Greedy Algorithms

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.

pseudocode (version 1):

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

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.          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

Problem Description: Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ 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 $0 \le s_i < f_i < \infty$. If selected, activity ai 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 $s_i \ge f_j$ or $s_j \le f_i$. 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):

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**.

```
// C++ program to find minimum
// number of denominations
#include <bits/stdc++.h>
using namespace std;

// All denominations of 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;
}
```

PROBLEM 04. Finding Minimum Stops [Link1]

Suppose you were to drive from A to B, which is **D** miles away, along a straight road. Your gas tank, when full, holds enough gas to travel **m** miles, and you have a map that gives distances between gas stations along the route. Let **d1** < **d2** < … < **dn** be the locations of all the gas stations along the route where **di** is the distance from St. Louis to the gas station. You can assume that the distance between neighboring gas stations is at most **m** miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm you can to determine at which gas stations you should stop.

Write a code to solve this problem using a **greedy algorithm**. Keep the time complexity of your code O(n).

Sample input **D** **m** **n** **d1 d2 … dn**	Sample output
20 10 8 2 4 5 8 12 14 16 19	stop at gas station 4 (8 miles) stop at gas station 7 (16 miles)
20 10 4 2 8 12 14	Can't reach destination

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;


int main(){
   int D, m, n;
   cin >> D >> m >> n;
   vector<int> dists;
   for (int i=0;i<n;i++){
      int d;
      cin >> d;
      dists.push_back(d);
   }

   int dist_covered = 0;
   int chosen = -1;
   for (int i=0;i<n;i++){
      if (dists[i]>=D) break;
      if (dists[i] - dist_covered <= m ){
         chosen = i;
      } else {
         printf("stop at gas station %d (%d miles)\n",chosen+1, dists[chosen]);
         dist_covered = dists[chosen];
      }
   }
   if (D-dist_covered>m)
      printf("can't reach destination. still %d miles away but gas ran out.\n",D-dist_covered );

}
```

PROBLEM 05. Minimum Swaps for Bracket Balancing

You are given a string of 2N characters consisting of N '[' brackets and N ']' brackets. A string is considered balanced if it can be represented in the form S2[S1] where S1 and S2 are balanced

strings. We can make an unbalanced string balanced by swapping adjacent characters. Calculate the minimum number of swaps necessary to make a string balanced.

Examples:

Input : []][][
Output : 2
First swap: Position 3 and 4
[][]][
Second swap: Position 5 and 6
[][][]
Input : [[][]]
Output : 0
The string is already balanced.

Procedure-1:

```
// C++ program to count swaps required to balance string
#include<bits/stdc++.h>
using namespace std;

// Function to calculate swaps required
int swapCount(string s)
{
        //To store answer
        int ans=0;

        //To store count of '['
        int count=0;

        //Size of string
        int n=s.size();

        //Traverse over the string
        for(int i=0;i<n;i++){
                //When '[' encounters
                if(s[i]=='['){count++;}
                //when ']' encounters
                else{count--;}
                //When count becomes less than 0
                if(count<0){
                        //Start searching for '[' from (i+1)th index
                        int j=i+1;
                        while(j<n){
                                //When jth index contains '['
                                if(s[j]=='['){break;}
                                j++;
                        }
                        //Increment answer
                        ans+=j-i;

                        //Set Count to 1 again
```

```
                        count=1;

                        //Bring character at jth position to ith position
                        //and shift all character from i to j-1
                        //towards right
                        char ch=s[j];
                        for(int k=j;k>i;k--){
                                s[k]=s[k-1];
                        }
                        s[i]=ch;
                    }
                }
        return ans;
}

// Driver code
int main()
{
        string s = "[]][][";
        cout << swapCount(s) << "\n";

        s = "[[][]]";
        cout << swapCount(s) << "\n";

        return 0;
}
```

Procedure-2:

```
// C++ program to count swaps required
// to balance string
#include <bits/stdc++.h>
using namespace std;

long swapCount(string chars)
{
        // Stores total number of Left and
        // Right brackets encountered
        int countLeft = 0, countRight = 0;

        // swap stores the number of swaps
        // required imbalance maintains
        // the number of imbalance pair
        int swap = 0 , imbalance = 0;

        for(int i = 0; i < chars.length(); i++)
        {
                if (chars[i] == '[')
                {
                        // Increment count of Left bracket
                        countLeft++;

                        if (imbalance > 0)
                        {
                                // swaps count is last swap count + total
                                // number imbalanced brackets
                                swap += imbalance;

                                // imbalance decremented by 1 as it solved
                                // only one imbalance of Left and Right
                                imbalance--;
```

```
                }
            }
            else if(chars[i] == ']' )
            {

                // Increment count of Right bracket
                countRight++;

                // imbalance is reset to current difference
                // between Left and Right brackets
                imbalance = (countRight - countLeft);
            }
        }
        return swap;
}

int main(){
        string s = "[]][][";
        cout << swapCount(s) << endl;
        s = "[[][]]";
        cout << swapCount(s) << endl;
        return 0;
}
```

Practice problems: https://leetcode.com/tag/greedy/