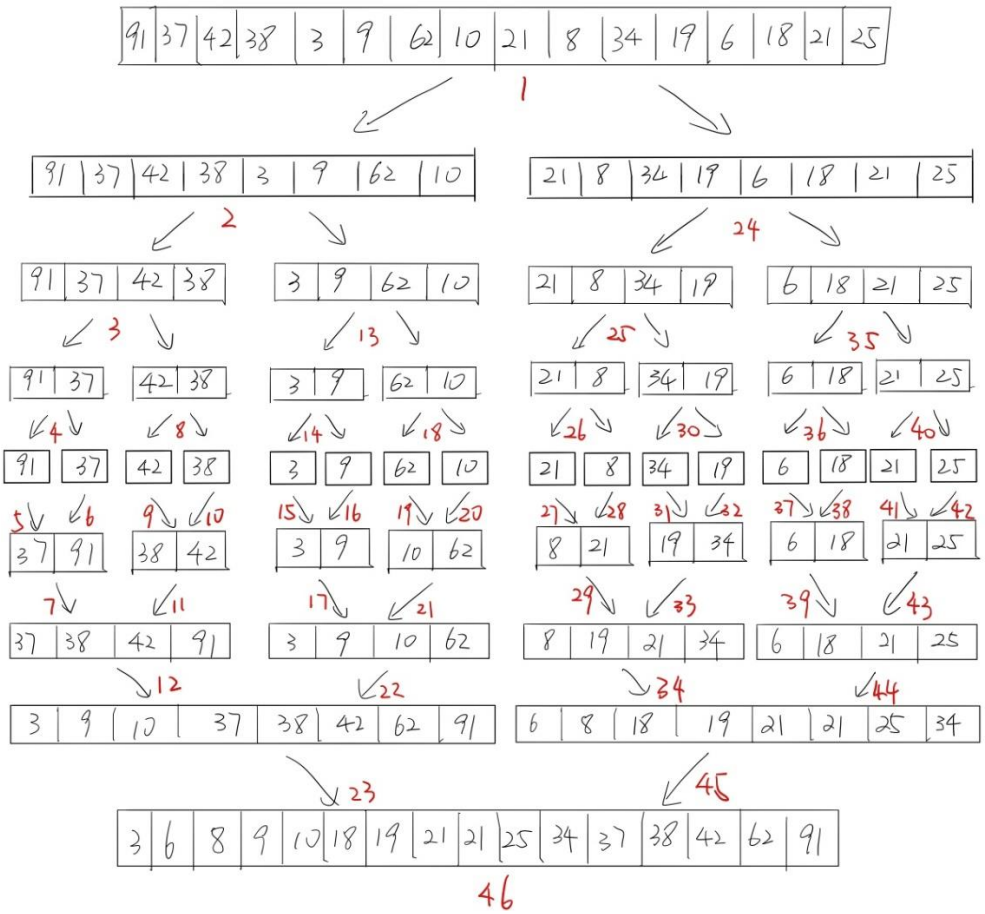
1.

A)

Take array {91, 37, 42, 38, 3, 9, 62, 10, 21, 8, 34, 19, 6, 18, 21, 25} as an example:

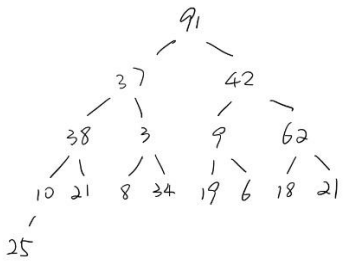
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	91	**37**	**42**	**38**	**3**	**9**	**62**	**10**	**21**	**8**	**34**	**19**	**6**	**18**	**21**	**25**
merge(a,aux,0,0,1)	**37**	**91**	42	38	3	9	62	10	21	8	34	19	6	18	21	25
merge(a,aux,2,2,3)	37	91	**38**	**42**	3	9	62	10	21	8	34	19	6	18	21	25
merge(a,aux,0,1,3)	**37**	**38**	**42**	**91**	3	9	62	10	21	8	34	19	6	18	21	25
merge(a,aux,4,4,5)	37	38	42	91	**3**	**9**	62	10	21	8	34	19	6	18	21	25
merge(a,aux,6,6,7)	37	38	42	91	3	9	**10**	**62**	21	8	34	19	6	18	21	25
merge(a,aux,4,5,7)	37	38	42	91	**3**	**9**	**10**	**62**	21	8	34	19	6	18	21	25
merge(a,aux,0,3,7)	**3**	**9**	**10**	**37**	**38**	**42**	**62**	**91**	21	8	34	19	6	18	21	25
merge(a,aux,8,8,9)	3	9	10	37	38	42	62	91	**8**	**21**	34	19	6	18	21	25
merge(a,aux,10,10,11)	3	9	10	37	38	42	62	91	8	21	**19**	**34**	6	18	21	25
merge(a,aux,8,9,11)	3	9	10	37	38	42	62	91	**8**	**19**	**21**	**34**	6	18	21	25
merge(a,aux,12,12,13)	3	9	10	37	38	42	62	91	8	19	21	34	**6**	**18**	21	25
merge(14,14,15)	3	9	10	37	38	42	62	91	8	19	21	34	6	18	**21**	**25**
merge(a,aux,12,13,15)	3	9	10	37	38	42	62	91	8	19	21	34	**6**	**18**	**21**	**25**
merge(8,11,15)	3	9	10	37	38	42	62	91	**6**	**8**	**18**	**19**	**21**	**21**	**25**	**34**
merge(0,7,15)	**3**	**6**	**8**	**9**	**10**	**18**	**19**	**21**	**21**	**25**	**34**	**37**	**38**	**42**	**62**	**91**

