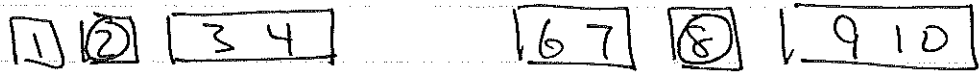
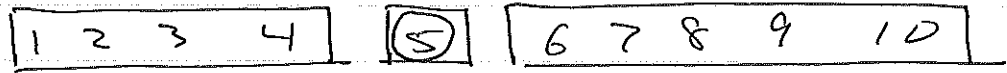
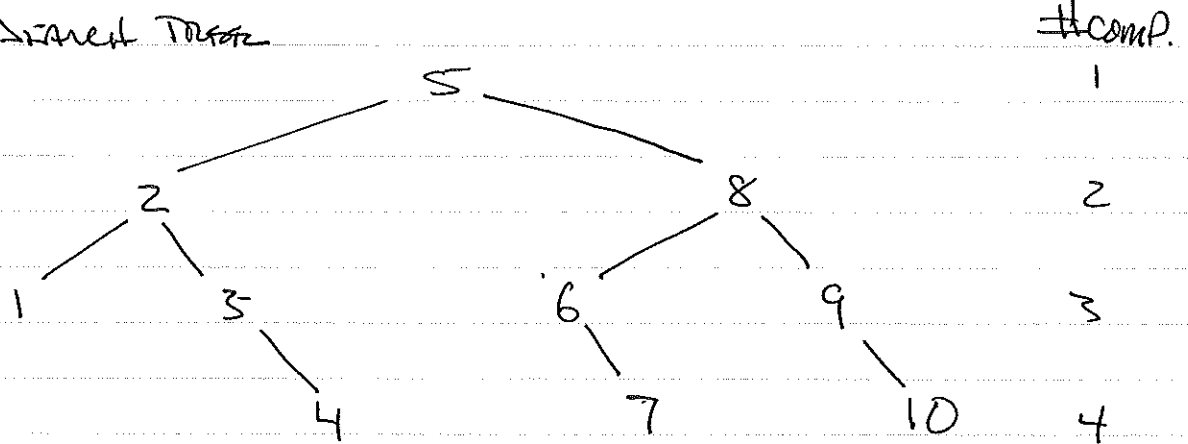


e.g) 1 2 3 4 5 6 7 8 9 10



Binary Search Tree



Binary Search should call itself on successively smaller subarrays.

NOTATION: if A is an array of length n , we write $A[p \dots q]$ for the subarray $(A[p], \dots, A[q])$.
 \therefore The full array is $A[0 \dots (n-1)]$, i.e. $p > q$
 $A[p \dots q]$ is understood to be empty.

So Binary Search Does Following:

- If $A[p \dots r]$ is empty return -1 , else
- Find index q in the "middle" of $A[p \dots r]$
- If ~~$A[q]$~~ target = $A[q]$, Done return q
- If target < $A[q]$, ~~recur~~ Recur on $A[p \dots (q-1)]$
- If target > $A[q]$ Recur on $A[(q+1) \dots r]$.

```
// BinarySearch.java
```

```
public class BinarySearch {
```

```
    public static int binarySearch(int[] A, int p, int r, int target){
        // pre: Array A[p..r] is sorted
```

```
        int q;
```

```
        if(p > r) {
            return -1;
```

```
        }else{
```

```
            q = (p+r)/2;
```

```
            if(target == A[q]){
```

```
                return q;
```

```
            }else if(target < A[q]){
```

```
                return binarySearch(A, p, q-1, target);
```

```
            }else{ // target > A[q]
```

```
                return binarySearch(A, q+1, r, target);
```

```
            }
```

```
        }
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        int[] B = {1,2,3,4,5,6,7,8,9,10};
```

```
        System.out.println(binarySearch(B, 0, 9, 7));
```

```
        System.out.println(binarySearch(B, 0, 9, 2));
```

```
        System.out.println(binarySearch(B, 0, 9, 11));
```

```
    }
```

```
}
```

NOTE: TOP level call is ON THE FULL ARRAY,
i.e. ON $A[0 \dots (n-1)]$.

