2 Divide and Conquer

The divide and conquer strategy involves three main steps.

- **Step 1:** Problem is divided into some subproblems that are smaller parts of the same problem.
- **Step 2:** Solve (Conquer) the subproblems in a recursive manner. If the subproblem is small, then solve it directly.
- **Step 3:** Combine solutions of all the subproblems into the original problem's solution.
- When the subproblem is not small enough to find solution directly (without recursion), i.e., with base case, it is divided into smaller subproblems.

The Maximum Subarray Problem

Problem:

Given an array A[1...n], one should find a continuous subarray that sums up to the maximum value.

Input: Array A[1...n]

Output:

The maximum sum of values possible in the subarray A[i...j] and the indices i and j.

Note:
Maximum subarray might not be unique.

Example:

Array A	5	−4	3	4	−2	1
	1	2	3	4	5	6

Maximum subarray: A[1...4] (i = 1, j = 4) and sum = 8.

Divide and Conquer Strategy:

1. **Divide:** The array A[l...h] is divided into two subarrays of as equal size as possible by calculating the mid.
2. **Conquer:** By algorithm given below.
   a) Finding maximum subarray of A[l...m] and A[m+1...h]
   b) Finding a maximum subarray that crosses the midpoint.
3. **Combine:** Returning the maximum of the above tree.

This process of finding solution works because any subarray may lie completely on one side of the midpoint, on the other side, or crossing the midpoint.

Algorithm:

```
FIND-MAXIMUM-CROSSING-SUBARRAY (Arr, l, m, h)
l-sum = −∞
sum = 0
   for i ← m downto l
      sum ← sum + Arr[i]
      if sum > l-sum
         l-sum ← sum
         max-left ← i
   r-sum ← −∞
   sum ← 0
   for j ← m + 1 to h
      sum ← sum + Arr[j]
      if sum > r-sum
         r-sum ← sum
         max-right ← j
  return (max-left, max-right, l-sum + r-sum)
```

Algorithm returns the indices i and j and the sum of two subarrays.

- If the subarray Arr[l...h] contain n entries, then n = h−l+1.
- Since, each iteration of two for loops takes θ(1) time, and the number of iterations is to be counted to get the total time complexity.

- The first for loop makes m-l+1 iterations, and the second for loop makes h-m iterations.

Total number of iterations = (m-l+1) + (h-m)

= h-l+1

= n

∴ FIND-MAXIMUM-CROSSING-SUBARRAY(Arr, l, m, h) takes θ(n) time.

- With the help of the FIND-MAXIMUM-CROSSING-SUBARRAY procedure in hand that runs in linear time, pseudocode for divide and conquer can be written to solve the maximum subarray sum problem.

Find-Maximum-Subarray(Arr, L, H)

```
If h == l
Return (l, h, Arr[l])
Else m = $\lfloor (l + h) / 2 \rfloor$
(L-low, l-high, l-sum) = FIND-MAXIMUM-SUBARRAY(Arr, l, m)
(R-low, r-high, r-sum) = FIND-MAXIMUM-SUBARRAY (Arr, m+1, h)
(Cross-low, cross-high, cross-sum) = FIND-MAXIMUM-CROSSING-SUBARRAY (Arr, l, m, h)
If l-sum ≥ r-sum and l-sum ≥ cross-sum
Return (l-low, l-high, l-sum)
Else if r-sum ≥ l-sum and r-sum ≥ cross-sum
Return (r-low, r-high, r-sum)
Else return(cross-low, cross-high, cross-sum)
```

Remarks:

1. Initial call: FIND-MAXIMUM-SUBARRAY (Arr, l, h).
2. Base case is used when the subarray can not be divided further, i.e., with 1 element.
3. Divide is based on the value of m.

   Conquer by the two recursive calls to FIND-MAXIMUM-SUBARRAY and a call to FIND-MAXIMUM-CROSSING-SUBARRAY.

   Combine the solutions by finding which of the three results gives the maximum sum.
4. **Complexity:**

   T(n) = 2.T(n/2) + θ(n) + θ(1)

   = θ(nlogn) by masters theorem.

Previous Years' Question

Consider a sequence of 14 elements:

A= [-5, -10, 6, 3, -1, -2, 13, 4, -9, -1, 4, 12, -3, O].

The sequence sum S(i, j) = ~t=i A[k]. Determine the maximum of S(i, j), where 0 ~ i ~j < 14.

(Divide and conquer approach may be used.)

Solution: Sum = 6 + 3 - 1 - 2 + 13 + 4 - 9 - 1 + 4 + 12 = 29

Strassen's Algorithm for Matrix Multiplication

Given 2 square matrices A and B of size n×n each, find the product of two matrices.

Naive method:

```
SQUARE–MATRIX–MULTIPLY (A, B)
1. n ← A.rows
2. let C be a new n×n matrix
3. for i ← 1 down to n
4. for j ← 1 down to n
5. $C_{ij}$ ← 0
6. for k ← 1 down to n
7. $C_{ij}$ ← $C_{ij}$ + $a_{ik}$ . $b_{kj}$
8. return C
```

a) Because each of the triply–nested loops runs for exactly n iterations, and each execution of line 7 takes constant time.

∴ Time complexity = O(n^3) time for naive method

A simple divide and conquer algorithm:

- Divide matrices A and B into four n/2 × n/2 matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (1)$$

So that we rewrite the equation C = A.B as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} . \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$$
$$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$$
$$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$$
$$C_{22} = A_{21}\ B_{12} + A_{22}\ B_{22}$$

- By using these equations, create a straight forward recursive, divide–and–conquer algorithm.

SQUARE–MATRIX–MULTIPLY–RECURSIVE (A,B)

```
1. N = A. rows
2. Let C be a new n×n matrix
3. If n == 1
4. $C_{11} = a_{11} \cdot b_{11}$
5. Else partition A, B, and C as in equations (1)
6. $C_{11}$ = SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{11}$, $B_{11}$)
   + SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{12}$, $B_{21}$)
7. $C_{12}$ = SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{11}$, $B_{12}$)
   + SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{12}$, $B_{22}$)
8. $C_{21}$ = SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{21}$, $B_{11}$)
   + SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{22}$, $B_{21}$)
9. $C_{22}$ = SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{21}$, $B_{12}$)
   + SQUARE–MATRIX–MULTIPLY–RECURSIVE ($A_{22}$, $B_{22}$)
10. Return C
```

- In the above method, we do eight multiplications for matrices of size n/2×n/2 and 4 additions.
- Addition of 2 matrices takes $O(n^2)$ time

So, the time complexity can be given as

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 8T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

By masters theorem, the time complexity is $O(n^3)$, which is the same as the naive method.

Strassen's method:

- Strassen's method makes the recursion tree sparse.
- Instead of eight recursive multiplications of n/2x n/2 matrices, it performs only seven.
- Strassen's method also uses the divide and conquer strategy, i.e., it divides the matrix into some smaller matrices of order n/2 x n/2.
- In Strassen's matrix multiplication, the result of four sub-matrices is calculated using the formula below:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{bmatrix}$$

Where

$P_1 = A_{11} (B_{12} - B_{22})$

$P_2 = B_{22} (A_{11} + A_{12})$

$P_3 = B_{11} (A_{21} + A_{22})$

$P_4 = A_{22} (B_{21} - B_{11})$

$P_5 = A_{11} (B_{11} + B_{22}) + A_{22} (B_{11} + B_{22})$

$= (A_{11} + A_{22}) (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) (B_{11} + B_{12})$

So, a total of 7 multiplications are required.

Time complexity of Strassen's method:

Adding and subtracting two matrices takes $O(n^2)$ time.

Hence, the time complexity taken by the algorithm can be written as

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 7T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

By master's theorem, the time complexity is $O(n\log_2 7)$, which is approximately $O(n^{2.8074})$

NOTE:

Strassen's method is not suggested for practical purposes due to the following reasons:

- More number of constants are used and naive methods performs better in most of the cases.
- There are better methods available for sparse matrices in particular.
- The sub-matrices occupy more space in recursion method.
- Strassen's algorithm gives more errors due to precision limitations in computer arithmetic on non-integer value.

