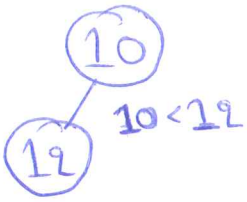
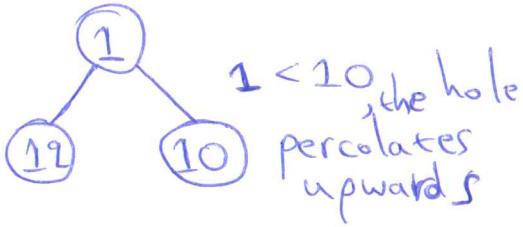
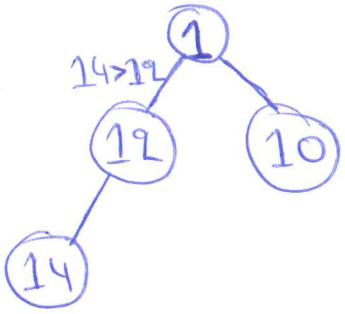
6.1) Check post on Moodle.

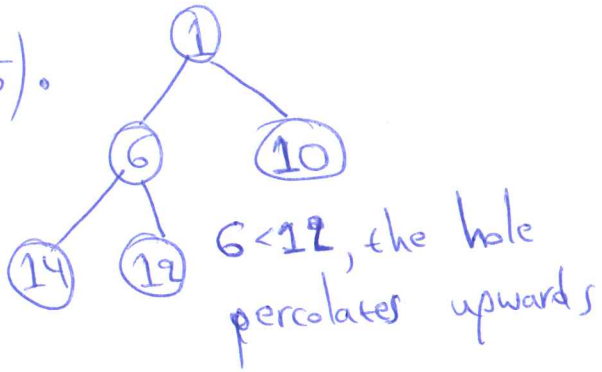
6.2) a)

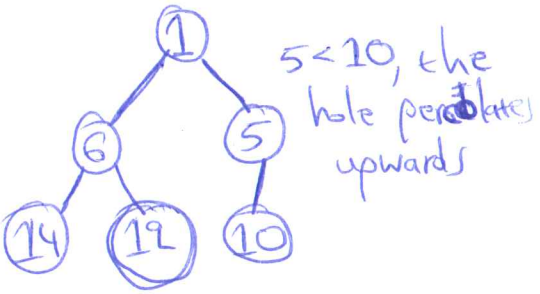
1). 

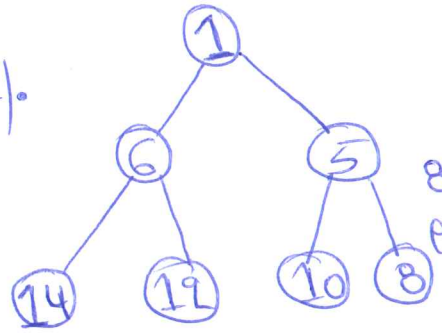
2). 

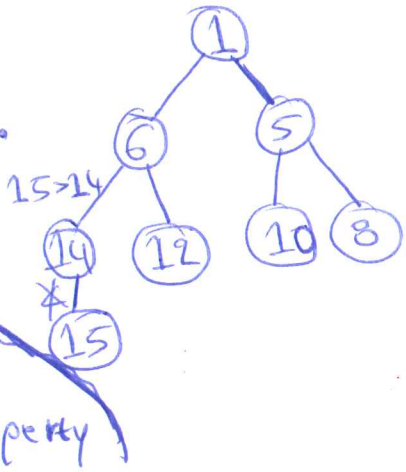
3). 

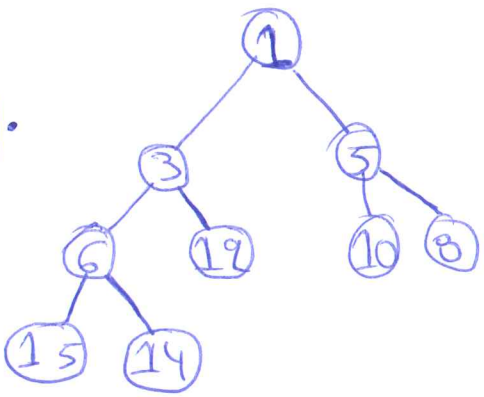
4). 

5). 

6). 

7). 

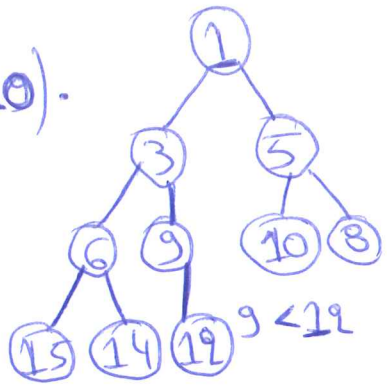
8). 

9). 

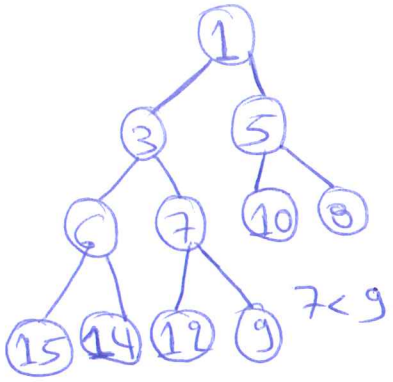
Create a hole in the right side of the node with value 14 which is the next available hole. 3<14, the hole percolates upwards. If 3 is to be inserted in the position where 14 was, its parent, i.e., 6 is still > 3.

