Greedy Algorithms

Let us solve a thief problem

A thief goes to a store….

- He has a bag of fixed size 20 kg
- Store has 5 items of different weights:

a1: 5kg, a2: 10 kg, a3: 4 kg, a4: 20kg, a5: 5kg

- Every items has different cost:

a1: Rs 100, a2: Rs 100 , a3: Rs 100 , a4: Rs 300, a5: Rs 200

- How should the thief fill his bag so that he/ she can get maximum value for it.

What is the meaning of Greedy here?

- Short-sightness
- Make a decision at a local level
- Solution of the problem is built in small steps choosing a decision at each step
- Decision at each step is taken myopically to optimize some underlying criteria
- It is applied to optimization problems
- We can design many greedy algorithms for the same problem, each one locally and optimizing some different measure on its way to a solution

Optimization problems??

- Optimization problems are those in which *we try to find/assign values to variables/configurations within given constraints* such that it either minimizes or maximizes the objective function defined on these variables.
- An Optimization problem can have many feasible solutions but has one optimal solution that evaluates the value of the *objective function to minimum or maximum* among all possible feasible solutions

Cont..

- The greedy approach does not always lead to an optimal solution
- But, for many problems it does give us an optimal solution
- Basically, certain Greedy algorithms have the property that a global optimal configuration can be reached by a series of locally optimal choices that are best from among the possibilities available at the that time.

Fractional KnapSack Problem

- Let there are ***n number of objects*** and each object is having a weight and contribution to profit.
- The knapsack of capacity M is given.
- The objective is to fill the knapsack in such a way that profit shall be maximum.
- We allow a fraction of item to be added to the knapsack

Mathematical formulation…

Maximize $\Sigma p_i x_i$ subjected to $\Sigma w_i x_i <= M$

Where p_i and w_i are the profit and weight of i^{th} object and x_i is the fraction of i^{th} object to be selected.

For example Given n = 3,

(p1, p2, p3) = {30, 24, 15}

(w1, w2, w3) = {10, 6, 3}

M = 12

(p1, p2, p3, p4) = {25, 24, 15, 30}

(w1, w2, w3) = {10, 12, 5, 5}

M = 20

Different algorithms…

Greedy strategy I:

- In this case, *the items are arranged by their profit values*.
- The item with the maximum profit is selected first.
- If the weight of the object is less than the remaining capacity of the knapsack then the object is selected full and the profit associated with the object is added to the total profit.
- Otherwise, a fraction of the object is selected so that the knapsack can be filled exactly.
- ***This process continues from selecting the highest profitable object to the lowest profitable object till the knapsack is exactly full.***

- **Greedy strategy II**:
- In this case, the items are arranged by weights.
- Here the item with minimum weight is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.
- **Greedy strategy III**: In this case, the items are arranged by profit/weight ratio and the item with maximum profit/weight ratio is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Cont..

- It is clear from the above strategies that the Greedy method III generates the optimal solution if we select the objects with respect to their profit to weight ratios that means the object with maximum profit to weight ratio will be selected first.

Coding for data Compression

- This is about encoding symbols using bits
- We know that computers ultimately operate on sequence of bits
- Each character is represented in 8 bits when characters are coded using standard codes such as ASCII.
- To compress, characters are coded fixed-length code word representation.
- In this fixed-length coding system the total code length is more.
- For example, suppose we have six characters (a, b, c, d, e, f) and their frequency of occurrence in a message is {45, 13, 12, 16, 9, 5}.
- In fixed-length coding system we can use three bits to represent each character. Then the total code length of the message is (45+13+12+16+9+5) x 3 = 100 x 3 = 300.

- The issue of reducing the average number of bits per letter is a fundamental problem in the area of data compression
- When large files needs to be sent across communication network or sometimes on disk, it is required to represent them as compactly as possible.

Prefix Code

- If we do variable size encoding then there arises the problem of ambiguity because there may exist pairs of letters where the bit string that encodes one letter is a prefix of the bit string that encodes another
- ***Hence we need to map letters to bit strings such that no encoding is prefix of another***
- ***Hence, we define that a prefix code for a set S of letters is a function Y that maps each letter x of S to some sequence of 0's and 1's in such a way that for distinct x, y of S, the sequence Y(x) is not a prefix of the sequence Y(y)***

Cont..

