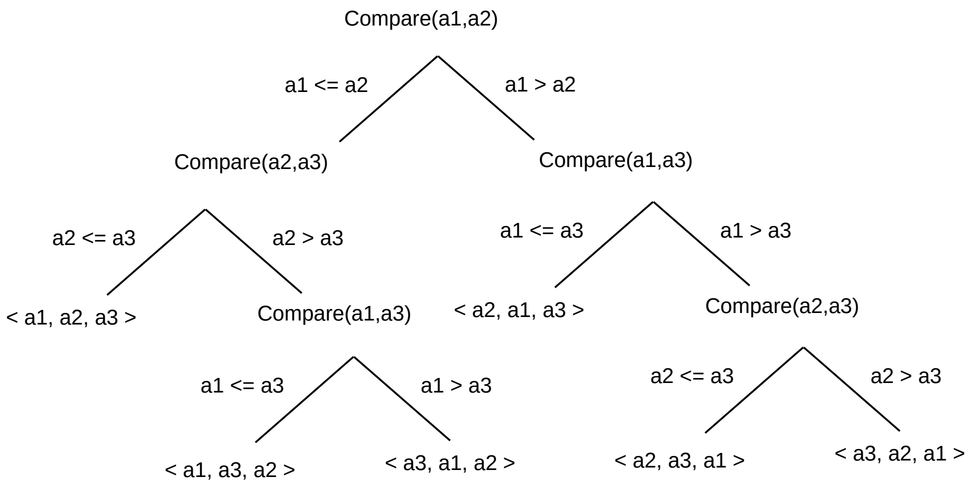


Figure 4: Decision Tree for Sorting three numbers, input: $< a_1, a_2, a_3 >$

$h \geq \log(n!)$. Note that $(\frac{n}{e})^n \leq n! \leq n^n$.
$h \geq \log(n!) \geq \log(\frac{n}{e})^n$.
This implies that $h = \log n^n - n \log e$. $h \geq n \log n - 1.44n$ which is $h = \Omega(n \log n)$.
Thus, our claim follows. Since our analysis is entirely depend on comparisons between a pair of elements, this lower bound holds good for any sorting algorithm based on comparison model.
Remark:

- Since heap sort and merge sort take $O(n \log n)$ in worst case and it matches with the lower bound, these two algorithms are optimal sorting algorithms. Note that the optimality is with respect to the number of comparisons.
- We also can get a lower bound for best case from the decision tree. Since the length of the shortest path is n, any algorithm for best case input incurs $\Omega(n)$.

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thanks all of them.

References:
1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.

References

[1] Friend, E: Sorting on electronic computer systems. J. ACM 3 (1956), 134-168.

[2] Gotlieb, C: Sorting on computers. Communications of the ACM 6, 5 (May 1963), 194-201.

[3] Bose, R. C., and Nelson, R. J: A sorting problem. Journal of the ACM (JACM) 9, 2 (1962), 282-296.

[4] McCracken, D., Weiss, H., and Lee, T: Programming Business Computers. John Wiley, 1959.

[5] Iverson, K: A Programming Language. John Wiley, 1962.

[6] Donald E. Knuth: The Art of Computer Programming, Volume 3: Sorting and Searching. Pearson Education. Inc., 1998.

[7] Jyrki Katajainen and Jesper Larsson Traff: A meticulous analysis of mergesort programs. Technical Report, (1997).

[8] Shustek, L: Interview: An interview with C.A.R. Hoare. Comm. ACM 52 (3): 3841, (2009).

[9] Hoare, C. A. R: Algorithm 64: Quicksort. Comm. ACM 4 (7): 321, (1961).

[10] Brass, Peter: Advanced Data Structures. Cam-bridge University Press, (2008).

[11] H.Cormen, T., C. E.Leiserson, R. L.Rivest and C. Stein: Introduction to Algorithms 3rd Edition. McGraw-Hill Higher Education, (2001).