Therefore, the hole percolates further upwards.

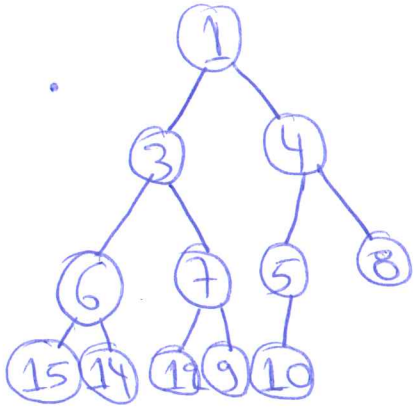
10).



11).

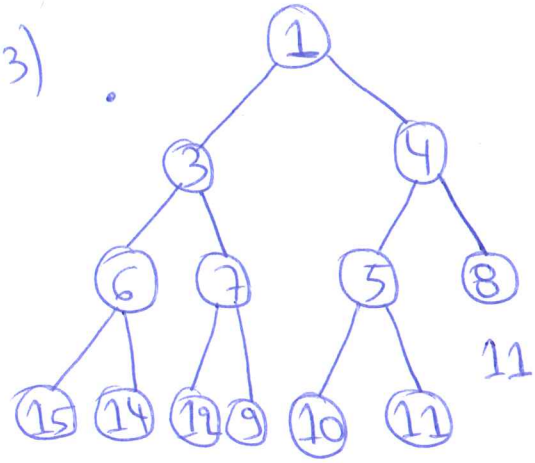


12).



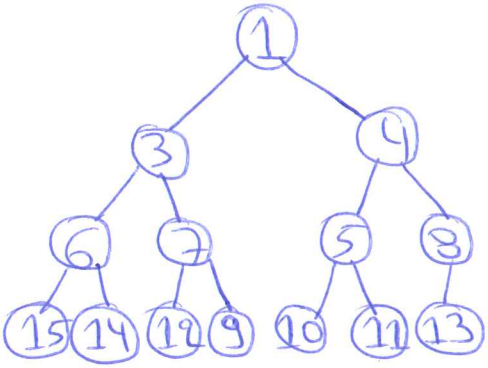
4<10, if 4 is to be inserted in the position where 10 was, its parent, i.e., 5 is > 4. Therefore, the hole percolates further upwards.

13).



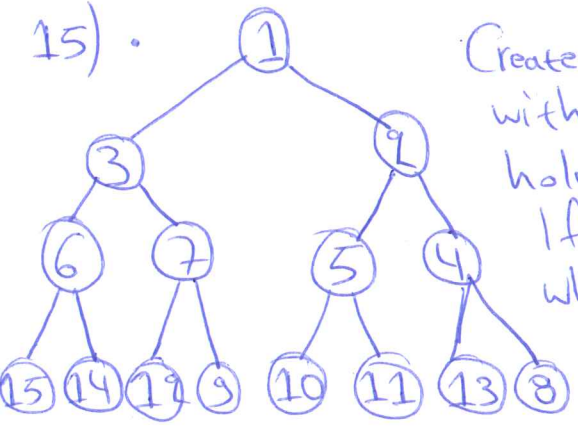
11, *

14).



13, *

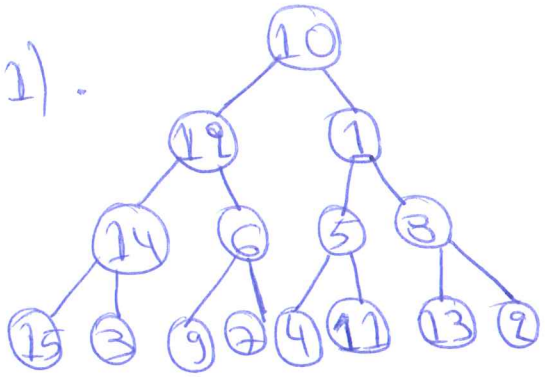
15).



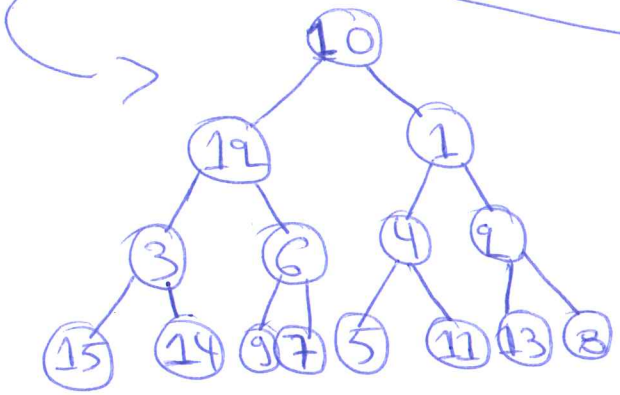
Create a hole in the right side of the node with value 8 which is the next available hole. 2<8, the hole percolates upwards. If 2 is to be inserted in the position where 8 was, its parent, i.e., 4 is still >2. The hole percolates further upwards.

6.2) b) A linear time algorithm begins by placing the values in the tree in the order they occur.

