SHELL SORT

1) A is an ordered integer array with 10 elements from small to large

Assume that array of A is → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

WORK STEP:

Gap = array.length / 2 = 5

While gap is bigger than zero (gap is 5)

Next position, according to gap, is 6
array[5] = 6
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 7
array[6] = 7
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 8
array[7] = 8
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 9
array[8] = 9
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 10
array[9] = 10
[1 2 3 4 5 6 7 8 9 10]

Gap = array.length / 2 = 2

While gap is bigger than zero (gap is 2)

Next position, according to gap, is 3
array[2] = 3
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 4
array[3] = 4
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 5
array[4] = 5
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 6
array[5] = 6
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 7
array[6] = 7
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 8
array[7] = 8
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 9
array[8] = 9
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 10
array[9] = 10
[1 2 3 4 5 6 7 8 9 10]

Gap = array.length / 2 = 1

While gap is bigger than zero (gap is 1)

Next position, according to gap, is 2
array[1] = 2
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 3
array[2] = 3
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 4
array[3] = 4
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 5
array[4] = 5
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 6
array[5] = 6
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 7
array[6] = 7
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 8
array[7] = 8
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 9
array[8] = 9
[1 2 3 4 5 6 7 8 9 10]

Next position, according to gap, is 10
array[9] = 10
[1 2 3 4 5 6 7 8 9 10]

2) B is an ordered integer array with 10 elements from large to small

Assume that array of B is → {9,8,7,6,5,4,3,2,1,0}

WORK STEP:

Gap = array.length / 2 = 5

While gap is bigger than zero (gap is 5)

Next position, according to gap, is 4
While next position index > gap-1 and array[index-gap] less than 4
Next position update -> (next position index - gap 0)
array[0] = 4
[4 8 7 6 5 9 3 2 1 0]

Next position, according to gap, is 3
While next position index > gap-1 and array[index-gap] less than 3
Next position update -> (next position index - gap 1)
array[1] = 3
[4 3 7 6 5 9 8 2 1 0]

Next position, according to gap, is 2
While next position index > gap-1 and array[index-gap] less than 2
Next position update -> (next position index - gap 2)
array[2] = 2
[4 3 2 6 5 9 8 7 1 0]

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 3)
array[3] = 1
[4 3 2 1 5 9 8 7 6 0]

Next position, according to gap, is 0
While next position index > gap-1 and array[index-gap] less than 0
Next position update -> (next position index - gap 4)
array[4] = 0
[4 3 2 1 0 9 8 7 6 5]

Gap = array.length / 2 = 2

While gap is bigger than zero (gap is 2)

Next position, according to gap, is 2
While next position index > gap-1 and array[index-gap] less than 2
Next position update -> (next position index - gap 0)
array[0] = 2
[2 3 4 1 0 9 8 7 6 5]

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 1)
array[1] = 1
[2 1 4 3 0 9 8 7 6 5]

Next position, according to gap, is 0
While next position index > gap-1 and array[index-gap] less than 0
Next position update -> (next position index - gap 2)
While next position index > gap-1 and array[index-gap] less than 0
Next position update -> (next position index - gap 0)
array[0] = 0
[0 1 2 3 4 9 8 7 6 5]

Next position, according to gap, is 9
array[5] = 9
[0 1 2 3 4 9 8 7 6 5]

Next position, according to gap, is 8
array[6] = 8
[0 1 2 3 4 9 8 7 6 5]

Next position, according to gap, is 7
While next position index > gap-1 and array[index-gap] less than 7
Next position update -> (next position index - gap 5)
array[5] = 7
[0 1 2 3 4 7 8 9 6 5]

Next position, according to gap, is 6
While next position index > gap-1 and array[index-gap] less than 6
Next position update -> (next position index - gap 6)
array[6] = 6
[0 1 2 3 4 7 6 9 8 5]

Next position, according to gap, is 5
While next position index > gap-1 and array[index-gap] less than 5
Next position update -> (next position index - gap 7)
While next position index > gap-1 and array[index-gap] less than 5
Next position update -> (next position index - gap 5)
array[5] = 5
[0 1 2 3 4 5 6 7 8 9]

Gap = array.length / 2 = 1

While gap is bigger than zero (gap is 1)

Next position, according to gap, is 1
array[1] = 1
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 2
array[2] = 2
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 3
array[3] = 3
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 4
array[4] = 4
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 5
array[5] = 5
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 6
array[6] = 6
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 7
array[7] = 7
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 8
array[8] = 8
[0 1 2 3 4 5 6 7 8 9]

Next position, according to gap, is 9
array[9] = 9
[0 1 2 3 4 5 6 7 8 9]

3) Assume that array of C is → {5,2,13,9,1,7,6,8,1,15,4,11}

WORK STEP:

Gap = array.length / 2 = 6

While gap is bigger than zero (gap is 6)

Next position, according to gap, is 6
array[6] = 6
[5 2 13 9 1 7 6 8 1 15 4 11]

Next position, according to gap, is 8
array[7] = 8
[5 2 13 9 1 7 6 8 1 15 4 11]

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 2)
array[2] = 1
[5 2 1 9 1 7 6 8 13 15 4 11]

Next position, according to gap, is 15
array[9] = 15
[5 2 1 9 1 7 6 8 13 15 4 11]

Next position, according to gap, is 4
array[10] = 4
[5 2 1 9 1 7 6 8 13 15 4 11]

Next position, according to gap, is 11
array[11] = 11
[5 2 1 9 1 7 6 8 13 15 4 11]

Gap = array.length / 2 = 2

While gap is bigger than zero (gap is 2)

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 0)
array[0] = 1
[1 2 5 9 1 7 6 8 13 15 4 11]

Next position, according to gap, is 9
array[3] = 9
[1 2 5 9 1 7 6 8 13 15 4 11]

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 2)
array[2] = 1
[1 2 1 9 5 7 6 8 13 15 4 11]

Next position, according to gap, is 7
While next position index > gap-1 and array[index-gap] less than 7
Next position update -> (next position index - gap 3)
array[3] = 7
[1 2 1 7 5 9 6 8 13 15 4 11]

Next position, according to gap, is 6
array[6] = 6
[1 2 1 7 5 9 6 8 13 15 4 11]

Next position, according to gap, is 8
While next position index > gap-1 and array[index-gap] less than 8

Next position update -> (next position index - gap 5)
array[5] = 8
[1 2 1 7 5 8 6 9 13 15 4 11]

