GREEDY ALGORITHM

* Greedy means लालची !

* "Greedy algorithms is the problem solving technique where we make the locally optimum choice at each stage & hope to achieve a global optimum."
→ this means, at every step we can make a choice that looks best at the moment & it will lead to an overall optimal solution.

Ex→ Company HR

Round 1 :	Online Test (1000)	or	Online Test (1000)
Round 2 :	Interview (100)	or	Interview (1000)
Round 3 :	Interview HR (20)	or	Interview HR (1000)
	10 Hired		
	Greedy Approach		Fair Approach

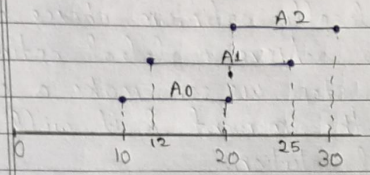
* Pros
→ Simple & Easy
→ Good Enough Time Complexity.

* Cons
→ A lot of time
→ Many times optimum is not achieved.

★ ACTIVITY SELECTION — You are given n activities with their start & end times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.
Activities are sorted according to end time.

* Sample Input : A0 A1 A2
[10, 12, 20] ← start
[20, 25, 30] ← end

* Sample Output : (2) (A0 & A2)
↓
max. 2 Activities perform कर सकता है A0 & A2



→ अगर A0 perform किए तो A1 का Activity time आधा खत्म हो जाएगा। इसीलिए A0 & A2

→ अगर A1 पहले perform किए तो A0 & A2 का Activity time खत्म हो जाएगा।

→ Maximum बोला है, तो A0 & A2.

* Approach

1. end time sort
2. We have to select 1st activity i.e. A0.
3. for (1 to n)
non-overlapping (disjoint)

start time >= last chosen activity end time

count ++;

4. Answer.

Input → start = [1, 3, 0, 5, 8, 5]
→ end = [2, 4, 6, 7, 9, 9]

Output ⇒ count = 4
ans - A0, A1, A3, A4


```
// Time Complexity = O(n)
public static void main (String args[]) {
    int start[] = {1, 3, 0, 5, 8, 5};
    int end[] = {2, 4, 6, 7, 9, 9};
    //end time basis sorted
    int maxAct = 0;
    ArrayList<Integer> ans = new ArrayList<>();
    //1st activity
    maxAct = 1;
    ans.add(0);
    int lastEnd = end[0];
    for (int i=1; i<end.length; i++) {
        if (start[i] >= lastEnd) {
            //activity select
            maxAct++;
            ans.add(i);
            lastEnd = end[i];
        }
    }
    System.out.println("max activities = " + maxAct);
    for (int i=0; i<ans.size(); i++) {
        System.out.print("A" + ans.get(i) + " ");
    }
    System.out.println();
}
```

Output: max activities = 4
A0 A1 A3 A4

If in a Question, there wasn't given that Activities are sorted then we have to write the sorted part also.

In Java, Comparator is an interface for sorting java objects.

// Time Complexity = O(n log n)

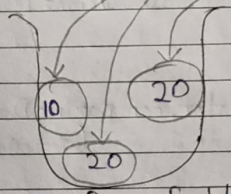
⇒
```
public static void main (String args[]) {
    int start[] = {1, 3, 0, 5, 8, 5};
    int end[] = {2, 4, 6, 7, 9, 9};
    //sorting
    int activities[][] = new int[start.length][3];
    for (int i=0; i<start.length; i++) {
        activities[i][0] = i;
        activities[i][1] = start[i];
        activities[i][2] = end[i];
    }
    //Lambda function -> shortform of any function
    Arrays.sort(activities, Comparator.comparingDouble
                (o -> o[2]));
    // end time basis sorted
    int maxAct = 0;
    ArrayList<Integer> ans = new ArrayList<>();
    //1st activity
    maxAct = 1;
    ans.add(activities[0][0]);
    int lastEnd = activities[0][2];
    for (int i=1; i<end.length; i++) {
        if (activities[i][1] >= lastEnd) {
        //activity select
        maxAct++;
        ans.add(activities[i][0]);
        lastEnd = activities[i][2];
        }
    }
    Syso("max activities = " + maxAct);
    for (int i=0; i<ans.length.size(); i++)
        System.out.print("A" + ans.get(i) + " ");
    Syso();
}
```


It is basically a bag.

★ FRACTIONAL KNAPSACK – Given the weights & values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

* Sample Input: value = [60, 100, 120], weight = [10, 20, 30], W = 50
* Sample Output: ans = 240

20 of from 30.