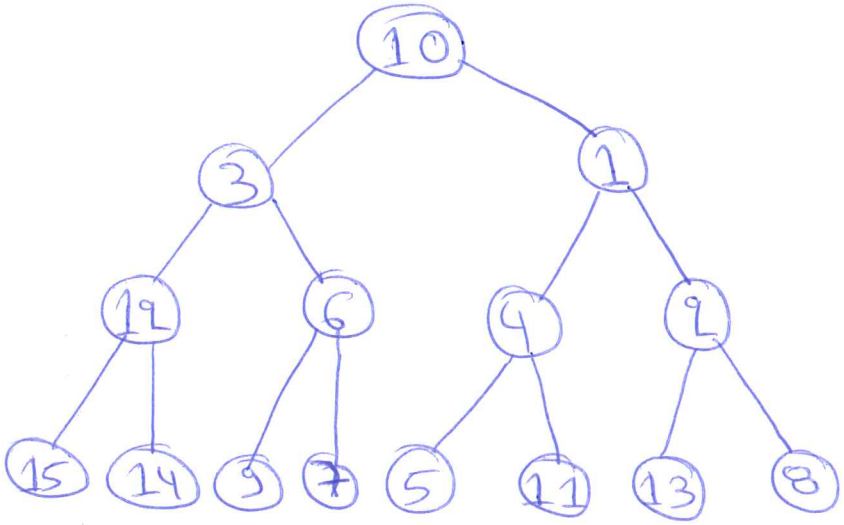
1).



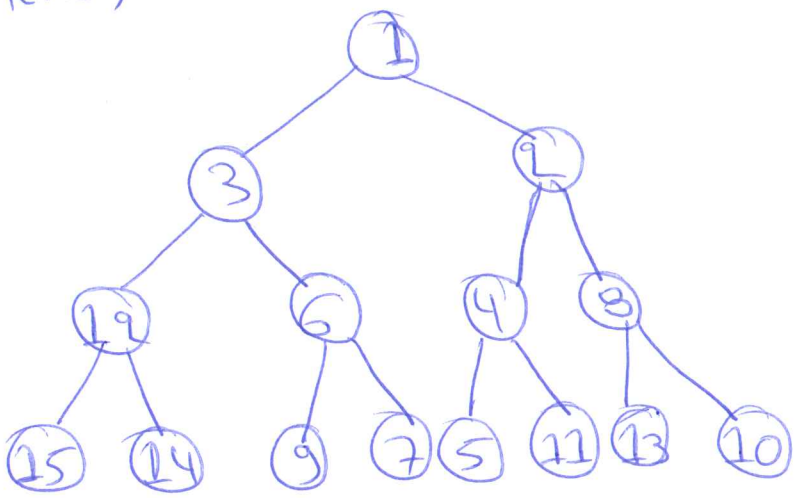
2). Check all the nodes on the last level and determine whether or not the heap-order is satisfied for all nodes. Notice that the parents of the nodes with values 3, 4, and 2 are greater than their values.



3). Now check the second ~~last~~ level and determine whether the heap-order is satisfied for all nodes or not. Notice that the parents of the nodes with values 3 and 6 are greater than their values. The lowest among the two is 3. Hence, 3 is rolled upwards.

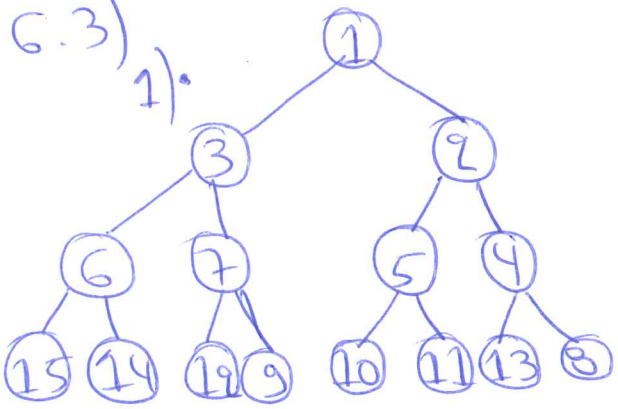


4). Now check the second level and determine whether the heap-order is satisfied for both the nodes or not. The parent nodes with values 3 and 1 are greater than their values. The lowest value among the two is 1. Hence, 1 is rolled upwards.

