1. [4 marks] Write a sorting algorithm, in pseudocode, that takes an array A with n elements and uses two stacks to sort it. You may only compare array elements to stack elements in your sort, that is, you may not compare array elements to each other. You may assume the following stack ADT methods:

   `push(o)`: Insert object o at the top of the stack
   `pop()`: Remove and return element from top of the stack, provided it is not empty
   `top()`: Returns the top element from the stack without removing it, provided it is not empty
   `isEmpty()`: Returns true is the stack is empty and false otherwise

   Analyze the worst-case running time of your algorithm.

Sol

```
Algorithm stackySort(A,n):
input: Array A of size n
output: Array A sorted
  for i<-0 to n-1 do
    stack1.push <- A[i]
      min = i
    for j<- i+1 to n-1 do
      {If (A[j] < stack1.peek())
        min <- j
        temp <- A[min]
        A[min] <- A[i]
        A[i] <- temp
      } swap A[i] <-> A[j]
    Stack2.push() <- stack1.pop()
    stack1.push <- A[i]

      end
  end
end
```

counting comparisons for my big O

Analyzing the worst case run time;

$$\sum_{i=0}^{n-1} \sum_{i=0}^{n-1} 1 \Rightarrow (n-1)-0+1 \sum_{i=0}^{n-1} 1$$

$$\Rightarrow n \sum_{i=0}^{n-1} 1$$

$$\Rightarrow n\,(n)\,1$$

$$\Rightarrow n^2$$

The array is iterated over once when the first for loop is applied to it. Then it is iterated once more when the second for loop is applied to it. The second iteration of the for loop will go over the entire array n times which is the worst possible sequence. If the array is not sorted, the outer for loop will run n times & the 2nd for loop will compare n times. So this will be the worst case.

So the worst case run-time of my algorithm is $O(n^2)$

2. [4 marks] Show the various steps of Selection Sort, Bubble Sort, Insertion Sort and Mergesort on the example array, $A = [5, 7, 0, 3, 4, 2, 6, 1]$.

Sol Selection Sort

A = [5, 7, 0, 3, 4, 2, 6, 1]

5	7	0	3	4	2	6	1

find the minimum element in the array my current minimum is set to A[0] right now, after I go through the array, I find out that 0 is the new minimum so I swap 0 & A[0] & now 0 is at A[0] & 0 is my new minimum

0	7	5	3	4	2	6	1

- ①

Now my minimum is pointing at A[1] & A[0] is sorted so I look for new minimum from A[1] to end of array.

1 is my new minimum

0	1	5	3	4	2	6	7

Min

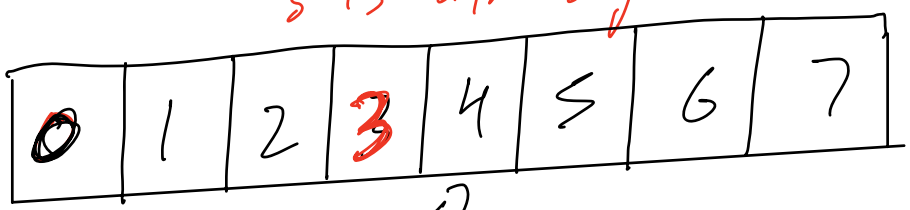
- ②

2 is my new minimum now

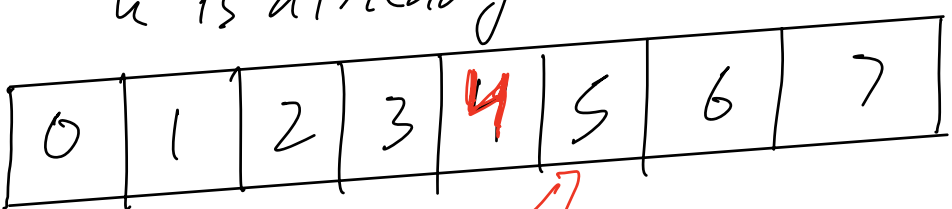
— 3

3 is already minimum

So



4 is already minimum

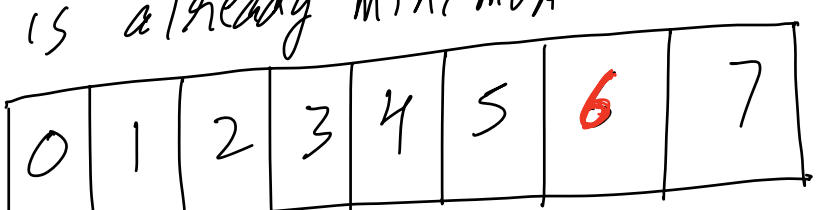


5 is already minimum

So

6 is already minimum

So



Now my last element of the array will be the largest element.

So my final array will look like

A =

0	1	2	3	4	5	6	7

(b) Bubble sort beginning

$A = [5, 7, 0, 3, 4, 2, 6, 1]$.