⇒ 60 + 100 + 80 = 240

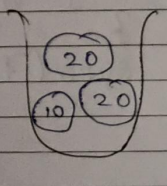
Bag of W = 50

* Approach:

⇒ We have to think like Greedy Bussinessman,
weight ↓ & value ↑ → ratio: value/weight

* Weight: [~~60~~ 10, ~~100~~ 20, ~~120~~ 30], W = 50
* Value: [60, 100, 120]
* ratio: [60/10, 100/20, 120/30] = [6, 5, 4]
  ↑ ↑ ↑
  ∵ Capacity × 4 = 20 × 4 ✓

→ Capacity = ~~50~~ ~~40~~ ~~20~~ 0
→ value = 60 + 100 + 80 = 240



⇒
```
public static void main (String args []) {
    int val [] = {60, 100, 120};
    int weight [] = {10, 20, 30};
    int W = 50;
    double ratio [][] = new double [val.length][2];
    // 0th col ⇒ idx ; 1st col ⇒ ratio
    for (int i=0; i<val.length; i++) {
        ratio [i][0] = i;
        ratio [i][1] = val [i] / (double) weight [i];
    }
    //ascending order
    Arrays.sort (ratio, Comparator.comparingDouble
                                    (o -> o[1]));
    int capacity = W;
    int finalVal = 0;
    for (int i=ratio.length-1; i>=0; i--) {
        int idx = (int) ratio [i][0];
        if (capacity >= weight [idx]) { //include full item
            finalVal += val [idx];
            capacity -= weight [idx];
        } else {
            //include fractional item
            finalVal += (ratio [i][1] * capacity);
            capacity = 0;
            break;
        }
    }
    Syso ("final value = " + finalVal);
}
```

*Output: final value = 240

★ MIN ABSOLUTE DIFFERENCE PAIRS — Given two arrays A & B of equal length n. Pair each element of array A to an element in array B, such that sum S of absolute differences of all the pairs is minimum.

* Sample Input: A = [1, 2, 3]
* ~~Sample Output~~ ⇒ B = [2, 1, 3]

Output: Ans: 0.

case ① $|1-2| + |2-1| + |3-3| = 1+1+0 = 2$
case ② $|1-3| + |2-1| + |3-2| = 2+1+1 = 4$
case ③ $|1-1| + |2-2| + |3-3| = 0+0+0 = 0$

⇒ Minimum ⇒ case ③ i.e., 0.

* Greedy Approach

a ⟷ b
जितने ज़ो Number close होंगे उतना absolute difference ↓.

* A = [4, 1, 8, 7] , B = [2, 3, 6, 5]

A = [1, 4, 7, 8] , B = [2, 3, 5, 6]

* Pair with the closest numbers means small no. with small one & big one with big one.

i,e, $|1-2| + |4-3| + |7-5| + |8-6|$

$|-1| + |1| + |2| + |2|$

$1 + 1 + 2 + 2$

(6)

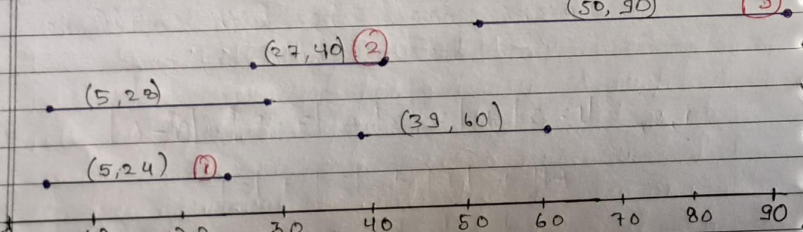
// Time Complexity, O(n log n)

⇒
```
public static void main (String args[]) {
    int A[] = {4, 1, 8, 7};
    int B[] = {2, 3, 6, 5};
    Arrays.sort (A);
    Arrays.sort (B);
    int minDiff = 0;
    for (int i=0; i<A.length; i++) {
        minDiff += Math.abs (A[i] - B[i]);
    }
    Syso ("min absolute diff of pairs = " + minDiff);
}
```

★ MAX LENGTH CHAIN OF PAIRS – We are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can come after pair (a, b) if b < c. Find the longest chain which can be formed from a given set of pairs.

Ex: pairs = (5,24) (39,60) (5,28) (27,40) (50,90)

⇒ ans = 3.



∴ ans = 3 set of pairs.

