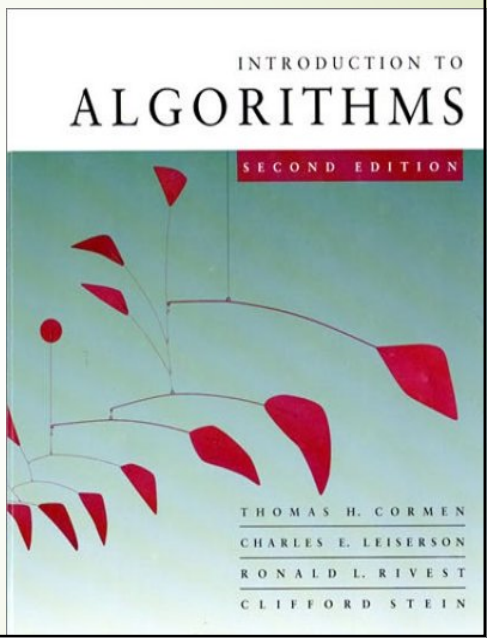
Greedy Algorithms

SUBJECT-CODE : KCS-503

Dr. Ragini Karwayun



1

Dynamic Programming

- Dynamic programming solves optimization problems by combining solutions to sub-problems.
- Dynamic programming is applicable when sub-problems are not independent, that is, Subproblems share sub-problems
- Solve every sub-problem only once and store the answer in the table for use when it reappear.
- The **key** is to store the solutions of sub-problems to be **reused** in the future
- E.g.: Fibonacci numbers:
  - Recurrence: F(n) = F(n-1) + F(n-2)
  - Boundary conditions: F(1) = 0, F(2) = 1
  - Compute: F(3) = 1, F(4) = 2, F(5) = 3
- A divide and conquer approach would repeatedly solve the common subproblems.
  - Top down approach.
- Dynamic programming solves every subproblem just once and stores the answer in a table.
  - Bottom up approach.

2

Divide-and-Conquer Approach

- Partition the problem into independent sub-problems
- Solve the sub-problems recursively
- Combine solutions of sub-problems
- **Example**: Merge Sort, Fibonacci numbers:
  - Recurrence: F(n) = F(n-1) + F(n-2)
- A divide-and-conquer approach will do more work than necessary

3

Greedy Algorithms

- Optimization problem: there can be many possible solution. Each solution has a value, and we wish to find *a* solution with the optimal (minimum or maximum) value
- Greedy algorithm: an algorithmic technique to solve optimization problems
  - Always makes the choice that looks best at the moment.
  - Makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- A greedy algorithm is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

4

Greedy Algorithms

- **Greedy is a strategy that works well on optimization problems with the following characteristics:**
- **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum. We make the choice that looks best in the current problem, without considering results for the subproblems.
- **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems. The second property may make greedy algorithms look like dynamic programming. This property is essential for both dynamic programming and greedy approach.
- In Dynamic programming, we make a choice at each step, but the choice usually depends on the solutions of the subproblems. It solves the problem in a bottom up manner.
- Whereas Greedy strategy usually progresses in a top-down manner.

5

Greedy Algorithms

Design/Correctness

1. Cast the problem as one where you make choices and each choice results in a smaller subproblem to solve.
2. Prove that the greedy choice property holds.
3. Demonstrate that the solution to the subproblem can be combined with the greedy choice to get an optimal solution for the original problem.

6

Elements of the Greedy Strategy

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, but, sometimes it does.
- Following steps are performed in a greedy algorithm:
  - Determine the optimal substructure of the problem.
  - Develop a recursive solution.
  - Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
  - Show that all but one of the subproblems induced by having made the greedy choice are empty.
  - Develop a recursive algorithm that implements the greedy strategy.
  - Convert the recursive algorithm to an iterative algorithm.