Bubble sort starts with the first two elements & compare them to check which one is greater

A =

5	7	0	3	4	2	6	1

step - 1

5	7	0	3	4	2	6	1

step - 2

5	0	7	3	4	2	6	1

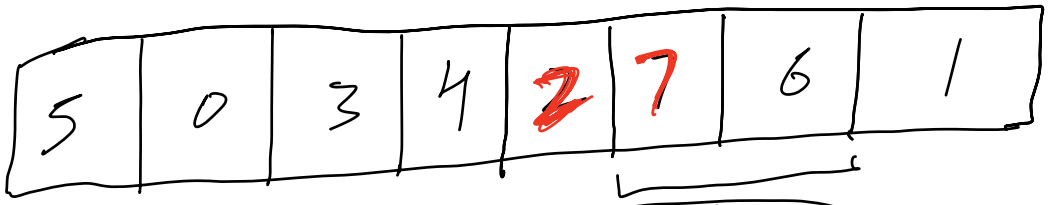
0 < 7

5	0	3	7	4	2	6	1

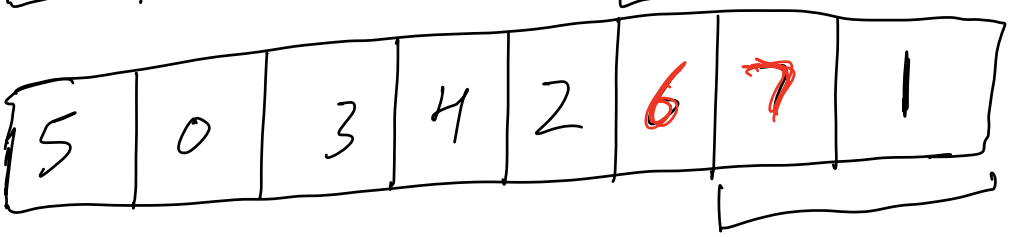
3 < 7

5	0	3	4	7	2	6	1

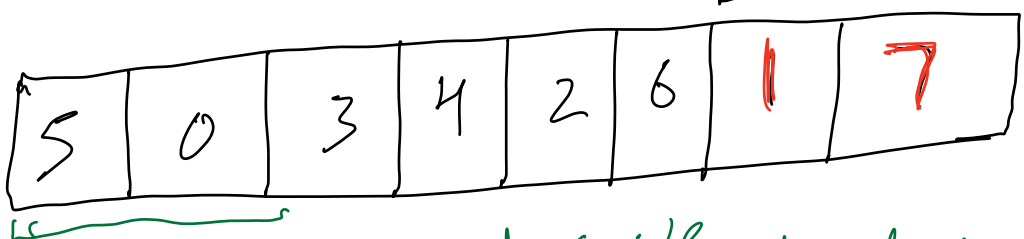
4 < 7



2 < 7

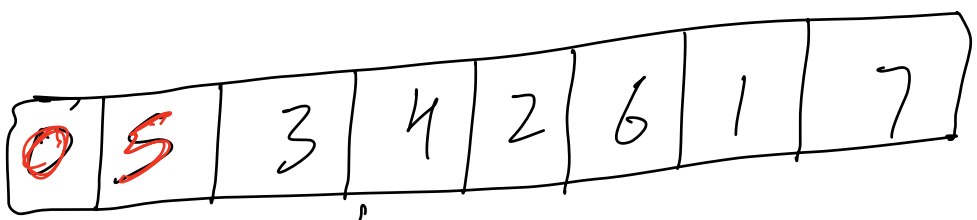


6 < 7

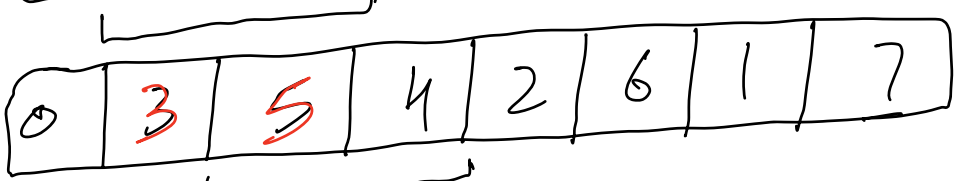


1 < 7

Start of second cycle, keep comparing adjacent values & swapping them.