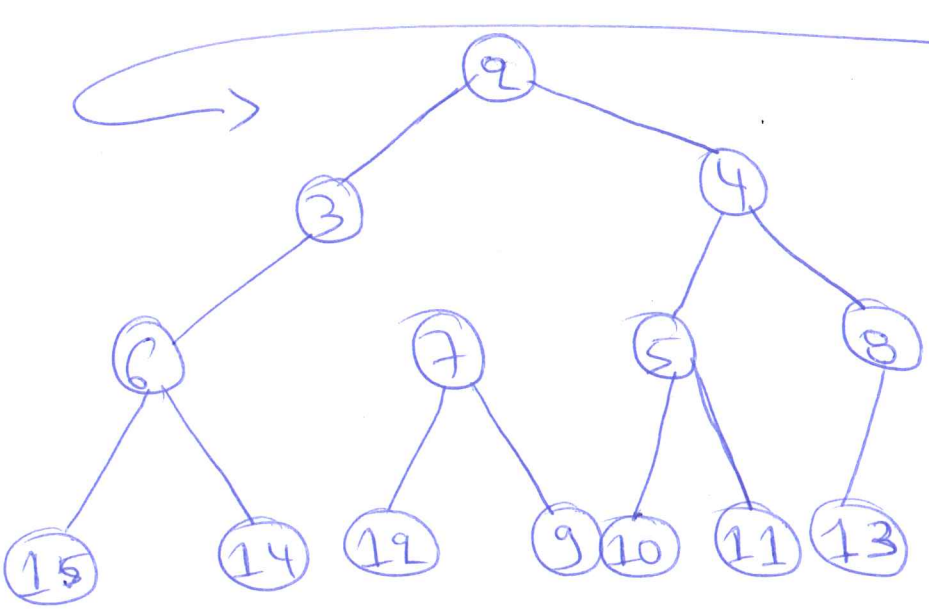


6.3)

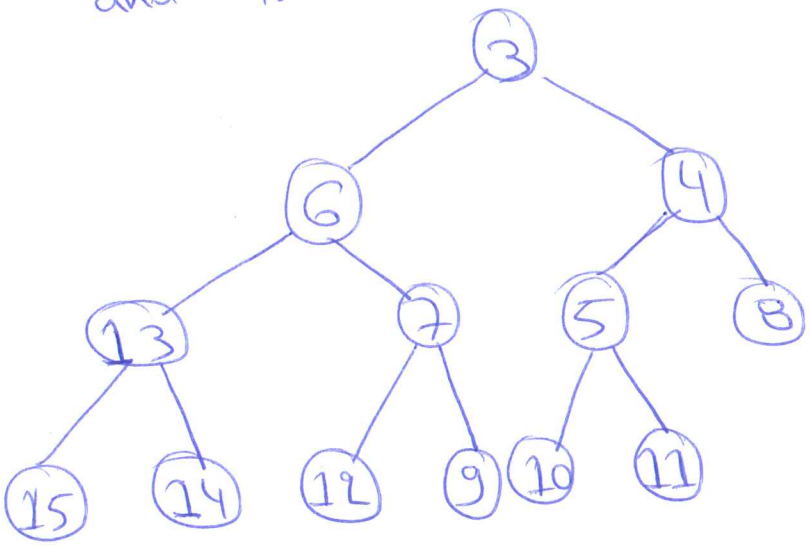
1).



2). The deleteMin deletes the minimum value from the heap which is the root. When 1 will be deleted, the candidates for the root will be 3 and 2. To satisfy the heap order, we choose 2 (the minimum value).

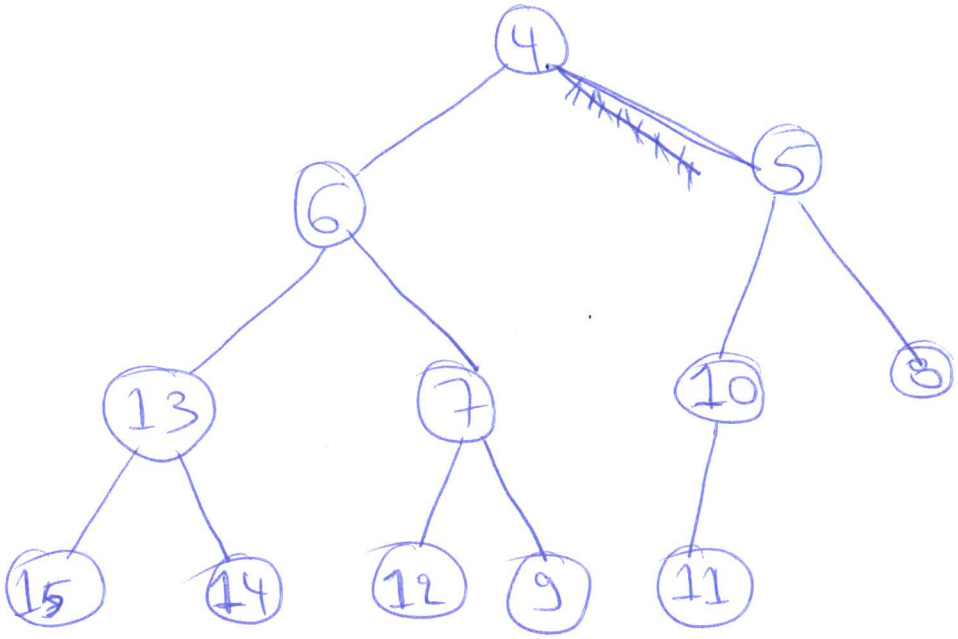


6.3) → Continuation 3). The second deleteMin will delete the next minimum value from the heap which is the root again. When 2 is deleted, the candidates for the root will be 3 and 4. We choose 3 (the minimum value)



Node with value 13 is moved to the left sub-tree to maintain the property of complete binary tree.

4). The third deleteMin will delete >>
>> >> >>. 3 will be deleted.
The candidates for the root will be 6 and 4.
We choose 4.



6.10) a) Here is a sketch of a recursive algorithm.

Start from the root of the heap
If the value of the root is smaller than X then print this value and call the procedure recursively once for its left child and once for its right child. If the value is bigger or equal than X the the procedure stops.
The complexity of this algorithm O(K), where K is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than X, so the procedure has to call each node of the heap.

