Data Structures and Algorithms Design

BITS Pilani

Hyderabad Campus

Febin. A. Vahab

2019-20

ONLNE SESSION 10 -PLAN

Session(#)	List of Topic Title	Text/Ref Book/external resource
10	**M5:Algorithm Design Techniques** Greedy Method - Design Principles and Strategy, Fractional Knapsack Problem, Task Scheduling Problem Minimum Spanning Tree, Shortest Path Problem - Dijkstra's Algorithm	T1: 5.1, 7.1, 7.3

Greedy method-Change making problem

25, 10, 5, 1

1, 25 coin
2, 5 coin
2, 1 coin
_____
4 coins

→ Infinite supply of each denomination
→ make a change for 32
→ minimum number of coins.

greedy — optimal

25, 10, 5, 1

2, 25 coin
1, 10 coin
4, 1 coin
_____
7 coins

→ Infinite supply of each coin
→ make a change for 64
→ minimum no: of coins

greedy — optimal

Example 2

denominations available are ✓✓

$$15, 10, 1 = \boxed{20} \quad \boxed{5}$$

X ⎰ 1, 15 coins ⎱
  ⎱ 5, 1 coins ⎰

6 coins

greedy
not optimal

2, 10 coins

2 coins.
optimal

→ make a change for 20
→ minimum no: of coins

→ no greedy choice property

→ no optimal substructure property.
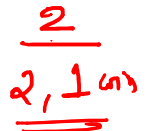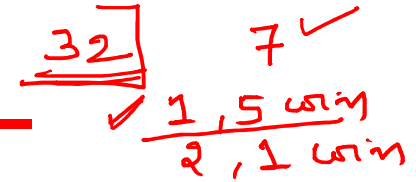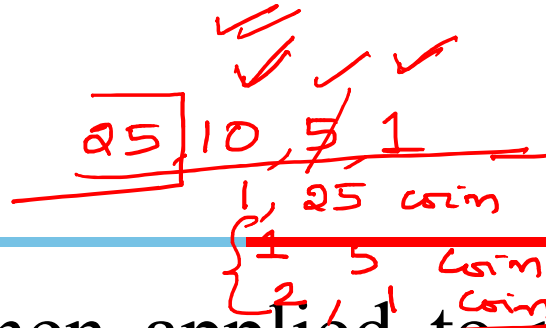
Greedy Method

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize

Greedy Method

(handwritten: 25 10 5 1 / 1, 25 coin / 1, 5 coin / 2, 1 coin / 32 7 / 1, 5 coin / 2, 1 coin / 2 / 2, 1 coin)

- It works best when applied to problems with the **greedy-choice** property:
  - ie, a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - When a locally optimal choice can lead to a globally optimal solution, the problem has this property.

- **Optimal Substructure property**: An optimal solution to the problem contains optimal solution to the sub problems.

Making Change-Example 2

- **Problem:** A dollar amount to reach and a collection of coin amounts to use to get there.

- **Configuration:** A dollar amount yet to return to a customer plus the coins already returned

- **Objective function:** Minimize number of coins returned.

- **Greedy solution:** Always return the largest coin you can

Making Change

40

- **Example 1:** Coins are valued $.32, $.08, $.01
  - Has the greedy-choice property, since no amount over $.32 can be made with a minimum number of coins by omitting a $.32 coin.

- **Example 2:** Coins are valued $.30, $.20, $.05, $.01
  - Does not have greedy-choice property, since $.40 is best made with two $.20's, but the greedy solution will pick three coins (which ones?)

40 [32,8,1]

1,32 coin
1, 8 com

2 coins

8 [8,1]

1,8 coins

1, 30 coin
9, 5 coins

3 coins

2, 20 coins

2 coins

10

9,5 coins

Greedy Method

- An algorithm design strategy that always tries to find the best solution for each sub problem with the hope that it will yield a good solution for the problem as a whole.

- It makes the choice that looks best at the moment.

- Don't worry about the effect these choices have in the future.