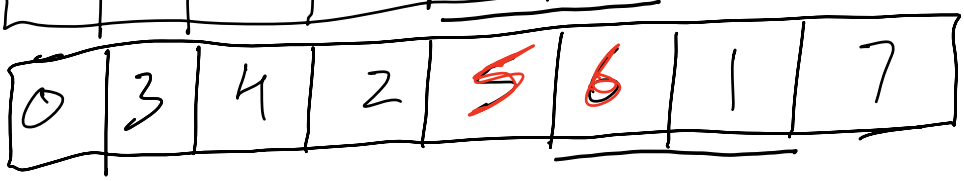
0 < 5



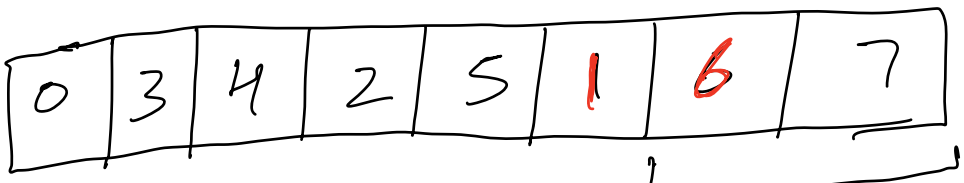
3 < 5

4 < 5

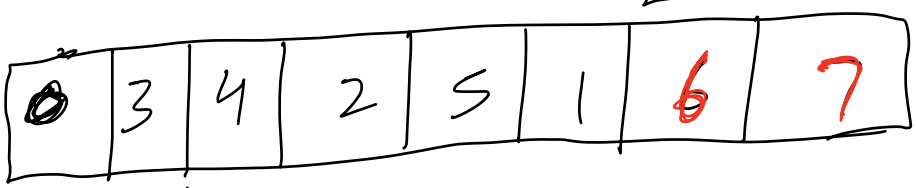
2 < 5



5 < 6

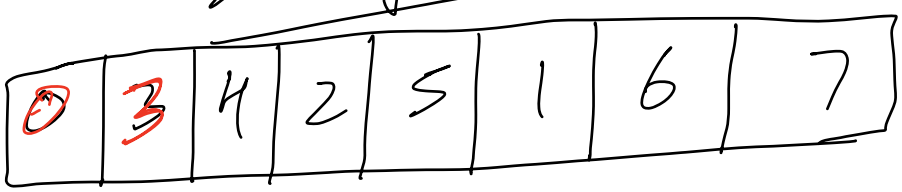


1 < 6



6 < 7

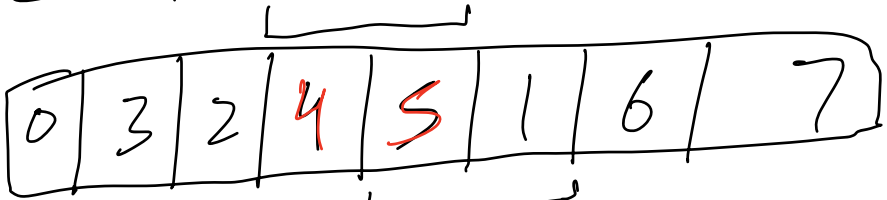
3rd cycle



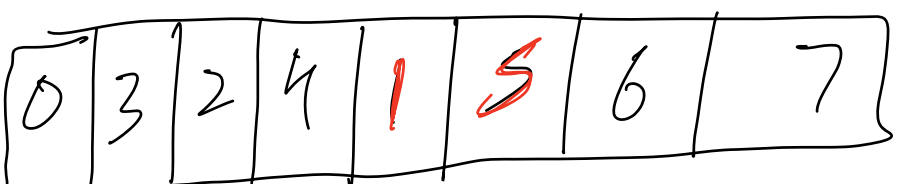
0 < 3

3 < 4

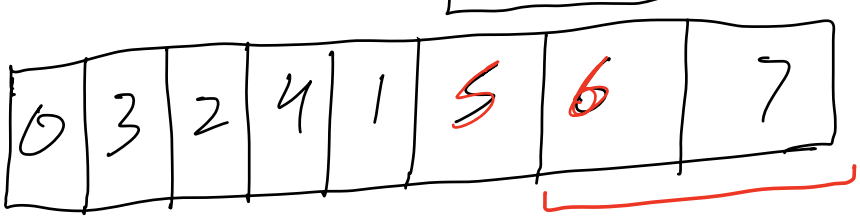
2 < 4



4 < 5



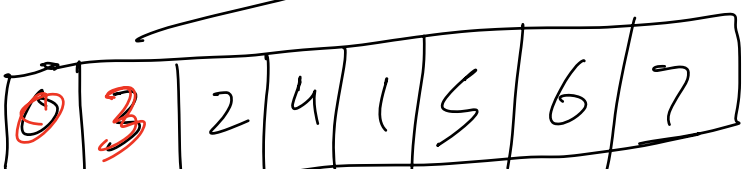
1 < 5



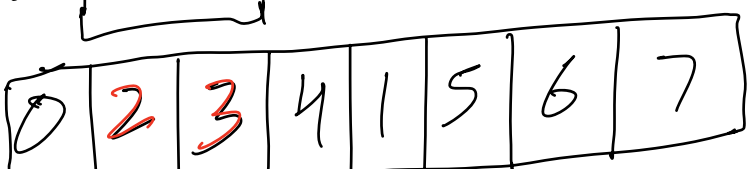
5 < 6

6 < 7

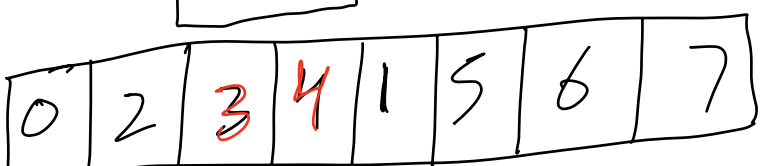
4th cycle



0 < 3



2 < 3

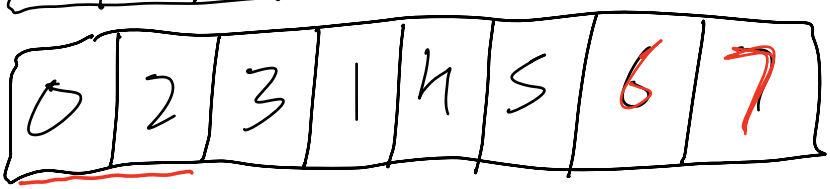


3 < 4

1 < 4

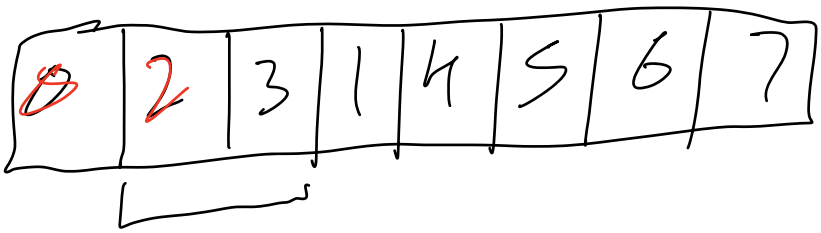
4 < 5

5 < 6

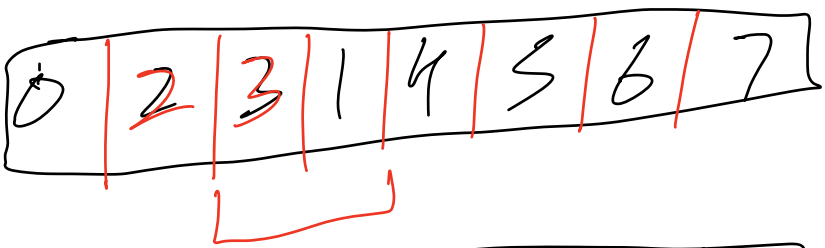


6 < 7

5th cycle



0 < 2



2 < 3

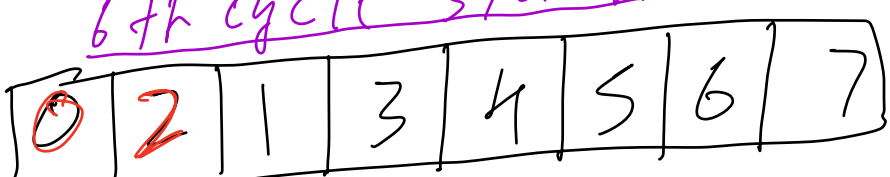
1 < 3

4 < 5

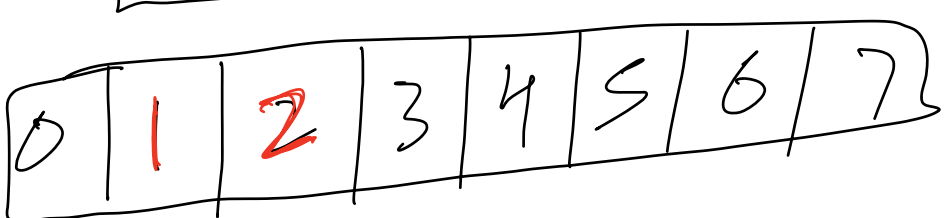
5 < 6

6 < 7

6th cycle start



0 < 2



1 < 2

The array is now sorted

& we get,

A =

0	1	2	3	4	5	6	7

(C) Insertion - Sort Begining

$$A = [5, 7, 0, 3, 4, 2, 6, 1].$$

see if the first two elements are sorted.