7.3) At least 1 inversion is removed:

To prove this, suppose an array a[5] (→ length/size) contains five elements: 1, 3, 2, 4, 5. a[2] and a[3] which are 3 and 2 are out of order, one inversion is removed after exchanging them. The sorted array now becomes 1, 2, 3, 4, 5. Hence, if the k value is 1, and a[i] and a[i+k] are not in order then, at least one inversion is removed after exchanging them.

(a[1], a[2], a[3], a[4], a[5])

At most 2k−1 inversions are removed:

To prove this, suppose an array a[5] again with five elements 5, 2, 3, 4, 1. All the elements in the previous array are out of order.

(a[1], a[2], a[3], a[4], a[5])

Here, a[1] that is 5 is bigger than all the rest elements, thus k=4 inversions are required to sort the array. In general, it can be stated, if a[i] is bigger than a[i+1], ..., a[i+k−1] then fix k−1 inversions. → 2, 3, 4, 1, 5

Also, a[5] that is 1 is smaller than all the elements before it, thus k=3 inversions are required to sort the array. In general, it can be stated as, if a[i+k] is smaller than a[i+1], ..., a[i+k−1] then fix k−1 inversions.

As a result, the maximum required inversions are: k−1+k−1+1 (the last 1 being the one that is required for the minimum inversions) which equals 2k−1.

Another example:

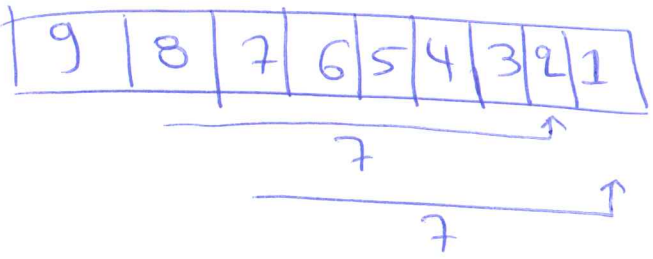
Position: 1 2 3 4 5 6 7 8 9 10

Elements: 1, 10, 3, 4, 5, 6, 7, 8, 9, 2

Switch: a[2] with a[10], k=8, 2 is now smaller than 10, and 2 is also smaller than $|3-9|=7$ numbers. Further, 10 is bigger than $|3-9|$ 7 numbers. So: $7+7+1=15=$

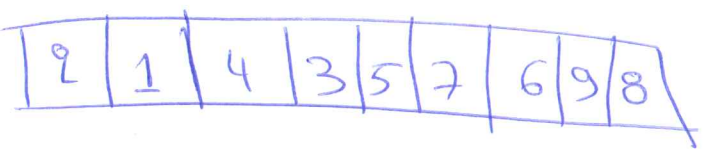
$=2 \cdot 8-1$

7.4) Original list:

- 

- First pass is for gap 7, compare the elements 2 with 9 and 1 with 8.

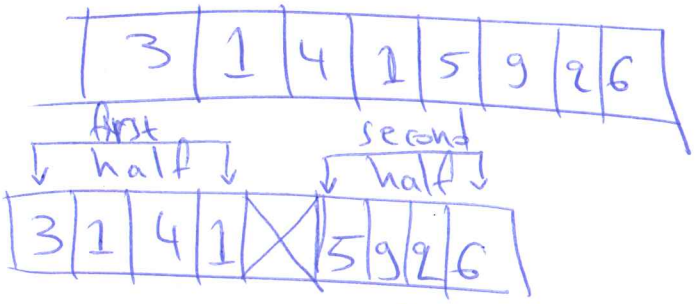
- Second pass with gap 3, compare elements 2 with 6, 1 with 5, 7 with 4, 6 with 3, 5 with 9, and 4 with 8.



7.4) → Continuation

Compare all consecutive elements because of gap = 1.

7.15) Merge sort uses divide and conquer. It devides the array into two halves and calls itself recursively and sort the list.

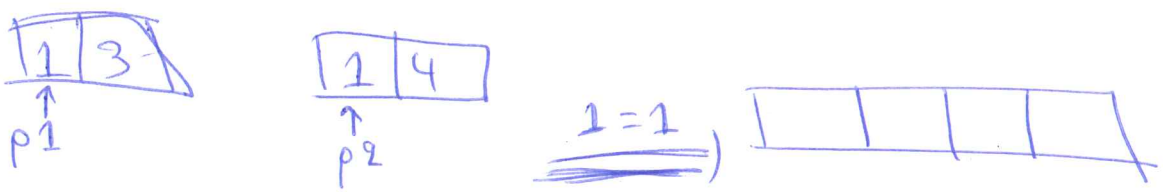


Two arrays after splitting

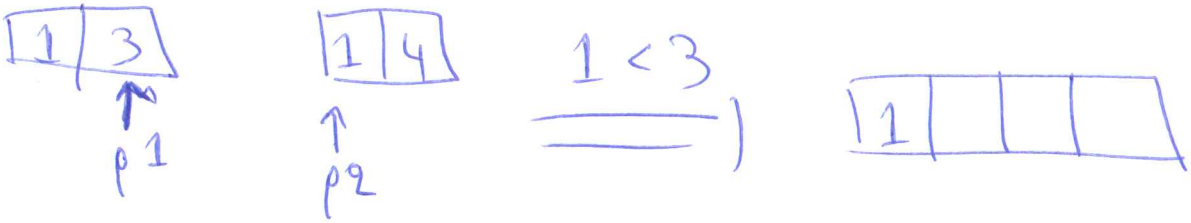
- Call again division:

- Now sort each array consisting of 2 elements. After sort, call merge().