- At each step the choice should be
  - Feasible
  - Locally optimal
  - Irrevocable

Greedy Method

- Applied for **optimization problems**

- In order to solve a given optimization problem, we proceed by a sequence of choices.

- The sequence starts from some well-understood starting configuration, and then iteratively makes the decision that seems best from all of those that are currently possible.

- Greedy approach does not always lead to an optimal solution. It works optimally for problems with **greedy-choice** property

The Fractional Knapsack Problem
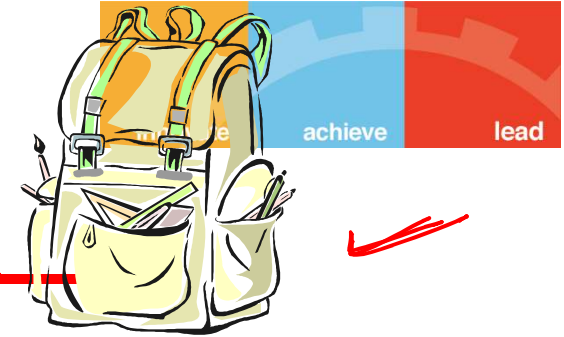
- Given: A knapsack of capacity M and n objects of weights w1,w2,..,wn with profits p1,p2,p3..,pn.

- **Goal: Choose items with maximum total benefit but with weight at most M.**

The Fractional Knapsack Problem

0/1

- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
- Let x1,x2,..,xn be the fractions of objects that are supposed to be added to knapsack.

 - Maximize $\sum_{i=1}^{n} p_i x_i$

  - Constraint : $\sum_{i=1}^{n} w_i x_i \leq M$

The Fractional Knapsack Problem

- Greedy choice:

  - **Keep taking item with highest value (profit to weight ratio)**

- Algorithm: Iteratively picks the item with the greatest value(pi/wi).

- If, at the end, the knapsack cannot fit the entire last item with greatest value among the remaining items, we will take a fraction of it to fill the knapsack.

The Fractional Knapsack Problem
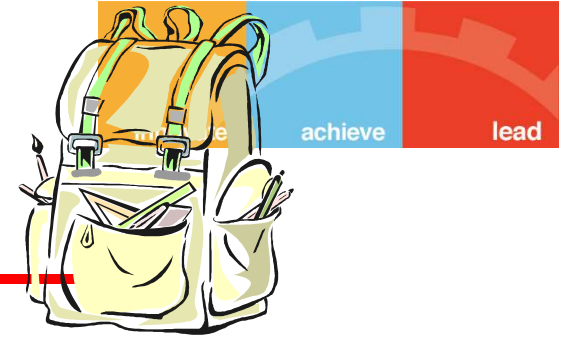
- Let n=3 M=40

Weight(wi)	Profit(Pi)
20	30
25	40
10	35

① Arrange the objects in decreasing order of $\left\{\dfrac{profit}{weight}\right.$

$\dfrac{P_1}{W_1} = \dfrac{30}{20} = 1.5$ $\dfrac{P_2}{W_2} = \dfrac{40}{25} = 1.6$ $\dfrac{P_3}{W_3} = \dfrac{35}{10} = 3.5$

$O_3 > O_2 > O_1$

The Fractional Knapsack Problem

- Let n=3 M=40

Weight(wi)	Profit(Pi)
20	30
25	40
10	35

obj (i)	w_i	P_i	$x = 1$ or $x = \frac{rc}{w_i}$	Profit = $x \times P_i$	$rc = rc - w_i x_i$
					40
3	10	35	1	$1 \times 35 = 35$	$40 - 10 = 30$
2	25	40	1	$1 \times 40 = 40$	$30 - 25 = 5$
1	20	30	$x = \frac{5}{20} = \frac{1}{4}$	$\frac{1}{4} \times 30 = 7.5$	$5 - \frac{20}{4} = 0$

$$82.5$$

The Fractional Knapsack Algorithm

Algorithm

fractionalKnapsack(m,n, w,x,p)

