Date 31/07/24
Page

GREEDY

It is a problem solving technique where we make the locally optimum choice at each stage & hope to acheive a global optimum.

Pros	Cons
Simple & Easy	A lot of time global optimum is not acheived
Good T. Complexity	

- **Activity Selection** (selecting Disjoints)

Q.→ n activities with start & end times. Select max. no. of activities that can performed by single person, assuming work on single activity at a time. (activities are sorted).

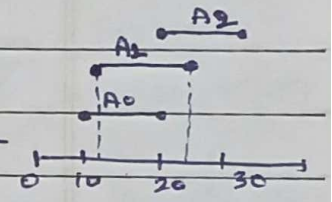
↳ we can sort by own (not needed)

eg. A_0 A_1 A_2

start = [10, 12, 20]

end = [20, 25, 30]

↳ Ans: 2 (A_0 & A_2)



concept of selecting disjoints & ignoring overlapping as :-

∵ only 1 (work) at a time ∴ we choose A_0 & A_2 ∵ 2 works

Disjoints.
(not overlapping)

eg.2 start = [1, 3, 0, 5, 8, 5]

end = [2, 4, 6, 7, 9, 9]

Approach :-

1. end time basis sort
2. Always select first activity (A_0)
3. For non-overlapping / Disjoint → start time >= last chosen activity end time
4. count++.

On observation of Greedy :-

A_0, A_1, A_3, A_4 works can be done at max by single person. (∵ since, sabhi ka next, prev se disjoint hai)

Note :- use sorted end time not start time because in case of starting of A0 is 0 & end is 9 ∴ it will cover all (gets overlapped) by all.

~~key~~ code :→

```
psvm {
    int start[] = { ---- };
    int end[] = { ---- };
```

sorted → (do by yourself if not sorted)

may use Array.sort func. also.

initially count=0 → `int maxAct = 0;`

`ArrayList<Integer> ans = ---;`

Choose 1st already start → `ans.add(0);`

update → (could have directly started from 1) `maxAct = 1;`

`int lastEnd = end[0];`

`for(int i=1; i<end.length; i++){`

checking non-overlapping → `if(start[i] >= lastEnd){`

selecting activity & updating end

```
    maxAct++;
    ans.add(i);
    lastEnd = end[i];
}
}
sout("max activities=" + maxAct);
for(int i=0; i<ans.size(); i++){
sout("A" + ans.get(i) + " ");
}
sout();
}
```

- Explanation of selection of Greedy (Activity Selection) :-

Sol^{n} A: A_0 ——— ——— ---- n_1 → Greedy Ans.

Sol^{n} B: K ——— ——— ---- n_2 → actual Ans. ∵ $(n_2 > n_1)$

to select Greedy :-

replace K with A_0 (non-overlapping) ∵ $[A_0] < [K]$ in size

∴ ——— ——— ——— ---- $n_2 - 1 + 1 = n_2$

∴ $[n_2 = n_1]$

∵ started with A_0 ∴ Greedy Approach Sol^{n}

* **Fractional Knapsack** (Greedy) ≡ Bag (container) 01/08/24

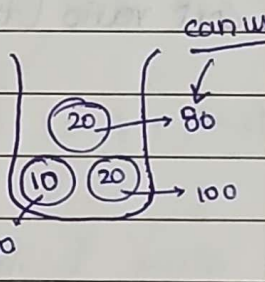
we'will study 0-1 Knapsack in DP

Q. Given the weights and values of N items, put these items in a knapsack of capacity 'W' to get the max. total value in knapsack.

eg. value = [60, 100, 120] ← price

weight = [10, 20, 30]

W = 50



max. ans. = 240

logically, Yes!

Greedy Approach :-

weight ↓ (to put more items) value ↑ (more profit)

∴ pr most favourable situation :-

ratio = value / weight → max.

∴ Jiska ratio highest → select that only.

∴ Process :-

weight = [10, 20, 30]

value = [60, 100, 120]

W = 50

ratio = v/w = [6, 5, 4]

capacity (left) = ~~40~~ 20 0

value = 60 + 100 + 80 = 240

For fraction → (wt highest ratio) * (space wt) = 4 × 20 = 80

A function ← Comparator.comparingDouble(o → o[2]);

Feature of function to write it into shortform | it is an interface for sorting Java objects | Sorting on the basis of col. → 2

1. take ratio always in descending order

Pseudo Code (for ascending ratio) →

```
for (int i = n-1; i>=0; i--){
    if (capacity >= weight[i]){
        include total → capacity - wt.[i]
                      → value + val[i]
    }
    fraction → else {
        value = val + (ratio * capacity)
        break; → knapsack filled
    }
}
```

Code →

```
psvm{
int[] val = {60,100,120};
int[] weight = {10,20,30};
int w = 50;
```

making a 2D array to sort ratio (type: double) →

```
double[][] ratio = new double[val.length][2];
```

0th col → index, 1st col → ratio

```
for(int i=0; i<val.length; i++){
    ratio[i][0] = i;        // index
    ratio[i][1] = val[i]/(double)weight[i];
}
```

Kisi ek ko bhi typecast karne se overall typecasted ∴ if 0.9 for int gives 0 ∴ double ✓

```
Arrays.sort(ratio, Comparator.comparingDouble
        (o → o[1]));
```

↑ (lambda function)

sorting on the basis of 1st col where ratio is stored ∴ ascending order obtained

```
int capacity = w; → initially
int finalValue = 0;
```

now to convert this ascending order ratio into descending :. opp. loop → for(int i = ratio.length; i>=0; i--)

~~kha~~ (kaunse index pr sabse high ratio hai) → int idx = (int) ratio[i][0];

if(capacity >= weight[idx]){

value stored ← finalVal += val[idx];

capacity decreasing step by step ← capacity -= weight[idx];

}else{

fractional value stored ← ~~fra~~ finalVal += (ratio[i][1] * capacity);

capacity = 0;

break;

}

}

sout(finalVal);

* Min. Absolute Difference Pairs.

e.g. A = [1, 2, 3] case 1 : |1-2| + |2-1| + |3-3| = 2

B = [2, 1, 3] case 2 : ...