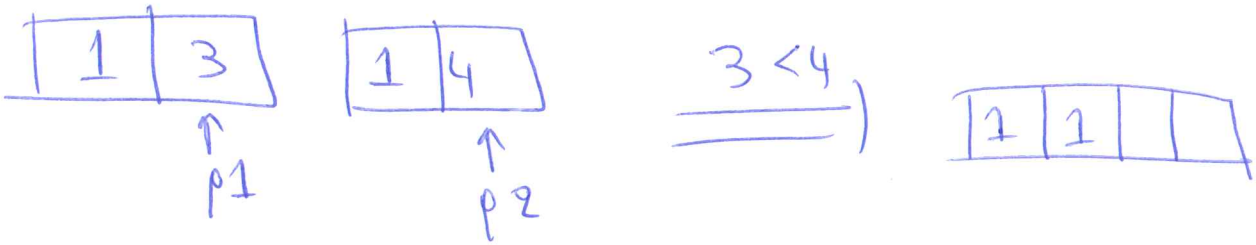
- Take first two arrays and an array of double size. Point at first elements of both arrays and compare:



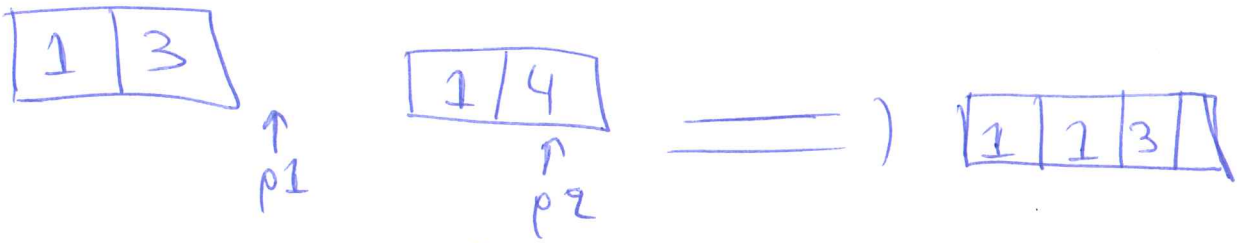
- Insert the smaller to the empty double array.



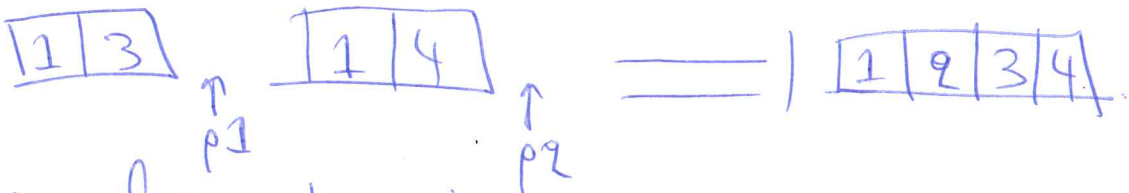
- Here 1 is smaller, thus:



- Now 3 is smaller, thus:



- Only 4 is left, thus:

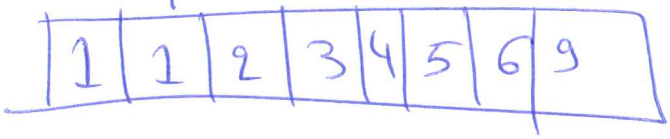


- Repeat for the other two arrays:

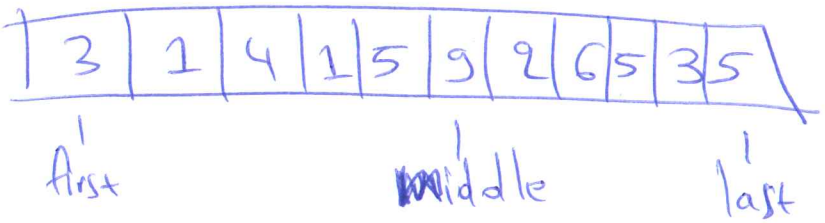
7.15) Continuation →

- Now, there are two sorted arrays as shown:

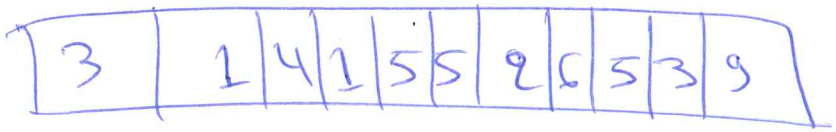
- Repeat the previous steps. The final array is:



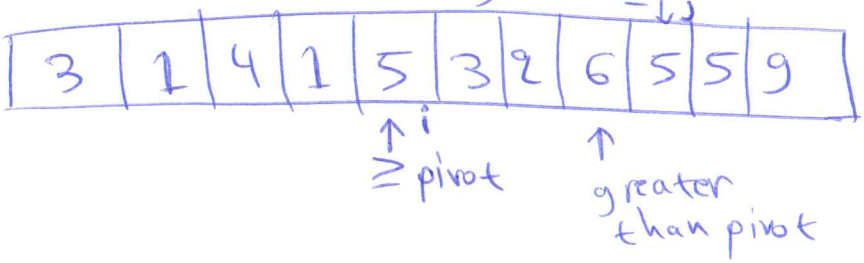
7.19) Quicksort: Insertion sort when below the cut-off. If it is 3, this means array size = 2 does not use quicksort.