7

Elements of the Greedy Strategy

- More generally, we design greedy algorithms according to the following sequence of steps:
  - Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  - Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
  - Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

8

Elements of the Greedy Strategy

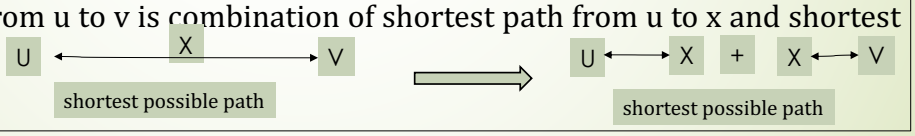
- How can one tell if a greedy algorithm will solve a particular optimization problem?
- There is no way in general, but the greedy-choice property and optimal sub-structure are the two key ingredients.
- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available

9

Optimal Substructure Property

- The first step in solving an optimization problem by dynamic or greedy programming is to characterize the structure of an optimal solution.
- We say that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming as well as greedy algorithms might apply.
- Investigating the optimal substructure of a problem by iterating on subproblem instances is a good way to infer a suitable space of subproblems for dynamic programming.
- For example, the Shortest Path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.



10

Optimal Substructure Property

- There is a common pattern in discovering optimal substructure :
  - Show that a solution to the problem consists of making a choice. Making a choice leaves one or more subproblems to be solved.
  - Suppose that for a given problem, a choice is given that leads to an optimal solution.
  - Given this choice, it is determined which subproblems occur as a result and how to best characterize the resulting space of subproblems.
  - Finally show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal.
  - Optimal substructure varies across problem domain in two ways :-
    1. How many subproblems are used in an optimal solution to the original problem, and
    2. How many choices we have in determining which subproblem(s) to use in an optimal solution.

Activity-Selection Problem

- **Scheduling several competing activities that require exclusive use of a common resource , with a goal of selecting a maximum size set of mutually compatible activities.**
- **Definition –**
- S={1, 2,…, n} – activities that wish to use a resource
- Each activity has a start time s_i and a finish time f_i , where $s_i \leq f_i$
- If selected, activity i takes place during the half-open interval $[s_i , f_i]$
- Activities i and j are compatible if $[s_i , f_i)$ and $[s_j , f_j)$ don't overlap » $s_i \geq f_j$ or $s_j \geq f_i$
- The activity-selection problem is to select a maximum-size set of mutually compatible activities where each activity is non interfering.
- Greedy-Activity-Selector Algorithm – Suppose $f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$ – $\Theta(n)$ if the activities are already sorted initially by their finish times

Activity-Selection Problem

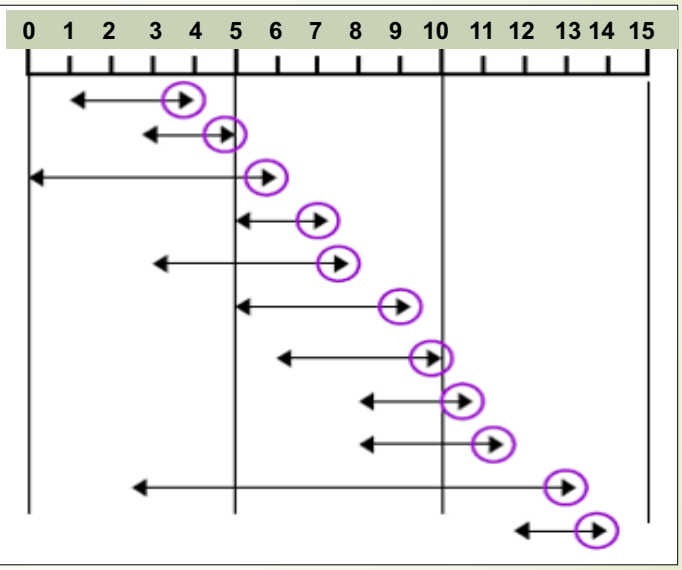
- Greedy-Activity-Selector Algorithm
  - Suppose $f_1 \le f_2 \le f_3 \le \ldots \le f_n$
  - Greedy Choice : $\Theta(n)$ if the activities are already sorted initially by their finish times
- Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- What is the maximum number of activities that can be completed?
  - Activities those are non interfering.
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