Input:

m-Capacity of Knapsack ,

n- no:of objects,

w-array of weights of all n objects,

p-array of profits of all n objects

Output:

x –array containing the solution

Arrange the objects in the decreasing order of pi/wi

$O(n\log n)$

```
for i ← 1 to n
    x[i]=0
end for

rc ← m

for i ← 1 to n
    if (w[i]>rc) break;
    x[i] ← 1
    rc←rc-W[i]
    //p←p+p[i]
end for
```

$O(n\log n)$

```
if(i<=n)
    {x[i]=rc/w[i]}
    //p=(p[i]*x[i])+p
```

$O(1)$

The Fractional Knapsack Algorithm

- Time complexity

- **O(nlogn)**

- Why?

- Specifically, we use a heap-based priority queue to store the items of S, where the key of each item is its value . With this data structure, each greedy choice, which removes the item with greatest value, takes O(log n) time. Process is repeated for n items. Hence O(nlogn)

Correctness

Greedy Choice Property: Let i be the index of the item with maximum pi/wi. Then there exists an optimal solution in which you take as much of item i as possible.

Given a set of n items $\{1, 2, ..., n\}$.

- Assume items sorted by per-pound values: $p_1 \geq p_2 \geq ... \geq p_n$.

Let the greedy solution be $G = \langle x_1, x_2, ..., x_k \rangle$

- x_i indicates fraction of item i taken (all $x_i = 1$, except possibly for $i = k$).

Consider any optimal solution $O = \langle y_1, y_2, ..., y_n \rangle$

- y_i indicates fraction of item i taken in O (for all i, $0 \leq y_i \leq 1$).
- Knapsack must be full in both G and O:
  $\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K$.

Consider the first item i where the two selections differ.

- By definition, solution G takes a greater amount of item i than solution O (because the greedy solution always takes as much as it can). Let $x = x_i - y_i$.

Correctness

- There must exist some item $j \neq i$ with $p_j/w_j < p_i/w_i$ that is in the knapsack.

- We can therefore take a piece of j, with x weight, out of the knapsack, and put a piece of i with x weight in.

- This increases the knapsack's value by

- $x[p_i/w_i] - x[p_j/w_j] = x[(p_i/w_i) - (p_j/w_j)] > 0$

- **Contradiction** to the original solution being optimal.

Applications

- Portfolio optimization:
- Cutting stock problems
- Huffman encoding for text compression : Compute high frequency characters with short code words.
- Web auction Optimization
- **K Means procedure in clustering**: It can be viewed as a greedy algorithm for partitioning the n examples into k clusters so as to minimize the sum of the squared distances to the cluster centers

Web Auction Optimization

- Suppose you are designing a new online auction website that is intended to process bids for multi-lot auctions.

- This website should be able to handle a single auction for 100 units of the same digital camera or 500 units of the same smartphone, where bids are of the form, "x units for $y," meaning that the bidder wants a quantity of x of the items being sold and is willing to pay $y for all x of them.

- The challenge for your website is that it must allow for a large number of bidders to place such multi-lot bids and it must decide which bidders to choose as the winners.

- Naturally, one is interested in designing the website so that it always chooses a set of winning bids that maximizes the total amount of money paid for the items being auctioned. **So how do you decide which bidders to choose as the winners?**

- A fractional knapsack problem can be formulated as a LPP(Linear programming problem) with a single constraint. Let us revisit the example

- Let n=3 M=40

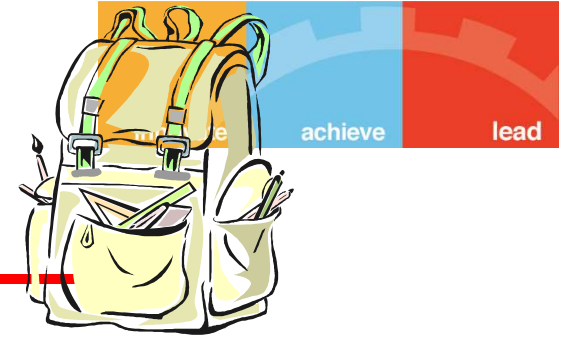| Sl No	W	Profit(Pi)
1	20	30
2	25	40
3	10	35

Problem

Let xi be the number of units of object i.