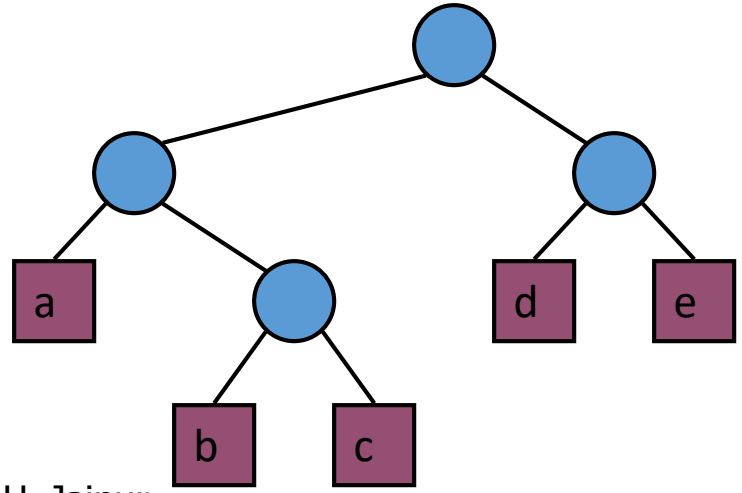
- Let us encode the characters with variable-length coding system.
- **In this coding system, the character with higher frequency of occurrence is assigned fewer bits for representation while the characters having lower frequency of occurrence in assigned more bits for representatio**n.
- The variable length code for the characters can be : a: 1,b and
- The total code length in variable length coding system is $1 \times 45 + 3 \times 12 + 3 \times 16 \times 4 \times 9 + 4 \times 5 = 224$.
- Hence fixed length code requires 300 bits while variable code requires only 224 bits

- Lecture 2

Encoding Tree Example

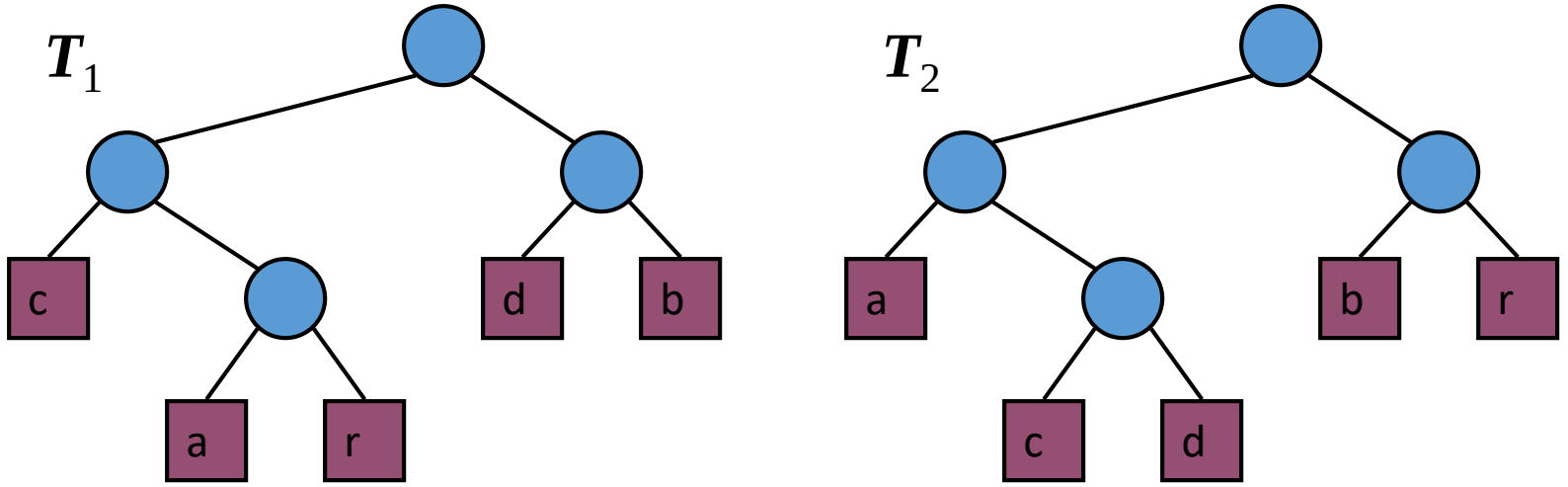
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X ***(not using Huffman coding)***
- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits

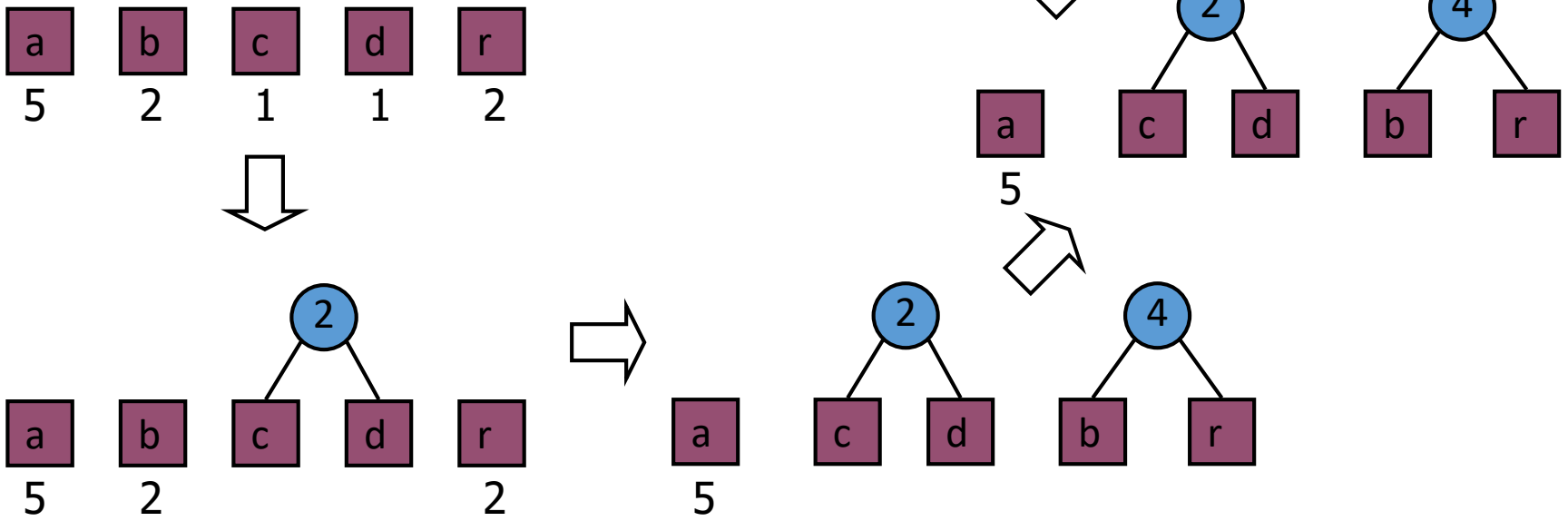


Example(Huffman)

X = abracadabra

Frequencies

a	b	c	d	r
5	2	1	1	2



Huffman Coding

There are two major steps in Huffman Coding-

- Building a Huffman Tree from the input characters.
- Assigning code to the characters by traversing the Huffman Tree.

How to build the tree:

- Create a leaf node for each character of the text.
- Leaf node of a character contains the occurring frequency of that character.
- Arrange all the nodes in increasing order of their frequency value.

Cont..

Taking two nodes having minimum frequency:

- Create a new internal node.
- *The frequency of this new node is the sum of frequency of those two nodes.*
- Make the first node as a left child and the other node as a right child of the newly created node.
- Keep repeating above steps until all the nodes form a single tree.
- The tree finally obtained is the desired Huffman Tree.

Time complexity analysis of Huffman Coding

- We are finding out two nodes having minimum elements frequency
- So, ExtractMin( ) is called 2 x (d-1) times if there are n nodes. As extractMin( ) calls minHeapify( ), it takes O(logd) time.
- Thus, Overall time complexity of Huffman Coding becomes **O(dlogd)**.
- d is the number of unique characters in the given text.
- If n is the size of the string then complexity of Huffman coding will be

