DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Analysis and Synthesis of Algorithms
Design of Algorithms

Greedy Algorithms
Basic Features and Examples

Outline

- Greedy Algorithms:
  - First Example: Selection of Activities
- Features of Greedy Algorithms
- Examples:
  - Knapsack Problem
  - Minimization of System Tasks
  - Huffman Codes
- Other Examples:
  - Minimum-Cost Spanning Trees: Kruskal, Prim,
  - Single-Source Shortest Paths: Dijkstra.

Algorithm Synthesis Techniques

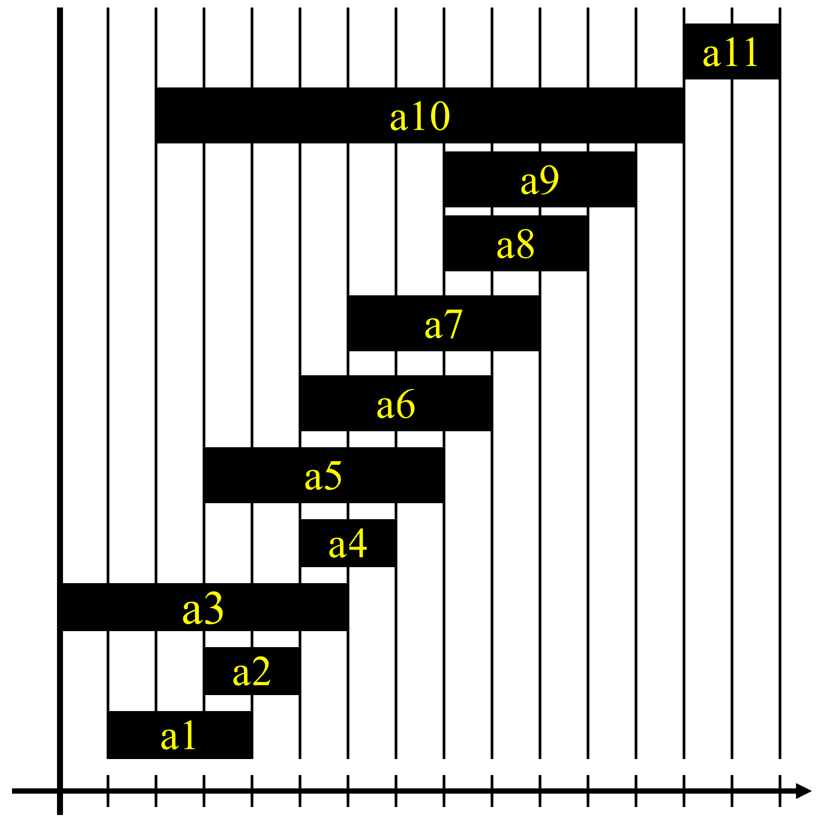
- Divide-and-Conquer
  - Split into independent sub-problems
- Dynamic Programming
  - Combination of dependent sub-problems
  - Use of table to avoid recomputation of sub-problems' solutions
- **Greedy Algorithms**
  - **Strategy:** At each Step of the Algorithm, Select the option that is locally the best to find the overall Optimal Solution
  - In many cases this strategy works
  - Examples:
    - Minimum-Cost Spanning Trees: Kruskal, Prim,
    - Single-Source Shortest Path: Dijkstra.

Example: Selection of Activities

- Let S = {1, 2, …, n} be a set of activities that share a common resource
  - Resource can only be used by one activity at a time
  - Activity `i` is characterized by:
    - start time: s_i
    - finish time: f_i
    - activity execution interval: $[s_i, f_i[$
  - Activities i and j are **compatible** if $[s_i, f_i[$ and $[s_j, f_j[$ are disjoint
- **Objective:** Find a/the Maximal set of Activities that are Mutually Compatible

Example: Activity Selection

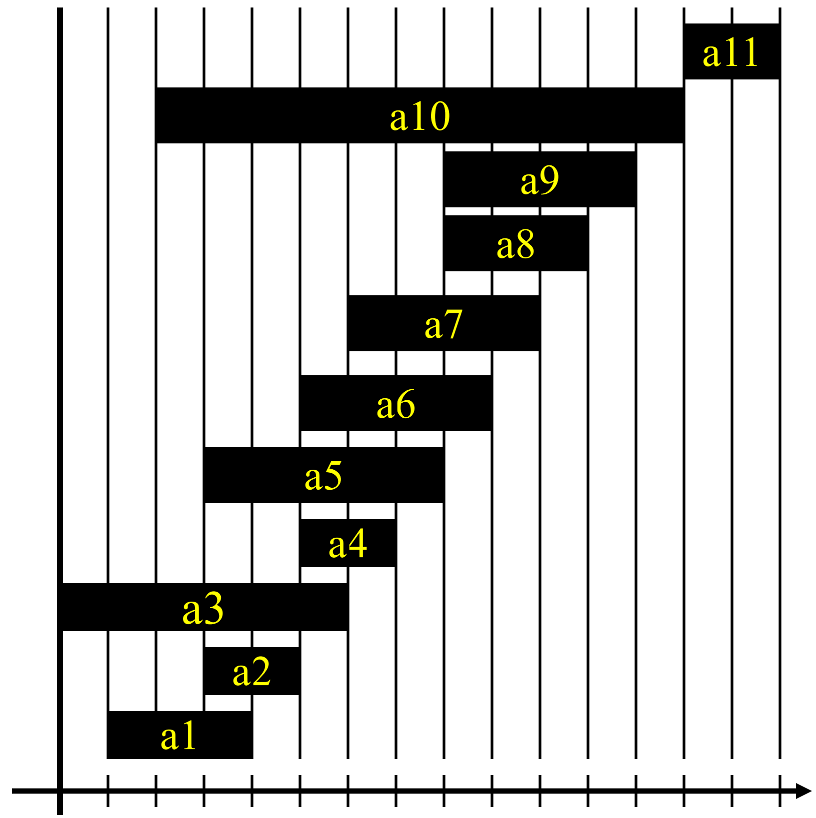
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Example: Activity Selection