Problem of Sorting

Input: An array A of size n.

Output: Reordering of the array such that A[0] ≤ A[1] ≤ … ≤ A[n–1].

Properties of Sorting algorithms:

In place:

“A sorting algorithm is said to be in place if a maximum of O(1) (in some cases it can be increased) elements are stored outside the array at any moment.

E.g., Insertion-sort is in place.

Stability:

A sorting algorithm is said to be stable if the order of repeated elements is not changed.

E.g., Insertion sort, merge sort, etc.

Adaptive:

A sorting algorithm falls into the adaptive sort family if it takes advantage of the existing order in its input.

E.g., Insertion sort

Merge sort:

The merge sort algorithm uses divide and conquer approach.

Divide:

Divide the array of n elements each into two subarrays of n/2 elements each.

Conquer:

Sort the subarrays in the recursive method by using merge sort.

Combine:

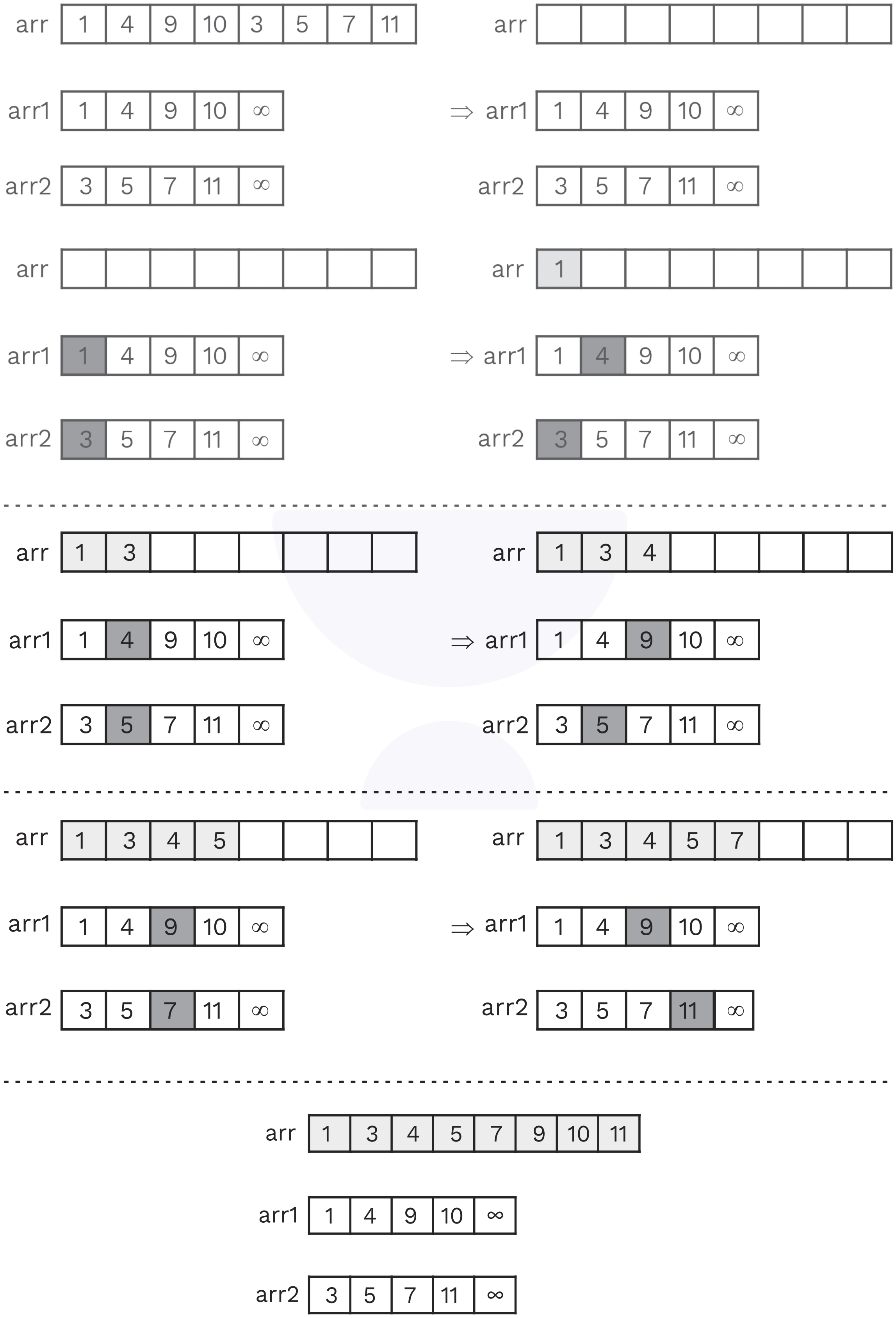
- Merge the two subarrays to result in the sorted array.
- The main operation of merge sort algorithm is to merge two sorted arrays.
- MERGE(A,p,q,r) is used to merge the arrays such that p ≤ q < r.It assumes the two arrays A[p…q] and A[q+1…r] are in sorted order.

Algorithm

```
MERGE_FUNCTION (arr, low, mid, high)
1.  Let n1 ¬ (mid – low) + 1
2.  Let n2 ¬ high – mid
3.  Let arr1[1...(n1 + 1)] & arr ¬ 2[1...(n2 + 1)] be
    the new arrays
4.  for i ¬ 1 to n1
5.  arr1[i] ¬ arr[low + i – 1]
6.  for j ¬ 1 to n2
7.  arr2[j] ¬ arr[mid + j]
8.  arr1[n1 + 1] ¬ ∞
9.  arr2[n2 + 1] ¬ ∞
10. i ¬ 1
11. j ¬ 1
12. for k ¬ low to high
13. if arr1[i] ≤ arr2[j]
14. arr[k] ¬ arr1[i]
15. i ¬ i + 1
16. else arr[k] = arr2[j]
17. j ¬ j + 1
```

- First line finds the length n_1 of the subarray arr[low…mid], and 2nd line finds the length n_2 of the subarray arr[mid+1…high].
- arr1 and arr2 arrays are created with lengths n_1 and n_2, respectively.
- The extra position in each array holds the sentinal.
- For loop of the line (4-5 and 6-7) copies subarray arr[low…mid] into arr1[1…n_1] and arr[mid + 1…high] into arr2[1…n_2].
- Traverse arr1 and arr2 simultaneously from left to right, and write the smallest element of the current positions to arr.

Example:



