Premaster Course Algorithms 1

Chapter 4: Dyanamic Programming and Greedy Algorithms

Christian Scheideler

SS 2018

Overview

- Dynamic Programming
- Greedy Algorithms

Dynamic Programming

- Typically used for optimization problems
- Similar to Divide&Conquer (cut problem into simpler problems that can be solved recursively)

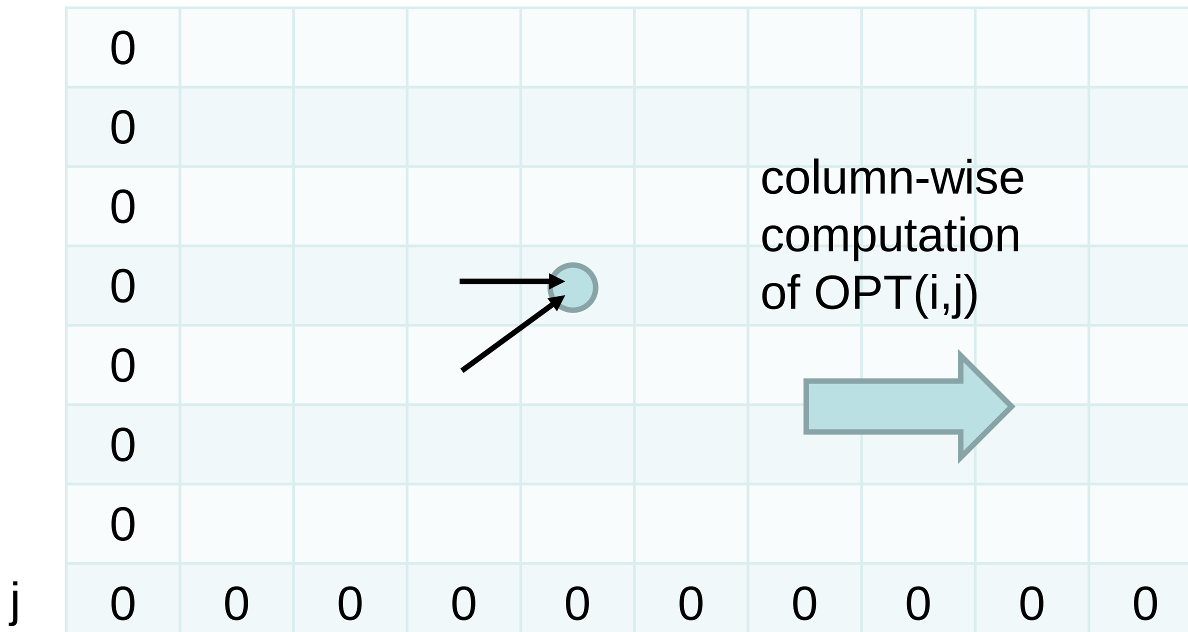
General approach:

1. Set up a recursive equation for the problem
   - initial case (i.e., OPT(i,0)=OPT(0,j)=0)
   - recursion (i.e., OPT(i,j)=…)
   - optimal value (i.e., OPT = OPT(n,W))
2. Formulate a dynamic program
   Important: compute OPT(i,j,..) values in an order so that all values in the recursion have already been computed!

Dynamic Programming

Example:

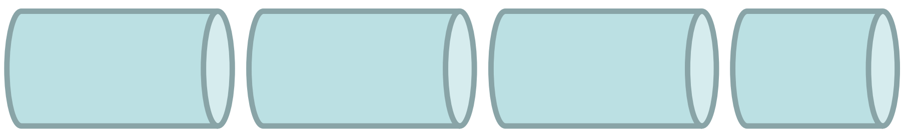
$$\mathrm{OPT}(i,j) = \begin{cases} 0 & i=0 \text{ or } j=0 \\ \mathrm{OPT}(i-1,j-1)+\mathrm{OPT}(i,j-1)+c & \text{otherwise} \end{cases}$$



i

Rod-cutting Problem

Rod-cutting problem: Given a rod of length n inches and a table of prices p_i for $i=1,2,\ldots,n$, for rods of length i, determine the maximum revenue OPT obtainable by cutting up the rod and selling the pieces.



Note that if the price for a rod of length n is large enough, an optimal solution may require no cutting at all.