Ans = 0 (min abs. diff.) case 3 : |1-1| + |2-2| + |3-3| = 0

logically → nos in pair jitne paas pr unka abs. diff. utna hi min. hoga.

Greedy → :. sort the arrays & apply logical approach.

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

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

=> |1-2| + |4-3| + |7-5| + |8-6| = 1+1+2+2 = 6

(min.) Ans

code → `[]A = ✓`

`[]B = ✓`

```
Arays.sort(A);
  ,,     ,, (B);
int  minDiff = 0;
for(int i=0; i< A.length; i++){
    minDiff += Math.abs(A[i] - B[i]);
}
sout(minDiff);
```

* Max. Length Chain of Pairs (type of activity selection)

Given 'n' pairs of nos. in every pair first no. is less than second no. Also a pair (c,d) can come after (a,b) if b<c.
Find the longest chain that can be formed.
(Observe Ques. with similar approach as well) e.g. starting time always less than ending time type Ques.

e.g. pairs = (5,24), (39,60), (5,28), (27,40), (50,90)
longest possible = 3 → (5,24), (27,40), (50,90). ✓

Greedy Approach :-

1. sort pairs (based on 2nd no.)
2. Always select first pair.

⇒ ∴

```
for(int i=1; i<n; i++){
    if(pair→start > last selected→end){
        ans++;
        update last selected
    }
}
```

code ⇒ `int pairs[][] = { {5,24}, {39,60}, {5,28}, {27,40}, {50,90} };`

`Arrays.sort(pairs, Comparator.comparingDouble(o -> o[1]));`

sorting on the basis of end col.

selecting first one initially

```
int chainLen = 1;
int chainEnd = pairs[0][1]   → last selected pair end.
```

```
for (int i = 1; i < pairs.length; i++){
    if (pairs[i][0] > chainEnd){
        (pairs[i][0] → starting)
        chainLen++;
        chainEnd = pairs[i][1];   → now selected wale ka end
    }                               & compare
}
sout(chainLen);
```

complexity: $O(n \log N)$ → loop; for sorting array

* **Indian Coins** → canonical coin system

Q. We are given notes & coins of [1, 2, 5, 10, 20, 50, 100, 500, 2000]. Find min. no. of coins/notes to make change for value 'V'.

eg. V = 551 → ans ⇒ 3 (500 + 50 + 1)

V = 590 → ans ⇒ 4 (500 + 50 + 20 + 20)

Note :- Since, Indian coin is a canonical coin system ∴ Greedy approach works here but for other notes/coins like [1, 50, 100] Greedy fails.

Greedy Approach

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

count = 0

amount = 590

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

Code => Integer coins[] = {1, 2, 5, 10, 20, 50, 100, 500, 2000};

to use comparator fn. (otherwise use loops to reverse)

```
Arrays.sort(coins, Comparator.reverseOrder());

int countCoins = 0;
int amt = 590;
ArrayList ans - - .

for(int i=0; i< coins.length ;i++){
  if ( coins[i] <= amount ){        eg. 100 then we 100 directly
     while ( " ) {
        countCoins++;
        ans.add(coins[i]);
        amt -= coins[i];
     }
  }
}
Sout (countCoins);
```

To show all those coins →

```
for (int i=0; i< ans.size(); i++){
    sout( ans.get(i) + " ");
}
sout();
```

Date 02/08/24
Page

* Job Sequencing Problem

Q. Array of Jobs with deadline & profit ~~if~~ job ~~is~~ should finished before deadline. Every Job takes single unit of time ∴ min. possible dealine for any job is 1. Maximize total profit.

eg
Job A = 4, 20
Job B = 1, 10
Job C = 1, 40
Job D = 1, 30
↓ max. deadline unit
→ Profit

cases:-

① A → time = 1, profit 20
↓
only A ∵ B, C, D has deadline of 1 unit & A covered it already.

② BA → time = 2; profit = 10+20=3
③ CA → time = 2; profit = 40+20=6
④ DA → time = 2; profit = 30+20=5

Hence, Ans. = CA.

Brute Force Approach → (all possible sequence) → takes long time

∴ Greedy Approach ✓ :-

① Jobs sort (based on profit (descending order))
∴ JobC, D, A, B.

②
```
time = 0
for (int i=0; i< jobs; i++){
  if (job(deadline) > time){
    add job in ans.
    time ++
  }
}
```

Check

Code ⇒

```
static class Job{
    int deadline;
    int profit;
    int id;
    (Job constr with all data members)
}

psvm {
int[][] jobsInfo = {{4,20},{1,10}, ... };

// Job[] jobs = new Job[jobsInfo.length]   ArrayList<Job> jobs = new Arra --

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 - obj1.profit);
```

descending order sort of objects ↵ profit

sorting objects ↑ on basis of (jiska ye jyada ho)

for ascending order :- wo to obj1.profit - obj2.profit

```
ArrayList<Integer> seq = new ArrayList<>();
int time = 0;
for(int i=0; i<jobs.size(); i++){
    Job current = jobs.get(i);
    if(current.deadline > time){
        seq.add(current.id);
        time++
    }
}
```

Print seq →

```
sout(seq.size() + "= max jobs");
for(int i=0; i<seq.size(); i++){
    sout(seq.get(i) + " ");
}
sout();
```