5	7	0	3	4	2	6	1

now move to the next two elements & compare them

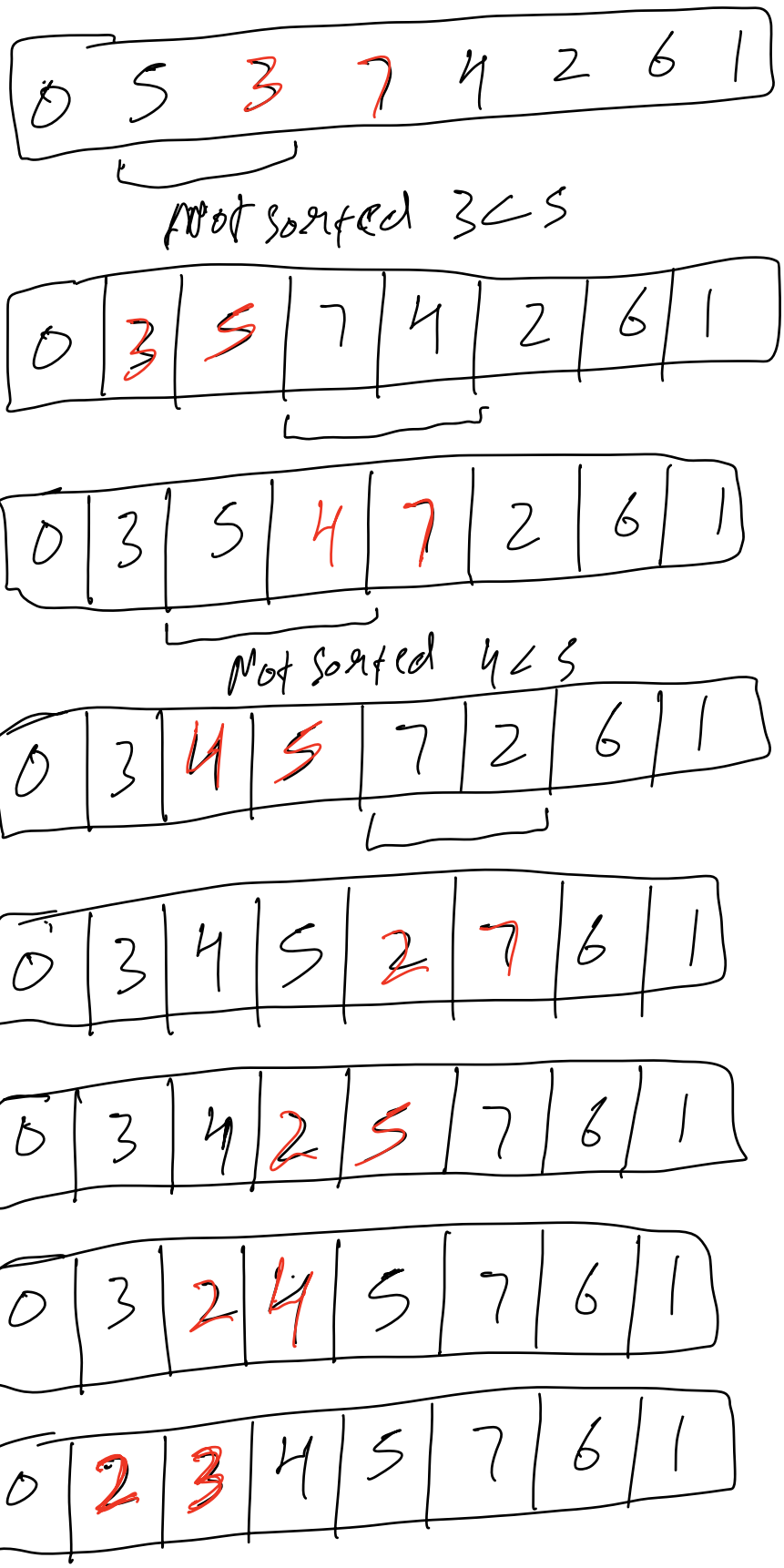
5	0	7	3	4	2	6	1

$0 < 7$

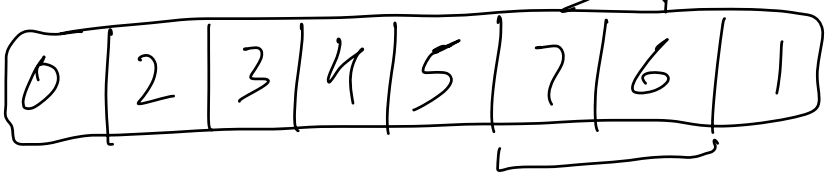
0	5	7	3	4	2	6	1

$0 < 5$

0	5	7	3	4	2	6	1

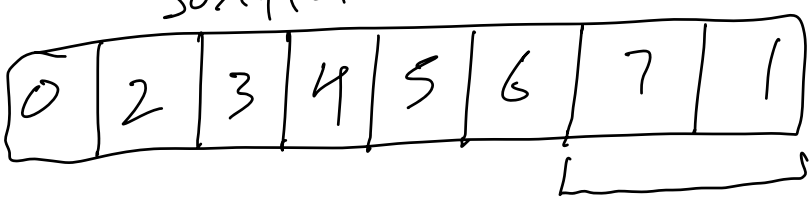

0 5 3 7 4 2 6 1
3 < 7
Not sorted 3 < 5
0 3 5 7 4 2 6 1
0 3 5 4 7 2 6 1
4 < 7
Not sorted 4 < 5
0 3 4 5 7 2 6 1
0 3 4 5 2 7 6 1
2 < 7
0 3 4 2 5 7 6 1
2 < 5