Rod-cutting Problem

OPT(n): optimal revenue for rod of length n

- Initial case: $OPT(0)=0$
- Recursion: $OPT(n)=\max_{1 \leq i \leq n} (p_i + OPT(n-i))$
- Optimal value: $OPT=OPT(n)$

Table of values:

OPT(0)	OPT(1)	…	OPT(n)

Dependencies:

So compute from left to right.

Rod-cutting Problem

OPT(n): optimal revenue for rod of length n

- Initial case: OPT(0)=0
- Recursion: $OPT(n)=\max_{1\leq i\leq n} (p_i + OPT(n-i))$
- Optimal value: OPT=OPT(n)

Dynamic program:

1. OPT(0):=0
2. for j:=1 to n do
3.   OPT(j):=-∞
4.   for i:=1 to n do
5.   OPT(j):=max(OPT(j),p[i]+OPT(j-i))
6. return OPT(n)

Matrix-chain Multiplication

Matrix-chain multiplication problem: Given matrices $A_1,\ldots,A_n$ and $p_0,\ldots,p_n \in \mathbb{N}$, where A_i is a $p_{i-1} \times p_i$-matrix, compute the minimal number of multiplications to compute $A_1 \cdot \ldots \cdot A_n$.

Observation:

- Let $A_{i..j} = A_i \cdot \ldots \cdot A_j$
- The matrix product $A_{i..k} \cdot A_{k+1..j}$ takes $p_{i-1} \cdot p_k \cdot p_j$ many multiplications (using the standard matrix multiplication method)

Matrix-chain Multiplication

- $m[i,j]$: minimal number of multiplications to compute $A_{i..j}$

Initial case:

$m[i,i]=0$ for all $i \in \{1,\ldots,n\}$.

Recursion:

$m[i,j] = \min_{i \le k < j} m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$ für alle $i<j$

Optimal Value:

$m[1,n]$

Dynamic program:

- First, execute initial case
- Compute $m[i,j]$ for all $i<j$ with $|j-i|=d$, starting with $d=1$

Longest Common Subsequence

Definition: Let $X=(x_1,\ldots,x_m)$ and $Y=(y_1,\ldots,y_n)$ be two sequences, where $x_i, y_j \in A$ for some finite alphabet A. Then we call Y a subsequence of X if there are indices $i_1<\ldots<i_n$ with $x_{i_j} = y_j$ for all $j = 1,\ldots,n$.

Example:

Sequence Y	B	C	A	C			
Sequence X	A	B	A	C	A	B	C

Longest Common Subsequence

Longest common subsequence problem: Given two sequences X and Y, find the longest common subsequence of X and Y.

$c[i,j]$: length of longest common subsequence of $(x_1,\dots,x_i)$ and $(y_1,\dots,y_j)$

- Initial case: $c[i,j]=0$ if $i=0$ or $j=0$
- Recursion:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j \end{cases}$$

- Optimal value: $c[m,n]$

Longest Common Subsequence

```
LCS-Length(X, Y)
    m:=X.length
    n:=Y.length
    new array C[0,..,m][0,..,n]
    for i:=0 to m do C[i][0]:=0
    for j:=0 to n do C[0][j]:=0
    for i:=1 to m do
        for j:=1 to n do
            if X[i]=Y[j] then
                C[i,j]:=C[i-1,j-1]+1
            else
                C[i,j]:=max(C[i,j-1],C[i-1,j])
    return C
```

Optimal Binary Search Tree

Optimal binary search tree problem: Given keys $k_1 < \ldots < k_n$ with access probabilities $p_1, \ldots, p_n \in [0,1]$ so that $\Sigma_{i=1}\ p_i = 1$, find a binary search tree T with minimal expected search time, i.e., $\Sigma_{i=1}\ p_i \cdot (\mathrm{depth}_T(k_i)+1)$ is minimal ($\mathrm{depth}_T(k)$: depth of node with key k in T, depth of the root of T is 0).