Maximise z=30x1+40x2+35x3

Constraint: 20x1+25x2+10x3<=40

0<=xi<=1

Exercise

"knapsack"

Items: 1 2 3 4 5

Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: $12 $32 $40 $30 $50

Value: 3 4 20 5 50
($ per ml)

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Task Scheduling Problem

- Job sequencing With deadlines
- Interval Scheduling

Job sequencing with deadlines

- Given a set of n jobs and associated with each job i is an integer deadline d_i >=0 and profit pi >0,it is required to find the set of jobs such that all the chosen jobs should be completed within their deadlines and the profit earned should be maximum with the following constraints.
- Only one machine is available for processing jobs
- Only one job must be processed at any point of time
- A job is said to be completed if it is processed on a machine for one unit time.
- **Greedy choice property**: Arrange the job in decreasing order of profits and pick the highest profit job first .
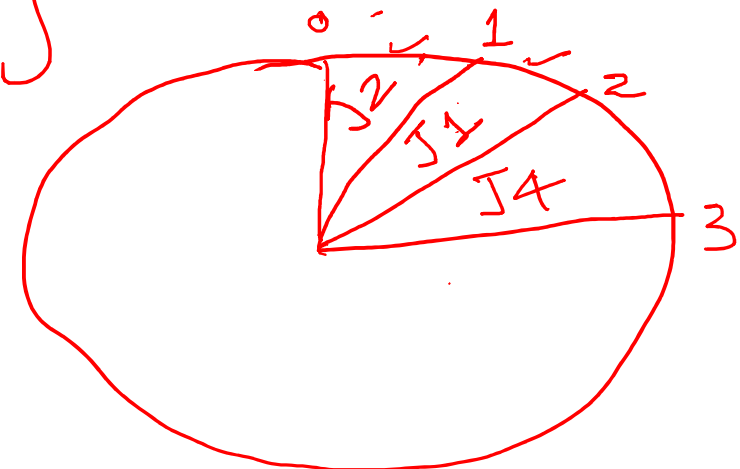
- Example

i	1	2	3	4	5
pi	20	10	15	1	5
di	2	1	2	3	3

Job sequencing with deadlines

- Example

i	1	2	3	4	5
pi	20	15	10	5	1
di	2	2	1	3	3

Algorithm JobSeq(job[1..n],p[1..n],d[1..n],list[1..n],n)

Arrange the objects in decreasing order of profits.

Initialise list with 0s

Profit=0

for i=1 to n do

    k=d[i]

    while (k>0)

        if(list[k]=0)

            list[k]=job[i]