0 3 2 4 5 7 6 1
2 < 4
0 2 3 4 5 7 6 1
2 < 3

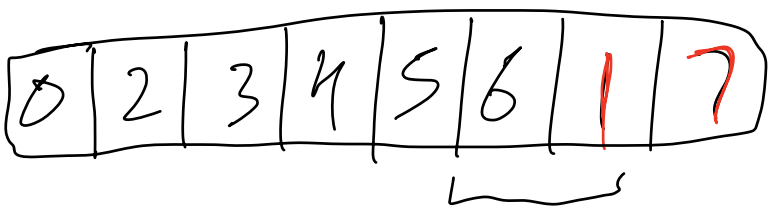
Now this is sorted



6 < 7

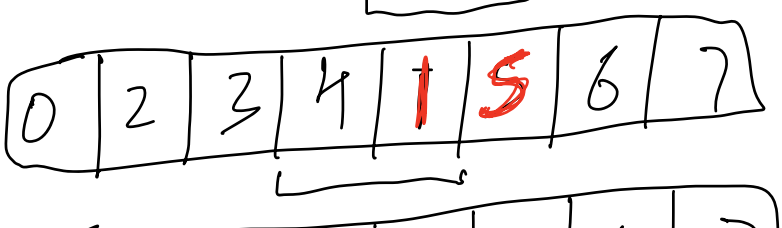
Sorted



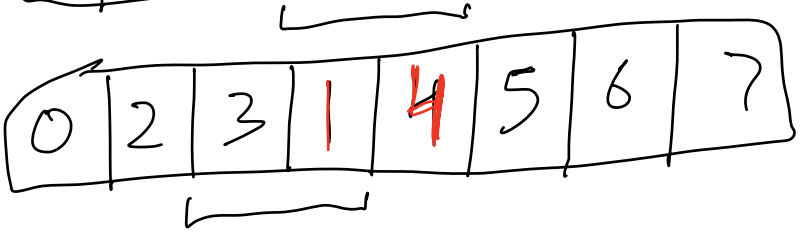


1 < 7

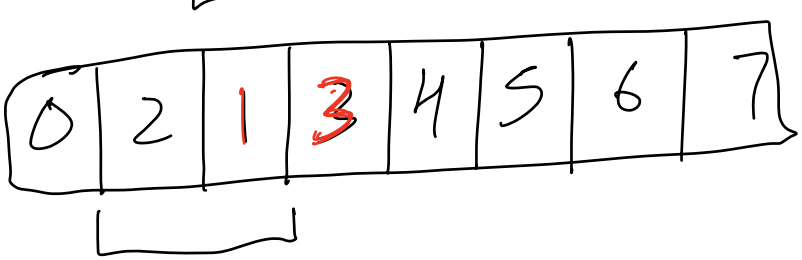
1 < 6



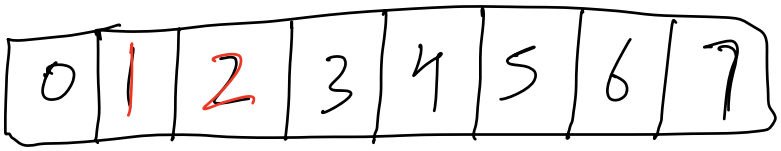
1 < 5



1 < 4

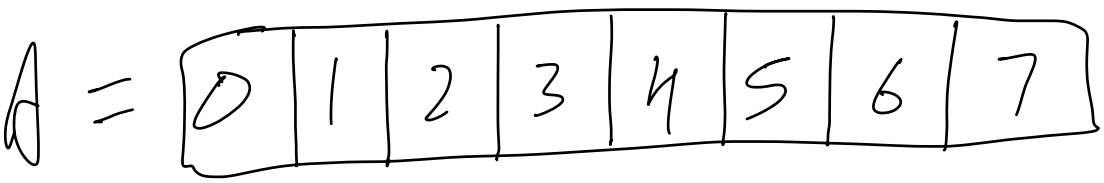


1 < 3



1 < 2

Now the Array is completely Sorted.