arr | 1 | 4 | 9 | 10 | 3 | 5 | 7 | 11

arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | | | | | | | |

⇒ arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | | | | | | | |

arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | 1 | | | | | | |

⇒ arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | 1 | 3 | | | | | |

arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | 1 | 3 | 4 | | | | |

⇒ arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | 1 | 3 | 4 | 5 | | | |

arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

arr | 1 | 3 | 4 | 5 | 7 | | |

⇒ arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

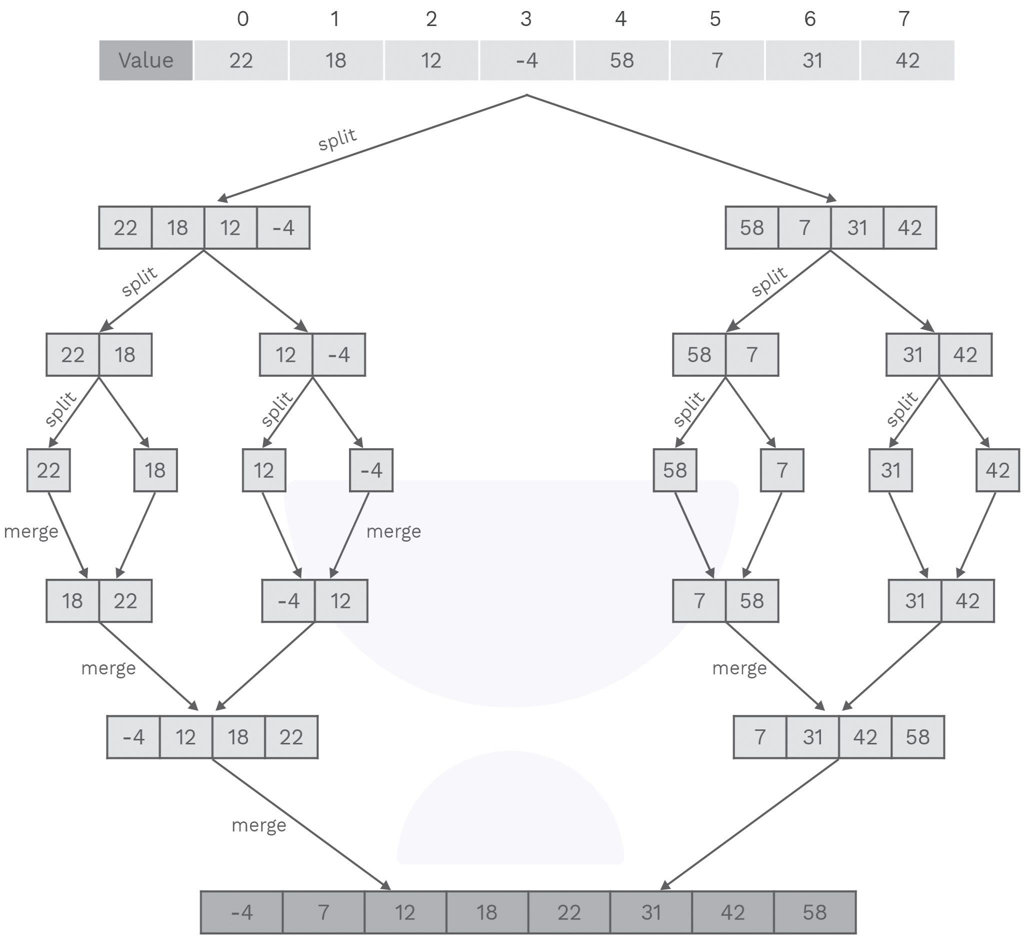
arr | 1 | 3 | 4 | 5 | 7 | 9 | 10 | 11

arr1 | 1 | 4 | 9 | 10 | ∞

arr2 | 3 | 5 | 7 | 11 | ∞

```
MERGE-SORT (arr, low, high)
1. if low < high
2.     mid = $\lfloor (low + high) / 2 \rfloor$
3.     MERGE-SORT (arr, low, mid)
4.     MERGE-SORT (arr, mid+1, high)
5.     MERGE (arr, low, mid, high)
```

Example:



Analysis:

- Even the Merge sort works correctly on odd-length arrays. Recurrence analysis is simplified if we assume the input array length is of power of 2.

Break down of running time:

Divide:

This step finds the middle of the subarray in constant time, i.e., O(1).

Conquer:

Solving two subproblems of $\frac{n}{2}$ size each recursively results in $2T\left(\frac{n}{2}\right)$ time complexity.

Combine:

MERGE_FUNCTION procedure applied on an n-sized subarray takes time θ(n).

Recurrence for the worst-case running time T(n) of merge sort.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

By Masters Theorem:

T(n) = θ(nlogn)

Rack Your Brain

Think about the properties of the merge sort and other kind of implementation of merge sort

Hint: There is an in place merge sort implementation.

Previous Years' Question

Assume that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

(A) 256 (B) 512

(C) 1024 (D) 2048

Solution: (B)

Quick sort:

Quick sort uses 3 step divide and conquer approach to sort an array A[p...r].

Divide:

Rearrange the array A into two subarrays A[p...q-1] and A[q+1...r].

The partition procedure computes the index q such that the elements A[p...q-1] are smaller than A[q] and the elements A[q+1...r] are greater.

Conquer:

Sort the 2 subarrays A[p...q-1] and A[q+1...r] recursively in quicksort.

Combine:

Since all the subarrays are already sorted, no extra work is needed to combine them together. Array A[p...r] is now sorted.

Algorithm:

QUICKSORT_ALGO (Arr, low, high)

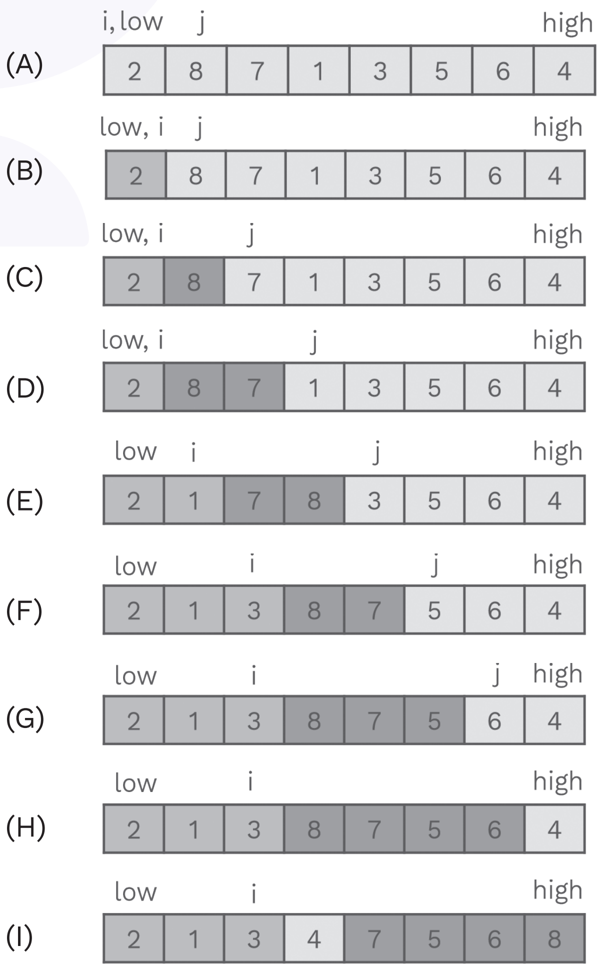
```
if low < high
    mid ← PARTITION_OF_ARRAY (Arr, low, high)
    QUICKSORT_ALGO (Arr, low, mid-1)
    QUICKSORT_ALGO (Arr, mid+1, high)
```

The main part of the algorithm is the PARTITION_OF_ARRAY procedure, which makes the subarray Arr[low...high] in place.

PARTITION_OF_ARRAY (Arr, low, high)

```
x ← Arr[high]
i ← low-1
for j ← low to high-1
    i   f Arr[j]<= x
        i ← i + 1
        swap Arr[i] with Arr[j]
swap Arr[i + 1] with Arr[high]
return i + 1
```

Example:



PARTITION_OF_ARRAY applied on an Array:

- Array entry A[r] be the pivot element x.
- Orange colour elements are all in the first partition values no greater than x.
- Pink colour elements are in the second partition with values greater than x.
- Blue colour elements, have not yet been put in one of the first two partitions.

Analysing quicksort:

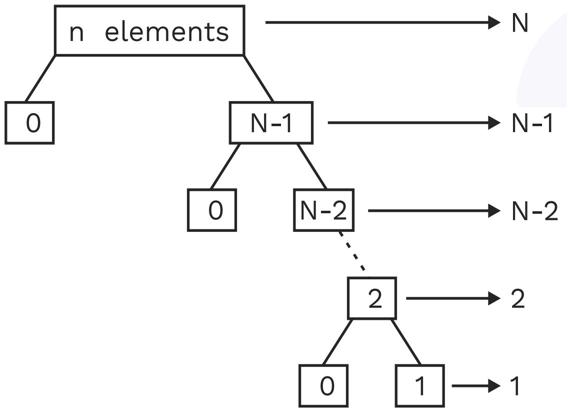
The choice of pivot is most crucial:

- A incorrect pivot may lead to worst-case time complexity O(n^2), whereas a pivot that divides the array into 2 equal-sized subarrays gives the best case time complexity, i.e., O(nlogn).

The worst-case choice:

The pivot may be the largest or the smallest of all the elements in the array.

- When the first iteration is done, then one subarray contains 0 elements, and the other subarray has n-1 elements.
- Quicksort is applied recursively on this second subarray of n-1 elements.



However, quicksort is fast on the “randomly scattered” pivots.

The partition step needs atleast n-1 comparisons.