Next position, according to gap, is 13
array[8] = 13
[1 2 1 7 5 8 6 9 13 15 4 11]

Next position, according to gap, is 15
array[9] = 15
[1 2 1 7 5 8 6 9 13 15 4 11]

Next position, according to gap, is 4
While next position index > gap-1 and array[index-gap] less than 4
Next position update -> (next position index - gap 8)

While next position index > gap-1 and array[index-gap] less than 4
Next position update -> (next position index - gap 6)

While next position index > gap-1 and array[index-gap] less than 4
Next position update -> (next position index - gap 4)
array[4] = 4
[1 2 1 7 4 8 5 9 6 15 13 11]

Next position, according to gap, is 11
While next position index > gap-1 and array[index-gap] less than 11
Next position update -> (next position index - gap 9)
array[9] = 11
[1 2 1 7 4 8 5 9 6 11 13 15]

Gap = array.length / 2 = 1

While gap is bigger than zero (gap is 1)

Next position, according to gap, is 2
array[1] = 2
[1 2 1 7 4 8 5 9 6 11 13 15]

Next position, according to gap, is 1
While next position index > gap-1 and array[index-gap] less than 1
Next position update -> (next position index - gap 1)
array[1] = 1
[1 1 2 7 4 8 5 9 6 11 13 15]

Next position, according to gap, is 7
array[3] = 7
[1 1 2 7 4 8 5 9 6 11 13 15]

Next position, according to gap, is 4
While next position index > gap-1 and array[index-gap] less than 4
Next position update -> (next position index - gap 3)
array[3] = 4

[1 1 2 4 7 8 5 9 6 11 13 15]

Next position, according to gap, is 8
array[5] = 8
[1 1 2 4 7 8 5 9 6 11 13 15]

Next position, according to gap, is 5
While next position index > gap-1 and array[index-gap] less than 5
Next position update -> (next position index - gap 5)

While next position index > gap-1 and array[index-gap] less than 5
Next position update -> (next position index - gap 4)
array[4] = 5
[1 1 2 4 5 7 8 9 6 11 13 15]

Next position, according to gap, is 9
array[7] = 9
[1 1 2 4 5 7 8 9 6 11 13 15]

Next position, according to gap, is 6
While next position index > gap-1 and array[index-gap] less than 6
Next position update -> (next position index - gap 7)

While next position index > gap-1 and array[index-gap] less than 6
Next position update -> (next position index - gap 6)

While next position index > gap-1 and array[index-gap] less than 6
Next position update -> (next position index - gap 5)
array[5] = 6
[1 1 2 4 5 6 7 8 9 11 13 15]

Next position, according to gap, is 11
array[9] = 11
[1 1 2 4 5 6 7 8 9 11 13 15]

Next position, according to gap, is 13
array[10] = 13
[1 1 2 4 5 6 7 8 9 11 13 15]

Next position, according to gap, is 15
array[11] = 15
[1 1 2 4 5 6 7 8 9 11 13 15]

4) Assume that array of D is → {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

WORK STEP:

Gap = array.length / 2 = 6

While gap is bigger than zero (gap is 6)

Next position, according to gap, is C
While next position index > gap-1 and array[index-gap] less than C
Next position update -> (next position index - gap 0)
array[0] = C
[C B I M H Q S L R E P K]

Next position, according to gap, is L
array[7] = L
[C B I M H Q S L R E P K]

Next position, according to gap, is R
array[8] = R
[C B I M H Q S L R E P K]

Next position, according to gap, is E
While next position index > gap-1 and array[index-gap] less than E
Next position update -> (next position index - gap 3)
array[3] = E
[C B I E H Q S L R M P K]

Next position, according to gap, is P
array[10] = P
[C B I E H Q S L R M P K]

Next position, according to gap, is K
While next position index > gap-1 and array[index-gap] less than K
Next position update -> (next position index - gap 5)
array[5] = K
[C B I E H K S L R M P Q]

Gap = array.length / 2 = 2

While gap is bigger than zero (gap is 2)

Next position, according to gap, is I
array[2] = I
[C B I E H K S L R M P Q]

Next position, according to gap, is E
array[3] = E
[C B I E H K S L R M P Q]

Next position, according to gap, is H

While next position index > gap-1 and array[index-gap] less than H
Next position update -> (next position index - gap 2)
array[2] = H
[C B H E I K S L R M P Q]

Next position, according to gap, is K
array[5] = K
[C B H E I K S L R M P Q]

Next position, according to gap, is S
array[6] = S
[C B H E I K S L R M P Q]

Next position, according to gap, is L
array[7] = L
[C B H E I K S L R M P Q]

Next position, according to gap, is R
While next position index > gap-1 and array[index-gap] less than R
Next position update -> (next position index - gap 6)
array[6] = R
[C B H E I K R L S M P Q]

Next position, according to gap, is M
array[9] = M
[C B H E I K R L S M P Q]

Next position, according to gap, is P
While next position index > gap-1 and array[index-gap] less than P
Next position update -> (next position index - gap 8)

While next position index > gap-1 and array[index-gap] less than P
Next position update -> (next position index - gap 6)
array[6] = P
[C B H E I K P L R M S Q]

Next position, according to gap, is Q
array[11] = Q
[C B H E I K P L R M S Q]

Gap = array.length / 2 = 1

While gap is bigger than zero (gap is 1)

Next position, according to gap, is B
While next position index > gap-1 and array[index-gap] less than B
Next position update -> (next position index - gap 0)
array[0] = B
[B C H E I K P L R M S Q]

Next position, according to gap, is H

array[2] = H
[B C H E I K P L R M S Q]

Next position, according to gap, is E
While next position index > gap-1 and array[index-gap] less than E
Next position update -> (next position index - gap 2)
array[2] = E
[B C E H I K P L R M S Q]

Next position, according to gap, is I
array[4] = I
[B C E H I K P L R M S Q]

Next position, according to gap, is K
array[5] = K
[B C E H I K P L R M S Q]

Next position, according to gap, is P
array[6] = P
[B C E H I K P L R M S Q]

Next position, according to gap, is L
While next position index > gap-1 and array[index-gap] less than L
Next position update -> (next position index - gap 6)
array[6] = L
[B C E H I K L P R M S Q]

Next position, according to gap, is R
array[8] = R
[B C E H I K L P R M S Q]

Next position, according to gap, is M
While next position index > gap-1 and array[index-gap] less than M
Next position update -> (next position index - gap 8)
While next position index > gap-1 and array[index-gap] less than M
Next position update -> (next position index - gap 7)
array[7] = M
[B C E H I K L M P R S Q]