Activity-Selection Problem

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



A Recursive Greedy Algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j) (Initial call (s,f,1,n))

```
1 m ← i +1
2 while m < j and s_m < f_i        ▷Find the first activity in S_ij
3    m ← m + 1
4 if m < j
5    return {a_m}∪RAS(s, f, m, j)
6 else
7    return Ø
```


An Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  n ← length[s]
2  a ← {a_1}
3  i ← 1
4  for m ← 2 to n
5     do if s_m ≥ f_i
6           then A ← A ∪ {a_m}
7                i ← m
8  return A
```

- **Assuming activities are sorted by finish time**

An Iterative Greedy Algorithm

i	N	M	Output
0	12	1	$\{a_1\}$
1	12	2,3,4	$\{a_1,a_4\}$
4	12	5,6,7,8	$\{a_1,a_4,a_8\}$
8	12	9,10,11	$\{a_1,a_4,a_8,a_{12}\}$

- **Assuming activities are sorted by finish time**

Few Examples

Q2.

Si. → 1 2 3 4 5 6 7 8 9 10

Fi. → 5 3 4 6 7 8 11 10 12 13

Q3.

Si → 1 2 3 4 7 8 9 9 11 12

Fi. → 3 5 4 7 10 9 11 13 12 14

i	n	m	Output

KNAPSACK PROBLEM

- A thief robbing a store finds n items: the *i* th item is worth v_i dollars and weighs w_i pounds, where both v_i and w_i are integers.
- The thief wants to take as valuable a load as possible but he can carry only W pounds in his knapsack, where W is some integer.
- Problem: Which items should the thief take?
- Two versions of the problem:
- *(a)* ***0-1*** : For each item, the thief must either take it or leave it, in other
- words, he cannot take a fraction of an item.
- *(b)* ***Fractional*** : The thief can take fractions of items.

19

0-1 Vs FRACTIONAL KNAPSACK

- One wants to pack n items in a knapsack(bag) with the capacity W.
- The i^{th} item is worth v_i (value) and weighs w_i (weight).
- Maximize the value but cannot exceed W pounds.
- v_i , w_i , W are integers.
- Each item has only one instance.
- 0-1 knapsack
  - each item as a whole can be taken or not taken. Items are not divisible.
- Fractional knapsack
  - fractions of items can be taken

20

0-1 Vs FRACTIONAL KNAPSACK

- Both exhibit the optimal-substructure property.
- **0-1:** If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W-w_j$
- **Fractional:** If w pounds of item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most W-w that can be taken from other n-1 items plus $(w_j - w)$ of item j.
- **0-1** knapsack can be solved using Dynamic programming.
- **Fractional** Knapsack can be solved with Greedy approach also.

21

0-1 Vs FRACTIONAL KNAPSACK

- The greedy approach does not work for 0-1 knapsack problem.

	A	B	C	capacity of the bag = 50