- The recurrence for the running time is

$$\left.\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ T(1) &= \theta(1) \end{aligned}\right\}$$

T(n)=T(n-1)+Θ(n)

=T(n-1)+cn (removing Θ with constant)

=T(n-2)+c(n-1)+cn

=T(n-3)+c(n-2)+c(n-1)

=T(1)+c(2)+c(3)+...+(n-2)+(n-1)+n

=c(1+2+3+...+n)

=c(n(n+1)/2)=O(n^2).

Average case:

T(n): Average number of comparisons during quicksort on n elements.

$$T(1) = \theta(1), T(0) = \theta(1)$$

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1)T(n-i) + n - 1\right)$$

$$T(n) = \frac{2}{n}\sum_{i=1}^{n}\left(T(i-1)\right) + n - 1$$

$$nT(n) = 2\sum_{i=1}^{n}\left(T(i-1)\right) + (n-1)n \qquad \text{...(i)}$$

For (n-1)

$$(n-1)T(n-1) = 2\sum_{i=1}^{n-1}\left(T(i-1)\right) + (n-1)(n-1-1)$$

$$(n-1)T(n-1) = 2\sum_{i=1}^{n-1}\left(T(i-1)\right) + (n-1)(n-2) \qquad \text{...(ii)}$$

Subtracting equation (2) from (1), we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

$$nT(n) - (n+1)T(n-1) = 2(n-1)$$

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

$$g(n) - g(n-1) = \frac{2(n-1)}{n(n+1)}$$

where $g(n) = \dfrac{T(n)}{n+1}$

Taking RHS-

$$\frac{2(n-1)}{n(n+1)} = \frac{2(n+1)-4}{n(n+1)}$$

$$= \frac{2}{n} - \frac{4}{n(n+1)}$$

$$= \frac{2}{n} - \frac{4}{n} + \frac{4}{n+1}$$

$$= \frac{4}{n+1} - \frac{2}{n}$$

Now,

$$g(n) - g(n-1) = \frac{4}{n+1} - \frac{2}{n}$$

Hence,

$$g(n) = \frac{4}{n+1} + \left(2\sum_{j=2}^{n}\frac{1}{j}\right) - 2$$

$$= \frac{4}{n+1} + \left(2\sum_{j=1}^{n}\frac{1}{j}\right) - 4$$

$$= \frac{4}{n+1} + 2H(n) - 4$$

Now,

$$T(n) = (n+1)\left(\frac{4}{n+1} + 2H(n) - 4\right)$$

$T(n) = 2(n+1)\ H(n) - 4(n)$

$T(n) = 2(n+1)\log_e n + 1.16(n+1) - 4n$

$T(n) = 2n\log_e n - 2.84n + O(1)$

$T(n) = 2n\log_e n$

The average number of comparisons during quicksort on n elements approaches.

$2n\log_2 n - 2.84n$

$= 1.39n\log_2 n - O(n)$

So, the average case time complexity as quicksort is O(nlogn).

Previous Years' Question

Let p be a quicksort program to sort numbers in ascending order using the first element as pivot. Let t and t be the number of comparisons 1 2 made by p for the inputs {1,2,3,4,5} and {4,1,5,3,2}, respectively. Which one of the following holds?

[CS 2014 (Set-1)]

(A) t = 5 1

(B) t1 < t2

(C) t1 > t2

(D) t = t 1 2

Solution: (C)

Previous Years' Question

In quicksort, for sorting n elements, the (n/4) th smallest element is selected as pivot using an O(n) time algorithm. What is the worstcase time complexity of the quicksort?

[CS 2009]

(A) θ(n)

(B) θ(nlogn)

(C) θ(n2)

(D) θ(n logn) 2

Solution: (B)

Previous Years' Question

Randomised quicksort is an extension of quicksort where the pivot is chosen randomly. What is the worst-case complexity of sorting n numbers using randomised quicksort?

[CS 2001]

(A) O(n)

(B) O(nlogn)

(C) $O(n^2)$

(D) O(n!)

Solution: (C)

Randomised quicksort:

- Assume all elements are distinct.
- Partition around a random element, i.e., the pivot is chosen randomly from the set of elements.
- This implies the splits n-1, n-2, ..., n-1 have the same probability of 1/n.

Note:
Randomisation is a general tool to improve algorithms with bad worst-case but good or average-case complexity.

```
Randomised_Partition (Arr, p, r)
  {
      i ← Random(p, r)      /* it generate a
                               random number
                               between p
                            and r and including
                            p and r */
      exchange Arr[r] ↔ Arr[i]
      return Partition (Arr, p, r)
  }
Randomised_Quicksort (Arr, p, r)
{
  if p < r
   q ← Randomised_Partition (Arr, p, r)
   Randomised_Quicksort (Arr, p, q-1)
   Randomised_Quicksort (Arr, q+1, r)
}
```

Time complexity:

- Let's assume T(n) be the number of comparisons needed to sort n numbers using quicksort.
- Since each split occurs with probability 1/n. In quicksort, every number requires atleast (n-1) comparison because the (n-1) number of elements has to be compared with the pivot element.

Let the pivot is i^{th} smallest element. Then we get (i-1) element on the left side and (n-i) element on the right side. And we have to do randomised-quicksort on $(i-1)^{th}$ and $(n-i)^{th}$ element. So, it take T(i-1) and T(n-i) time, respectively.

∴ T(n) = T(i-1) + T(n-i) + (n-1) with probability 1/n.

↑

this is because atleast (n-1) comparison is required.

Hence,

$$T(n)=\frac{1}{n}\sum_{j=1}^{n}\left(T(j-1)+T(n-j)+n-1\right)$$

(replacing i with j for simplicity)

= This is the expectation.

$\sum_{j=1}^{n} T(j-1)$= This quantity varies from T(0) to T(n-1)

$\sum_{j=1}^{n} T(n-j)$= This quantity varies from T(n-1) to T(0)

So, every term T(0) to T(n-1) appears twice.

So, we write it as:

$$=\frac{2}{n}\sum_{j=0}^{n-1}T(j)+\frac{1}{n}\sum_{j=1}^{n}n-1$$

$$=\frac{2}{n}\sum_{j=0}^{n-1}T(j)+\frac{n(n-1)}{n}$$

$$=\frac{2}{n}\sum_{j=0}^{n-1}T(j)+n-1 \qquad ...(3)$$

$$=O(n\log_2 n)$$

∴ The expected number of comparisons is O(nlogn). But the worst-case time complexity of randomised quicksort is $O(n^2)$.

Note:

- The running time of quicksort does not depend on the input. It depends on the random numbers provided by the generation.
- Thus, for the same input the program might take 2sec today and 5sec tomorrow.
- The average time taken over many different runs of the program would give us the expected time.
- Same as saying that we are taking average over all possible random number sequences provided by the generator.

Solution of quicksort( ) recurrence relation:

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n - 1 \qquad \text{...(i) (from eq (3))}$$

Note that T(0) = 0, T(1) = 0.

Now, $T(n-1) = \frac{2}{(n-1)}\sum_{i=0}^{n-2} T(i) + (n-2)$

or, $(T(n-1) - n + 2) = \frac{2}{(n-1)}\sum_{i=0}^{n-2} T(i)$

Multiply both sides by $\frac{(n-1)}{n}$

$$\frac{2}{(n-1)}\sum_{i=0}^{n-2}\left(T(i)\right) \times \frac{(n-1)}{n} = \frac{(n-1)}{n} \times \left(T(n-1) - n + 2\right)$$

or, $\frac{2}{n}\sum_{i=0}^{n-2} T(i) = \frac{(n-1)}{n} = \left(T(n-1) - n + 2\right)$ ...(ii)

from (1) and (2);

$$T(n) = \frac{(n-1)}{n}\left(T(n-1) - n + 2\right) + \frac{2}{n}T(n-1) + (n-1)$$

$$= T(n-1)\left(\frac{n-1+2}{n}\right) - n + 1 + \frac{2(n-1)}{n} + (n-1)$$

$$= \frac{(n+1)}{n}T(n-1) + \frac{2(n-1)}{n}$$

So, we got $T(n) = \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n}$

$$< \left(\frac{n+1}{n}\right)T(n-1) + 2 \qquad \left[2 * \frac{(n-1)}{n} < 2\right]$$