Next position, according to gap, is S
array[10] = S
[B C E H I K L M P R S Q]

Next position, according to gap, is Q

While next position index > gap-1 and array[index-gap] less than Q
Next position update -> (next position index - gap 10)

While next position index > gap-1 and array[index-gap] less than Q
Next position update -> (next position index - gap 9)
array[9] = Q

[B C E H I K L M P Q R S]

MERGE SORT

1) A is an ordered integer array with 10 elements from small to large

Assume that array of A is → {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

WORK STEP:

- 1) Split the selected array
[0,1,2,3,4] [5,6,7,8,9]

Right Subarray

- Select the left subarray
[0,1,2,3,4]

- 2) Split the selected array
[0,1,2] [3,4]

- Select the left subarray
[0,1,2]

- 3) Split the selected array
[0,1] [2]

-Select the left subarray
[0,1]

- 4) Split the selected array
[0] [1]

(4. Step)
- Last two arrays can not split because its length is 1
- Merge selected arrays back together, in sorted order
-Select the minimum of the two values
-Add the selected value to the sorted array
- [0]
- [0,1] (Finished merging)

(3. Step)
- Last array can not split because its length is 1
- Merge selected arrays back together, in sorted order
- Select the number and add the sorted array
- a)Select the smaleest value form the front of each array
- (0 and 2) → [0,1] [2]
- Select minimum of the two values
[0] [1] [2]
jump a th step
[0,1] [2]
jump a th step

[0,1,2]

(2. Step)
- Select the rigth subarray
- Split the selected array
[3] [4]
- The array can not split
-Select the minimum of two values
- Add the selected value to the sorted array
[3]
- When array becoems empty, copy all values from the remaininh array into the sorted array
[3,4]

(1. Step left subarray)
- Merge selected arrays back together, in sorted order
- Select the smallest value from the front of arrays.
- Add the selected value to the sorted array
[0… [1,2] [3,4]
[0,1… [1] [3,4]
[0,1,2… [3,4]
- When one list becoems empty, copy all values from the remaining array into the sorted array
[0,1,2,3,4]

<u>Right Subarray</u>

- Select right subarray
- Split the selected array
[5,6,7] [8,9]
- Select the left subarray
- Sptlit the selected array
[5,6] [7]
- Select the leftsubarray
[5] [6]
- Merge selected arrays back together
- Select the minimum of values and add the selected value to the sorted array
- When one array becomes empty, copy all values from the remaining array into the sorted array
[5,6]
- Finish merging for leftsubarray
- An array of length 1 can not split
[7]

- Select the smallest value from the front of each array
- If no expression is provided, the element of the other array is taken.
[5… [6] [7]
[5,6… [7]
[5,6,7]

- Select last right subarray

[8,9]

- Split selected array

[8] [9]

- Select the minimum of the two values

- When the one array becomes emtpy, copt all values from the remaining array into the sorted array

[8,9]

- Merge sekected array back together

- Select and add smallest value from the front of each array

[5… [6,7] [8,9]
[5,6… [7] [8,9]
[5,6,7… [8,9]
[5,6,7,8… [9]
[5,6,7,8,9]

- Finish merging

- Select and add the smallest value from the front of each array to sorted array

-
[0… [1,2,3,4] [5,6,7,8,9]
[0,1… [2,3,4] [5,6,7,8,9]
[0,1,2… [3,4] [5,6,7,8,9]
[0,1,2,3… [4] [5,6,7,8,9]
[0,1,2,3,4… [5,6,7,8,9]
[0,1,2,3,4,5… [6,7,8,9]
[0,1,2,3,4,5,6… [7,8,9]
[0,1,2,3,4,5,6,7… [8,9]
[0,1,2,3,4,5,6,7,8… [9]
[0,1,2,3,4,5,6,7,8,9]

- Finish merging

34 Replacement

2) B is an ordered integer array with 10 elements from large to small

Assume that array of B is → {9,8,7,6,5,4,3,2,1,0}

WORK STEP:

Split the selected array

[9,8,7,6,5] [4,3,2,1,0]

Select the left subarray

[9,8,7,6,5]

Split the selected array

[9,8,7] [6,5]

Select the left subarray

[9,8,7]

Split the selected array

[9,8] [7]

Select the left subarray

[9,8]

Split the selected array
[9] [8]
Arrays can not be splitted

Select the minimum of the two values
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[8… [9]
[8,9]
Finish merging

[7] can not be splitted

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[7… [8,9]
[7,8… [9]
[7,8,9]
Finish merging

Select right subarray → [6,5]
Split the selected subarray → [6] [5]
Select minimum of two values and add the value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[5… [6]
[5,6]
Finish merging

Select the smallest value from the front of each array
Select the minimum of the two values
Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[5… [7,8,9] [6]
[5,6… [7,8,9]
[5,6,7… [8,9]
[5,6,7,8… [9]
[5,6,7,8,9]

Split the right subarray of main array → [4,3,2] [1,0]
Split the left subarray → [4,3,2]
Select the left subarray → [4,3] [2]
Select the left subarray → [4,3]
Split the left subarray → [4] [3]
Array can not be splitted

Select minimum of the two values
Add selected value to the sorted array → [3]
When the one array is empty, copy all values from the remaining array into the sorted array → [3,4]
Finish merging

Array can not be splitted ([2])
Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[2… [3,4]
[2,3… [4]
[2,3,4]
Finish merging

Select the right subarray → [1,0]
Split the selected array → [1] [0]
Array can not be splitted

Select minimum of the two values
Add selected value to the sorted array → [0]
When the one array is empty, copy all values from the remaining array into the sorted array
[0,1]
Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[0… [2,3,4] [1]
[0,1… [2,3,4]
[0,1,2… [3,4]
[0,1,2,3… [4]
[0,1,2,3,4]
Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[0… [1,2,3,4] [5,6,7,8,9]
[0,1… [2,3,4] [5,6,7,8,9]
[0,1,2… [3,4] [5,6,7,8,9]
[0,1,2,3… [4] [5,6,7,8,9]
[0,1,2,3,4… [5,6,7,8,9]
[0,1,2,3,4,5… [6,7,8,9]
[0,1,2,3,4,5,6… [7,8,9]
[0,1,2,3,4,5,6,7… [8,9]
[0,1,2,3,4,5,6,7,8… [9]
[0,1,2,3,4,5,6,7,8,9]
Finish merging