n + dlogd

Huffman's Algorithm

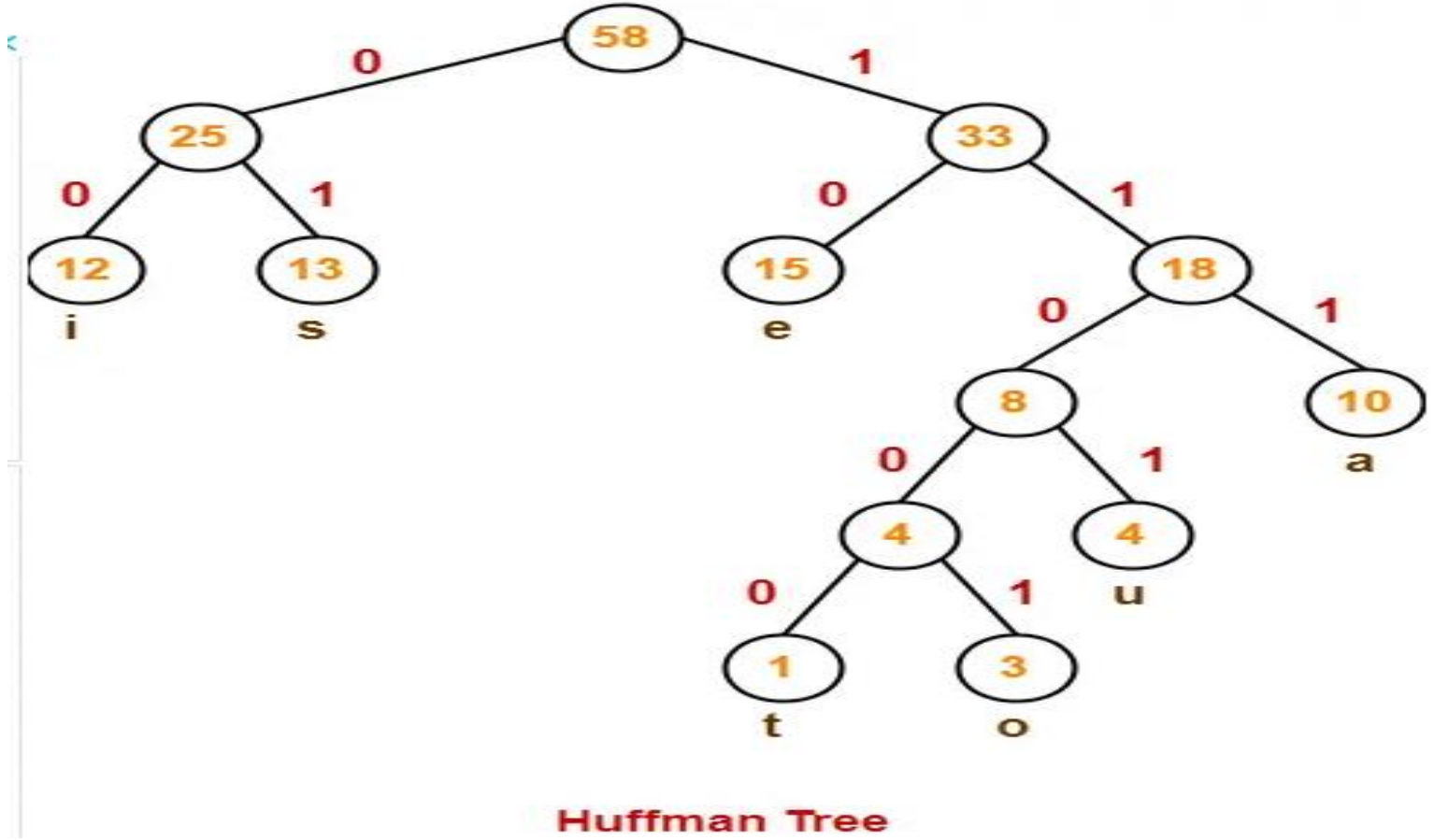
- Given a string X, Huffman's algorithm construct a prefix code that minimizes the size of the encoding of X
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Huffman's Algorithm

Algorithm Huffman(X):
Input: String X of length n with d distinct characters
Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
  Create a single-node binary tree T storing c.
  Insert T into Q with key $f(c)$.
while len(Q) > 1 **do**
  $(f_1, T_1) = Q$.remove_min()
  $(f_2, T_2) = Q$.remove_min()
  Create a new binary tree T with left subtree T_1 and right subtree T_2.
  Insert T into Q with key $f_1 + f_2$.
$(f, T) = Q$.remove_min()
return tree T

