Greedy Algorithms

* A Greedy Algorithm is like a problem-solving strategy where you make the best choice at each step without thinking too much about the future. It's like going for the immediate advantage without worrying if it's the absolute best in the end.
* For example, think of a game where you need to collect coins to reach a target amount. The greedy approach would be grabbing the biggest coin available at each moment until you hit the goal. It might not always give you the absolute best result, but it's a quick and simple way to get close.

**Basic:**

* A greedy algorithm always makes the choice that looks best at this moment.
* We hope that a locally optimal choice will lead to a globally optimal solution.
* For some problems, it works.
* Greedy algorithms tend to be easier to code

# 

# A Simple Example

Pick k numbers out of n numbers such that the sum of these k numbers is the largest.

## Algorithm

FOR i = 1 to k

Pick out the largest number and

Delete this number from the input. ENDFOR

1. **for** i=1 to k **do**
2. pick out the largest number
3. delete this number from the input
4. **end for**

# 

# **Optimization Problems**

An optimization problem is the problem of finding the best solution from all feasible solutions

* Fractional knapsack: we maximize our profit
* Activity selection: we maximize the number of activities
* Shortest path problem: we minimize the path length.
* Minimum spanning tree: we minimize the spanning tree weight

## PROBLEM 01. Fractional knapsack

The weights and values of **n** items are given. *The items are such that you can take a whole item or some fraction of it (divisible).*You have a knapsack to carry those items, whose weight capacity is **W**. Due to the capacity limit of the knapsack, it might not be possible to carry all the items at once. In that case, pick items such that the profit (total values of the taken items) is maximized.

Write a program that takes the weights and values of **n** items, and the capacity **W** of the knapsack from the user and then finds the items which would maximize the profit using a greedy algorithm.

|  |  |
| --- | --- |
| **sample input**  n  weight, value  …  W | **sample output** |
| 4  4 20  3 9  2 12  1 7  5 | item 4: 1.0 kg 7.0 taka  item 3: 2.0 kg 12.0 taka  item 1: 2.0 kg 10.0 taka  profit: 29 taka |

**Possible greedy strategies:**

* Pick the lightest item first, then pick the next lightest item and so on.
* Pick the costliest (per-unit value wise) item first, then pick the next costliest item and so on. **(optimal answer)**
* Pick the costliest (total value wise) item first, then pick the next costliest item and so on.

### Pseudo-code (version 1)

**Function FractionalKnapsack**(W, arr[], N):

Sort items in arr by ratio (value / weight) in descending order

cap\_left = W

profit = 0

i = 0

**while** cap\_left > 0 and i < N:

**if** cap\_left >= arr[i].weight:

profit += arr[i].value

cap\_left -= arr[i].weight

**else:**

profit += arr[i].ratio \* cap\_left

cap\_left = 0

i = i + 1

**end while**

return profit

### 

### pseudocode (version 2):

|  |
| --- |
| **Function** FractionalKnapsack(*W, v[n], w[n]*)   1. sort *items* by *vi/wi* descending // vi, wi = value and weight of the ith item 2. *cap\_left* = *W*, *profit* = 0 // cap\_left = capacity left 3. *i* = 1 4. **while** *cap\_left* > 0 **and** *i* <= *n* **do** 5. **if** *cap\_left ≥ wi* **then** 6. *profit = profit + vi* 7. *cap\_left = cap\_left - wi* 8. **else:** 9. *profit = profit + vi \* cap\_left/wi* 10. *cap\_left = 0* 11. *i = i+1* 12. **end if** 13. **end while** |

### 

### pseudocode (version 3):

|  |
| --- |
| **Function** FractionalKnapsack(*W, v[n], w[n]*)   1. sort *items* by *vi/wi* descending // vi, wi = value and weight of the ith item 2. *cap\_left* = *W*, *profit* = 0 // cap\_left = capacity left 3. *i* = 1 4. **while** *cap\_left* > 0 **and** *i* <= *n* **do** 5. *fraction* = min(1.0, *cap\_left/wi*) // fraction = fraction taken from ith item 6. *cap\_left* = *cap\_left* - *fraction* \* *wi* 7. *profit* = *profit* + *fraction* \* *vi* 8. *i++* 9. **end while** |

### 

### 

## // C++ program to solve fractional Knapsack Problem

## #include <bits/stdc++.h>

## using namespace std;

## // Structure for an item which stores weight and

## // corresponding value of Item

## struct Item {

## int profit, weight;

## // Constructor

## Item(int profit, int weight){

## this->profit = profit;

## this->weight = weight;

## }

## };

## // Comparison function to sort Item

## // according to profit/weight ratio

## static bool cmp(struct Item a, struct Item b){

## double r1 = (double)a.profit / (double)a.weight;

## double r2 = (double)b.profit / (double)b.weight;

## return r1 > r2;

## }

## // Main greedy function to solve problem

## double fractionalKnapsack(int W, struct Item arr[], int N){

## // Sorting Item on basis of ratio

## sort(arr, arr + N, cmp);

## double finalvalue = 0.0;

## // Looping through all items

## for (int i = 0; i < N; i++) {

## 

## // If adding Item won't overflow,

## // add it completely

## if (arr[i].weight <= W) {

## W -= arr[i].weight;

## finalvalue += arr[i].profit;

## }

## // If we can't add current Item,

## // add fractional part of it

## else {

## finalvalue+= arr[i].profit\* ((double)W / (double)arr[i].weight);

## break;

## }

## }

## return finalvalue;

## }

## int main(){

## int W = 50;

## Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

## int N = sizeof(arr) / sizeof(arr[0]);

## cout << fractionalKnapsack(W, arr, N);

## return 0;

## }

## PROBLEM 02. Activity Selection Problem

Suppose we have a set ***S = {a\_1, a\_2, …, a\_n}*** of ***n*** proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity ***a\_i*** has a start time ***s\_i*** and a finish time ***f\_i***, where *0 ≤ s\_i < f\_i <* . If selected, activity a\_i takes place during the half-open time interval *[s\_i, f\_i)*. Activities *a\_i* and *a\_j* are compatible if the **intervals *[s\_i, f\_i)*** and *[s\_j, f\_j)* do not overlap. That is, *a\_i* and *a\_j* are compatible if *s\_i ≥ f\_j* or *s\_j ≤ f\_i*. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities.