(d) # Merge sort Start

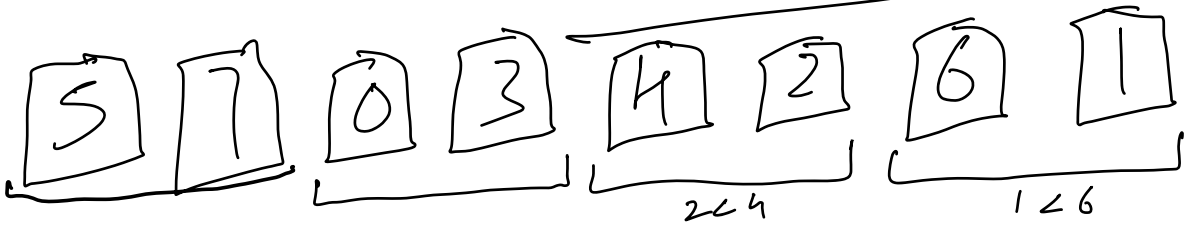
$A = [5, 7, 0, 3, 4, 2, 6, 1].$

Divide the Array in half

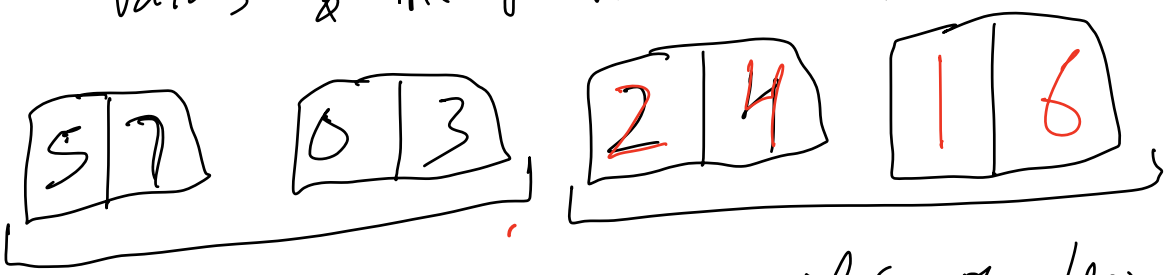
Divide in half

Divide in half

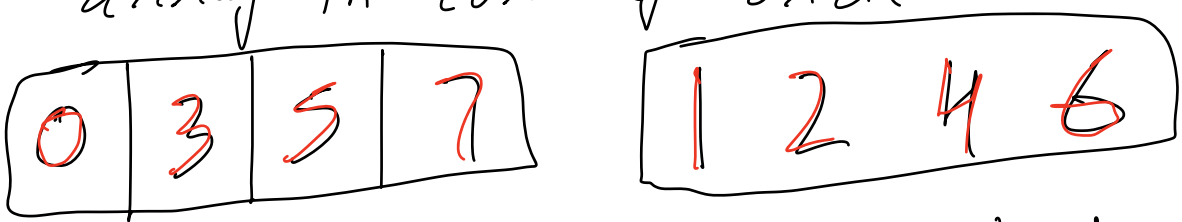
Divide again



Now compare these two individual values & merge them into temporary arrays



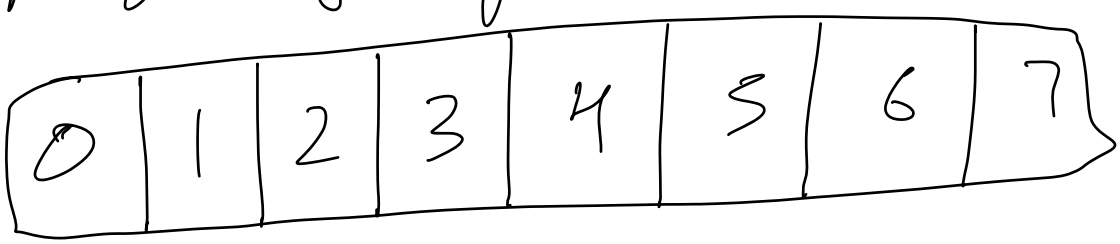
Now we merge these groups of two tempora. arrays into sorted (larger) temporary array in correct order.



Now we are going to do one last merge of these two arrays in correct order.

Now our array is sorted.

A =



3. [4 marks] Consider a version of the quicksort algorithm that uses the element at rank $\lfloor n/2 \rfloor$ as the pivot for a sequence on n elements. What is the running time of this version on a sequence that is already sorted? Describe the kind of sequence that would cause this version of quick-sort to run in $\Theta(n^2)$ time. Justify your answers.