v_i	60	100	120	
w_i	10	20	30	
v_i/w_i	6	5	4	(per unit cost)

- According to the greedy strategy we will take A & B but cannot take C as the total weight will exceed the capacity > 50.
- Total value of items in the bag = 60 + 100 = 160 which is not optimal as the optimal answer will be to take B & C, value = 220.
- Greedy will have A,B & 2/3 of C , value = 160 + (2/3 * 120) = 240.

22

FRACTIONAL KNAPSACK

- Fractional Knapsack can be solvable by the greedy strategy
  1. Compute the value per pound v_i / w_i for each item.
  2. Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  3. If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room.
  4. O(n lg n) (we need to sort the items by value per pound)

FRACTIONAL KNAPSACK

Fractional-Knapsack(v,w,W)

1. Sort the list in decreasing order of v_i/w_i
2. Load ← 0; i← 1; value ← 0;
3. while load < W AND i <= n
4. if w_i < (W-load)
5. Then take all of item i else take (W-load)/ w_i of item i
6. add what was taken to load and value

   [load = load + w_i]

   [value = value + v_i]

FRACTIONAL KNAPSACK

Fractional-Knapsack(v,w,W)

1. Sort the list in decreasing order of v_i/w_i
2. Load ← 0; i← 1; value ← 0;
3. while load < W AND i <= n
4. if w_i < (W-load)
5. Then take all of item i else take (W-load)/ w_i of item i
6. add what was taken to load and value [load = load + w_i]. [value = value + v_i]

Q1. n=3, W = 20. w_i = (18,15,10) v_i = (25,24,15)

v_i/ w_i = 25/18, 24/15, 15/10. → 1.3, 1.6, 1.5. sort → I_2 I_3. I_1

i.	Load in the bag.	Remaining Load.	Value in the bag
	0	20	0
1	15	20-15=5	24
2	15+[(20-15)/10.	0	25+ (5/10 * 15) = 31.5

25

FRACTIONAL KNAPSACK

Q2. W=15 v_i = (10,5,15,7,6,18,3). w_i = (2,3,5,7,1,4,1)

v_i/ w_i = 10/2, 5/3, 15/5, 7/7, 6/1, 18/4, 3/1

= 5 1.6 3 1 6 4.5 3

(2) (6) (4) (7) (1) (3) (5)

i.	Load in the bag.	Remaining Load.	Value in the bag
	0	15	0
1	1	15-1 =14	6
2.	1+2=3	14-2 = 12	6+10 = 16
3.	3+4=7	12-4 = 8	16+18 = 34
4	7+5=12	8-5 = 3	34+15 = 49
5	12+1=13	3-1 = 2	49+3 = 52
6.	13+[(15-13)/3]	0	52+(2/3 * 5) = 55.3

26

FRACTIONAL KNAPSACK

c. Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. Find the value of $V_{opt} - V_{greedy}$

Item	I_1	I_2	I_3	I_4
W	10	7	4	2
V	60	28	20	24

Huffman codes

- Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.
- Huffman's algorithm constructs an optimal prefix code called a Huffman code.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree.
- A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

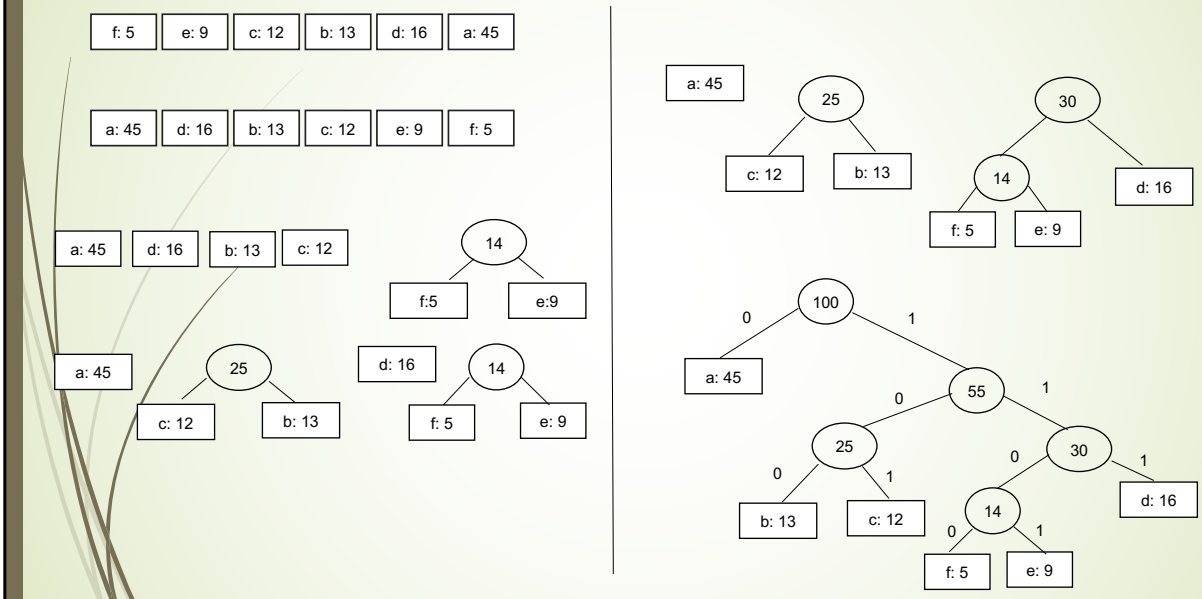
Huffman codes

HUFFMAN(C)