**Possible greedy strategies:**

* Select the activity that starts first, next select the activity that starts first and does not conflict with the already picked activities
* Select the activity that ends first (this one gives the optimal answer)
* Select the activity that has the shortest duration first

### Pseudocode (version 1.1):

**Note:** *activities are sorted* by their finishing time ascending

procedure MaxActivities(startTimes[], finishTimes[], numberOfActivities)

i = 1 *// The first activity always gets selected*

output i

**for** j := 2 **to** numberOfActivities - 1 **do**

*// If this activity has a start time greater than or equal to the finish time of the previously selected activity, then select it*

if startTimes[j] >= finishTimes[i] then

output j

i = j *//update current activity index*

end if

end for

end procedure

### Pseudocode (version 1.2):

|  |
| --- |
| **Function** Greedy-Activity-Selector (*activities*):   1. sort *activities* by finish time ascending 2. *n* = *activities*.length 3. *A*= {*activities*[1]} // A = selected activities 4. *k* = 1 // k = last chosen activity 5. **for** m=2 **to** n **do** 6. **if** *activities*[*m*]*.start\_time >= activities*[*k*]*.finish\_time* **then** 7. *A*.add( *activities*[*m*] ) 8. *k = i* 9. **end if** 10. **end for** 11. **return** *A* |

### Pseudocode (version 2):

|  |
| --- |
| **Function** Greedy-Activity-Selector (s, f):   1. sort *activities* by finish time ascending 2. n = s.length 3. A = {a\_1} 4. k = 1 5. **for** m=2 **to** n **do** 6. **if** s[m] ≥ f[k] **then** 7. A = A ∪ {a\_m} 8. k = m 9. **end for** 10. **return** A |

## 

## // The following implementation assumes that the activities

## // are already sorted according to their finish time

## // Prints a maximum set of activities that can be done by a

## // single person, one at a time.

## void printMaxActivities(int s[], int f[], int n){

## int i, j;

## cout << "Following activities are selected" << endl;

## // The first activity always gets selected

## i = 0;

## cout << i << " ";

## // Consider rest of the activities

## for (j = 1; j < n; j++) {

## // If this activity has start time greater than or

## // equal to the finish time of previously selected

## // activity, then select it

## if (s[j] >= f[i]) {

## cout << j << " ";

## i = j;

## }

## }

## }

## int main(){

## int s[] = { 1, 3, 0, 5, 8, 5 };

## int f[] = { 2, 4, 6, 7, 9, 9 };

## int n = sizeof(s) / sizeof(s[0]);

## // Function call

## printMaxActivities(s, f, n);

## return 0;

## }

## PROBLEM 03. Greedy Coin Change

Given a value of **V**Rs and an infinite supply of each of the denominations {1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, The task is to find the minimum number of coins and/or notes needed to make the change?

**Examples:**

***Input:****V = 70****Output:****2****Explanation:****We need a 50 Rs note and a 20 Rs note.*

***Input:****V = 121****Output:****3****Explanation:****We need a 100 Rs note, a 20 Rs note, and a 1 Rs coin.*

Follow the steps below to implement the idea:

* Sort the array of coins in decreasing order.
* Initialize **ans**vector as empty.
* Find the largest denomination that is smaller than **remaining amount**and while it is smaller than the **remaining amount**:
  + Add found denomination to **ans**. Subtract value of found denomination from **amount**.
* If amount becomes **0**, then print **ans**.

### Pseudo code (version 1):

**Procedure** minCoinChange(Note):

coins[] = {1000, 500, 100, 50, 20, 10, 5, 2, 1}

num\_of\_coin = length(coins)

change\_coin = []

**for** i from 0 **to** num\_of\_coin - 1 **do**:

**while** Note >= coins[i]:

Note -= coins[i]

change\_coin.append(coins[i])

Output "Minimum number of coins needed: ", length(change\_coin)

Output "Display coins from change\_coin"

for coin in change\_coin:

Output coin, " "

**End Procedure**

### // C++ program to find minimum

### // number of denominations

### #include <bits/stdc++.h>

### using namespace std;

### // All denominations of Indian Currency

### int denomination[]

### = { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };

### int n = sizeof(denomination) / sizeof(denomination[0]);

### void findMin(int V){

### sort(denomination, denomination + n);

### // Initialize result

### vector<int> ans;

### // Traverse through all denomination

### for (int i = n - 1; i >= 0; i--) {

### // Find denominations

### while (V >= denomination[i]) {

### V -= denomination[i];

### ans.push\_back(denomination[i]);

### }

### }

### // Print result

### for (int i = 0; i < ans.size(); i++)

### cout << ans[i] << " ";

### }

### int main(){

### int n = 93;

### cout << "Following is minimal"

### << " number of change for " << n << ": ";

### findMin(n);

### return 0;

### }

## Practice problems: <https://leetcode.com/tag/greedy/>