34 Replacement

3) Assume that array of C is → {5,2,13,9,1,7,6,8,1,15,4,11}

WORK STEP:

Split selected array → [5,2,13,9,1,7] [6,8,1,15,4,11]
Select left subarray → [5,2,13,9,1,7]
Split selected array → [5,2,13] [9,1,7]
Select left subarray → [5,2,13]
Split selected array → [5,2] [13]
Select left subarray → [5] [2]
Selected arrays can not be splitted

Select minimum of the two values → [2]
Add the selected value to the sorted array
When one array becomes empty, copy all values from the remaining array into the sorted array → [2,5]
Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[2… [5] [13]
[2,5… [13]
[2,5,13]
Finish merging

Select the right subarray → [9,1,7]
Split selected array → [9,1] [7]
Select left subarray → [9,1]
Split selected array → [9] [1]
Arrays can not be splitted

Select minimum of the two values → [1]
Add the selected value to the sorted array
When one array becomes empty, copy all values from the remaining array into the sorted array → [1,9]
Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array
[1… [9] [7]
[1,7… [9]
[1,7,9]
Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array

When the one array is empty, copy all values from the remaining array into the sorted array

[1… [2,5,13] [7,9]
[1,2… [5,13] [7,9]
[1,2,5… [13] [7,9]
[1,2,5,7… [13] [9]
[1,2,5,7,9… [13]
[1,2,5,7,9,13]

Finish merging

Select the right subarray → [6,8,1,15,4,11]
Split the selected array → [6,8,1] [15,4,11]
Select the left subarray → [6,8,1]
Split the selected array → [6,8] [1]
Select the left subarray → [6,8]
Split the selected array → [6] [8]

Select the min value of two values → 6
Add the selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array → [6,8]
Finish merging

Select the right subarray → [1]
[1] → can not be splitted

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array

[1… [6,8]
[1,6… [8]
[1,6,8]

Finish merging

Select the right subarray → [15,4,11]
Split the selected array → [15,4] [11]
Select the leftsubarray → [15,4]
Split the selected array → [15] [4]
Arrays can not be splitted

Select min of the two values → 4

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array → [4,15]
Finish merging

Select the smallest value from the front of each array

And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array

[4… [15] [11]
[4,11… [15]
[4,11,15]

Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array

[1… [6,8] [4,11,15]
[1,4… [6,8] [11,15]
[1,4,6… [8] [11,15]
[1,4,6,8… [11,15]
[1,4,6,8,11… [15]
[1,4,6,8,11,15]

Finish merging

Select the smallest value from the front of each array
And Add selected value to the sorted array
When the one array is empty, copy all values from the remaining array into the sorted array

[1… [2,5,7,9,13] [1,4,6,8,11,15]
[1,1… [2,5,7,9,13] [4,6,8,11,15]
[1,1,2… [5,7,9,13] [4,6,8,11,15]
[1,1,2,4… [5,7,9,13] [6,8,11,15]
[1,1,2,4,5… [7,9,13] [6,8,11,15]
[1,1,2,4,5,6… [7,9,13] [8,11,15]
[1,1,2,4,5,6,7… [9,13] [8,11,15]
[1,1,2,4,5,6,7,8… [9,13] [11,15]
[1,1,2,4,5,6,7,8,9… [13] [11,15]
[1,1,2,4,5,6,7,8,9,11… [13] [15]
[1,1,2,4,5,6,7,8,9,11,13… [15]
[1,1,2,4,5,6,7,8,9,11,13,15]

Finish merging

44 Replacement

4) Assume that array of D is → {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

WORK STEP:

Split the selected array → [S B I M H Q] [C L R E P K]
Select left subarray → [S B I M H Q]
Split the selected array → [S B I] [M H Q]
Select left subarray → [SBI]
Split the selected array → [S B] [I]
Select left subarray → [S] [B]
Select smallest value of front of array and push sorted array
→ [B,S]

Select smallest value of front of array and push sorted array
→ [B,I,S]

Select left subarray → [M,H,Q]
Split the selected array → [M,H] [Q]
Select left subarray → [M] [H]
Select smallest value of front of array and push sorted array
→ [H,M]
Select smallest value of front of array and push sorted array
→ [H,M,Q]

Select smallest value of front of array and push sorted array([B,I,S] vs [H,M,Q])
→ [B, H, I, M, Q, S]

Split the selected array → [C L R] [E P K]
Select left subarray → [CLR]
Split the selected array → [CL] [R]
Select left subarray → [C] [L]
Select smallest value of front of array and push sorted array
→ [C,L]

Select smallest value of front of array and push sorted array
→ [C L R]

Select left subarray → [E P K]
Split the selected array → [E P] [K]
Select left subarray → [E] [P]
Select smallest value of front of array and push sorted array
→ [E P]
Select smallest value of front of array and push sorted array
→ [E K P]

Select smallest value of front of array and push sorted array([C L R] vs [E K P])
→ [C E K L P R]

Select smallest value of front of array and push sorted array
([B, H, I, M, Q, S] vs [C E K L P R])

→ [B C E H I K L M P Q R S]

44 Replacement

HEAP SORT

1) A is an ordered integer array with 10 elements from small to large

Assume that array of A is → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

WORK STEP:

In this algorithm we first build the heap using the given elements
We create a Max Heap to sort he elements in ascending order
Building heap:
Array -> 1 2 3 4 5 6 7 8 9 10

Child: 1
Parent: 0
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 2 1 3 4 5 6 7 8 9 10
New Child: 0
New Parent: 0

Child: 2
Parent: 0
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 3 1 2 4 5 6 7 8 9 10
New Child: 0
New Parent: 0

Child: 3
Parent: 1
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 3 4 2 1 5 6 7 8 9 10
New Child: 1
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 4 3 2 1 5 6 7 8 9 10
New Child: 0
New Parent: 0

Child: 4
Parent: 1
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 4 5 2 1 3 6 7 8 9 10
New Child: 1
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 5 4 2 1 3 6 7 8 9 10
New Child: 0
New Parent: 0

Child: 5
Parent: 2
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 5 4 6 1 3 2 7 8 9 10
New Child: 2
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 6 4 5 1 3 2 7 8 9 10
New Child: 0
New Parent: 0

Child: 6
Parent: 2
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 6 4 7 1 3 2 5 8 9 10
New Child: 2
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 7 4 6 1 3 2 5 8 9 10
New Child: 0
New Parent: 0