            add p[i] to profit

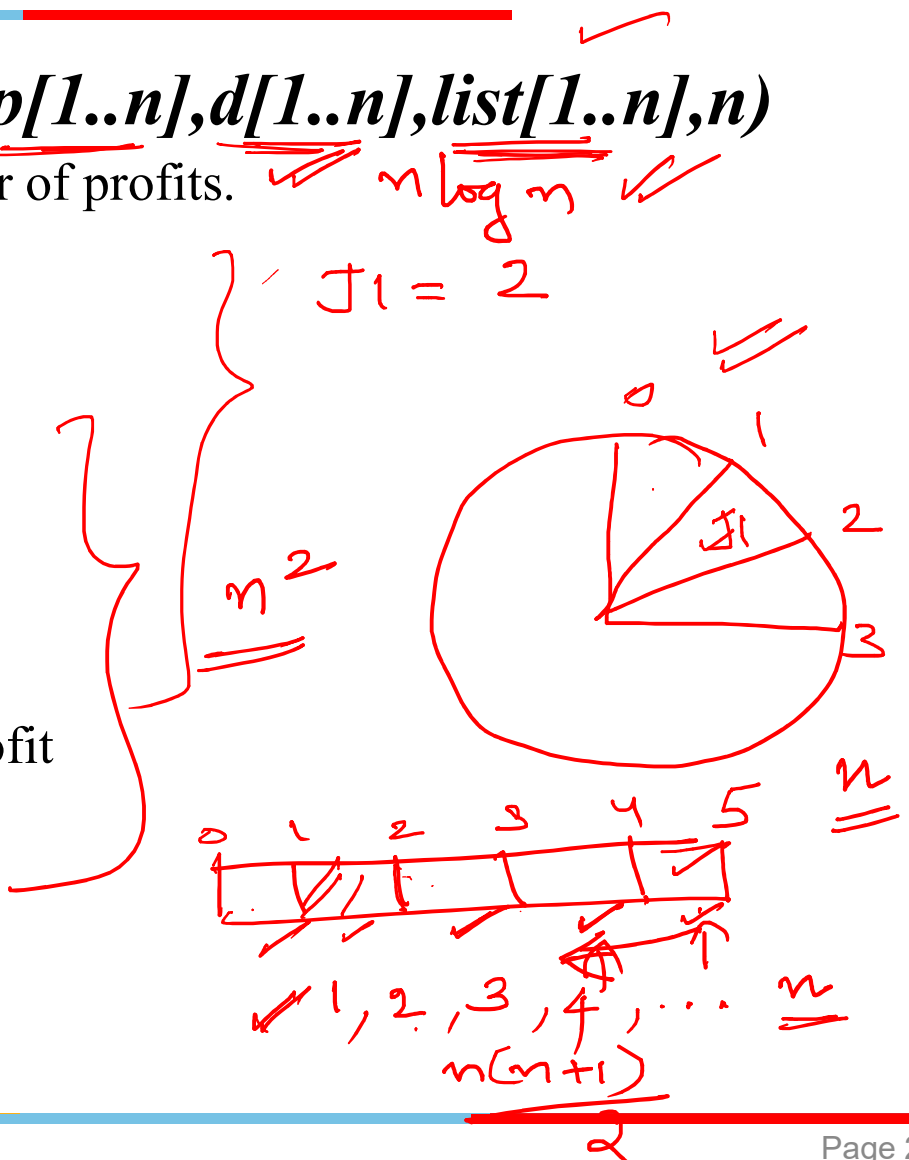   break;

        end if

        decrement k by 1

    end while

End for

Print list

Job sequencing with deadlines

- Time complexity
- $O(n^2)$
- Why?

Example

- Given a set of 9 jobs where each job has a deadline and profit associated to it .Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done.

Jobs	Profit	Deadline
J1	85	5
J2	25	4
J3	16	3
J4	40	3
J5	55	4
J6	19	5
J7	92	2
J8	80	3
J9	15	7

Task Scheduling Problem [Interval Partitioning]

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of "machines."

Task Scheduling Algorithm [Interval Partitioning]

- Suppose we are given a set T of n tasks, such that each task i has a start time, s_i , and a finish time, f_i (where $s_i < f_i$). Task i must start at time s_i and it is guaranteed to be finished by time f_i.

- Each task has to be performed on a machine and each machine can execute only one task at a time.

- Two tasks i and j are nonconflicting if $f_i <= s_j$ or $f_j <= s_i$.

- Two tasks can be scheduled to be executed on the same machine only if they are nonconflicting