- 

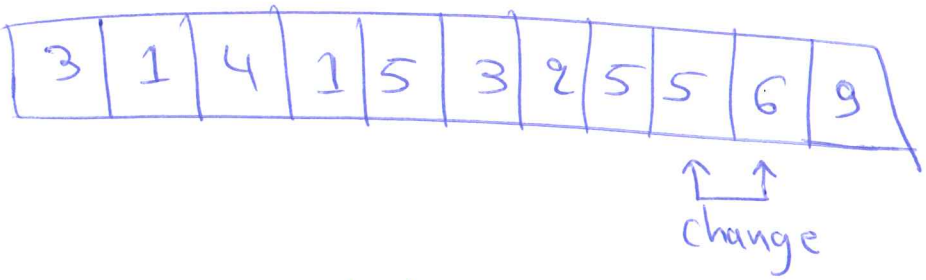
- After sorting the first, middle, and last elements, we have:



- The pivot is 5. Hiding it ≤ gives us:



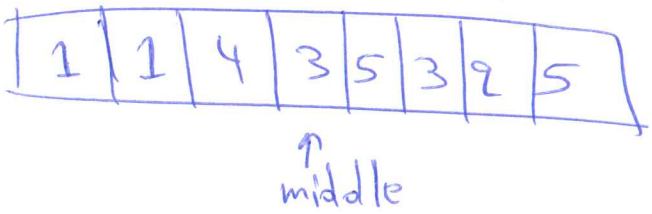
- The first swap is between two fives.
  The next swap has the i and j crossing.
  Thus the pivot is swapped back with i.
  As a result, 5 is moved to 6's place and
  6 is moved to 5's place.



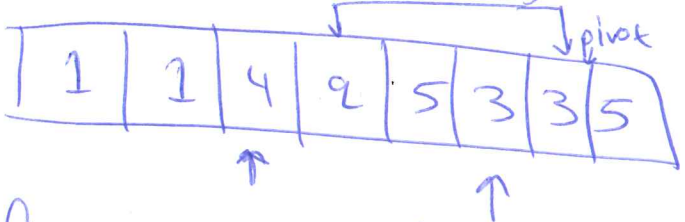
- We now recursively quicksort the first eight elements.

⑤ -> previous pivot.

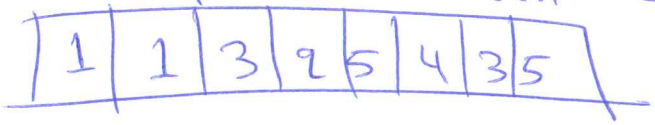
- Sorting the three appropriate elements.



- The pivot is 3, which gets hidden.



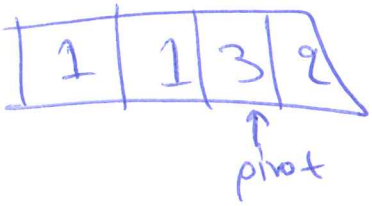
- The first swap is between 4 and 3.



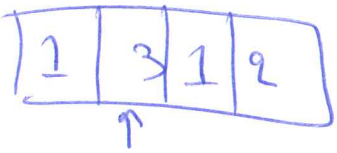
- The next swap crosses pointers, so is undone;
  i points at 5, and so the pivot is swapped.

7.19) Continuation →

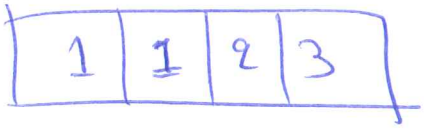
- Take the first 4 elements and proceed the sorting.



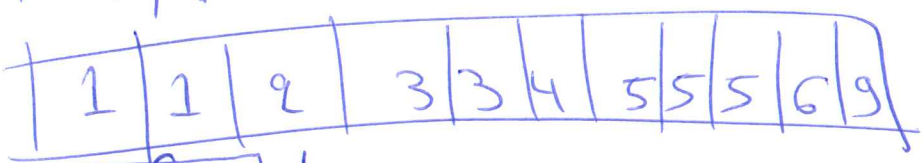
- The median element is getting hidden at the second position.



- In the next step the pivot get back in place.



- Finally:



7.20) Worst Case!

If the input given for sorting is sorted, then it depends upon the position where pivot is chosen. If it is first position, then after restoring its place it will remain at first place itself. On partition it will contain 0 elements in one partition and $n-1$ in the other.

This can be shown as:

$$T(n) = T(0) + T(n-1) + cn \quad \text{---} \quad (1)$$

Now, $n-1$ elements will call quicksort again. Thus, choosing the pivot is not able to divide the problem to reduce time. This is the worst case.

Solving (1): $T(n) = T(0) + T(n-2) + c(n-1) + T(0) + cn$

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

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

by expanding $T(n-2)$ $= T(n-3) + c(n-2+n-1+n)$

$= T(n-4) + c(n-3+n-2+n-1+n)$

$= \_ \_ \_ \_$

$= \_ \_ \_ \_ \_$

$= T(n-i) + c(n-i+1+\_ \_ n)$

$= T(n-i) + c\sum_{k=0}^{i-1}(n-k)$

It is going from 1 to $n-1$. Substitute:

$i = n-1$.