B)

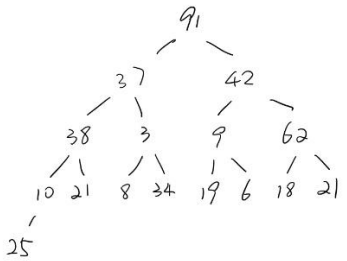


In this diagram, step 1, 2, 3, 4, 8, 13, 14, 18, 24, 25, 26, 30, 35, 36, 40 are pushing mergeSort

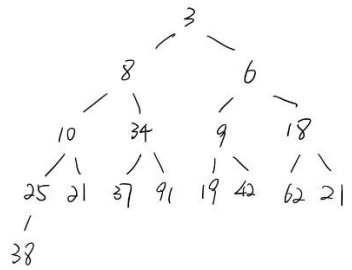
method into the stack. Step 5, 6, 7, 9, 10, 11, 12, 15, 16, 17, 19, 20, 21, 22, 23, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46 are pushing merge method into stack and pop mergeSort and merge methods from the stack.
The stack is the same as shown in question A). The termination point is r>l
2.
A)
build heap:



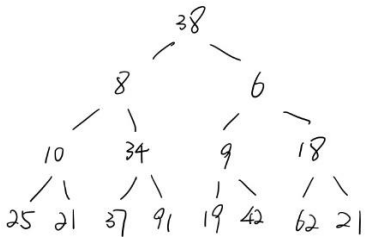
First, i=n/2-1. Method heapify is implemented to the node n/2-1 which is 7 in this case.
Then 10 is compared to its childnode 15, then remain the same:



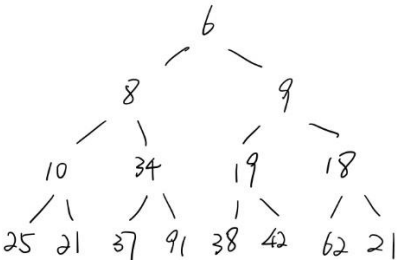
Then method heapify is implemented to node 6, 5, 4, 3, 2, 1 and 0 and finally get the MinHeapify heap:



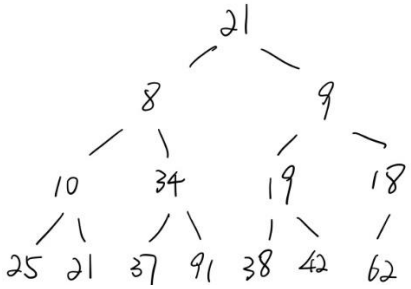
Then we change the 0 element with the last element in the tree. In this case, we change the place of 3 and 38, and delete 3 from the tree:



The output array becomes: {38,8,6,10,34,9,18,25,21,37,91,19,42,62,21,3}.
Then, heapify the 0 node to get a min heap again.



Then we change the 0 element with the last element in the tree. In this case, we change the place of 6 and 21, and delete 6 from the tree:



The output array becomes: {21,8,9,10,34,19,18,25,21,37,91,38,42,62,6,3}

Repeat the steps until the tree is empty, then we get the output:

{91,62,42,38,37,34,25,21,21,19,18,10,9,8,6,3}

which is the sorted data in Descending order.