Sol

Choosing a sequence that is already sorted, for example

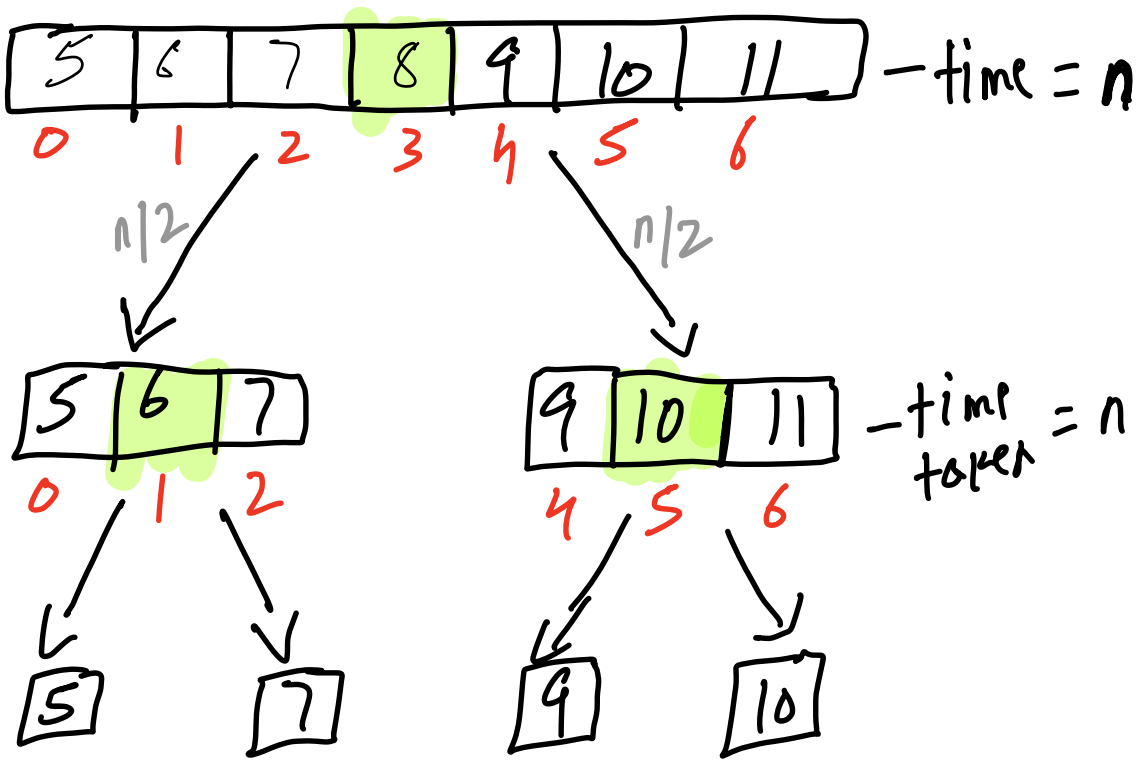
[5, 6, 7, (8), 9, 10, 11]

Now our pivot = rank[n/2]

n = 7

So, pivot = rank[7/2] = rank[3]

So our pivot in this case is element at rank[3] which is 8 in this case



each level is taking time = n

Time complexity of entire recursion tree = Number of levels * time taken for one level

Number of levels: $\frac{n}{2^k} = 1$ { k is no. of levels here.

$\Rightarrow n = 2^k$

Take log on both sides $\Rightarrow \log_2 n = \log_2 2^k$ (base 2 log)