// Time Complexity → O(n log n)

⇒
```
public static void main (String args[]) {
    int pairs[][] = {{5,24}, {39,60}, {5,28}, {27,40}, {50,90}};
    Arrays.sort (pairs, Comparator.comparingDouble
                        (o -> o[1]));

    int chainLen = 1;
    int chainEnd = pairs[0][1]; //Last selected pair end
                                //chain end
    for (int i=1; i<pairs.length; i++) {
        if (pairs[i][0] > chainEnd) {
            chainLen++;
            chainEnd = pairs[i][1];
        }
    }
    Syso ("max length of chain = " + chainLen);
}
```

★ INDIAN COINS – We are given an infinite supply of denominations [1, 2, 5, 10, 20, 50, 100, 500, 2000]. Find min number of coins / notes to make change for a value V.

* Input: V = 121
* ans : 3 (100+20+1)

* Input: 590
* ans : 4 (500+50+20+20)

* Approach

① Sort descending [2000, 500, 100, 50, 20, 10, 5, 2, 1]

⇒ Amount = ~~590~~ ~~90~~ ~~40~~ ~~20~~ 0

② count = 0

```
for (int i=0; i<n; i++)
    if (coin[i] < amount)
        while (coin[i] < amount)
            count++
            amount = amount - coin[i];
```

⇒
```
public static void main (String args[]) {
    Integer coins [] = {1, 2, 5, 10, 20, 50, 100, 500, 2000};
    Arrays.sort (coins, Comparator.reverseOrder());
    int countOfCoins = 0;
    int amount = 590;
    ArrayList <Integer> ans = new ArrayList <>();
    For (int i=0; i<coinb.length; i++) {
        if (coins [i] <= amount) {
            While (coins [i] <= amount) {
                countOfCoins ++;
                ans.add (coins [i]);
                amount -= coins [i];
            }
        }
    }
    Syso ("Total (min) coins used = " + countOfCoins);
    For (int i=0; i<ans.Size(); i++) {
        System.out.print (ans.get (i) + " ");
    }
    Syso ();
}
```

★ JOB SEQUENCING PROBLEM - Given an array of jobs where every job has a deadline & profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

⇒ Job A = 4, 20
Job B = 1, 10
Job C = 1, 40
Job D = 1, 30

⇒ ans = C, A

अगर हम C को पहले करें तो Job C के बाद D और B नहीं हो सकता क्योंकि time=1 Job C में use हो चुका है।

Case 1: A time = 1; profit = 20
Case 2: B, A time = 2; profit = 10 + 20 = 30
Case 3: C, A time = 2; profit = 40 + 20 = 60 — Maximum Profit
Case 4: D, A time = 2; profit = 30 + 20 = 50

* Approach

Job A = 4, 20
Job B = 1, 10
Job C = 1, 40
Job D = 1, 30

Sorting based on Profit

Job C = 1, 40
Job D = 1, 30
Job A = 4, 20
Job B = 1, 10

① Jobs sorting (based on Digit).

② time = ~~0~~ 1 2

C	A		

ArrayList

```
for (int i=0; i<jobs; i++)
    if (job (deadline) > time)
        add job in ans
        time ++
    }
}
print ArrayList;
```

Brute Force Approach is also applied i.e., going to all Jobs to check time & profit.

We will learn sorting of objects in below code :-

Page No. 341
Date : / /

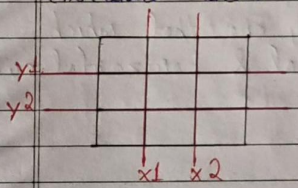
⇒
```
static class Job {
    int deadline;
    int profit;
    int id;
    public Job (int i, int d, int p) {
        id = i;
        deadline = d;
        profit = p;
    }
}
public static void main (String args[]) {
    int jobsInfo[][] = {{4,20}, {1,10}, {1,40}, {1,30}};
                        indexes → 0      1      2      3
    ArrayList<Job> jobs = new ArrayList<>();
    for (int i=0; i<jobsInfo.length; i++) {
        jobs.add (new Job (i, jobsInfo[i][0], jobsInfo[i][1]));
    }
    Collections.sort (jobs, (obj1,obj2)->obj2.profit -
    //descending order of profit          obj1.profit);
    ArrayList<Integer> seq = new ArrayList<>();
    int time = 0;
    for (int i=0; i<jobs.size(); i++)
        Job curr = jobs.get(i);
        if (curr.deadline > time)
            seq.add (curr.id);
            time++;
    Sysо ("max jobs = " + seq.size());
    for (int i=0; i<seq.size(); i++)
        System.out.print (seq.get(i) + " ");
    Sysо();
}
```

Output: max jobs = 2
2 0

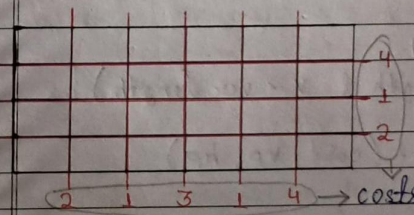
Min cost to cut board into squares.