Child: 7
Parent: 3
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 7 4 6 8 3 2 5 1 9 10
New Child: 3
New Parent: 1

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 7 8 6 4 3 2 5 1 9 10
New Child: 1
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 8 7 6 4 3 2 5 1 9 10
New Child: 0
New Parent: 0

Child: 8
Parent: 3
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 8 7 6 9 3 2 5 1 4 10
New Child: 3
New Parent: 1

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 8 9 6 7 3 2 5 1 4 10
New Child: 1

New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 9 8 6 7 3 2 5 1 4 10
New Child: 0
New Parent: 0

Child: 9
Parent: 4
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 9 8 6 7 10 2 5 1 4 3
New Child: 4
New Parent: 1

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 9 10 6 7 8 2 5 1 4 3
New Child: 1
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 10 9 6 7 8 2 5 1 4 3
New Child: 0
New Parent: 0

Once the heap is created we swap the root node with the last node anddelete the last node from the heap

Swapping root and last element of heap
Array -> 3 9 6 7 8 2 5 1 4 10

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 3 < 9
assign max child index to parent
Array -> 9 3 6 7 8 2 5 1 4 10

left child index: 3
right child index: 4
max child index: 4
swapping element that has parent and maxchild index
because 3 < 8
assign max child index to parent
Array -> 9 8 6 7 3 2 5 1 4 10

Swapping root and last element of heap
Array -> 4 8 6 7 3 2 5 1 9 10

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 4 < 8
assign max child index to parent
Array -> 8 4 6 7 3 2 5 1 9 10

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 4 < 7
assign max child index to parent
Array -> 8 7 6 4 3 2 5 1 9 10

left child index: 7
right child index: 8
max child index: 7
Swapping root and last element of heap
Array -> 1 7 6 4 3 2 5 8 9 10

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 7
assign max child index to parent
Array -> 7 1 6 4 3 2 5 8 9 10

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 1 < 4
assign max child index to parent
Array -> 7 4 6 1 3 2 5 8 9 10

Swapping root and last element of heap
Array -> 5 4 6 1 3 2 7 8 9 10

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 5 < 6
assign max child index to parent
Array -> 6 4 5 1 3 2 7 8 9 10

left child index: 5
right child index: 6

max child index: 5
Swapping root and last element of heap
Array -> 2 4 5 1 3 6 7 8 9 10

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 2 < 5
assign max child index to parent
Array -> 5 4 2 1 3 6 7 8 9 10

Swapping root and last element of heap
Array -> 3 4 2 1 5 6 7 8 9 10

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 3 < 4
assign max child index to parent
Array -> 4 3 2 1 5 6 7 8 9 10

left child index: 3
right child index: 4
max child index: 3
Swapping root and last element of heap
Array -> 1 3 2 4 5 6 7 8 9 10

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 3
assign max child index to parent
Array -> 3 1 2 4 5 6 7 8 9 10

Swapping root and last element of heap
Array -> 2 1 3 4 5 6 7 8 9 10

left child index: 1
right child index: 2
max child index: 1
Swapping root and last element of heap
Array -> 1 2 3 4 5 6 7 8 9 10

Swapping root and last element of heap
Array -> 1 2 3 4 5 6 7 8 9 10

29 Replacement
39 Compare

2) B is an ordered integer array with 10 elements from large to small

Assume that array of A is → {9,8,7,6,5,4,3,2,1,0}

WORK STEP:

In this algorithm we first build the heap using the given elements
We create a Max Heap to sort he elements in ascending order
Building heap:
Array -> 9 8 7 6 5 4 3 2 1 0

The array ascending ordered array...

Once the heap is created we swap the root node with the last node anddelete the last node from the heap

Swapping root and last element of heap
Array -> 0 8 7 6 5 4 3 2 1 9

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 0 < 8
assign max child index to parent
Array -> 8 0 7 6 5 4 3 2 1 9

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 0 < 6
assign max child index to parent
Array -> 8 6 7 0 5 4 3 2 1 9

left child index: 7
right child index: 8
max child index: 7
swapping element that has parent and maxchild index
because 0 < 2
assign max child index to parent
Array -> 8 6 7 2 5 4 3 0 1 9

Swapping root and last element of heap
Array -> 1 6 7 2 5 4 3 0 8 9

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 1 < 7

assign max child index to parent
Array -> 7 6 1 2 5 4 3 0 8 9

left child index: 5
right child index: 6
max child index: 5
swapping element that has parent and maxchild index
because 1 < 4
assign max child index to parent
Array -> 7 6 4 2 5 1 3 0 8 9

Swapping root and last element of heap
Array -> 0 6 4 2 5 1 3 7 8 9

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 0 < 6
assign max child index to parent
Array -> 6 0 4 2 5 1 3 7 8 9

left child index: 3
right child index: 4
max child index: 4
swapping element that has parent and maxchild index
because 0 < 5
assign max child index to parent
Array -> 6 5 4 2 0 1 3 7 8 9

Swapping root and last element of heap
Array -> 3 5 4 2 0 1 6 7 8 9

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 3 < 5
assign max child index to parent
Array -> 5 3 4 2 0 1 6 7 8 9

left child index: 3
right child index: 4
max child index: 3
Swapping root and last element of heap
Array -> 1 3 4 2 0 5 6 7 8 9

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 1 < 4

assign max child index to parent
Array -> 4 3 1 2 0 5 6 7 8 9

Swapping root and last element of heap
Array -> 0 3 1 2 4 5 6 7 8 9

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 0 < 3
assign max child index to parent
Array -> 3 0 1 2 4 5 6 7 8 9

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 0 < 2
assign max child index to parent
Array -> 3 2 1 0 4 5 6 7 8 9

Swapping root and last element of heap
Array -> 0 2 1 3 4 5 6 7 8 9

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 0 < 2
assign max child index to parent
Array -> 2 0 1 3 4 5 6 7 8 9

Swapping root and last element of heap
Array -> 1 0 2 3 4 5 6 7 8 9

left child index: 1
right child index: 2
max child index: 1
Swapping root and last element of heap
Array -> 0 1 2 3 4 5 6 7 8 9

Swapping root and last element of heap
Array -> 0 1 2 3 4 5 6 7 8 9

22 Replacement
12 Compare

3) C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}

WORK STEP:

In this algorithm we first build the heap using the given elements
We create a Max Heap to sort he elements in ascending order
Building heap:

Array -> 5 2 13 9 1 7 6 8 1 15 4 11
Child: 1
Parent: 0

Child: 2
Parent: 0
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 2 5 9 1 7 6 8 1 15 4 11
New Child: 0
New Parent: 0

Child: 3
Parent: 1
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 9 5 2 1 7 6 8 1 15 4 11
New Child: 1
New Parent: 0

Child: 4
Parent: 1
Child: 5
Parent: 2
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 9 7 2 1 5 6 8 1 15 4 11
New Child: 2
New Parent: 0

Child: 6
Parent: 2
Child: 7
Parent: 3
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 9 7 8 1 5 6 2 1 15 4 11
New Child: 3
New Parent: 1

Child: 8
Parent: 3
Child: 9

Parent: 4
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 9 7 8 15 5 6 2 1 1 4 11
New Child: 4
New Parent: 1

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 13 15 7 8 9 5 6 2 1 1 4 11
New Child: 1
New Parent: 0

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 15 13 7 8 9 5 6 2 1 1 4 11
New Child: 0
New Parent: 0

Child: 10
Parent: 4
Child: 11
Parent: 5
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 15 13 7 8 9 11 6 2 1 1 4 5
New Child: 5
New Parent: 2

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> 15 13 11 8 9 7 6 2 1 1 4 5
New Child: 2
New Parent: 0

Once the heap is created we swap the root node with the last node anddelete the last node from the heap

Swapping root and last element of heap
Array -> 5 13 11 8 9 7 6 2 1 1 4 15

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 5 < 13
assign max child index to parent
Array -> 13 5 11 8 9 7 6 2 1 1 4 15

left child index: 3
right child index: 4
max child index: 4

swapping element that has parent and maxchild index
because 5 < 9
assign max child index to parent
Array -> 13 9 11 8 5 7 6 2 1 1 4 15

left child index: 9
right child index: 10
max child index: 10
Swapping root and last element of heap
Array -> 4 9 11 8 5 7 6 2 1 1 13 15

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 4 < 11
assign max child index to parent
Array -> 11 9 4 8 5 7 6 2 1 1 13 15

left child index: 5
right child index: 6
max child index: 5
swapping element that has parent and maxchild index
because 4 < 7
assign max child index to parent
Array -> 11 9 7 8 5 4 6 2 1 1 13 15

Swapping root and last element of heap
Array -> 1 9 7 8 5 4 6 2 1 11 13 15

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 9
assign max child index to parent
Array -> 9 1 7 8 5 4 6 2 1 11 13 15

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 1 < 8
assign max child index to parent
Array -> 9 8 7 1 5 4 6 2 1 11 13 15

left child index: 7
right child index: 8
max child index: 7
swapping element that has parent and maxchild index
because 1 < 2
assign max child index to parent

Array -> 9 8 7 2 5 4 6 1 1 11 13 15

Swapping root and last element of heap
Array -> 1 8 7 2 5 4 6 1 9 11 13 15

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 8
assign max child index to parent
Array -> 8 1 7 2 5 4 6 1 9 11 13 15

left child index: 3
right child index: 4
max child index: 4
swapping element that has parent and maxchild index
because 1 < 5
assign max child index to parent
Array -> 8 5 7 2 1 4 6 1 9 11 13 15

Swapping root and last element of heap
Array -> 1 5 7 2 1 4 6 8 9 11 13 15

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 1 < 7
assign max child index to parent
Array -> 7 5 1 2 1 4 6 8 9 11 13 15

left child index: 5
right child index: 6
max child index: 6
swapping element that has parent and maxchild index
because 1 < 6
assign max child index to parent
Array -> 7 5 6 2 1 4 1 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 5 6 2 1 4 7 8 9 11 13 15

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 1 < 6
assign max child index to parent
Array -> 6 5 1 2 1 4 7 8 9 11 13 15

left child index: 5

right child index: 6
max child index: 5
swapping element that has parent and maxchild index
because 1 < 4
assign max child index to parent
Array -> 6 5 4 2 1 1 7 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 5 4 2 1 6 7 8 9 11 13 15

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 5
assign max child index to parent
Array -> 5 1 4 2 1 6 7 8 9 11 13 15

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because 1 < 2
assign max child index to parent
Array -> 5 2 4 1 1 6 7 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 2 4 1 5 6 7 8 9 11 13 15

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because 1 < 4
assign max child index to parent
Array -> 4 2 1 1 5 6 7 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 2 1 4 5 6 7 8 9 11 13 15

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because 1 < 2
assign max child index to parent
Array -> 2 1 1 4 5 6 7 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 1 2 4 5 6 7 8 9 11 13 15

left child index: 1

right child index: 2
max child index: 1
Swapping root and last element of heap
Array -> 1 1 2 4 5 6 7 8 9 11 13 15

Swapping root and last element of heap
Array -> 1 1 2 4 5 6 7 8 9 11 13 15

38 Replacement
26 Compare

4) D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

WORK STEP:

In this algorithm we first build the heap using the given elements
We create a Max Heap to sort he elements in ascending order
Building heap:
Array -> S B I M H Q C L R E P K
Child: 1
Parent: 0
Child: 2
Parent: 0
Child: 3
Parent: 1
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S M I B H Q C L R E P K
New Child: 1
New Parent: 0

Child: 4
Parent: 1
Child: 5
Parent: 2
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S M Q B H I C L R E P K
New Child: 2
New Parent: 0

Child: 6
Parent: 2
Child: 7
Parent: 3
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S M Q L H I C B R E P K
New Child: 3
New Parent: 1

Child: 8
Parent: 3
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S M Q R H I C B L E P K
New Child: 3
New Parent: 1

while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S R Q M H I C B L E P K
New Child: 1
New Parent: 0

Child: 9
Parent: 4
Child: 10
Parent: 4
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S R Q M P I C B L E H K
New Child: 4
New Parent: 1

Child: 11
Parent: 5
while array[parent] < table[child]
swapping array[parent] and array[child]
Array -> S R Q M P K C B L E H I
New Child: 5
New Parent: 2

Once the heap is created we swap the root node with the last node anddelete the last node from the heap

Swapping root and last element of heap
Array -> I R Q M P K C B L E H S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because I < R
assign max child index to parent
Array -> R I Q M P K C B L E H S

left child index: 3
right child index: 4
max child index: 4
swapping element that has parent and maxchild index
because I < P
assign max child index to parent

Array -> R P Q M I K C B L E H S

left child index: 9
right child index: 10
max child index: 10
Swapping root and last element of heap
Array -> H P Q M I K C B L E R S