```
1 n ← |C|
2 Q ← C
3 for i ← 1 to n – 1
4     allocate a new node z
5     left[z]← x ← EXTRACT-MIN(Q)      // min priority queue
6     right[z]← y ← EXTRACT-MIN(Q)
7     f [z]← f [x] + f [y]
8     INSERT(Q, z)
9 return EXTRACT-MIN(Q)     // Return the root of the tree.
```

Running time = O(n lg n)

Huffman codes



Huffman codes

Final Coding

Character	Coding	No. bits required	Frequency	Total bits
a	0	1	45	45
b	101	3	13	39
c	100	3	12	36
d	111	3	16	48
e	1101	4	9	36
f	1100	4	5	20

Total characters = 100
Total bits required by the normal text using 8 bit Ascii code = 8*100 = 800bits
Total bits required by the compressed text using Huffman coding = 224

31

Huffman codes

Ex: Given the chars S <a, b, c, d, e, f> with the following probability P <29, 25, 20, 12, 05, 09>. Build a binary tree according to the greedy approach.

32

A Task-Scheduling Problem

- A ***unit-time task*** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete.
- Given a finite set *S* of unit-time tasks, a ***schedule*** for *S* is a permutation of *S* specifying the order in which these tasks are to be performed.
- The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

33

A Task-Scheduling Problem

A task-scheduling problem

- The problem of scheduling unit-time tasks with deadlines and penalties for a single processor has the following inputs:
  - a set $S = \{a_1, a_2, \ldots, a_n\}$ of n unit-time tasks;
  - a set of n integer deadlines $d_1, d_2, \ldots, d_n$, such that each d_i satisfies $1 \le di \le n$ and task a_i is supposed to finish by time d_i;
  - a set of n nonnegative weights or penalties $w_1, w_2, \ldots, w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.
- We are asked to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

34

A Task-Scheduling Problem

GreedyJob(d. J, n)

J = {1} // assign 1st job

for i = 2 to n

if (all jobs in J U{i} can be completed by their deadlines)

J = J U {i}

Time slots : [i-1, i] where $1 \le i \le b$ such that

$b = \min \{ n, \max(d_i)\}$

where n = no. of tasks and

d_i = maximum deadline

The optimal greedy scheduling algorithm:

1. Sort penalties/profits in non-increasing order.
2. Place each job at latest time that meets its deadline.
3. Nothing is gained by scheduling it earlier but scheduling it earlier could prevent another more profitable job from being done.

A Task-Scheduling Problem

- Q1. N = 5 $\{P_1..P_5\}$ = {20,15,10,5,1} and $\{d_1..d_5\}$={2,2,1,3,3}
- Number of slots b = min[n, max(d_i)] = min[5,3] = 3
- = [α-1, α] for 1<=α<= b → [0,1], [1,2], [2,3]

J.	Assigned slots	Job considered	Action	Profit
ϕ	None	1	Assign to [1,2]	20
{1}	[1,2]	2	Assign to [0,1]	20+15=35
{1,2}	[0,1] [1,2]	3	Reject	35
{1,2}	[0,1] [1,2]	4	Assign to [2,3]	35+5=40
{1,2,4}	[0,1] [1,2] [2-3]	5	Reject	40

Penalty = 10+1 (remaining jobs)

A Task-Scheduling Problem

- Q2. N = 5 $\{P_1..P_4\}$ = {50,10,15,30} and $\{d_1..d_4\}$={2,1,2,7}
- Arrange these in decreasing order of profits
- P_1. P_4. $P_3..P_2$, b= min {4,7} = 4 → [0,1],[1,2],[2,3],[3,4]

J.	Assigned slots	Job considered	Action	Profit