Page No. 342
Date : / /

★ CHOCOLA PROBLEM – We are given a bar of chocolate composed of m * n square pieces. One should break the chocolate into single squares. Each break of a part of the chocolate is charged a cost expressed by a positive integer. This cost does not depend on the size of the part that is being broken but only depends on the line the break goes along.
Let us denote the costs of breaking along consecutive vertical lines with x1, x2, ..., xm-1 & along horizontal lines with y1, y2, ..., yn-1.
Compute the minimal cost of breaking the whole chocolate into single squares.

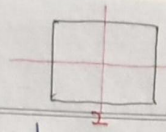


* Approach n=4, m=6



अब हमें सारे pieces of chocolate को cut करना है तो final cost → minimum

① It's necessary to have (n-1)+(m-1) cuts.
② After each cut → chocolate pieces ↑
* expensive pieces को पहले काटेंगे।
* cheap pieces को बाद में काटेंगे।



Case 1: Vertical cuts → ③ ③ → cost = 2+3+3 = 8

Case 2: Horizontal cuts → ② → cost = 3+2+2 = 7
②

* cost of vertical cuts → horizontal pieces * x.
* cost of horizontal cuts → vertical pieces * y.

* Vertical cuts = [4, 3, 2, 1, 1]
* horizontal cuts = [4, 2, 1]

* Now, we will compare one by one that which cut is expensive in vertical & horizontal cuts, we cut them first.
→ Total Cuts = m x n = 24
→ Total Costs = ~~26~~ 42

Pseudo Code ⇒

```
hor, ver → Descending sort
int h=0, v=0
int hp=1, vp=1
while (h < hor.length & v < ver.length) {
    h < v {
        vertical cut (cost, vp, hp)
    }else {
        horizontal cut
    }
}
horizontal
Vertical
Total min Cost
```


⇒

```
public static void main (String args[]) {
  int n=4, m=6;
  Integer costVer[] = {2, 1, 3, 1, 4}; //m-1
  Integer costHor[] = {4, 1, 2}; //n-1
  Arrays.sort (costVer, Collections.reverseOrder ());
  Arrays.sort (costHor, Collections.reverseOrder ());
  int h=0, v=0;
  int hp=1, vp=1;
  int cost=0;
  while (h < costHor.length && v < costVer.length) {
    if (costVer[v] <= costHor[h]) { //Horizontal cut
      cost += (costHor[h] * vp);
      hp++;
      ~~hp~~ vp++;
    } else { //Vertical cut
      cost += (costVer[v] * hp);
      vp++;
      v++;
    }
  }
  while (h < costHor.length) {
    cost += (costHor[h] * vp);
    hp++;
    h++;
  }
  while (v < costVer.length) {
    cost += (costVer[v] * hp);
    vp++;
    v++;
  }
  Syso ("min cost of cuts = " + cost); //42
}
```

★ COMPARATORS IN JAVA

It is an interface that is used to order the objects of user-defined classes.

* A comparator object is capable of comparing two objects of the same class.
* This interface is present in the java.util package & contains 2 methods compare (Object obj1, Object obj2) & equals (Object element).
* Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

⇒ public void sort (List list, ComparatorClass c)

★ LAMBDA EXPRESSIONS IN JAVA

* A lambda expression is a short block of code which takes in parameters & returns a value.
* Lambda expressions are similar to methods, but they do not need a name & they can be implemented right in the body of a method.

Format: Comparator <ClassName> comparator = Comparator.comparing (o -> o.property);

Example: Comparator <Student> comparator = Comparator.comparing (o -> o.age);
Collections.sort (students, comparator);

Que 1. Maximum Balanced String Partitions – We have balanced string str of size N with an equal numbers of L & R, the task is to find a maximum number X, such that a given string can be partitioned into X balanced substring. A string is called to be balanced if the number of 'L's' in the string equals the number of 'R's'.

* Input - "LRRRRLLRLLRL"
* Output - 3

⇒
```
public static int BalancedPartition (String str, int n) {
    if (n == 0) {
        return 0;                    // TC → O(N)
    }                                // SC → O(1)
    int r = 0, l = 0;
    int ans = 0;
    for (int i = 0; i < n; i++) {
        if (str.charAt(i) == 'R') {
            r++;
        } else if (str.charAt(i) == 'L') {
            l++;
        }
        if (r == l) {
            ans++;
        }
    } return ans;
}
psvm {
    String str = "LRRRRLLRLLRL";
    int n = str.length();
    System.out.print (BalancedPartition (str,n) + "\n");
}
```

Que 2. Kth Largest Odd number in a given range — We have two variables L & R, indicating a range of integers from L to R inclusive, & a number K, the task is to find Kth largest odd number. If K > number of odd numbers in the range L to R then return 0.