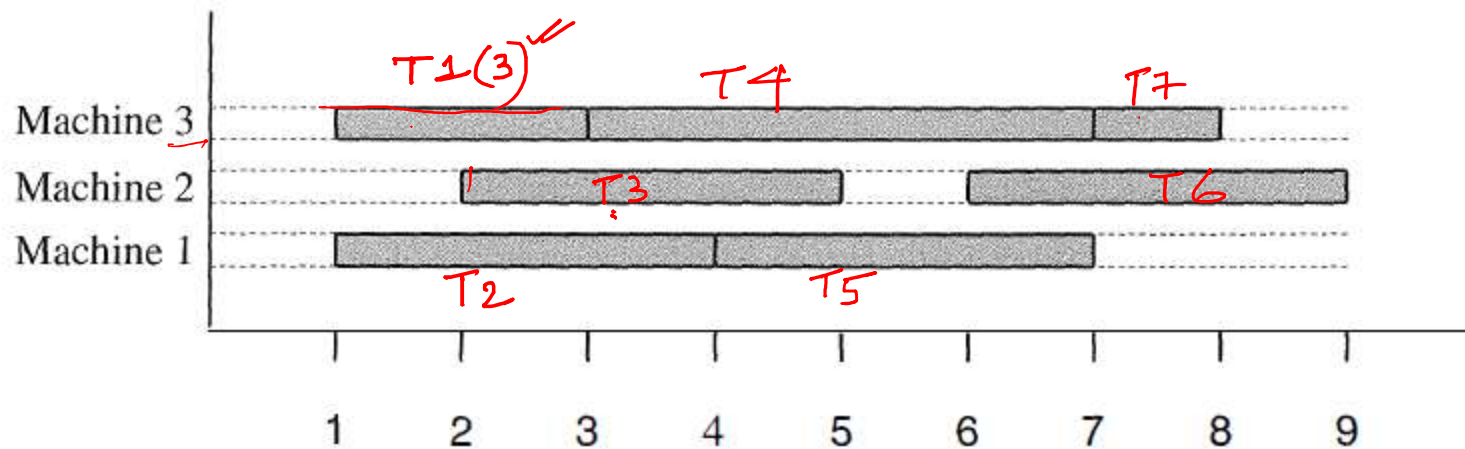# Task Scheduling Problem [Interval Partitioning]

- The task scheduling problem we consider here is to schedule all the tasks in T on the fewest machines possible in a nonconflicting way.

- Greedy choice.
  - [Earliest start time] Consider tasks in ascending order of sj.
  - [Earliest finish time] Consider tasks in ascending order of fj
  - [Shortest interval] Consider tasks in ascending order of fj - sj.

non-conflicting

An illustration of a solution to the task scheduling problem, for tasks whose collection of pairs of start times and finish times is { (1 , 3) , (1 , 4) , (2, 5) , (3, 7) , (4, 7) , (6, 9) , (7, 8) } .

- Consider tasks in increasing order of start time:

Algorithm TaskSchedule(T):

    Input: A set T of tasks, such that each task has a start time s_i and a finish time f_i

    Output: A nonconflicting schedule of the tasks in T using a minimum number of machines

    $m \leftarrow 0$ {optimal number of machines}

    while $T \neq \emptyset$ **do**

        remove from T the task i with smallest start time s_i → greedy choice

        if there is a machine j with no task conflicting with task i **then**

            schedule task i on machine j

        else

            $m \leftarrow m+1$ {add a new machine}

            schedule task i on machine m

Task Scheduling Problem [Interval Partitioning]-
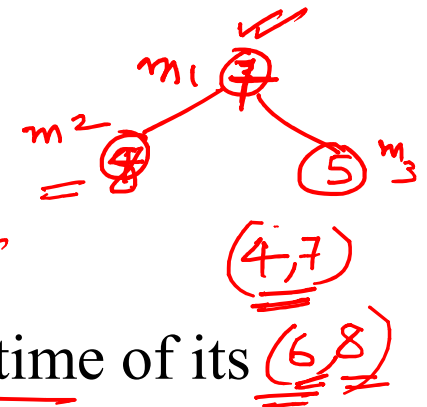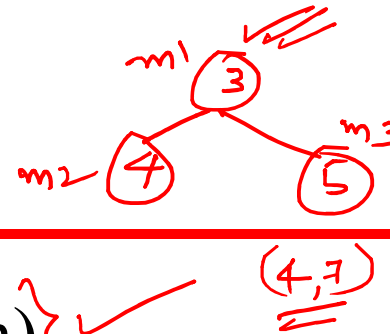
- Time complexity : O(n log n)
  - Sorting by start times takes O(n log n) time.
  - Store machines in a priority queue (key = finish time of its last job).
  - To allocate a new machine, INSERT machine onto priority queue.
  - To schedule Task j in machine k, INCREASE-KEY of classroom k to fj.
  - To determine whether Task j is compatible with any machine, compare sj to FIND-MIN
  - Total # of priority queue operations is O(n); each takes O(log n) time.
  - **This implementation chooses a machine k whose finish time of its last job is the earliest**