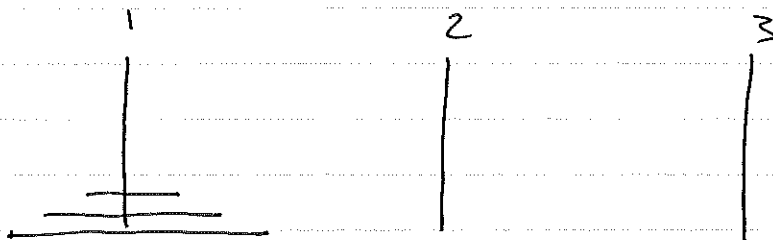
NOTE: EACH RECURSIVE INVOCATION CUTS SEARCH SPACE IN 'HALF'. HOW MANY TIMES CAN YOU 'HALVE' n UNTIL YOU GET DOWN TO 1?

$$n, n/2, n/2^2, \dots, n/2^k = 1$$

$$\therefore k = \log_2(n)$$

$$\text{RUN TIME} = \# \text{ RECURSIVE CALLS} = \log(n)$$

EX. TOWERS OF HANOI.



MOVE ALL DISCS FROM 1 TO 3!

1 → 3

1 → 2

3 → 2

1 → 3

2 → 1

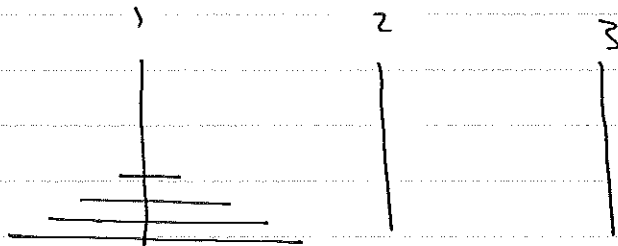
2 → 3

1 → 3

$n = \# \text{ DISCS} = 3$

(ALSO DO CASES: $n=1, n=2$)

SUPPOSE $n=4$



TO MOVE ALL DISCS FROM 1 TO 3 DO:

- MOVE TOP 3 DISCS FROM 1 TO 2 (ABOVE SUBPROB)
- MOVE REMAINING DISC FROM 1 TO 3
- MOVE 3 DISCS FROM 2 TO 3 (ABOVE)

OBSERVE THAT IF WE CAN SOLVE ANY INSTANCE OF SIZE $n-1$, WE CAN SOLVE ANY INSTANCE OF SIZE n .

```

public static void hanoi(int n, int i, int j) {
  // PRINTS INSTRUCTIONS FOR MOVING n
  // DISCS FROM ROD i TO ROD j.
  // Pre: i, j ∈ {1, 2, 3}, i ≠ j, n ≥ 1.

```

```

  int k;

```

```

  if (n == 1)

```

```

    System.out.println(i + " -> " + j);

```

```

  else { // n > 1

```

```

    k = 6 - i - j;

```

```

    hanoi(n-1, i, k);

```

```

    System.out.println(i + " -> " + j);

```

```

    hanoi(n-1, k, j);
  }
}

```

EXERCISE: Run this on successively larger ints n.

How many instructions are printed? Let

$T(n) = (\# \text{ INSTRUCTIONS TO MOVE } n \text{ DISCS})$.