Optimal Binary Search Tree

$m[i,j]$: minimal expected search time for binary tree containing k_i to k_j

Initial case:

- $m[i,i-1]=0$ for all $i \in \{1,\dots,n\}$
- $m[i,i]=p_i$ for all $i \in \{1,\dots,n\}$

Recursion:

$m[i,j] = \min_{i \le k \le j} m[i,k-1] + m[k+1,j] + \Sigma_{l=i} \, p_l$ for all $i<j$

Optimal value:

$m[1,n]$

Dynamic program:

- First compute initial case
- Compute $m[i,j]$ for all $i<j$ with $|j-i|=d$, starting with $d=1$
- At the end, output $m[1,n]$

Overview

- Dynamic Programming
- Greedy Algorithms

Greedy Algorithms

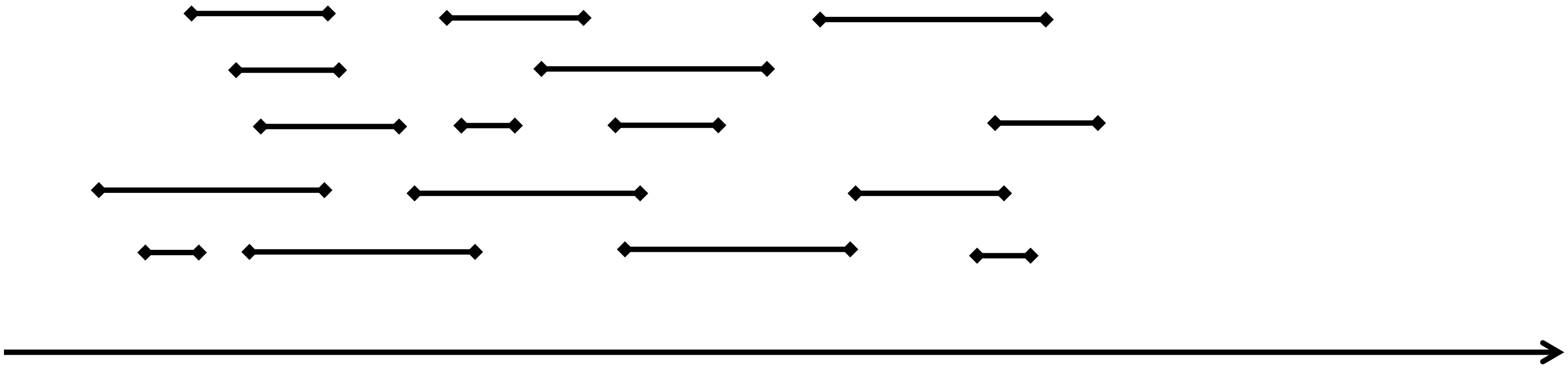
General approach:

- Arrange the input into a sequence of small pieces that are considered one after the other
- For each piece, make an irreversible decision in a Greedy fashion (based on the given objective function) without taking the remaining pieces into account

Often, Greedy algorithms produce non-optimal solutions, so one has to be careful about when to use Greedy approaches!

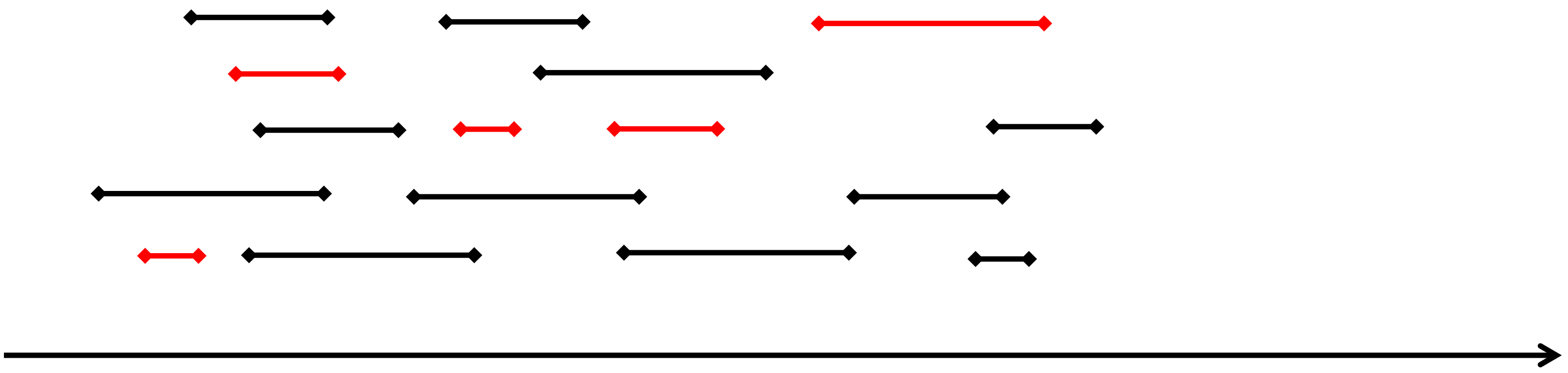
Interval Scheduling

Interval scheduling problem: given a resource (room, computer,…) and a set of requests to use that resource for a certain time interval, schedule as many requests as possible.