B)

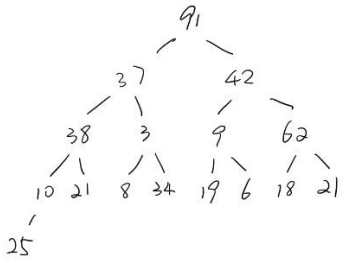
The code is Ascending order. It creates max heap each time and move the first element in the tree to the end of the tree array and remove it from the tree. So the result array becomes an ascending order.

D)

No. Because the result of problem-1 is in ascending order while the output of problem 2-C is in descending order.

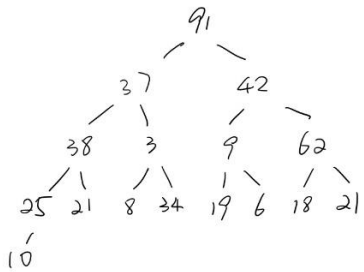
E)

build heap:

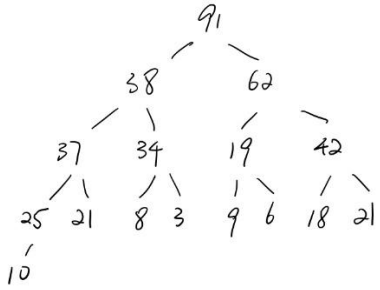


First, i=n/2-1. Method heapify is implemented to the node n/2-1 which is 7 in this case.

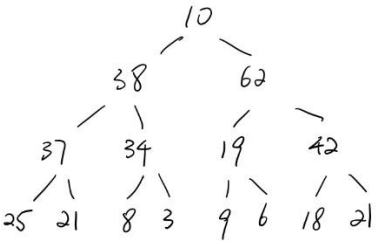
Then 10 is compared to its childnode 15, then change 10 and 25:



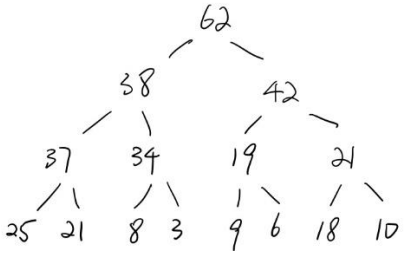
Then method heapify is implemented to node 6, 5, 4, 3, 2, 1 and 0 and finally get the MaxHeapify heap:



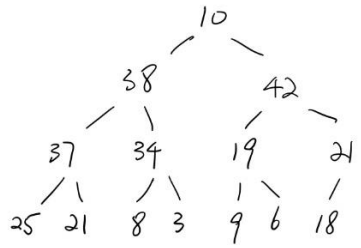
Then we change the 0 element with the last element in the tree. In this case, we change the place of 91 and 10, and delete 91 from the tree:



The output array becomes: {10,38,62,37,34,19,42,25,21,8,3,9,6,18,21,91}.
Then, heapify the 0 node to get a max heap again.



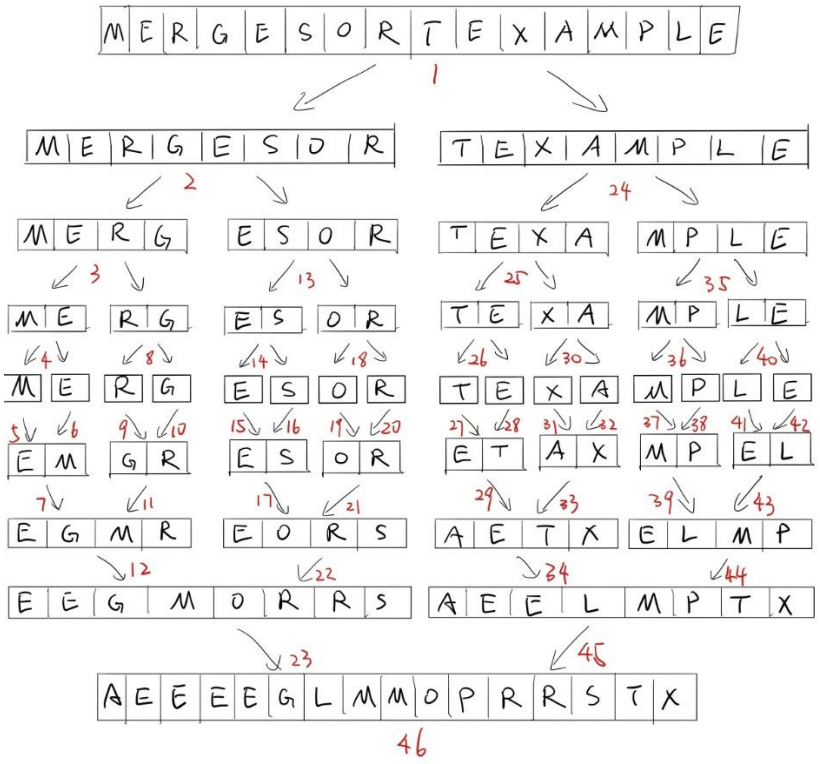
Then we change the 0 element with the last element in the tree. In this case, we change the place of 62 and 10, and delete 62 from the tree:



The output array becomes: {10,38,42,37,34,19,21,25,21,8,3,9,6,18,62,91}

Repeat the steps until the tree is empty, then we get the output:
{3,6,8,9,10,18,19,21,21,25,34,37,38,42,62,91}
which is the sorted data in Ascending order.
G)
Yes. They are both in the Ascending order.

4.



In this diagram, step 1, 2, 3, 4, 8, 13, 14, 18, 24, 25, 26, 30, 35, 36, 40 are pushing mergeSort method into the stack. Step 5, 6, 7, 9, 10, 11, 12, 15, 16, 17, 19, 20, 21, 22, 23, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46 are pushing merge method into stack and pop mergeSort and merge methods from the stack.
The change of the stack is listed as the following:
(1) push

merge(a, aux, 0, 0, 1)
~~mergeSort(a, aux, 1, 1)~~
~~mergeSort(a, aux, 0, 0)~~
mergeSort(a, aux, 0, 1)
mergeSort(a, aux, 0, 3)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(2) pop

mergeSort(a, aux, 0, 3)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(3) push

merge(a, aux, 2, 2, 3)
~~mergeSort(a, aux, 3, 3)~~
~~mergeSort(a, aux, 2, 2)~~
mergeSort(a, aux, 2, 3)
mergeSort(a, aux, 0, 3)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(4) pop

mergeSort(a, aux, 0, 3)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(5) push

merge(a, aux, 0, 1, 3)
mergeSort(a, aux, 0, 3)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(6) pop

mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(7) push

merge(a, aux, 4, 4, 5)
~~mergeSort(a, aux, 5, 5)~~
~~mergeSort(a, aux, 4, 4)~~
mergeSort(a, aux, 4, 5)
mergeSort(a, aux, 4, 7)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(8) pop

mergeSort(a, aux, 4, 7)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(9) push

merge(a, aux, 6, 6, 7)
~~mergeSort(a, aux, 7, 7)~~
~~mergeSort(a, aux, 6, 6)~~
mergeSort(a, aux, 6, 7)

mergeSort(a, aux, 4, 7)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(10) pop
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(11) push
merge(a, aux, 0, 3, 7)
mergeSort(a, aux, 0, 7)
mergeSort(a, aux, 0, 15)

(12) pop
mergeSort(a, aux, 0, 15)

(13) push
merge(a, aux, 8, 8, 9)
~~mergeSort(a, aux, 9, 9)~~
~~mergeSort(a, aux, 8, 8)~~
mergeSort(a, aux, 8, 9)
mergeSort(a, aux, 8, 11)
mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(14) pop
mergeSort(a, aux, 8, 11)
mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(15) push
merge(a, aux, 10, 10, 11)
~~mergeSort(a, aux, 11, 11)~~
~~mergeSort(a, aux, 10, 10)~~
mergeSort(a, aux, 10, 11)
mergeSort(a, aux, 8, 11)
mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(16) pop
mergeSort(a, aux, 8, 11)
mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(17) push

```
                merge(a, aux, 8, 9, 11)
            mergeSort(a, aux, 8, 11)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(18) pop

```
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(19) push

```
                    merge(a, aux, 12, 12, 13)
~~                  mergeSort(a, aux, 13, 13)~~
~~                  mergeSort(a, aux, 12, 12)~~
                mergeSort(a, aux, 12, 13)
            mergeSort(a, aux, 12, 15)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(20) pop

```
            mergeSort(a, aux, 12, 15)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(21) push

```
                    merge(a, aux, 14, 14, 15)
~~                  mergeSort(a, aux, 15, 15)~~
~~                  mergeSort(a, aux, 14, 14)~~
                mergeSort(a, aux, 14, 15)
            mergeSort(a, aux, 12, 15)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(22) pop

```
            mergeSort(a, aux, 12, 15)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(23) push

```
                merge(a, aux, 12, 13, 15)
            mergeSort(a, aux, 12, 15)
        mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)
```

(24) pop
    mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(25) push
        merge(a, aux, 8, 11, 15)
    mergeSort(a, aux, 8, 15)
mergeSort(a, aux, 0, 15)