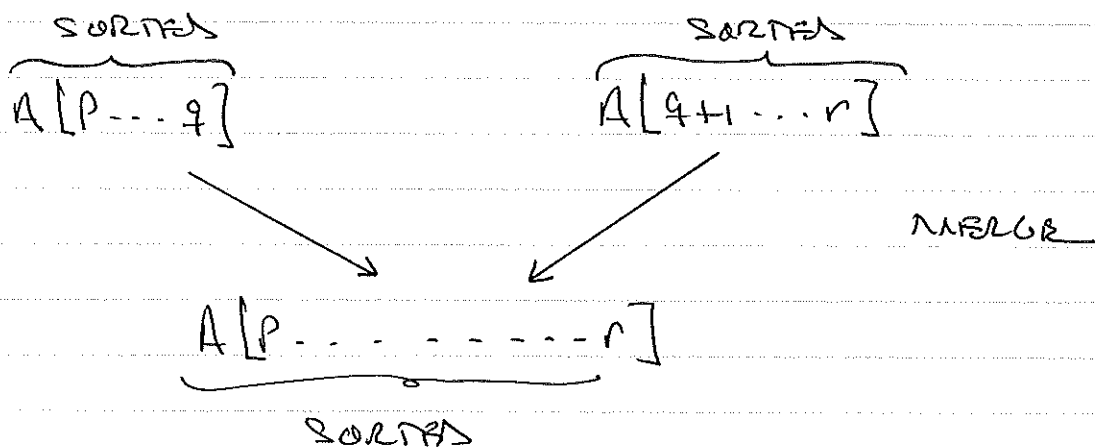
Then

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n-1) + 1 & \text{if } n>1 \end{cases}$$

check: $T(n) = 2^n - 1$

Ex SORTING AN ARRAY OF INTEGERS

Merge Sort is a Recursive Algorithm which sorts an array in increasing order. It calls a (non-recursive) sub-routine called Merge which combines two sorted subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ into a single sorted subarray $A[p \dots r]$

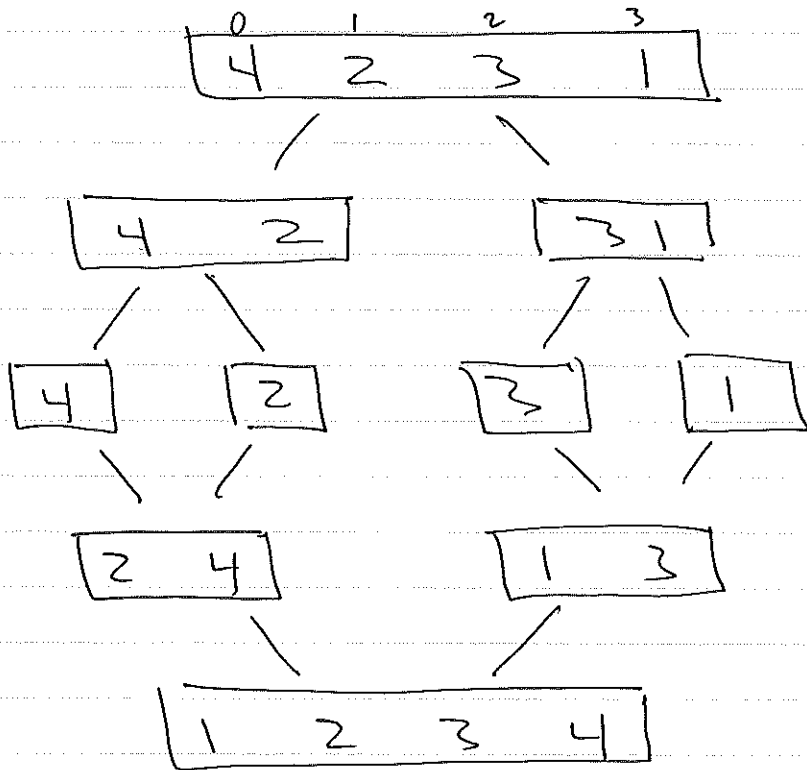


Assuming the correctness of Merge, it is easy to describe the operations of Merge Sort:

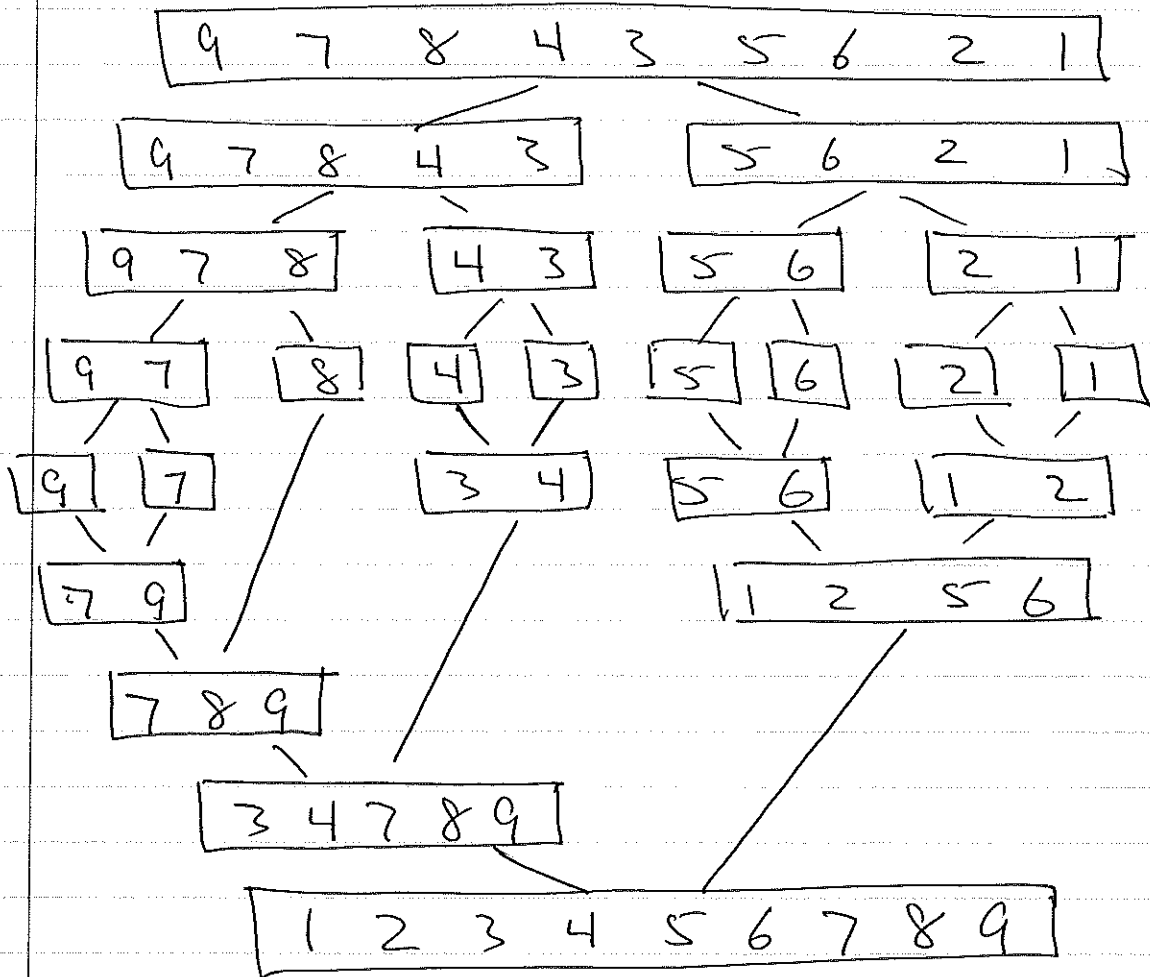
- Split the array $A[0 \dots (n-1)]$ in 'half'
- Call Merge Sort recursively on the two halves
- Merge the (now sorted) halves into a sorted whole array.

```
public static void mergeSort(int[] A, int p, int r) {  
    // Sort subarray A[p...r].  
  
    if (p < r) {  
        int q = (p+r)/2;  
        mergeSort(A, p, q);  
        mergeSort(A, q+1, r);  
        merge(A, p, q, r);  
    }  
}
```

Box trees: length = n = 4



Box Merge:



~~How to do merge operations? Let $n_1 = \text{length } A[p \dots q] = q - p + 1$, and let $n_2 = \text{length } A[q+1 \dots r]$~~

How is MERGE ACCOMPLISHED?

- let $n_1 = \text{length } A[p \dots q] = q - p + 1$
- let $n_2 = \text{length } A[q+1 \dots r] = r - q$
- Copy $A[p \dots q]$ into $L[0 \dots (n_1 - 1)]$
- Copy $A[q+1 \dots r]$ into $R[0 \dots (n_2 - 1)]$
- Merge items in L and R back into $A[p \dots r]$ in correct order.