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because H < Q
assign max child index to parent
Array -> Q P H M I K C B L E R S

left child index: 5
right child index: 6
max child index: 5
swapping element that has parent and maxchild index
because H < K
assign max child index to parent
Array -> Q P K M I H C B L E R S

Swapping root and last element of heap
Array -> E P K M I H C B L Q R S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because E < P
assign max child index to parent
Array -> P E K M I H C B L Q R S

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because E < M
assign max child index to parent
Array -> P M K E I H C B L Q R S

left child index: 7
right child index: 8
max child index: 8
swapping element that has parent and maxchild index
because E < L
assign max child index to parent
Array -> P M K L I H C B E Q R S

Swapping root and last element of heap

Array -> E M K L I H C B P Q R S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because E < M
assign max child index to parent
Array -> M E K L I H C B P Q R S

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because E < L
assign max child index to parent
Array -> M L K E I H C B P Q R S

left child index: 7
right child index: 8
max child index: 7
Swapping root and last element of heap
Array -> B L K E I H C M P Q R S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because B < L
assign max child index to parent
Array -> L B K E I H C M P Q R S

left child index: 3
right child index: 4
max child index: 4
swapping element that has parent and maxchild index
because B < I
assign max child index to parent
Array -> L I K E B H C M P Q R S

Swapping root and last element of heap
Array -> C I K E B H L M P Q R S

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because C < K
assign max child index to parent
Array -> K I C E B H L M P Q R S

left child index: 5

right child index: 6
max child index: 5
swapping element that has parent and maxchild index
because C < H
assign max child index to parent
Array -> K I H E B C L M P Q R S

Swapping root and last element of heap
Array -> C I H E B K L M P Q R S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because C < I
assign max child index to parent
Array -> I C H E B K L M P Q R S

left child index: 3
right child index: 4
max child index: 3
swapping element that has parent and maxchild index
because C < E
assign max child index to parent
Array -> I E H C B K L M P Q R S

Swapping root and last element of heap
Array -> B E H C I K L M P Q R S

left child index: 1
right child index: 2
max child index: 2
swapping element that has parent and maxchild index
because B < H
assign max child index to parent
Array -> H E B C I K L M P Q R S

Swapping root and last element of heap
Array -> C E B H I K L M P Q R S

left child index: 1
right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because C < E
assign max child index to parent
Array -> E C B H I K L M P Q R S

Swapping root and last element of heap
Array -> B C E H I K L M P Q R S

left child index: 1

right child index: 2
max child index: 1
swapping element that has parent and maxchild index
because B < C
assign max child index to parent
Array -> C B E H I K L M P Q R S

Swapping root and last element of heap
Array -> B C E H I K L M P Q R S

Swapping root and last element of heap
Array -> B C E H I K L M P Q R S

37 Replacement
25 Compare

QUICK SORT

1) A is an ordered integer array with 10 elements from small to large

Assume that array of A is → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

WORK STEP:

Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 1
pivot should be greater of equals 1
up index -> 1
down index -> 0
down should be greater than up
Swap pivot and 0th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 2
pivot should be greater of equals 2
up index -> 2
down index -> 1
down should be greater than up
Swap pivot and 1th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:

Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 3
pivot should be greater of equals 3
up index -> 3
down index -> 2
down should be greater than up
Swap pivot and 2th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 4
pivot should be greater of equals 4
up index -> 4
down index -> 3
down should be greater than up
Swap pivot and 3th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 5
pivot should be greater of equals 5
up index -> 5
down index -> 4
down should be greater than up
Swap pivot and 4th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 6
pivot should be greater of equals 6
up index -> 6
down index -> 5
down should be greater than up
Swap pivot and 5th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 7
pivot should be greater of equals 7
up index -> 7
down index -> 6
down should be greater than up
Swap pivot and 6th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 8
pivot should be greater of equals 8
up index -> 8
down index -> 7
down should be greater than up
Swap pivot and 7th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 5 6 7 8 9 10
Select the pivot first element
Pivot -> 9
pivot should be greater of equals 9
up index -> 9
down index -> 8
down should be greater than up
Swap pivot and 8th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
1 2 3 4 5 6 7 8 9 10

54 Compare
9 Replacement

2) B is an ordered integer array with 10 elements from large to small

Assume that array of B is → {10,9,8,7,6,5,4,3,2,1}

WORK STEP:

Partition subarrays
Array -> 10 9 8 7 6 5 4 3 2 1
Select the pivot first element
Pivot -> 10
pivot should be greater of equals 10
up index -> 9
down index -> 9
down should be greater than up
Swap pivot and 9th index of array
Array -> 1 9 8 7 6 5 4 3 2 10

Quick Sort left subarray:
Partition subarrays
Array -> 1 9 8 7 6 5 4 3 2 10
Select the pivot first element
Pivot -> 1
pivot should be greater of equals 1
up index -> 1
down index -> 0
down should be greater than up
Swap pivot and 0th index of array
Array -> 1 9 8 7 6 5 4 3 2 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 9 8 7 6 5 4 3 2 10
Select the pivot first element
Pivot -> 9
pivot should be greater of equals 9
up index -> 8
down index -> 8
down should be greater than up
Swap pivot and 8th index of array
Array -> 1 2 8 7 6 5 4 3 9 10

Quick Sort left subarray:
Partition subarrays
Array -> 1 2 8 7 6 5 4 3 9 10
Select the pivot first element
Pivot -> 2
pivot should be greater of equals 2
up index -> 2
down index -> 1
down should be greater than up
Swap pivot and 1th index of array

Array -> 1 2 8 7 6 5 4 3 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 8 7 6 5 4 3 9 10
Select the pivot first element
Pivot -> 8
pivot should be greater of equals 8
up index -> 7
down index -> 7
down should be greater than up
Swap pivot and 7th index of array
Array -> 1 2 3 7 6 5 4 8 9 10

Quick Sort left subarray:
Partition subarrays
Array -> 1 2 3 7 6 5 4 8 9 10
Select the pivot first element
Pivot -> 3
pivot should be greater of equals 3
up index -> 3
down index -> 2
down should be greater than up
Swap pivot and 2th index of array
Array -> 1 2 3 7 6 5 4 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 7 6 5 4 8 9 10
Select the pivot first element
Pivot -> 7
pivot should be greater of equals 7
up index -> 6
down index -> 6
down should be greater than up
Swap pivot and 6th index of array
Array -> 1 2 3 4 6 5 7 8 9 10