Example…

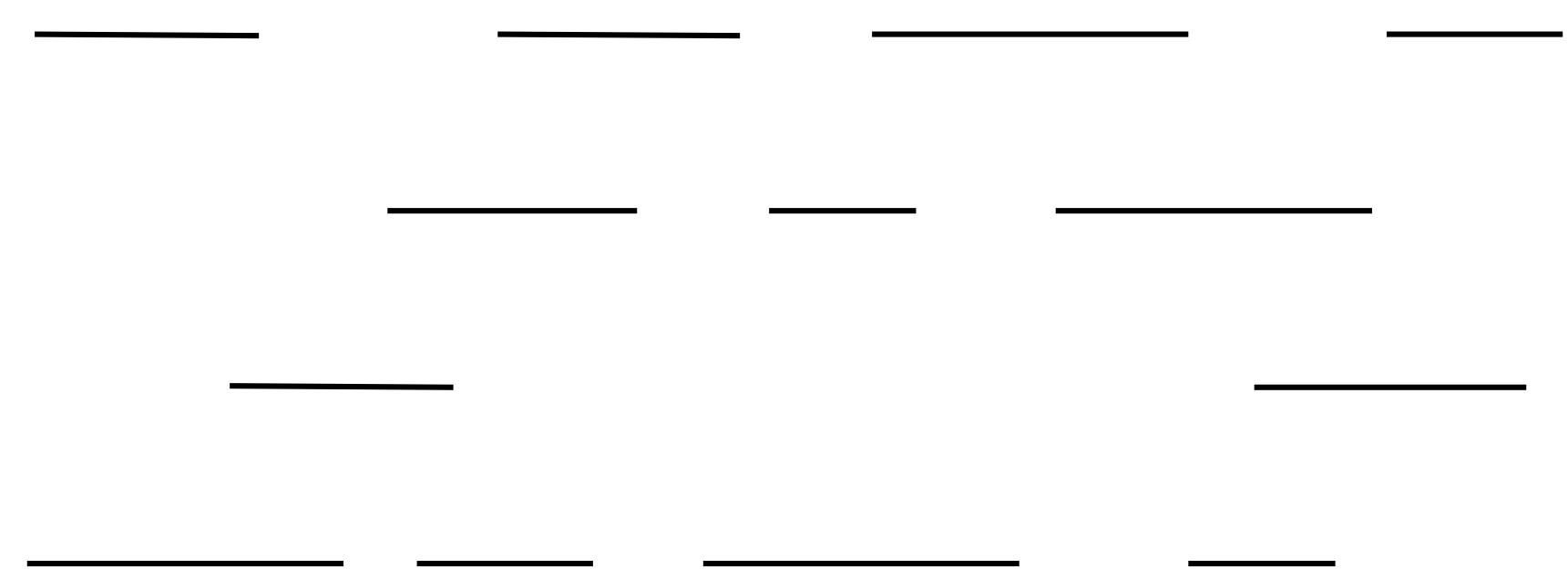


Huffman Tree

Interval Scheduling

- Suppose we have a resource: ***a lecture room***, **a super computer** or something else and many people want to use the resource for some periods of time.
- A request will be of this form: **May I reserve the resource S starting at time s and ending at f**
- A **scheduler** wants taccepted requests do not overlap in timeo accept a subset of these requests, rejecting others, so that the and goal is to maximize the number of requests accepted
- So we can say that we have n requests/ intervals specifying (si and fi) and we want to choose maximum numbers of compatible requests/non overlapping intervals

An instance of the Interval Scheduling problem (there is only single compatible set of size 4)

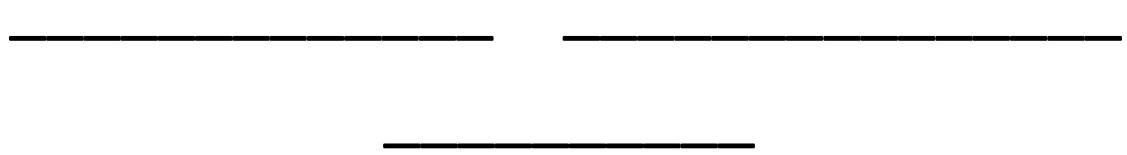


Designing a greedy algorithm…

- **First Idea**: Use a simple rule to select a first request i_1. Once i_1 is accepted, all other requests are rejected which are not compatible with i_1
- We continue like this until we run out of requests.
- This is the most obvious and simple solution because we would like to start with minimal start time s_i so that resource does not get wasted.
- But this method does not give optimal solution
- Our objective is to fulfil maximum requests and if first request is very long or finish time fi is maximum then no other request will be accepted. Example: **Not a solution**
- _____
- _____ _____ __ _____

Cont..