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

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

This is quadratic $(n^2) = O(N^2)$.

7.20 (a)(b) Continuation

If in sorted input median is chosen as pivot then it will be the best case that divides the array in two equal parts.

As each problem is divided into 2 sub problems it will be the logarithmic function.

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

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

$$= 2\left(T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + c\frac{n}{2}\right)$$

$$= 2^2 T\left(\frac{n}{4}\right) + cn \cdot 2$$

$$= 2^i T\left(\frac{n}{2^i}\right) + cn \cdot i$$

i which goes from $\log n$.

$$T(n) = n \cdot T(1) + \underbrace{cn}_{\text{constant}} \cdot \log n$$

running time is $n \log n = \Theta(N \log N)$.

If the input given for sorting is in reverse order, if pivot is chosen at first or last point it is the worst case. If chosen in the middle, we have the best case.

$O(N^2)$ and $\Theta(N \log N)$.

c) If random order, then we are in the average case. Suppose randomly it is decided that either the pivot will be in the middle 50% elements or it will not be in the middle 50% elements. At one side, 25% elements will be there and at 75% cases will be there. $\log_{\frac{4}{3}} n$ times which is $\Theta(n \log n)$. Because the median-of-three partition and cutoff help to avoid the worst cases.

Check also Moodle!

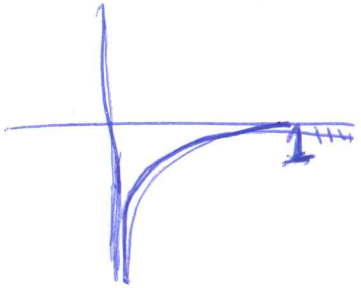
7.32) a) Insert each of the additional $f(n)$ elements, as in insertion sort.

This takes: $O(f(n)) * O(n) = O(1) * O(n) =$

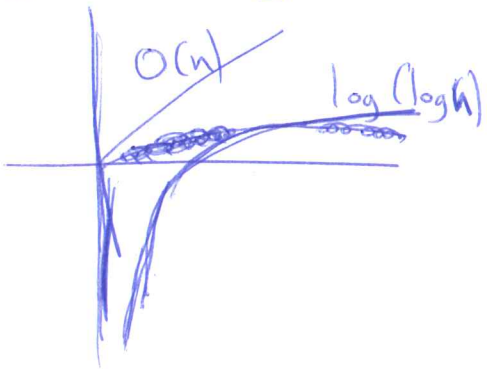
$= O(n)$ total time.

b) $O(f(n) * \log f(n)) = (\underbrace{\log n}_{①} * \underbrace{\log(\log n)}_{②})$

①:



①:



Use merge sort with a time complexity of $O(N \log N)$

7.32) c) First sort the additional $f(n)$ elements using insertion or selection sort or bubble sort.
Then merge the original n elements with the additional $f(n)$ elements, as in merge sort.
The first step takes $O(\sqrt{n^2}) = O(n)$ time.
And the second step takes $O(n + f(n)) = O(n)$ time.

d) First sort the additional $f(n)$ elements by using either merge sort or heap sort.
Then merge the original n elements with the additional $f(n)$ elements, as in merge sort.
The first step takes $O(f(n)\log f(n))$ time, and the second step takes $O(n+f(n)) = O(n)$ time.
If the total time must be $O(n)$, then we must have $f(n)$ equal to $O(n/\log n)$.
Because: $O((n/\log n)\log(n/\log n)) =$

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

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

7.33) Comparison sort: is compare the elements and then sort in either increasing or decreasing order. There are many algorithms who find an element X in $\Omega(\log N)$ comparisons such as: quicksort, merge sort, heapsort.

Lower-bound for sorting comparison: the worst case number of comparisons depends on the length or height of the tree. Height: root to the leaf of the tree. The binary tree has height h equal to $2^{h+1}-1$ leaves maximum.

~~The least N~~

The maximum is: $N = 2^{h+1} - 1$.

log on both sides:

$$\log(N) = \log(2^{h+1} - 1)$$
$$= h+1 (\log_2 2) - \log 1$$
$$= h+1$$

$$\Rightarrow h+1 = \Omega(\log N)$$

for ~~the~~ height of the tree, it needs $\Omega(\log N)$ time.

7.46| a) The number of elements are 4, so, the number of comparisons are $4! = 24$. 24 possible outcomes. If an algorithm uses 5 comparisons, then $2^4 = 16$ which is less than 24 and $2^5 = 32 > 24$. An algorithm using 4 or less than 4 comparisons cannot sort all the cases.

b) Check Moodle.

7.51) a) 2, 3, 4, ---, N-1, N, 1

This is not the worst case of the quick sort. When all the elements are already sorted is the worst case.

Here, 1 is the last element which is not sorted. The average and best cases are $O(N \log_2 N)$. The reason of being $O(N \log_2 N)$ is the median of the left, right, and center element. Since the right number of the array is not sorted which is a smaller number. If it will be greater than the N then we have the worst case.

b) The reverse:

1, N, N-1, ---, 4, 3, 2

Again as before the first element violates the sorting sequence. Same as previously, we have the best or average case:

O(NlogN) time.

7.53) Check Moodle.