$$\left[ex : 2 * \frac{4}{5} = 2 * 0.8 = 1.6\right]$$

$$< \left(n + \frac{1}{n}\right)\left(\frac{n-1+1}{n-1}T(n-2) + 2\right) + 2$$

$$< \left(\frac{n+1}{n}\right)\left(\frac{n}{n-1}T(n-2) + 2\right) + 2$$

$$< \frac{n+1}{(n-1)}T(n-2) + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-1)}\left(\frac{n-2+1}{n-2}T(n-3) + 2\right) + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-1)}\left(\frac{n-1}{(n-2)}T(n-3) + 2\right) + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-2)}T(n-3) + \frac{2(n+1)}{(n-1)} + \frac{2(n+1)}{n} + 2$$

$$< \frac{n+1}{(n-2)}T(n-3) + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{n+1}{(n-2)}\left[\frac{n-3+1}{n-3}T(n-4) + 2\right]$$

$$+ 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{n+1}{(n-2)}\left[\frac{n-2}{(n-3)}T(n-4) + 2\right]$$

$$+ 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{n+1}{(n-3)}T(n-4) + \frac{2(n+1)}{(n-2)} + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)}\right] + 2$$

$$< \frac{(n+1)}{(n-3)}T(n-4) + 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)}\right] + 2$$

$$< \frac{(n+1)}{\left(n - (n+3)\right)}T(2)$$

$$+ 2(n+1)\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)} + \ldots + \frac{1}{4}\right] + 2$$

$$< \frac{(n+1)}{(n-n+1)} T(0) \cdot$$

$$+1)\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)} + \ldots + \frac{1}{3} + \frac{1}{2}\right] + 2$$

$$< 2(n+1)\underbrace{\left[\frac{1}{n} + \frac{1}{(n-1)} + \frac{1}{(n-2)} + \ldots + \frac{1}{2}\right]} + 2$$

$$\left[\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \ldots + \frac{1}{(n-1)} + \frac{1}{n}\right.$$

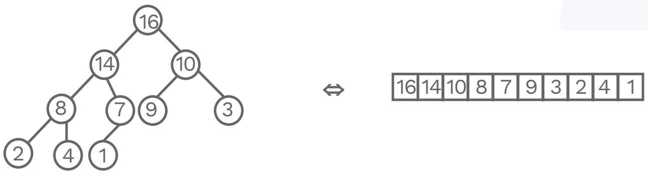
$$\left.\begin{array}{l} = \text{harmonic series} \\ = (\log n) \end{array}\right]$$

$$< 2(n+1)(\log n - 1) + 2$$

$$= O(n \log n)$$

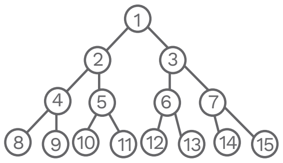
Heap sort:

- Heap is a data structure used to store and manage information. It is the design technique used by heapsort.
- A binary heap is an almost complete binary tree.

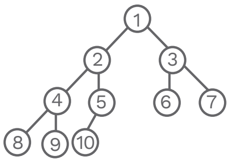


- The tree is said to be almost complete if all the levels are filled except the last level. The nodes in the last level are also filled from left to right.

Binary tree representation:



A full binary tree of height 3.



A complete binary tree with 10 nodes and height 3.

Heap as a tree:

- Root:
  First element in the array
- parent i) = $\lfloor i/2 \rfloor$:

  Returns index of node's parent
- left i) = 2i :

  Returns index of node's left child.
- right i) = 2i + 1:

  Returns index of node's right child.

  Height of a binary heap is O(logn).

There are two types of binary heaps:

Max–heaps and min–heaps

In both the heaps, the values in the nodes satisfy the heap property.

In a max heap, the max–heap property is that for every node i other than the root node, A[PARENTi)] ≥ A[i]

A min–heap property is that for every node i other than the root,

A[PARENT i)] ≤ A[i]

The smallest valued node in a min-heap is at the root.

- For the heap sort algorithm, we use max–heap.
- Min–heaps commonly implement priority queues.

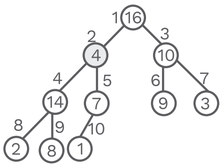
Heap operations:

- Build–Max–Heap:
  Produce a max–heap from an unordered array
- Max_heapify:
  Changes every single violation of the heap property in every subtree at its root.
- Insert, extract_max, heap sort

Max_heapify:

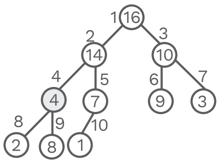
- Assume that the trees rooted at left i) and right i) are max_heap.
- If element A[i] violates the max_heap property, correct violation by "trickling" element A[i] down the tree, making the subtree rooted at index i a max_heap.

Max–heapify (example):



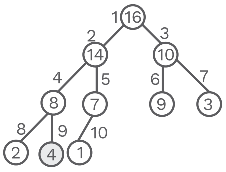
MAX–HEAPIFY (A,2)
Heapsize[A] = 10

⇓



Exchange A[2] with A[4]
Call MAX–HEAPIFY(A,4)
Because max–heap property is Violated

⇓



Exchange A[4] with A[9]
No more calls

- **Max–Heapify Pseudocode**

```
MAX–HEAPIFY(Arr, i)
l ← left_node i)
r ← right_node i)
if (l ≤ Arr.size_of_heap && Arr[i]<Arr[l])
maximum ← l
else maximum ← i
if (r ≤ Arr.size_of_heap && Arr[maximum]
    < Arr[r])
maximum ← r
if (maximum ≠ i)
swap Arr[i] and Arr[maximum]
MAX–HEAPIFY (Arr, maximum)
```

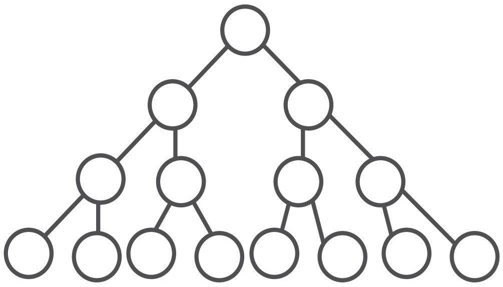
- Convert array A[1...n] into a max–heap, where n=A.length.
- The elements in the subarray $A\left[\left(\lfloor n/2 \rfloor + 1\right)\ldots n\right]$ are all leaves of the tree, and so each is a 1–element heap to begin with.

Build–max–heap (a):

```
A. size_of_heap = A. length
for i = ⌊A.length / 2⌋ down to 1
MAX–HEAPIFY (A, i)
```

Note:
The number of nodes present in a complete binary tree at height h is $\left\lceil \frac{n}{2^{h-1}} \right\rceil$, where n is the total nodes of the tree.

- Observe, however, that MAX–HEAPIFY takes O(1) time for nodes that are a level above the leaves, and O(L) for the nodes that are L levels above the leaves.



We have $\lceil n/4 \rceil$ nodes with level logn

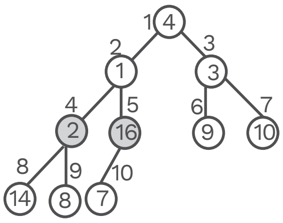
We have $\lceil n/8 \rceil$ nodes with level logn–1,

We have 1 root node that is 0 level above the leaves.

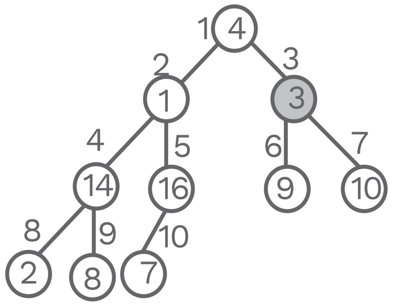
- Total amount of work in BUILD–MAX–HEAP for loop can be summed as
  $n/4\ (1C) + n/8\ (2C) + n/16\ (3C) + \ldots + 1(\log_n C)$
  substitute $n/4 = 2^K$
  $C\ 2^K\ (1/2^0 + 2/2^1 + 3/2^2 + \ldots(K+1)/2^K)$
  The term in brackets is bounded by a constant.
- This means that build–max–heap is O(n).