ϕ	None	1	Assign to [1,2]	50
{1}	[1,2]	4	Assign to [3,4]	50+30=80
{1,4}	[1,2] [3,4]	3	Assign to [0,1]	80+15=95
{1,4,3}	[0,1] [1,2] [3,4]	2	Reject	95

Penalty = 10 (remaining job)

37

A Task-Scheduling Problem

- N = 7 $\{P_1..P_7\}$ = {3,5,20,18,1,6,30} and $\{d_1..d_7\}$={1,3,4,3,2,1,2}

  Number of slots b = min[n, max(d_i)]

J.	Assigned slots	Job considered	Action	Profit
ϕ	None	1		

Penalty = (remaining job)

38

A Task-Scheduling Problem

- N = 7 $\{P_1..P_7\} = \{70,60,50,40,30,20,10\}$ and $\{d_1..d_7\}=\{4,2,4,3,1,4,6\}$

  Number of slots b = min[n, max(d_i)]

J.	Assigned slots	Job considered	Action	Profit
ϕ	None	1		

Penalty = (remaining job)

Milk Delivery

A private dairy has *n* milk delivery vans. The company has mapped out *n* delivery routes. Each route has to be served once in the morning and once in the evening. Each van covers one morning route and one evening route, but these may be different routes. Each route has fixed delivery volumes in the morning and evening, possibly different.

The dairy's license limits the number of packets a van can deliver to *p* packets per day. If a van delivers more than *p* packets, the company has to pay a fine of *f* per additional packet.

Given the delivery volumes of the morning and evening routes, your task is to find the minimum fine the company has to pay if it optimally allocates morning and evening routes to each delivery van.

For instance, suppose there are 3 routes, the packet limit per day is 24, the fine per additional packet is 4, the morning volumes for the three routes are [10,17,12] and the evening volumes for the three routes are [11,9,24]. Then, the minimum fine to be paid is 48. This can be achieved by pairing the routes as follows: (10,24), (17,9), (12,11).

- Solution hint :Minimize the average morning plus evening volume by pairing up small volumes with large volumes.

Input format

The first line of the input has three space-separated integers *n*, *p* and *f*, where *n* is the number of milk routes, *p* is the daily packet limit and *f* is the fine to be paid for each packet above the limit.

The second line has *n* space-separated integers, corresponding to the morning delivery volumes of delivery routes *1,2,...n*.

The third line has *n* space-separated integers, corresponding to the evening delivery volumes of delivery routes *1,2,...n*.

Output format

Your output should be a single integer, the minimum fine to be paid if the routes are paired up optimally.

Huffman codes

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n=0,p=0,f=0,fine=0, morning[5000], evening[5000];

   cin >> n >>p >>f;

   for (int i = 0; i < n; ++i)
        cin >> morning[i];

   for (int i = 0; i < n; ++i)
        cin >> evening[i];
```

```
  sort(morning, morning+n);

   sort(evening, evening+n, greater<int>());

   for (int i = 0; i < n; ++i)
   { if (morning[i] + evening[i] > p)
      { fine = fine + ((morning[i]+evening[i]) - p);
      }}
   fine = fine *f;
   cout << fine <<endl;



     return 0;
}
```