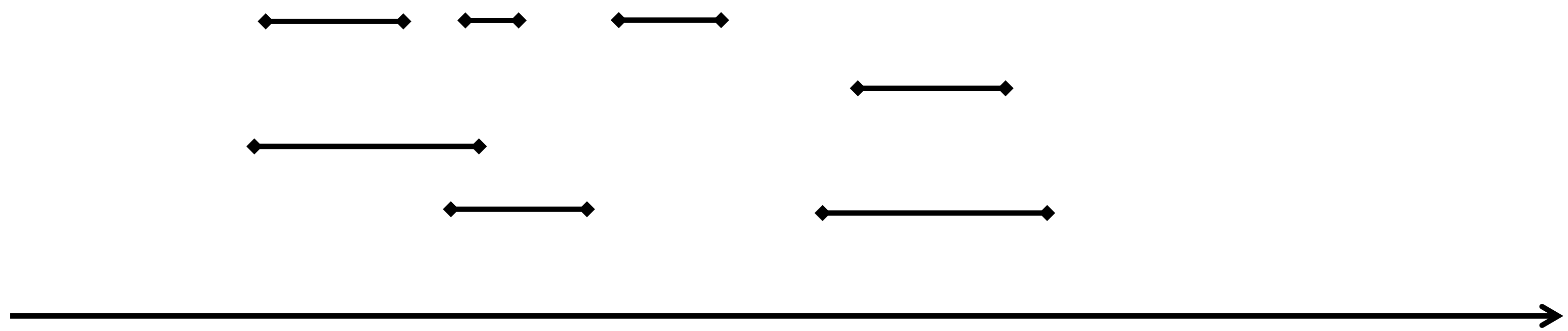
Interval Scheduling

Interval scheduling problem: given a resource (room, computer,…) and a set of requests to use that resource for a certain time interval, schedule as many requests as possible.



Interval Scheduling

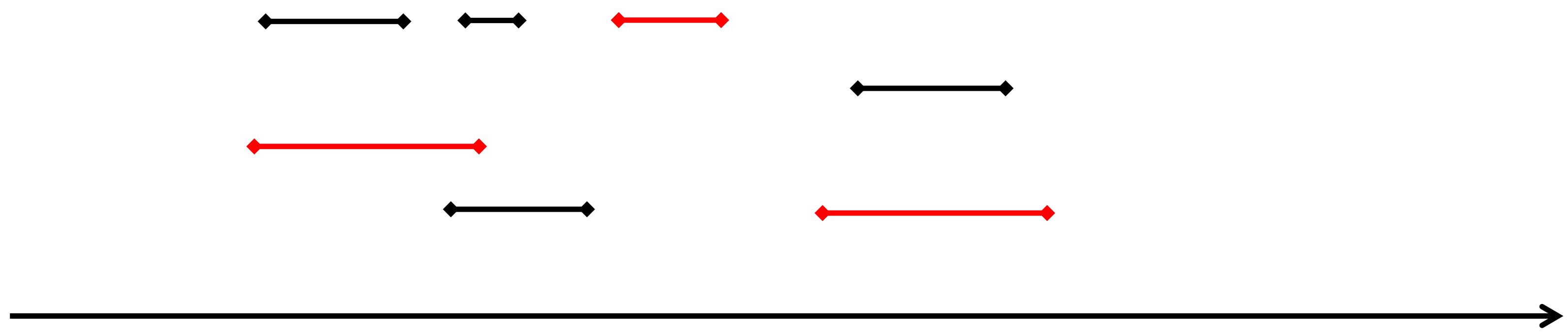
Strategy 1: consider requests in order in which they start



