Algorithms: Greedy Algorithms - Fractional Knapsack Problems, Task Scheduling Problem - with C Program source codes

These are two popular examples of Greedy Algorithms.

1. Fractional Knapsack Problem

Given the weight of items along with its value and also the maximum weight that we can to take, we have to maximize the value we can get. Unlike 0-1 knapsack, here we are allowed to take fractional items. Hence this problem is called fractional knapsack. Fractional knapsack problem is solvable by greedy strategy.

Greedy algorithm – It obtains the solution of a problem by making a sequence of choices. At each decision point, the best possible choice is made.

The basic idea is to calculate for each item the ratio of value/weight, and sort them according to this ratio. Then take the items with the highest ratios and add them until we can't add the next item as whole. Finally add as much as you can of the next item.

Properties

If the items are already sorted into decreasing order of vi / wi, then the for-loop takes O(n) time. Where n is the total number of items. Therefore, the total time including the sort is in O(n log n) as quicksort's average time complexity is O(nlogn).

Complete Tutorial for the Fractional Knapsack Problem :

Fractional Knapsack

Given the weight of items along with its value and also the maximum weight that we can to take, we have to maximize the value we can get. Unlike 0-1 knapsack, here we are allowed to take fractional items. Hence this problem is called fractional knapsack. Fractional knapsack problem is solvable by greedy strategy.

Greedy algorithm – It obtains the solution of a problem by making a sequence of choices. At each decision point, the best possible choice is made.

The basic idea is to calculate for each item the ratio of value/weight, and sort them according to this ratio. Then take the items with the highest ratios and add them until we can't add the next item as whole. Finally add as much as you can of the next item.

Algorithm:

Firstly, declare a structure item with fields weight and value of each item. Input the total number of items (items) and initialize an array I[items] of structure item. Input the weight and value of each item I, and calculate for each item the ratio of value/weight, and sort them according to this ratio using qsort. Where, qsort() is standard C function for sorting arrays. qsort() takes four arguments:

void **qsort**(void *base, size_t nel, size_t width, int (*compare))

base - is a pointer to the beginning of data array, nel - is a number of elements, width - is a size of each element (in bytes) and compare — is a callback function (pointer to function), which does comparison and returns positive or negative integer depending on result.

Fractional Knapsack Problem - C Program Source Code

```
#include<stdio.h>
/* Given the weight of items along with its value and also the maximum weight that we can to take,
   we have to maximise the value we can get. Unlike 0-1 knapsack, here we are allowed to take fractional
   items. Hence this problem is called fractional knapsack */
typedef struct item
{
        int weight;
        int value;
}item;
int compare(const void *x, const void *y)
{
        item *i1 = (item *)x, *i2 = (item *)y;
        double ratio1 = (*i1).value*1.0 / (*i1).weight;
        double ratio2 = (*i2).value*1.0 / (*i2).weight;
        if(ratio1 < ratio2) return 1;
        else if(ratio1 > ratio2) return -1;
        else return 0;
}
int main()
{
        int items;
        scanf("%d",&items);
        item I[items];
        int iter;
        for(iter=0;iter<items;iter++)
        {
                scanf("%d%d",&I[iter].weight,&I[iter].value);
        }
        qsort(I,items,sizeof(item),compare);
        int maxWeight;
        scanf("%d",&maxWeight);
```

```
        double value = 0.0;
        int presentWeight = 0;
        for(iter=0;iter<items;iter++)
        {
                if(presentWeight + I[iter].weight <maxWeight)
                {
                        presentWeight = presentWeight + I[iter].weight ;
                        value += I[iter].value;
                }
                else
                {
                        int remaining  = maxWeight - presentWeight;
                        value += I[iter].value*remaining *1.0/I[iter].weight;
                        break;
                }

        }
        printf("Maximum value that can be attained is %.6lf\n",value);
}
```

2. Task Scheduling Problem

Task Scheduling

Given a set of events, our goal is to maximize the number of events that we can attend. Let E ={1,2,3…n} be the events we need to attend. Each event has a start time si and a finish time fi, where si < fi. Events i and j are compatible if the intervals [si , fi) and [sj , fj) do not overlap (i.e., i and j are compatible if si ≥ fj or sj ≤ fi. The goal is to select a maximum-size set of mutually compatible events. Greedy algorithm is used for solving this problem. For this we need to arrange the events in order of increasing finish time: f1 ≤ f2 ≤ ……≤ fn.