Build–max–heap (example):

A	4	1	3	2	16	9	10	14	8	7

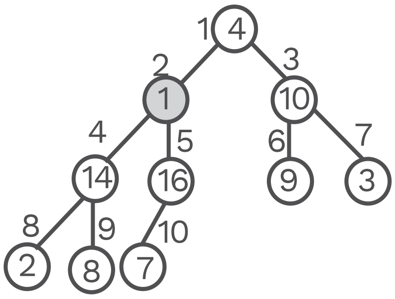


MAX–HEAPIFY (A, 5)
No change
MAX–HEAPIFY (A, 4)
Swap A[4] and A[8]



MAX–HEAPIFY (A, 3)

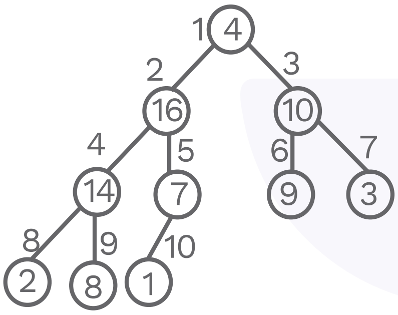
Swap A[3] and A[7]



MAX–HEAPIFY (A, 2)

Swap A[2] and A[5]

Swap A[5] and A[10]



MAX–HEAPIFY (A, 1)

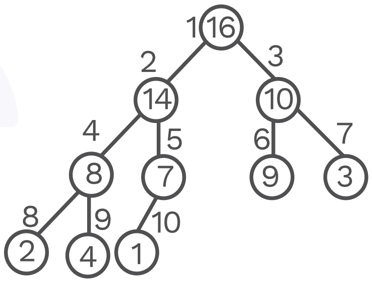
Swap A[1] with A[2]

Swap A[2] with A[4]

Swap A[4] with A[9]

A	4	1	3	2	16	9	10	14	8	7

⇒



Heap Sort

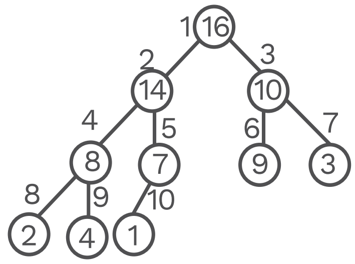
Sorting strategy:

1. A max-heap is built using an unordered array.
2. Maximum element A[i] is found.
3. Exchange the values of A[n] and A[1], which results in changing the maximum to the end of the array.
4. Delete node n from the heap.
5. The new root added may violate the max-heap property but its children are already max-heaps. Run max-heap to solve this.
6. Go to step 2 if the heap is not empty.

Heapsort(A)

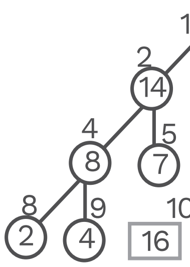
```
{
1.  BUILD–MAX–HEAP(A)
2.  for (i = A.length() down to 2)
3.  swap A[1] with A[i]
4.  A.heap–size = (A.heap–size–1)
5.  MAX–HEAPIFY (A, 1)
}
```

Heap–sort demo:



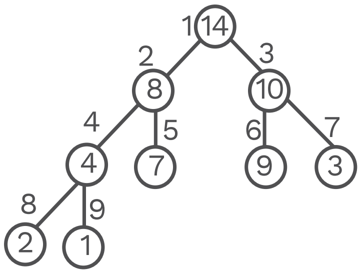
⇓ Swap A[10] and A[1]

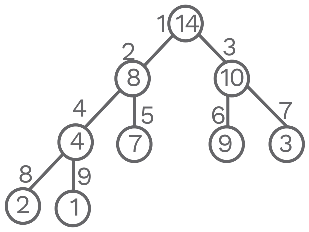
heap – size = 9



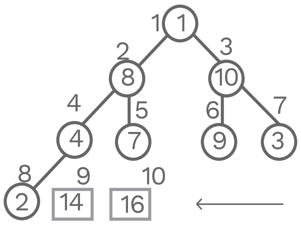
← Not part of heap

⇓ MAX–HEAPIFY (A, 1)





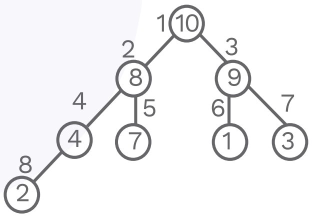
⇓ Swap A[9] and A[1]

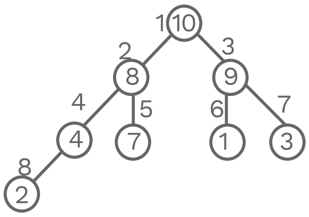


← Not part of heap

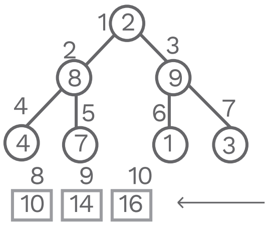
MAX–HEAPIFY (A, 1)

⇓





⇓ Swap A[8] and A[1]



← Not part of heap

Running time:

- The heap is empty after n iterations.
- Each iteration does a swap and a max-heapify operation, which involves O(logn) time.
- So it takes O(nlogn) in total.

Note:
Heapsort is an efficient algorithm and has its own uses, but quicksort beats it, if it is implemented correctly.

- The most commonly used application of heaps is priority queues.

- Same as with heaps, priority queues are of two types,
  Max-priority queue
  Min-priority queue
- A max priority queue has the following operations:

  Insert(S,x)

  Adds an element to set S ,i.e., S=S U {x}

  Maximum(S)
- Returns the largest element in the set.

  Extract max(S)
- Returns the largest element in S and deletes it from the set.

  Increase(S,x,k)
- Increases the value of x to k, and it is assumed that k ≥ x.
- Apart from these, a min-priority queue supports the operations MINIMUM, EXTRACT MIN, INSERT and DECREASE KEY.
- The procedure HEAP MAXIMUM on the max-priority queue implements the maximum operations in θ(1) time.

Heap–maximum (A):

```
1. return A[1]
```

The procedure HEAP–EXTRACT–MAX implements the EXTRACT–MAX operation.

Heap–extract–max (A):

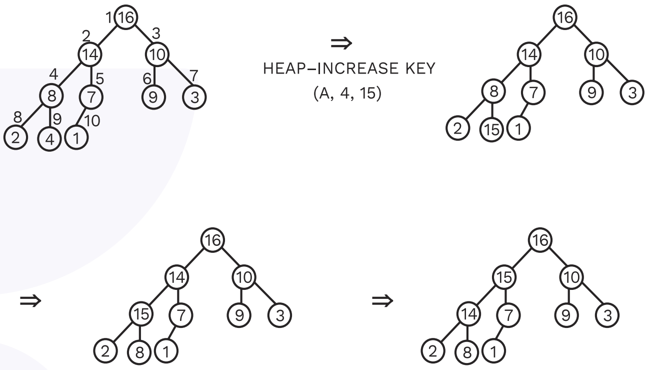
```
1. if A. heap–size <1
2. error “heap underflow”
3. max = A[1]
4. A[1] = A[A. heap–size]
5. A. heap–size = A. heap–size – 1
6. MAX–HEAPIFY (A, 1)
7. return max
```

- The HEAP-EXTRACT-MAX takes O(logn) since it performs O(1) the amount of time on the top of O(logn) time for MAX_HEAPIFY.
- The HEAP-INCREASE_KEY implements the INCREASE KEY operation.

Heap_increase_element (Arr, i, key):

```
1. if key < Arr[i]
2. ERROR /* since the key is lesser than the
   element */
3. Arr[i] ← key
4. while (i > 1) && [Arr[PARENT(i)] < Arr[i]]
5. exchange Arr[i] with Arr[PARENT(i)]
6. i ← PARENT(i)
```

The HEAP-INCREASE-KEY takes O(log n) time on an n element heap. The path from the updated node in the 3rd line to the root has length O(logn).



- The procedure MAX–HEAP–INSERT implements the INSERT operation.

Max–heap–insert (A, key):

```
1. A. heap–size = A. heap–size + 1
2. A [A. heap–size] = – ∞
3. HEAP–INCREASE–KEY (A, A. heap–size, key)
```

The running time of MAX–HEAP–INSERT on an n–element heap is O(logn).

