DEsign and analysis of algorithms

By Utsav Balar

201903103510391

BTech CE-C Batch 1

LAB MANUAL

**Practical 5**

**Aim: A thief robbing a store finds n items, ith item is worth vi dollers and weights wi pounds where vi and wi are integers. He wants to take as valuable a load as possible but he can carry atmost W pounds in his knapsack where W is**

**an integer. Which items should he take, where condition is that he is**

**allowed to take or select fractional part of an item?**

**w = 50, n = 3**

Table

Description automatically generated

**Code:**

// Knapsack problem algorithm

#include <stdio.h>

/\*\*

\* swap - Swapping algorithm

\* @x: Number 1

\* @y: Number 2

\*/

void swap(float \*x, float \*y)

{

float temp = \*x;

\*x = \*y;

\*y = temp;

}

/\*\*

\* fractional\_knapsack - Fractional knapsack algorithm using Greedy approach

\* @w: List of all the objects weight

\* @V: List of all the objects value

\* @W: Capacity of the knapsack

\* @n: Total numbers of objects

\*/

void fractional\_knapsack(float w[], float V[], float W, int n)

{

float x[3], weight = 0.0, profit = 0.0;

// Initialization

for (int i = 0; i < n; i++)

x[i] = 0.0;

for (int i = 0; weight < W; i++)

{

if (weight + w[i] < W)

{

x[i] = 1.0;

weight = weight + w[i];

profit = profit + V[i];

// printf("Object: %d: \tx[%d] = %0.1f\tweight = %0.1f\tw[%d] = %0.1f\tV[%d] = %0.1f\tW = %0.1f\n",

// i + 1, i + 1, x[i], weight, i + 1, w[i], i + 1, V[i], W);

}

else

{

x[i] = (W - weight) / w[i];

weight = W;

profit = profit + (x[i] \* V[i]);

// printf("Object: %d: \tx[%d] = %0.1f\tweight = %0.1f\tw[%d] = %0.1f\tV[%d] = %0.1f\tW = %0.1f\n",

// i + 1, i + 1, x[i], weight, i + 1, w[i], i + 1, V[i], W);

}

}

// for (int i = 0; i < n; i++)

// {

// if (x[i] == 1.0)

// printf("\nObject added = %d\t\tProfit = $%0.1f\tWeight = %0.1f",

// i + 1, V[i], w[i]);

// else if (x[i] > 0.0)

// printf("\nAdded %0.1f part of Object %d\tProfit = $%0.1f\tWeight %0.1f",

// x[i], i + 1, (x[i] \* V[i]), w[i]);

// }

printf("\nThe Maximum Value for %d objects with load %0.1f = $%0.1f\n\n",

n, W, profit);

}

/\*\*

\* knapsack - Knapsack algorithm without considering fractional weight

\* @w: List of all the objects weight

\* @V: List of all the objects value

\* @W: Capacity of the knapsack

\* @n: Total numbers of objects

\*/

void knapsack(float w[], float V[], float W, int n)

{

float x[3], weight = 0.0, profit = 0.0;

// Initialization

for (int i = 0; i < n; i++)

x[i] = 0.0;

for (int i = 0; i < n; i++)

{

if (weight + w[i] < W)

{

x[i] = 1.0;

weight = weight + w[i];

profit = profit + V[i];

// printf("Object: %d: \tx[%d] = %0.1f\tweight = %0.1f\tw[%d] = %0.1f\tV[%d] = %0.1f\tW = %0.1f\n",

// i + 1, i + 1, x[i], weight, i + 1, w[i], i + 1, V[i], W);

}

}

// for (int i = 0; i < n; i++)

// {

// if (x[i] == 1.0)

// printf("\nObject added = %d\t\tProfit = $%0.1f\tWeight = %0.1f",

// i + 1, V[i], w[i]);

// }

printf("\nThe Maximum Value for %d objects with load %0.1f = $%0.1f\n\n",

n, W, profit);

}

int main(void)

{

// Start: provided from the question

int object[3] = {1, 2, 3};

float weight[3] = {10, 20, 30};

float value[3] = {60, 100, 150};

float capacity = 50;

int n = 3;

// End:

/\* NOTE: Create an array for storing value/weight

\* for all the objects

\*/

float ratio[n];

for (int i = 0; i < n; i++)

{

ratio[i] = value[i] / weight[i];

// printf("Object: %d\t ratio=%0.1f\n", i, ratio[i]);

}

/\* NOTE: Re-Arrange the array in such a way

\* that value/weight is in descending order

\*/

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (ratio[i] > ratio[j])

{

swap(&ratio[i], &ratio[j]);

swap(&weight[i], &weight[j]);

swap(&value[i], &value[j]);

}

}

}

int choice = 0; // Default is Fractional(Greedy)

printf("As a thief yourself Enter the choice:\n0 = fractional\t1 = not fractional: \n");

scanf("%d", &choice);

// Validation for user choice

if (choice > 1 || choice < 0)

{

printf("Please input with either 0 or 1: ");

scanf("%d", &choice);

}

switch (choice)

{

case 0:

printf("Using Fractional values\n");

fractional\_knapsack(weight, value, capacity, n);

break;

case 1:

printf("Not using fractional values\n");

knapsack(weight, value, capacity, n);

break;

default:

break;

}

return 0;

}

**Output: (When fractional part was allowed)**

A picture containing text

Description automatically generated

**Output: (When fractional part was not allowed)**

Text

Description automatically generated

**Practical 6**

**Aim: You are given n activities with their start and finish 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.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Created macro for Random number generation