(26) pop
mergeSort(a, aux, 0, 15)

(27) push
    merge(a, aux, 0, 7, 15)
mergeSort(a, aux, 0, 15)

(28) pop

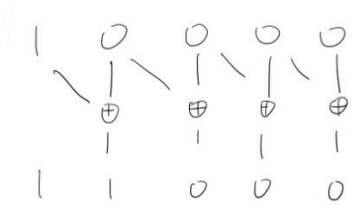
5.
a) 00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111, 01000, 01001, 01010, 01011, 01100, 01101, 01110, 01111, 10000, 10001, 10010, 10011, 10100, 10101, 10110, 10111, 11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111
b)

Binary number	Number of bits differences
00000	5 (comparing with 11111)
00001	1
00010	2
00011	1
00100	3
00101	1
00110	2
00111	1
01000	4
01001	1
01010	2
01011	1
01100	3
01101	1
01110	2
01111	1
10000	5
10001	1
10010	2
10011	1

10100	3
10101	1
10110	2
10111	1
11000	4
11001	1
11010	2
11011	1
11100	3
11101	1
11110	2
11111	1

d)

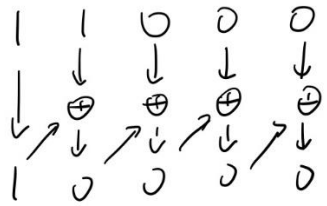
Binary: 10000



Gray code: 11000

e)

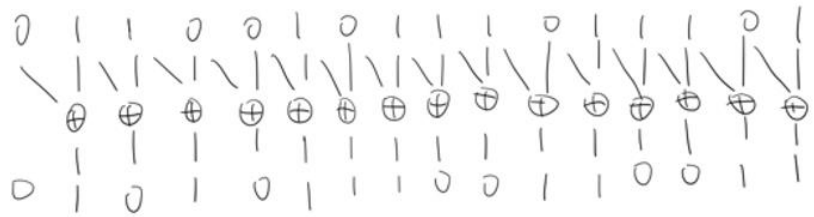
Gray code: 11000



Binary: 10000

g)

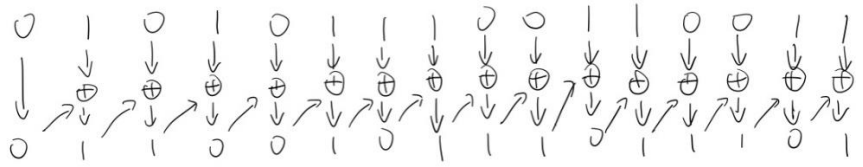
Binary Code: 0110010111011101



Gray Code: 0101011100110011

h)

Gray Code: 0101011100110011



Binary Code: 0110010111011101

6.

By the nature of dynamic programming, a matrix consisting of n^2 n elements is generated. The first row of the matrix is a zero vector. For each index value, corresponding row in the matrix is incrementally updated using the loop below. Note that each row will hold the binary representation of its index.

```
for i 1 to 2^n do
    for j 1 to n do
        if (i mod 2^j = 0) b[j] (b[j] XOR 1)
```

Take 3-bits binary code as an example:

The first row is a zero vector (0, 0, 0)

Then, for the second row, 1 mod 2^0=0, which means b[0]=b[0] XOR 1,

Then the second row becomes (0, 0, 1)

For the third row, 2 mod 2^1=0 and 2 mod 2^0=0,

which means b[1]=b[1] XOR 1, b[0]=b[0] XOR 1

Then the third row becomes (0, 1, 0)

So on and so forth, until we get the whole matrix for 3-bits binary code:

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

As shown in the figure 1, the jth column of gray code binary string of length n is actually stored in the vertical block starting with 2 j+1th index of (j+1)th column of (n+1)-length ordinary binary code.

Then we can consider i as i+2^j and j as j+1,

Then i mode 2^j=0 is equivalent to i+2^j mod 2^{j+1}=0

Which means i mod 2^{j+1}=2^j

Take 3-bits gray code as an example:

The first row is a zero vector (0, 0, 0)

Then, for the second row, 1 mod 2^1=2^0, which means b[0]=b[0] XOR 1,

Then the second row becomes (0, 0, 1)
For the third row, 2 mod $2^2=2^1$, which means b[1]=b[1] XOR 1
Then the third row becomes (0, 1, 1)
So on and so forth, until we get the whole matrix for 3-bits gray code:

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

The decimal code equals the i in the algorithm discussed before.