Properties

If the events are already sorted into increasing order of fi , then the for-loop takes O(n) time. Where n is the total number of events. Therefore, the total time including the sort is in O(n log n) as quicksort's average time complexity is O(nlogn).

Complete Tutorial for the Task Scheduling Problem :

Task Scheduling

Given a set of events, our goal is to maximize the number of events that we can attend. Let E = {1,2,3…n} be the events we need to attend. Each event has a start time s_i and a finish time f_i, where $s_i < f_i$. Events i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., i and j are compatible if $s_i \geq f_j$ or $s_j \leq f_i$. The goal is to select a maximum-size set of mutually compatible events. Greedy algorithm is used for solving this problem. For this we need to arrange the events in order of increasing finish time: $f_1 \leq f_2 \leq ...... \leq f_n$.

Algorithm:

Firstly declare a structure event with fields start_time (starting time of the event), end_time (finishing time of the event) and event_number (the number of event). Input the total number of events (number_of_events) and initialize an array T[number_of_events] of structure event.
Input the start_time and end_time for every event T, and give it an event_number.
Sort the events according to their respective finish time using qsort() function. Where, qsort() is standard C function for sorting arrays. qsort() takes four arguments:

void **qsort**(void *base, size_t nel, size_t width, int (*compare))

base - is a pointer to the beginning of data array, nel - is a number of elements, width - is a size of each element (in bytes) and compare — is a callback function (pointer to function), which does comparison and returns positive or negative integer depending on result.In this case base = T, size_t nel = number_of_events, size_t width = sizeof(event) and a compare function. compare function takes the end_time of every event in T as argument, compares the two and returns end_time of first event – end_time of second

Task Scheduling - C Program Source Code

```
#include<stdio.h>
typedef struct event
{
        int start_time;
        int end_time;
        int event_number;
}event;
int compare(const void *x, const void *y)
{
        event *e1 = (event *)x, *e2 = (event *)y;
        return (*e1).end_time - (*e2).end_time;
}
/* Given the list of events, our goal is to maximise the number of events we can attend. */
int main()
{
        int number_of_events;
        scanf("%d",&number_of_events);
        event T[number_of_events];
        int iter;
        for(iter=0;iter<number_of_events;iter++)
        {
                scanf("%d%d",&T[iter].start_time,&T[iter].end_time);
                T[iter].event_number = iter;
        }
        /* Sort the events according to their respective finish time. */
        qsort(T,number_of_events,sizeof(event),compare);
```

```
        int events[number_of_events]; // This is used to store the event numbers that can be attended.

        int possible_events = 0; // To store the number of possible events

        //Taking the first task
        events[possible_events++] = T[0].event_number;
        int previous_event = 0;

        /* Select the task if it is compatable with the previously selected task*/
        for(iter=1;iter<number_of_events;iter++)
        {
                if(T[iter].start_time >= T[previous_event].end_time)
                {
                        events[possible_events++] = T[iter].event_number;
                        previous_event = iter;
                }
        }
        printf("Maximum possible events that can be attended are %d. They are\n",possible_events);
        for(iter=0;iter<possible_events;iter++)
        {
                printf("%d\n",events[iter]);
        }


}
```

Related Tutorials (Common examples of Greedy Algorithms) :

Elementary cases : Fractional Knapsack Problem, Task Scheduling	Elementary problems in Greedy algorithms - Fractional Knapsack, Task Scheduling. Along with C Program source code.
Data Compression using Huffman Trees	Compression using Huffman Trees. A greedy technique for encoding information.

Some Important Data Structures and Algorithms, at a glance:

Arrays : Popular Sorting and Searching Algorithms			
Bubble Sort	Insertion Sort	Selection Sort	Shell Sort
Merge Sort	Quick Sort	Heap Sort	Binary Search Algorithm
Basic Data Structures and Operations on them			
Stacks	Queues	Single Linked List	Double Linked List
Circular Linked List	1.		

Tree Data Structures			
Binary Search Trees	Heaps	Height Balanced Trees	

Graphs and Graph Algorithms			
Depth First Search	Breadth First Search	Minimum Spanning Trees: Kruskal Algorithm	Minumum Spanning Trees: Prim's Algorithm
Dijkstra Algorithm for Shortest Paths	Floyd Warshall Algorithm for Shortest Paths	Bellman Ford Algorithm	
Popular Algorithms in Dynamic Programming			
Dynamic Programming	Integer Knapsack problem	Matrix Chain Multiplication	Longest Common Subsequence
Greedy Algorithms			
Elementary cases : Fractional Knapsack Problem, Task Scheduling	Data Compression using Huffman Trees		