Task Scheduling Problem Algorithm-Correctness

So, suppose the algorithm does not work. That is, suppose the algorithm finds a nonconflicting schedule using k machines but there is a nonconflicting schedule that uses only $k - 1$ machines. Let k be the last machine allocated by our algorithm, and let i be the first task scheduled on k. By the structure of the algorithm, when we scheduled i, each of the machines 1 through $k - 1$ contained tasks that conflict with i. Since they conflict with i and because we consider tasks ordered by their start times, all the tasks currently conflicting with task i must have start times less than or equal to s_i, the start time of i, and have finish times after s_i. In other words, these tasks not only conflict with task i, they all conflict with each other. But this means we have k tasks in our set T that conflict with each other, which implies it is impossible for us to schedule all the tasks in T using only $k - 1$ machines. Therefore, k is the minimum number of machines needed to schedule all the tasks in T.

Sample problem1-Meeting Room Scheduling

- Schedule the meetings in as few conference rooms as possible.

- *Given a list of meeting times, checks if any of them overlaps. The follow-up question is to return the minimum number of rooms required to accommodate all the meetings.*

- Suppose we use interval (start, end) to denote the start and end time of the meeting, we have the following meeting times: (1, 4), (5, 6), (8, 9), (2, 6).

Sample problem1

- In the above example, apparently we should return true for the first question since (1, 4) and (2, 6) have overlaps.

- For the follow-up question, we need at least 2 meeting rooms so that (1, 4), (5, 6) will be put in one room and (2, 6), (8, 9) will be in the other.

Sample problem2

- A teacher assign homework at the beginning of the first day of class. Each problem carries 1 mark + bonus if submitted within specified number of days. The deadline for submitting and bonus marks are different for each problem. Each problem takes exactly one day to complete. Maximum bonus one can obtain is 15.Your task is to find a schedule to complete all homework problems so as to maximize bonus marks..

Problem number	1	2	3	4	5	6	7
Deadline for bonus	1	1	3	3	2	2	6
Bonus	6	7	2	1	4	5	1

- A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers may have different weights. Let w_i be the weight of the ith container, $1<=i<=n$. The cargo capacity of the ship is c. We wish to load the ship with the maximum number of containers.

Sample problem3

lead

Formulation of the problem :

- Variables : x_i $(1 \leq i \leq n)$ is set to 0 if the container i is not to be loaded and 1 in the other case.

- Constraints : $\sum_{i=1}^{n} w_i x_i \leq c.$

- Optimization function : $\sum_{i=1}^{n} x_i$

Sample problem3

- Using the greedy algorithm the ship may be loaded in stages; one container per stage.

- At each stage, the greedy criterion used to decide which container to load is the following :

  - *From the remaining containers, select the one with least weight.*

- This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers.

Sample problem3

- Suppose that:

  - $n = 8$,

  - $[w_1, \ldots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$,

  - and $c = 400$.

- When the greedy algorithm is used, the containers are considered for loading in the order 7,3,6,8,4,1,5,2.

- Containers 7,3,6,8,4 and 1 together weight 390 units and are loaded.

- The available capacity is now 10 units, which is inadequate for any of the remaining containers.

- In the greedy solution we have:

$$[x_1, \ldots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1] \text{ and } \sum x_i = 6.$$

Sample problem4

- Lecture j starts at sj and finishes at fj.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.