Interval Scheduling

Strategy 1: consider requests in order in which they start

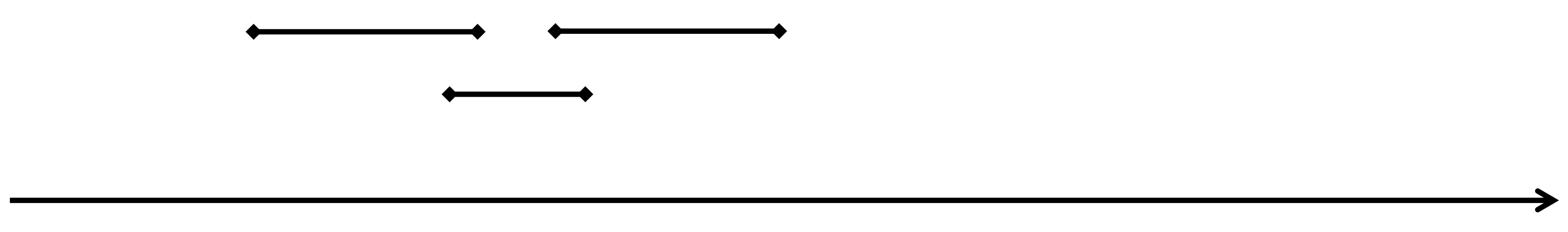
Not optimal!



Interval Scheduling

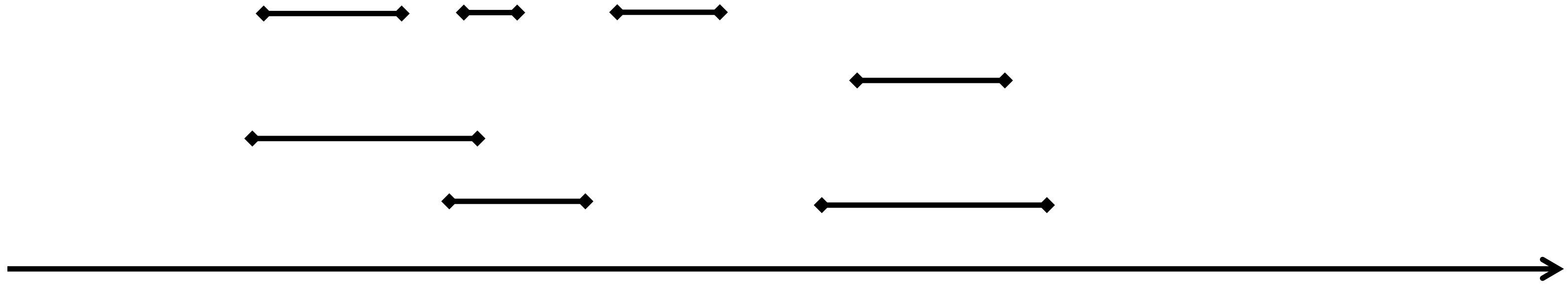
Strategy 2: consider requests in increasing order of length

Not optimal!



Interval Scheduling

Strategy 3: consider requests in order in which they finish



Always produces optimal result!

Huffman Trees

Minimum code length problem: Given a text, we want to find a code for its letters that minimizes the code length of the text.

Problem: Code is not allowed to be ambiguous, i.e., there should be a unique way of recovering the text from the code.

Unique coding strategy: prefix coding

Definition: A prefix code for an alphabet Σ is a function γ that assigns to each letter $x \in \Sigma$ a bit string so that for any $x,y \in \Sigma$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

Example:

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	0100	0110	0111	1001	1010	1011	1101	1110	1111

Huffman Trees

Definition: The frequency $f[x]$ of a letter $x \in \Sigma$ is the fraction of letters in the given text that is equal to x.

Example:

- $\Sigma = \{0,1,2\}$
- text =„0010022001“ (10 letters)
- $f[0] = 3/5$
- $f[1] = 1/5$
- $f[2] = 1/5$

Huffman Trees

Definition: The code length of a text T of n letters w.r.t. code γ is defined as

$$BL(T) = \Sigma_{x \in \Sigma}\, n \cdot f[x] \cdot |\gamma(x)|$$

Example:

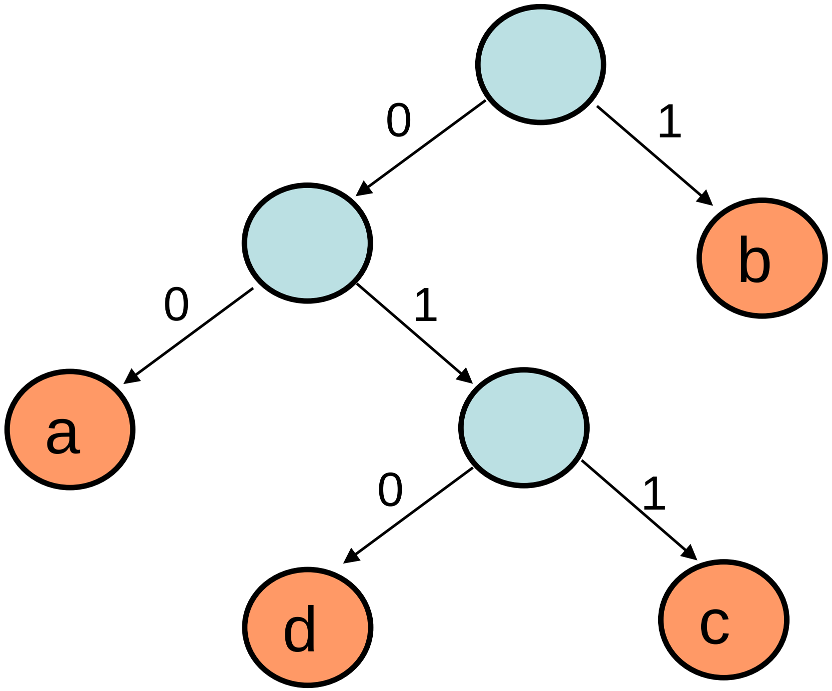
- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- T = „aacdaabb“
- code length = 16

Optimal prefix code problem: Given an alphabet Σ and a frequency function $f:\Sigma \to [0,1]$, find a prefix code that minimizes

$$ABL(\gamma) = \Sigma_{x \in \Sigma}\, f[x] \cdot |\gamma(x)|$$

Huffman Trees

Binary trees and prefix codes:

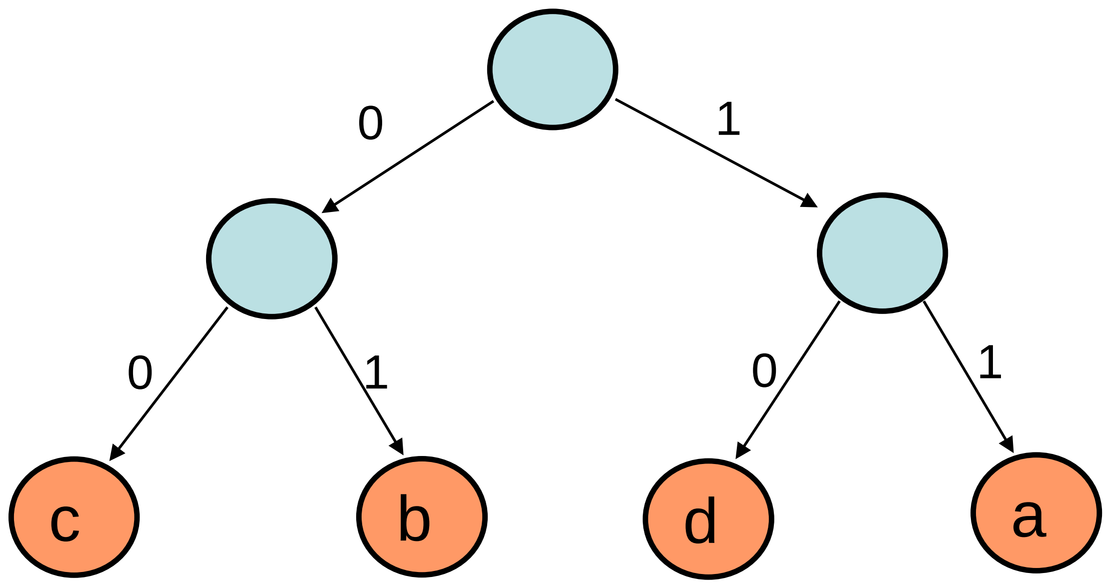


$x \in \Sigma$	$\gamma(x)$
a	00
b	1
c	011
d	010

Huffman Trees

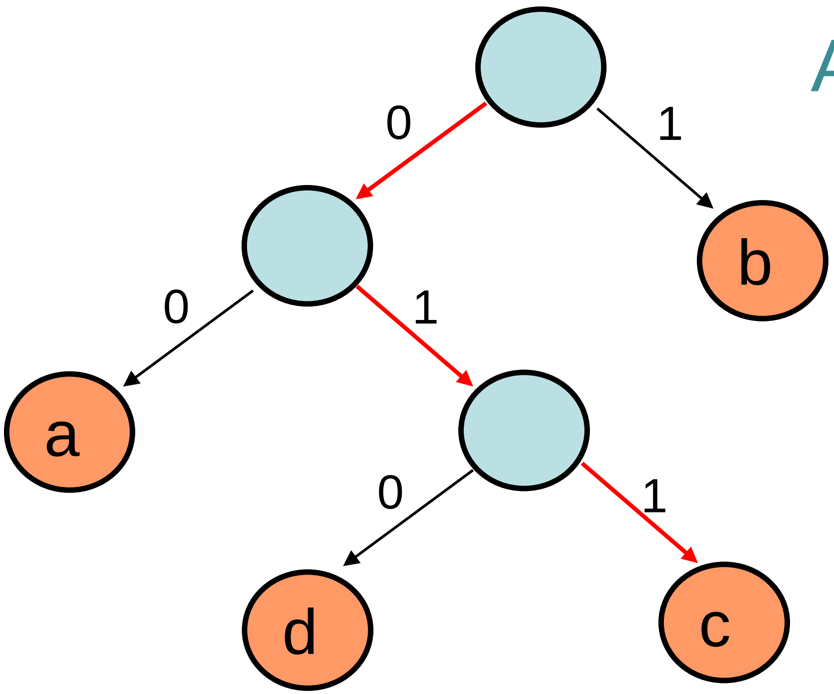
Binary trees and prefix codes:

$x \in \Sigma$	$\gamma(x)$
a	11
b	01
c	00
d	10



Huffman Trees

Definition: The depth of a tree node is the length of its path from the root.



$ABL(T) = \Sigma_{x \in \Sigma} f[x] \cdot depth_T(x)$

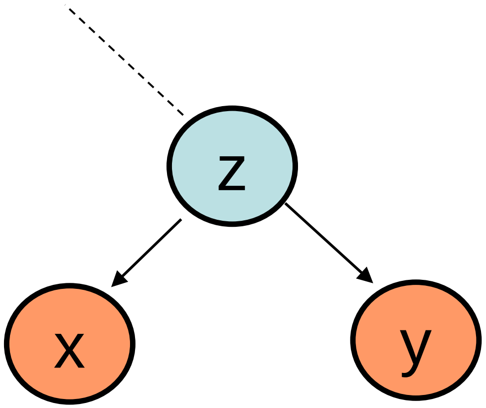
depth(c) = 3

Huffman Trees

Idea of Huffman‘s algorithm:

Repeatedly do the following until only one letter is left in Σ:

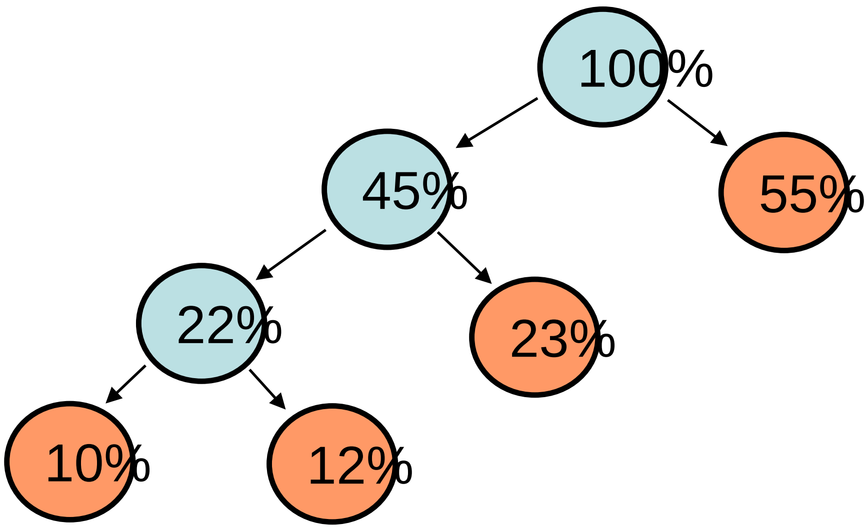
- Pick the two letters $x,y \in \Sigma$ of lowest frequencies and connect them to a tree with root z. Remove x,y from Σ and add instead z to Σ with frequency $f[z]=f[x]+f[y]$.



Huffman Trees

Example:

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%



Greedy Algorithms

There is a general approach:

Whenever an optimization problem can be modeled as a matroid, it can be solved by a Greedy algorithm.

Next Lecture

Topic: Basic graph algorithms