Note:

Time–taken to perform some operations in Max–heap is given below

Fing max	Delete max	Insert	Increase	Decrease key	Find min	Search	Delete
O(1)	O(Logn)	O(Logn)	O(Logn)	O(Logn)	O(n)	O(n)	O(n)

Rack Your Brain

- Give an O(nlogk) – Time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. Hint : Use a min-heap for k–way merging.
- Write Pseudocode for the procedures HEAP MINIMUM, HEAP–EXTRACT–MIN, HEAP–DECREASE–KEY, and MIN–HEAP INSERT that implement a min–priority queue w ith a min–heap.

Previous Years' Question

The number of elements that can be sorted in O(logn) time using heap sort is **[CS 2013]**

(A) $\emptyset(1)$

(B) $\emptyset\left(\sqrt{\log n}\right)$

(C) $\emptyset\left(\frac{\log n}{\log \log n}\right)$

Solution: (C)

Previous Years' Question

Consider a max heap, represented by the array 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is **[CS 2015]**

(A) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35
(B) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
(C) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15
(D) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

Solution: (B)

Previous Years' Question

Consider the following array of elements (89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100). The minimum number of interchanges needed to convert it into a maxheap is: **[CS 2015]**

(A) 4	(B) 5
(C) 2	(D) 3

Solution: (D)

Insertion Sort

Insertion sort is an efficient algorithm to sort arrays with a small number of elements.

- It is the same as sorting and playing cards in your hand.
- Start with an empty hand and pick one card at a time from the cards that are faced down on the table.
- Check the card in the right hand with the cards in the left hand. Compare it with cards from right to left and place it in its correct position.
- At any given time, the cards in the left hand are sorted and are the top cards in the file.

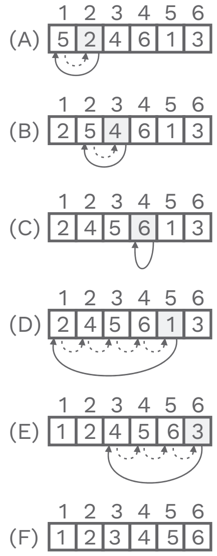
Pseudocode for insertion sort:

Insertion–sort (A):

```
for j from 2 to A. length
    Key = A[j]
    // Insert A[j] into sorted sequence A[1...j–1]
    i = j – 1
    While i > 0 & A[i] > Key
        A[i+1] = A[i]
        i = i –1
    A[i + 1] = Key
```

- INSERTION–SORT, takes as a parameter an array A[1--n] containing a sequence of n length that is to be sorted.
- The algorithm sorts the input within the array A, with at most O(1) or O(logn) numbers of them are stored outside the array.

- The operation of INSERTION–SORT on array A = {5, 2, 4, 6, 1, 3}



Analysis of Insertion Sort

Worst-case behaviour on an array of n length:

- Inner loop could be executed 'i' times
- 'i' swaps per loop ⇒ O(n^2) total swaps

Rack Your Brain

What sort of input leads to the worst case time complexity in insertion sort?

Best case behaviour on an array of length 'n'

- When the input array is sorted
- Inner loop executed 0 times ⇒ 0 swaps
- While condition is entry condition (always performed at least once)

So, O(n) comparisons are in the best case to verify the array is indeed sorted.

Binary Insertion Sort

BINARY–INSERTION–SORT (A, n)

```
    for j ← 2 to n
    insert A[j] Key into the (already sorted)
    sub–array A[1...j–1]
use binary search to find its right position
```

- Binary search will take O(logn) time. However, shifting the elements after insertion will still take O(n) time

Complexity:

O(nlogn) comparisons

O(n^2) swaps. So overall time complexity is O(n^2).

Rack Your Brain

Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

NOTE:

INSERTION_SORT gives time complexity depending on how nearly the input sequence is sorted. It is better for almost sorted sequences.

Previous Years' Question

Consider the following array

23	32	45	69	72	73	89	97

Which algorithm out of the following options uses the least number of comparisons (among the array elements) to sort the above array in ascending order?

[CS-2021 (Set-1)]

(A) Selection sort
(B) Merge sort
(C) Insertion sort
(D) Quicksort using the last element as pivot

Solution: (C)

Bubble sort:

Bubble sort is a popular yet inefficient sorting algorithm.

- It works by repeatedly swapping adjacent elements that are out of order.

Bubble sort (Arr)

1. for i ← 1 up to Arr. length – 1
2. for j ← Arr. length down to i + 1
3. if Arr [j] < Arr [j – 1]
4. swap Arr[j] and Arr[j–1]

Analysis:

Worst-Case

- The i-th iteration of the “for loop” of lines 1–4 will cause “n – i” iterations of the for loop of lines 2–4, each with constant time execution, so the worst–case running time of bubble sort is O(n^2)

Selection Sort

The selection sort finds the minimum repeatedly and places it at the beginning.

- It divides the array into two subarrays. One is sorted, and another one is unsorted.
- The minimum element is found in an unsorted array and is added at the end of the sorted array.

Example:

A = {1, 20, 6, 30, 42}

Find minimum element in A[0...4] and place it at the beginning.

	0	1	2	3	4
(A)	1	20	6	30	42

Find minimum element in A[1...4] and place it at the beginning of A[1...4]

	0	1	2	3	4
(B)	1	6	20	30	42

Find minimum element in A[2...4] and place it at the beginning of A[2...4]

	0	1	2	3	4
(C)	1	6	20	30	42

Find minimum element in A[3...4] and place it at the beginning of A[3...4]

	0	1	2	3	4
(D)	1	6	20	30	42

Pseudocode of selection sort:

```
Sleection–sort (A):
for j ← 1 to n–1
smallest ← j
for i ← j + 1 to n
if A[i] < A[smallest]
        smallest ← i
        Exchange A[j] ↔ A[smallest]
```

Time complexity:

- (n–1) comparisons are needed to find the minimum element from an array of ‘n’ elements.
- The size of an unsorted array decreases to (n–1) after the minimum element is placed in its proper position, and then (n–2) comparisons are required to find the minimum in the unsorted array.

Therefore,

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} \text{ comparisons}$$

and n swaps

$\therefore$ Time complexity = O(n^2)

Counting Sort

- Counting sort assumes every element in the n sized array is in range 0 to k, where k is an integer.

The algorithm:

Input

Array [1...n], where A[j] $\in$[1,2,3...k]

Output

The array [1...n] holds the sorted output and the array count[0...k] provides temporary working storage.

Counting_sort (array, array_size):

1. Max ← maximum element in the array and create an array Count of size max and initialize with zeroes
2. For j ← 0 to size
3. Find the total count of each unique element and store the count at jth index in count array
4. For i ← 1 to max
5. Find the cumulative sum and store it in count array itself
6. For j ← size down to 1 restore the elements to array
7. Decrement count_arr[arr[j]] and make arr[count_arr[arr[j]]]=j

Counting sort example:

	1	2	3	4	5	6	7	8
(A)	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
(C)	2	0	2	3	0	1

(A)

	0	1	2	3	4	5
(C)	2	2	4	7	7	8

(B)

	1	2	3	4	5	6	7	8
(B)							3	

	0	1	2	3	4	5
(C)	2	2	4	6	7	8

(C)

	1	2	3	4	5	6	7	8
(B)		0					3	

	0	1	2	3	4	5
(C)	1	2	4	6	7	8

(D)

	1	2	3	4	5	6	7	8
(B)		0				3	3	

	0	1	2	3	4	5
(C)	1	2	4	5	7	8

(E)

	1	2	3	4	5	6	7	8
(B)	0	0	2	2	3	3	3	5

(F)

Finding the maximum element takes O(n) time.

Creating an array and initializing it to zeroes takes O(k). therefrore, total time complexity=O(n+k) space complexity=O(k) for creating new array.

Since the elements are changed in-place,it is an inplace algorithm

Rack Your Brain

Why don't we always use counting sort?
Hint: Depends on the range K of elements.