$\Rightarrow \log_2 n = k \log_2 2$

$\Rightarrow k = \log_2 n$

So when the array size is n then the no. of levels $k = \log n$

So, time complexity = No. of levels * time taken for 1 level

$= \log n * n$

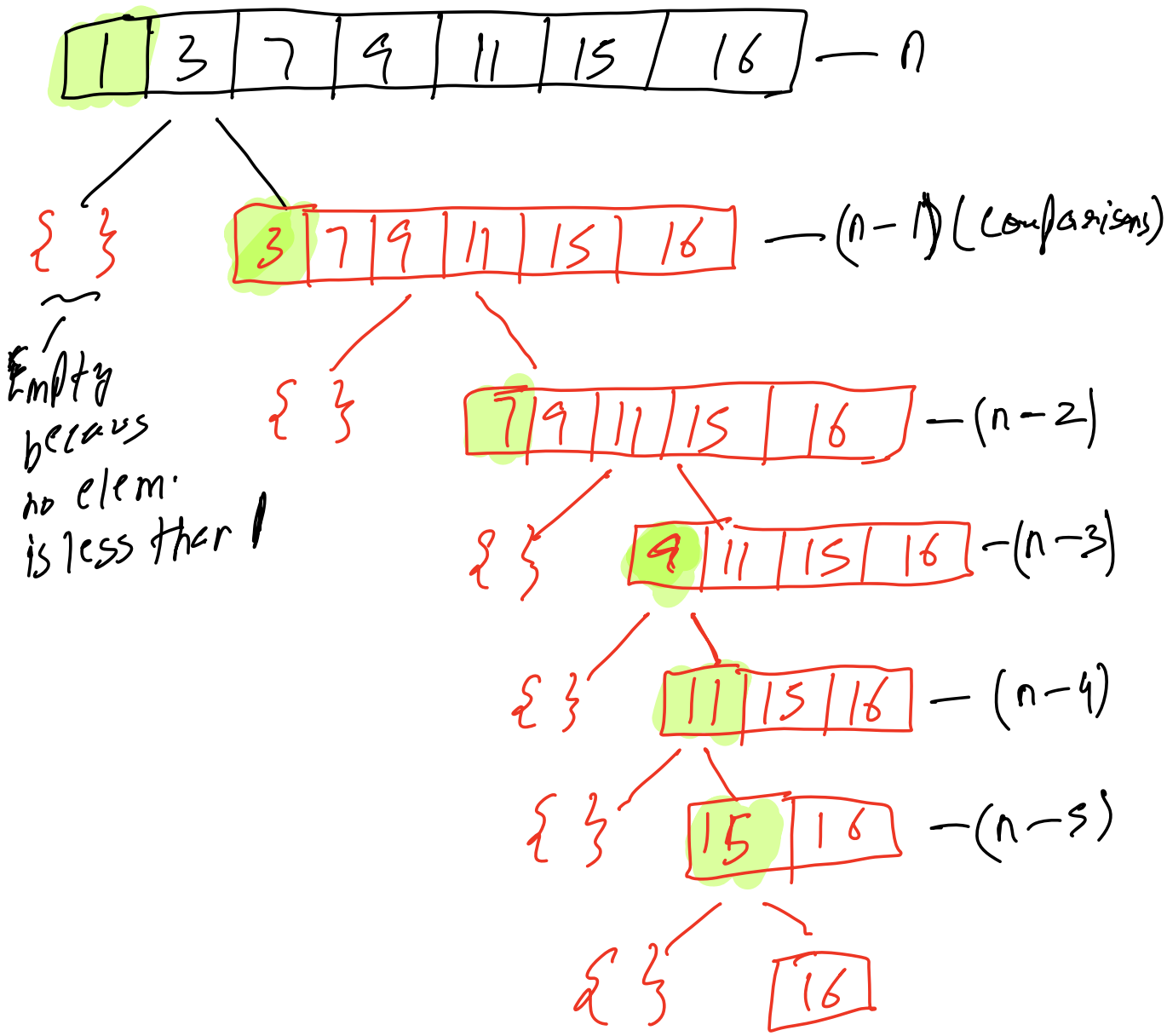
So, time complexity $= O(n \log n)$

Example 2 for worst case $O(n^2)$

n = 7

1	3	7	9	11	15	16

I'm going to choose p.pivot = rank[0]
pivot = 1



n = 7

Time complexity $\Rightarrow (n-5)+(n-4)+(n-3)+(n-2)+(n-1)+n$

$\Rightarrow 2 + 3 + 4 + 5 + 6 + 7$

$\Rightarrow \frac{n(n+1)}{2}$ (start is 2 & not 1 so) just subtract 1 from the formula

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

$\Rightarrow \frac{n^2+n}{2} - 1 \Rightarrow \frac{n^2+n-2}{2}$

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

4. [4 marks] Below is the pseudocode implementation of insertion sort that I gave in class. Use a loop invariants proof to show that insertion sort is correct.

```
Algorithm insertionSort(A,n):
    Input: Array A of size n
    Output: Array A sorted
1.  for k ← 1 to n-1 do
2.  val ← A[k]
3.      j ← k–1
4.      while j ≥ 0 and A[j] > val do
5.          A[j+1] ← A[j]
6.          j ← j – 1
        end
7.      A[j+1] = val
    end
end
```

Proof :-

Loop Invariant

At the start of each iteration for the 'for' loop of lines 1-8, the sub array A[1.....,k-1] consists of elements originally in A[1.....k-1], but in sorted order.

Initialization :- k = 1

Therefore, A[1..... k-1] = A[1]

So, Array of single element is always sorted

Maintainence :- for a particular k.

A[1... k-1] is sorted

while loop → A[K] in the correct position.

which means, A[1 ..... K] is now sorted

At the begining of next iteration
K is K+1, so now which means

A[1 ..... K] is sorted → which means our loop invariant holds true

Termination: The 'for' loop will terminate when you have K > n (i.e, K = n+1)

The subarray is A[1 ..... n]

By definition A[1 ..... n] is in sorted order.

So this proves Insertion Sort is correct.

5. [4 marks] Define the ***internal path length***, $I(T)$, of a tree T to be the sum of the depths of all the internal nodes in T. Likewise, define the ***external path length***, $E(T)$, of a tree T to be the sum of the depths of all the external nodes in T. Use induction to show that if T is a proper binary tree with n internal nodes, then $E(T) = I(T) + 2n$.

Base case — depth of all the internal node

$n=1$, $I(T)=0$ (because there is only 1 node)

$E(T)$ = Internal path length + 2 * no. of nodes

$= 0 + 2(1)$

$E(T) = 2 \Rightarrow I(T) + 2n$

Hypothesis

when n is greater than 0 & less than k then our statement holds $0 < n < k$

Induction step

Now, let's say $k > 1$ & $n = k$

Partition the tree into root & left subtree(L) & right subtree(R)

Now L tree will have k_1 internal nodes & righ subtree will have k_2 inter. nodes

then, $k = k_1 + k_2$

root = 1 so

$$k = k_1 + k_r + 1 \quad -(1)$$

which means both internal & external node has (plus 1) length of the depth in tree

& external nodes is equal to 1 plus the no. of internal nodes for all binary tree.

$$I(T) = (I(L) + k_1) + (I(R) + k_r) \quad -(2)$$

$$E(T) = (E(L) + k_1 + 1) + (E(R) + k_r + 1)$$

Now

$$E(T) = E(L) + E(R) + (k_1 + k_r + 1) + 1$$

$$= E(L) + E(R) + k + 1 \quad (\text{from } (1))$$

$$= \underbrace{I(L) + 2k_1}_{\downarrow} + \underbrace{I(R) + 2k_r}_{\searrow} + k + 1$$

$$= (I(L) + k_1) + (I(R) + k_r) + k_1 + k_r + k + 1$$

$$= I(T) + k_1 + k_r + k + 1 \quad - \text{from } (2)$$

$$= \underline{I}(\hat{I}) + k_1 + k_{n+1} + k$$

$$= \underline{I}(\hat{I}) + k + k \quad \text{- from (1)}$$

$$= \underline{I}(\hat{I}) + 2k$$

which is what we wanted, So according to the P.M.I our statement $E(\hat{I}) = \underline{I}(\hat{I}) + 2n$ is true.