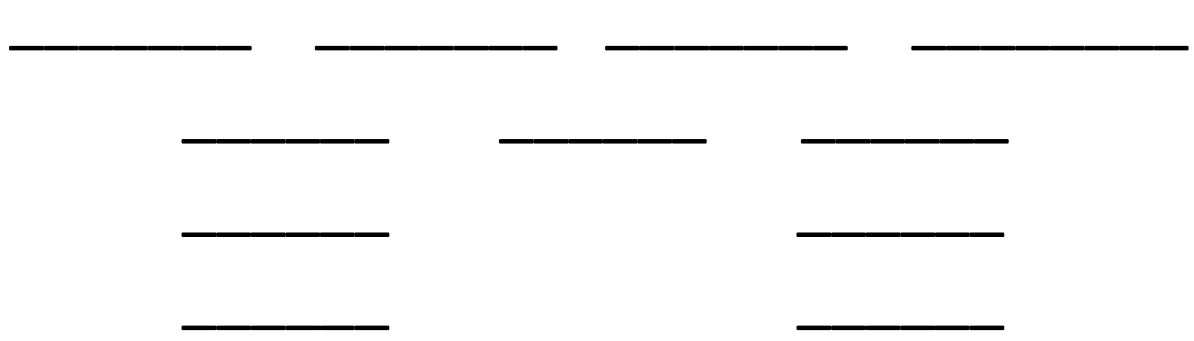
- This suggests that we should start with the request whose interval (fi – si) is as small as possible
- This is somewhat better but again it does not give optimal solution
- Example: (**Not a solution**)



- Notice: Here second request competes with 1st and 3rd both or it overlaps with both so this approach will choose only one request.

Cont..

- So now we design a greedy algorithm based on this concept that for each request we count the number of requests which are not compatible and choose the one which has fewest number of non-compatible requests
- Or choose the interval with fewest conflicts
- This would lead to an optimal solution in previous example. Actually it is difficult to design a bad example for this rule. Let us examine this: (it will fail here) (**Not a solution**)



Cont..

- **A greedy algorithm that leads to optimal solution is based on fourth idea**: We should accept the request that finishes first. This means choose the request whose fi is as small as possible

Algorithm:

- Let R is the set of requests (A is a empty set)
- While R is not empty
- choose a request i from R that has smallest finishing time
- Add request i to A
- Delete all requests from R that are not compatible (which can not be accepted if I is accepted) with request i
- End of while
- Return A

Let us examine few examples…

- _____(6) _____(8)
- _____(1) _____(3) _____(5) _____(9)
- _____(2) ____(4) _____(7)

- Number are assigned according to finish time
- The solution according to last algorithm would be:

(1,3,5,8)

Analysis of earliest-finish-time-first algorithm

- Theorem. The earliest-finish-time-first algorithm is optimal.
- Pf. [by contradiction]
- Assume greedy is not optimal, and let’s see what happens.
- Let i1, i2, ... ik denote set of jobs selected by greedy in set A
- Let j1, j2, ... jm denote set of jobs in an optimal solution in set O
- Our goal is to prove k =m
- Assume that the requests in O are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points.
- We know that the requests in O are compatible, which implies that the start points have the same order as the finish points.

Proof cont..

- Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request.
- So, our greedy rule guarantees that $f(i1) \leq f(j1)$. This is way in which we want to show that our greedy rule “stays ahead”—that each of its intervals finishes at least as soon as the corresponding interval in the set O.
- Thus we now prove that for each $r \geq 1$, the rth accepted request in the algorithm’s schedule finishes no later than the rth request in the optimal schedule

- This situation is not possible
- _____i_{r-1} _____i_r
- _____j_{r-1} _____j_r

Lemma: For all indices r<= k we have f(i_r) <= f(j_r)

Proof: We will prove this by induction.