Bucket Sort

- Counting sort assumes every element in the array of size n is in the range 0 to k, where k is an integer.
- Bucket sort assumes the input is given a uniform distribution function and runs in O(n) time.
- Same as counting sort, bucket sort is also fast since it assumes something about the input.
- Counting sort thinks that the input consists of n integers in a smaller range, whereas bucket sort assumes that the input is generated by a random process that distributes elements in the array uniformly and independently over an interval [0, 1).
- Bucket sort divides the [0, 1) interval into n equal-sized intervals (known as buckets) and distributes n input elements into the buckets.
- Since the numbers are uniformly and independently distributed over [0, 1), the chances of getting more elements in the same bucket are very, very less.

- To get the output, sort the elements in each bucket and list them in order to get the final sorted order.
- Bucket sort assumes that the input is an array A of n elements, and each element A[i] satisfies the condition $0 \leq A[i] \leq 1$.
- The code requires an auxiliary array B[0...n-1] of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

Bucket-sort (A):

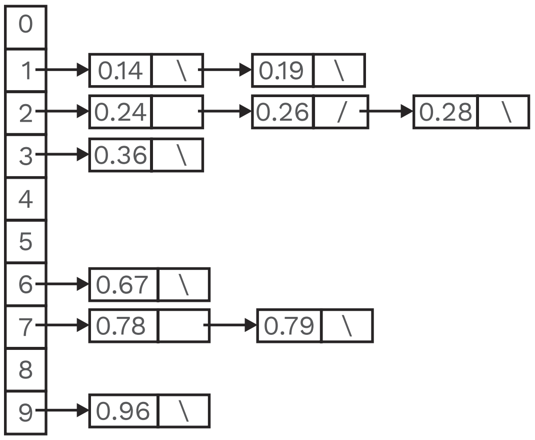
```
1. Let B[0....size-1] be a new array  created.
2. n = A.length
3. For itr<-0 to size-1
4. Make B[itr] empty
5. For itr<-1 to size
6. Do insert A[itr] into list B[n A[itr]]
7. For itr<-0 to size-1
8. Do sort list B[itr] with insertion-sort
9. Merge all lists B[0], B[1],.., B[size-1] together
   in order
```

Example:

Let array A have 10 elements as shown below:

A	0.79	0.19	0.36	0.28	0.78	0.96	0.24	0.14	0.26	0.67
	1	2	3	4	5	6	7	8	9	10

The array B[0...9] of sorted lists (buckets) after line 8 of the algorithm are shown below:



Time Complexity

Best Case

- In case if all the elements are uniformly distributed, then every insertion take O(1) time (assuming that every number that we are going to insert next has to be just inserted at the beginning of the list).
- We are going to insert all the number 'n' times, then the total time complexity is going to be O(n) for all the insertion, and then we have to scan all the number one time, so again O(n).
- Therefore, T(n) = O(n).

Worst-case:

- Let us assume that we have 'k' bucket, and since elements, are uniformly distributed. So, each bucket might get $\left(\frac{n}{k}\right)$ element as a list and if we are going to apply insertion sort with all the element, which are present in a bucket then we might have to compare with $\left(\frac{n}{k}\right)$ element then in that case time complexity is $O\left(\left(\frac{n}{k}\right)n\right) = O\left(n^2\right)$.
- Therefore, $O(n^2)$ is the worst-case time complexity.

Space complexity:

- If we assume that the number of bucket is 'k', then 'k' cells have been dedicated for the bucket, and for all the n-elements, we should have space of size n.
- So, space complexity = O(n+k).

Radix-sort:

- Alike counting and bucket sorting algorithms, even radix-sort assumes that the input has some information.
- It assumes that the input values are to be sorted from the base, i.e., means all numbers in the array have d digits.
- In radix sort, sorting done with each digit from the last to the first.
- A stable sort algorithm is used to sort all the numbers by least significant, then last but one, so on.
- Counting sort gives a time complexity of $O(nd) \approx O(n)$ for this procedure in radix sort.

Algorithm:

- Radix sort (A, d) /* Each key in A[1...n] is a d digit integer, and if the keys are not of d-digit, then we have to make 'd' digit number by appending zero. */
- For (i=1 to d) do
- Use a stable sorting
- Algorithm to sort A
- On digit i.
- /* digits are numbers 1 to d from right to left */

Example:

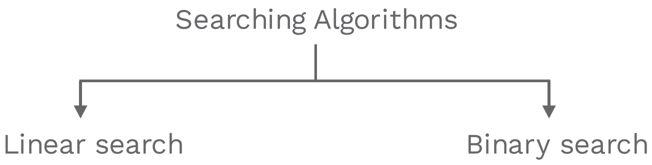
- The operation of radix sort on a list of seven 3-digit numbers is shown below:

329		720		720		329
457		355		329		355
657		436		436		436
839	⇨	457	⇨	839	⇨	457
436		657		355		657
720		329		457		720
355		839		657		839

- Let n d-digited numbers be given, in which each digit can take k values possible.
- RADIX-SORT sorts the input numbers correctly in θ(d(n+k)) time. If the intermediate stable algorithm used to sort the numbers takes d(n+k).
- When each digit is between 0 to k-1 and k is not too large, counting sort is the best choice. Then each pass over n d-digited numbers takes θ(n+k). Such passes are d. Therefore, radix sort takes θ(d(n+k)) time in total.
- When d is constant and K = O(n), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

Searching

- Searching is a process of locating a specific element among a set of elements.
- If the required element is found, the search is considered successful. Otherwise, it is unsuccessful.

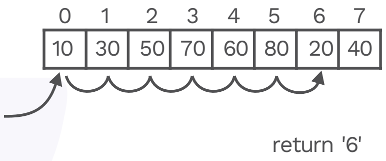


Linear Search

- Given an array arr[ ] of n elements and a search element 'x' to be searched in arr[ ]
- Begin with the leftmost element of arr[ ] and compare x with each element of arr[ ] one by one.
- Return the index if x matches an element.
- Return -1 if x does not match any of the elements.

Example:

Find 20, i.e., x = 20.



Time complexity of the linear search is written as O(n)

Note:
Binary search and hash tables are used more significantly in practice compared to Linear search since they allow faster searching.

Binary Search

- Binary search is one of the most efficient search algorithms available.
- It is based on the divide-conquer strategy.

Note:
Binary search algorithms can be applied only on sorted arrays.

- As a result, the elements must be arranged in a certain order.
- If the elements are numbers, use either ascending or descending order.
- If the elements are strings, use dictionary order.

To use binary search on an array that is not sorted.

- Sort the array first using the sorting technique.

- After that, employ the binary search algorithm.

```
Binary search (A, low, high, key)
if high < low
return (low-1)
mid ← $\left\lfloor low + \left(\frac{high - low}{2}\right)\right\rfloor$
if key = A[mid]
return mid
else if key < A[mid]
return Binary search (A, low, mid-1, key)
else
return Binary search (A, mid+1, high, key)
```

- Every call to the binary search algorithm is dividing the array into two equal half's and comparing the key with middle element. Based on the comparison with middle element, recursing to (left/right) half.

  $\therefore T(n) = T(n/2) + C$

  By master method

  $T(n) = \Theta(\log n)$

- $O(1)$ space in case of iterative implementation, and $O(\log n)$ recursion call stack space in case of recursive implementation.

Chapter Summary

Sorting Algorithms	Time Complexcity			Space Complexity Worst Case
	Best Case	Average Case	Worst Case	
Bubble Sort	$O(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n\log n)$	$\theta(n\log n)$	$O(n\log n)$	$O(n)$
Quick Sort	$O(n\log n)$	$\theta(n\log n)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n\log n)$	$\theta(n\log n)$	$O(n\log n)$	$O(n)$

Sorting Algorithms	In-place	Stable
Bubble Sort	Yes	Yes
Insertion Sort	Yes	Yes
Selection Sort	Yes	No
Merge Sort	No	Yes
Quick Sort	Yes	No

- The algorithm that can sort a dynamic input added while sorting is called online sorting algorithm.
  Eg: Insertion sort algorithm.
- The application and implementation details play a big role in determining which algorithm is best.
- Quicksort out performs heapsort, it is practically used for sorting large input arrays since its worst case time complexity is $O(n\log n)$.
- If stability and space issues are considered, then merge sort is the best choice.
- When the input array is nearly sorted or the input size is small, insertion sort is preferable.