#define randnum(min, max) ((rand() % (int)(((max) + 1) - (min))) + (min))

void printActivities(int start[], int finish[], int n)

{

int i = 0, j;

printf("\nActivities selected: ");

printf("%d\t", i);

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

{

if (start[j] >= finish[i])

{

printf("%d\t", j);

i = j;

}

}

}

// Function to swap two numbers

void swap(int array[], int i, int j)

{

int temp = 0;

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

// Function to all the elements in an array

void printArr(int array[], int size)

{

for (int i = 0; i < size; ++i)

{

printf("%d\t", array[i]);

}

printf("\n");

}

int main()

{

int end = 10, i, temp = 0, r;

int \*start\_t, \*finish\_t;

int size = end - 1;

// Dynamic memory allocation for size of activities

start\_t = malloc(sizeof(int) \* size);

if (start\_t == NULL)

{

printf("malloc of size %d failed!\n", size);

return -1;

}

finish\_t = malloc(sizeof(int) \* size);

if (finish\_t == NULL)

{

printf("malloc of size %d failed!\n", size);

return -1;

}

// To avoid repetition with each run

srand((unsigned int)time(0));

// Fill array with numbers in range (start, end)

for (int temp = 0, i = 1; temp < size; i++, temp++)

start\_t[temp] = i;

// Store all the random numbers in the array

for (i = size - 1; i > 0; i--)

{

r = randnum(1, size);

swap(start\_t, i, r);

}

// Store all the random numbers in the array

for (i = size - 1; i > 0; i--)

{

r = randnum(1, size);

finish\_t[i] = r;

}

printf("\nActivities: \t");

for (int j = 1; j < size + 1; j++)

printf("%d\t", j);

printf("\n\nStart times: \t");

printArr(start\_t, size);

printf("Finish times: \t");

printArr(finish\_t, size);

printActivities(start\_t, finish\_t, size);

free(start\_t);

free(finish\_t);

return 0;

}

**Output:**

Graphical user interface, text

Description automatically generated with medium confidence

**Practical 7**

**Aim: Supposed customer has purchased some items from Mall. He has the bill of Rs. 732 to be paid at the billing counter. Customer has the options in his**

**Wallet of Rs. {500,100, 50, 20, 10, 1}. Your problem is to devise an algorithm**

**For paying given amount to billing counter using the smallest possible**

**number of coins. You can use one option more than once.**

**Code:**

#include <stdio.h>

#define min(x, y) ((x) < (y) ? (x) : (y))

#define MAX 999999

/\*\*

\* coin\_change - Making coin change algorithm using dynamic programming

\* @d: pointer for list of available coins

\* @N: Total amount given

\* @n: Total No. of different coins available

\*/

int coin\_change(int \*d, int N, int n)

{

int c[n][N + 1];

int i = 0, j = 0;

// Initialize whole 2-D array to 0

for (i = 0; i < n; i++)

for (j = 0; j < N + 1; j++)

c[i][j] = 0;

// Set column 1 [j=0] as 0

for (i = 0; i < n; i++)

c[i][0] = 0;

for (i = 1; i < n + 1; i++)

{

for (j = 1; j < N + 1; j++)

{

if (i == 1 && j < d[i - 1])

c[i - 1][j] = MAX;

else if (i == 1)

{

c[i - 1][j] = (1 + c[i - 1][j - d[i - 1]]);

}

else if (j < d[i - 1])

{

c[i - 1][j] = c[(i - 1) - 1][j];

}

else

{

c[i - 1][j] += min(c[(i - 1) - 1][j], 1 + c[i - 1][j - d[i - 1]]);

}

}

}

// for (i = 0; i < n; i++)

// {

// printf("i = %d\td[i] = %d\t", i, d[i]);

// for (j = 0; j < N + 1; j++)

// {

// printf("c[%d][%d]=%d\t", i, j, c[i][j]);

// }

// printf("\n");

// }

return c[n - 1][N];

}

/\*\*

\* swap - Swapping algorithm

\* @x: Number 1

\* @y: Number 2

\*/

void swap(int \*x, int \*y)

{

float temp = \*x;

\*x = \*y;

\*y = temp;

}

/\*\*

\* partition - Function to partition array and return the pivot element index

\* @arr: The required Array to sort

\* @start: Starting index of the array to sort

\* @end: Ending index of the array to sort

\*/

int partition(int arr[], int start, int end)

{

int pivot = arr[end];

int pindex = start - 1;

for (int i = start; i < end; i++)

{

if (arr[i] <= pivot)

{

pindex++;

swap(&arr[i], &arr[pindex]);

}

}

swap(&arr[pindex + 1], &arr[end]);

return pindex + 1;

}

/\*\*

\* quicksort - Function to sort an array using quicksort algorithm

\* @arr: The required Array to sort

\* @start: Starting index of the array to sort

\* @end: Ending index of the array to sort

\*/

void quicksort(int arr[], int start, int end)

{

if (start < end)

{

int pindex = partition(arr, start, end);

quicksort(arr, start, pindex - 1);

quicksort(arr, pindex + 1, end);

}

}

int main()

{

int d[] = {500, 100, 50, 20, 10, 1};

int N = 732;

int n = sizeof(d) / sizeof(d[0]);

// Why quicksort?: Provides Best time complexity for average case

quicksort(d, 0, n - 1);

printf("\nNo. of coins required are: %d\n", coin\_change(d, N, n));

return 0;

}

**Output:**

Text

Description automatically generated