Quick Sort left subarray:
Partition subarrays
Array -> 1 2 3 4 6 5 7 8 9 10
Select the pivot first element
Pivot -> 4
pivot should be greater of equals 4
up index -> 4
down index -> 3
down should be greater than up
Swap pivot and 3th index of array

Array -> 1 2 3 4 6 5 7 8 9 10

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 2 3 4 6 5 7 8 9 10
Select the pivot first element
Pivot -> 6
pivot should be greater of equals 6
up index -> 5
down index -> 5
down should be greater than up
Swap pivot and 5th index of array
Array -> 1 2 3 4 5 6 7 8 9 10

9 Replacement
44 Compare

3) C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}

WORK STEP:

Partition subarrays
Array -> 5 2 13 9 1 7 6 8 1 15 4 11
Select the pivot first element
Pivot -> 5
pivot should be greater of equals 5
up index -> 2
down index -> 10
down should be greater than up
swapping...
up -> 2 down -> 10
pivot should be greater of equals 4
up index -> 3
down index -> 8
down should be greater than up
swapping...
up -> 3 down -> 8
pivot should be greater of equals 1
up index -> 5
down index -> 4
down should be greater than up
Swap pivot and 4th index of array
Array -> 1 2 4 1 5 7 6 8 9 15 13 11

Quick Sort left subarray:
Partition subarrays
Array -> 1 2 4 1 5 7 6 8 9 15 13 11
Select the pivot first element

Pivot -> 1
pivot should be greater of equals 1
up index -> 1
down index -> 3
down should be greater than up
swapping...
up -> 1 down -> 3
pivot should be greater of equals 1
up index -> 2
down index -> 1
down should be greater than up
Swap pivot and 1th index of array
Array -> 1 1 4 2 5 7 6 8 9 15 13 11

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 1 4 2 5 7 6 8 9 15 13 11
Select the pivot first element
Pivot -> 4
pivot should be greater of equals 4
up index -> 3
down index -> 3
down should be greater than up
Swap pivot and 3th index of array
Array -> 1 1 2 4 5 7 6 8 9 15 13 11

Quick Sort left subarray:

Quick Sort right subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 1 2 4 5 7 6 8 9 15 13 11
Select the pivot first element
Pivot -> 7
pivot should be greater of equals 7
up index -> 7
down index -> 6
down should be greater than up
Swap pivot and 6th index of array
Array -> 1 1 2 4 5 6 7 8 9 15 13 11

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 1 2 4 5 6 7 8 9 15 13 11
Select the pivot first element
Pivot -> 8
pivot should be greater of equals 8

up index -> 8
down index -> 7
down should be greater than up
Swap pivot and 7th index of array
Array -> 1 1 2 4 5 6 7 8 9 15 13 11

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 1 2 4 5 6 7 8 9 15 13 11
Select the pivot first element
Pivot -> 9
pivot should be greater of equals 9
up index -> 9
down index -> 8
down should be greater than up
Swap pivot and 8th index of array
Array -> 1 1 2 4 5 6 7 8 9 15 13 11

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> 1 1 2 4 5 6 7 8 9 15 13 11
Select the pivot first element
Pivot -> 15
pivot should be greater of equals 15
up index -> 11
down index -> 11
down should be greater than up
Swap pivot and 11th index of array
Array -> 1 1 2 4 5 6 7 8 9 11 13 15

Quick Sort left subarray:
Partition subarrays
Array -> 1 1 2 4 5 6 7 8 9 11 13 15
Select the pivot first element
Pivot -> 11
pivot should be greater of equals 11
up index -> 10
down index -> 9
down should be greater than up
Swap pivot and 9th index of array
Array -> 1 1 2 4 5 6 7 8 9 11 13 15

Quick Sort right subarray:
1 1 2 4 5 6 7 8 9 11 13 15

37 Compare
11 Replacement

4) D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

WORK STEP:

Partition subarrays
Array -> S B I M H Q C L R E P K
Select the pivot first element
Pivot -> S
pivot should be greater of equals S
up index -> 11
down index -> 11
down should be greater than up
Swap pivot and 11th index of array
Array -> K B I M H Q C L R E P S

Quick Sort left subarray:
Partition subarrays
Array -> K B I M H Q C L R E P S
Select the pivot first element
Pivot -> K
pivot should be greater of equals K
up index -> 3
down index -> 9
down should be greater than up
swapping...
up -> 3 down -> 9
pivot should be greater of equals E
up index -> 5
down index -> 6
down should be greater than up
swapping...
up -> 5 down -> 6
pivot should be greater of equals C
up index -> 6
down index -> 5
down should be greater than up
Swap pivot and 5th index of array
Array -> C B I E H K Q L R M P S

Quick Sort left subarray:
Partition subarrays
Array -> C B I E H K Q L R M P S
Select the pivot first element
Pivot -> C
pivot should be greater of equals C
up index -> 2
down index -> 1
down should be greater than up
Swap pivot and 1th index of array
Array -> B C I E H K Q L R M P S

Quick Sort left subarray:

Quick Sort right subarray:
Partition subarrays
Array -> B C I E H K Q L R M P S
Select the pivot first element
Pivot -> I
pivot should be greater of equals I
up index -> 4
down index -> 4
down should be greater than up
Swap pivot and 4th index of array
Array -> B C H E I K Q L R M P S

Quick Sort left subarray:
Partition subarrays
Array -> B C H E I K Q L R M P S
Select the pivot first element
Pivot -> H
pivot should be greater of equals H
up index -> 3
down index -> 3
down should be greater than up
Swap pivot and 3th index of array
Array -> B C E H I K Q L R M P S

Quick Sort left subarray:

Quick Sort right subarray:

Quick Sort right subarray:

Quick Sort right subarray:
Partition subarrays
Array -> B C E H I K Q L R M P S
Select the pivot first element
Pivot -> Q
pivot should be greater of equals Q
up index -> 8
down index -> 10
down should be greater than up
swapping...
up -> 8 down -> 10
pivot should be greater of equals P
up index -> 10
down index -> 9
down should be greater than up
Swap pivot and 9th index of array
Array -> B C E H I K M L P Q R S

Quick Sort left subarray:
Partition subarrays

Array -> B C E H I K M L P Q R S
Select the pivot first element
Pivot -> M
pivot should be greater of equals M
up index -> 8
down index -> 7
down should be greater than up
Swap pivot and 7th index of array
Array -> B C E H I K L M P Q R S

38 Compare
10 Replacement