* Input - L = -3, R = 3, K = 1
* Output - 3.

⇒
```
public static int kthOdd (int[] range, int k) {
    if (k<=0) {
        return 0;          //O(1)
    }                      //O(1)
    int L = range[0];
    int R = range[1];
    if ((R&1) > 0) {
        int Count = (int) Math.ceil ((R-L+1)/2);
        if (k > Count) {
            return 0;
        } else {
            return (R-2*k+2);
    } else {
        int Count = (R-L+1)/2;
        if (k > Count) {
            return 0;
        } else {
            return (R-2*k+1);
        }
    }
psvm {
    int[] p = {-10, 10}; int k = 8;
    Syso (kthOdd(p,k));
}
```


Que 3. Lexicographically smallest string of length N & sum K - We have two integers N & K. The task is to print the lexicographically smallest string of length N consisting of lower-case English alphabets such that the sum of the characters of the string equals to K where 'a' = 1, 'b' = 2, 'c' = 3 --- & 'z' = 26.

* Input = N=5, K=42
* Output = aaamz (1+1+13+26)
* Input = N=3, K=25
* Output = aaw (1+1+23)

⇒
```
public static char[] lexo_small (int n, int k) {
    char arr[] = new char[n];
    Arrays.fill (arr, 'a');
    for (int i=n-1; i>=0; i--) {
        k -= i;
        if(k>=0){
            if (k >= 26) {
                arr[i] = 'z';
                k -= 26;
            } else {
                arr[i] = (char) (k+97-1);
                k -= arr[i] - 'a' + 1;
            }
        } else {
            break;
        }                          // TC -> O(n)
        k += i;                    // SC -> O(n)
    }
    return arr;
}
psvm {
    int n=5, k=42;
    char arr[] = lexo_small (n,k);
    System.out.print (new String (arr));
}
```

Ques 4. Best time to Buy & Sell Stock – Given an array prices[] of length N, representing the prices of the stock on different days, the task is to find the maximum profit possible for buying & selling the stocks on different days using transactions where at most one transaction is allowed

Stock must be brought before being sold.

* Input: prices[] = {7, 6, 4, 3, 1}
* Output: 0

* Input: prices[] = {7, 1, 5, 3, 6, 4}
* Output: 5

```
⇒ public static int maxProfit (int prices[], int n) {
    int buy = prices[0], max_profit = 0;
    for (int i = 1; i < n; i++) {
        if (buy > prices[i]) {
            buy = prices[i];
        } else if (prices[i] - buy > max_profit) {
            max_profit = prices[i] - buy;
        }
    }
    return max_profit;
}
public static void main (String args[]) {
    int prices[] = {7, 1, 5, 6, 4};
    int n = prices.length;
    int max_profit = maxProfit (prices, n);
    Syso (max_profit);
}
```

// Time Complexity – O(n)
// Space Complexity – O(1)

// TC → O((N-1)C(K-1)) – here 'C' depicts combinations i.e., ((N-1)! / ((N-K)! * (K-1)!)) where N is no. of elements of array & K is no. of divisions that we are having.

Ques 5. Split the given array into K sub-arrays – We have arr[] of N elements & a number K. (1 <= K <= N) Split arr[] into K subarrays (must cover all elements). The maximum subarray sum achievable out of K subarrays formed must be the minimum possible. Find that possible subarray sum.

* Input ⇒ Arr[] = {1, 1, 2}, K = 2
* Output ⇒ 2

* Input ⇒ Arr[] = {1, 2, 3, 4}, K = 3
* Output ⇒ 4

```
⇒ public static int ans = 10000000;
public static void solve (int a[], int n, int k, int index, int sum, int maxSum) {
    if (k == 1) {
        maxsum = Math.max (maxsum, sum);
        sum = 0;
        for (int i = index; i < n; i++) {
            sum += a[i];
        }
        maxsum = Math.max (maxsum, sum);
        ans = Math.min (ans, maxsum);
        return n;
    }                                   // Space Complexity - O(n)
    sum = 0;
    for (int i = index; i < n; i++) {
        sum += a[i];
        maxsum = Math.max (maxsum, sum);
        solve (a, n, k-1, i+1, sum, maxsum);
    }
}
public static void main (String args[]) {
    int arr[] = {1, 2, 3, 4};
    int k = 3; // K divisions
    int n = 4; // size of an Array
    solve (arr, n, k, 0, 0, 0);
    Syso (ans + "\n");
}
```