- For r = 1 the statement is clearly true: the algorithm starts by selecting the request i1 with minimum finish time.
- Now let r > 1. We will assume that the statement is true for r − 1, and we will try to prove it for r.

- In Figure, we assume that $f(i_{r-1}) \leq f(j_{r-1})$.
- The rth interval not to finish earlier, would need to “fall behind” as shown.
- But this could not happen because the greedy algorithm always chooses earliest finish so it would have chosen j_r
- So if j_r was finishing earlier than i_r than greedy algo would have chosen j_r.
- Hence, for each r, the rth interval it selects finishes at least as soon as the rth interval in O.

Cont..

The greedy algorithm returns an optimal set A

- We will prove the statement by contradiction.
- If A is not optimal, then an optimal set O must have more requests, that is, we must have m > k.
- Applying lemma with r = k, we get that $f(i_k) \leq f(j_k)$. Since m > k, there is a request j_{k+1} in O. This request starts after request j_k ends, and hence after i_k ends.
- So after deleting all requests that are not compatible with requests i1,..., i_k the set of possible requests R still contains j_{k+1}.
- But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—so we get a contradiction, hence proved.

Extensions

- The problem which have considered is very simple scheduling problem
- There are many complications which could arise in real life
- In this problem, we assumed that all requests were known to the scheduling algo while choosing the subset for accepting requests
- But we need to think about the version where scheduler needs to make decision about accepting or rejection without knowing the a full set of requests
- Also, customers may be impatient and can leave without waiting

Another version…

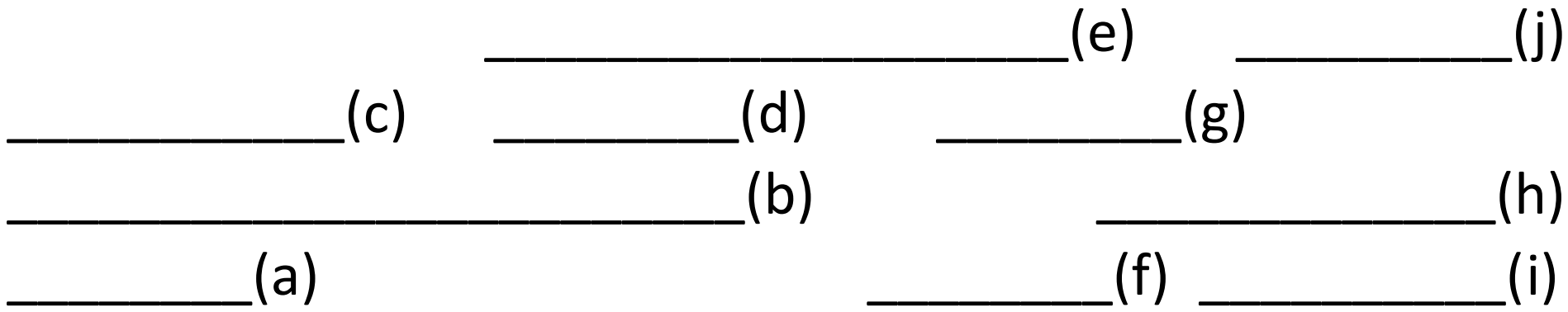
- In the simple version, our goal was to maximize the number of satisfied requests
- But we could picture a situation in which each request i has a different value vi to us
- Goal would be to maximize our income
- This leads to weighted interval scheduling problem

Third version....Scheduling all intervals

- Till now we had only one resource and many requests
- A related problem arises when we have multiple identical resources and we wish to schedule all of them using as few resources as possible.
- This is called **Interval Partitioning problem**
- **Examples:**

1. Scheduling lectures using minimum class rooms
2. Processing of jobs for a specific period of time on machines

Example..



(c, d, f, j)

(b, g, i)

(a, e, h)

How many resources:?

Depth of a set of intervals..

- The maximum number that pass over a single point on the time line

- Claim:
- In any instance of interval partitioning, the number of resources needed is at least the depth of the set of intervals
- Can we design an efficient algorithm that schedules all intervals using minimum no. of resources?
- Is there always a schedule using a no. of resources equal to depth?