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

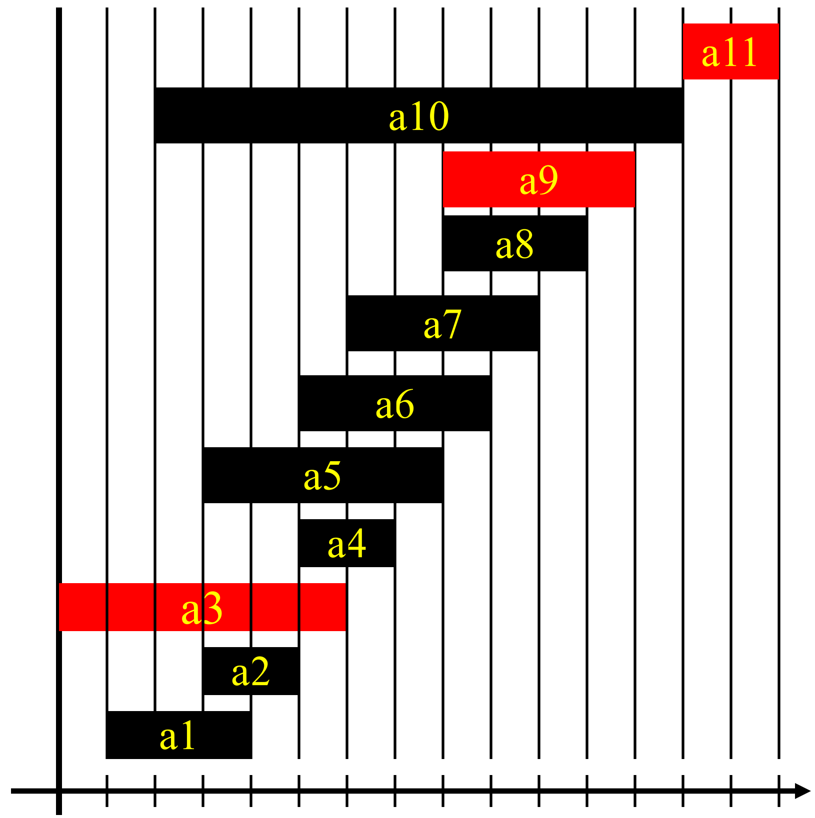
- It is possible to choose many mutually exclusive sets of tasks:



Example: Activity Selection

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

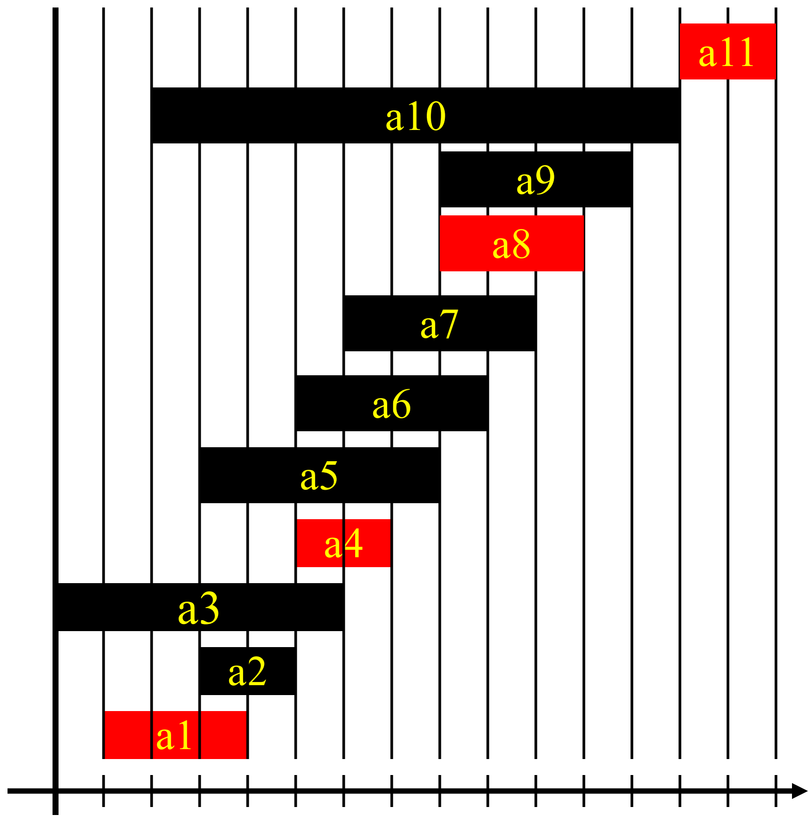
- It is possible to choose many mutually exclusive sets of tasks:
  - $\{a_3, a_9, a_{11}\}$



Example: Activity Selection

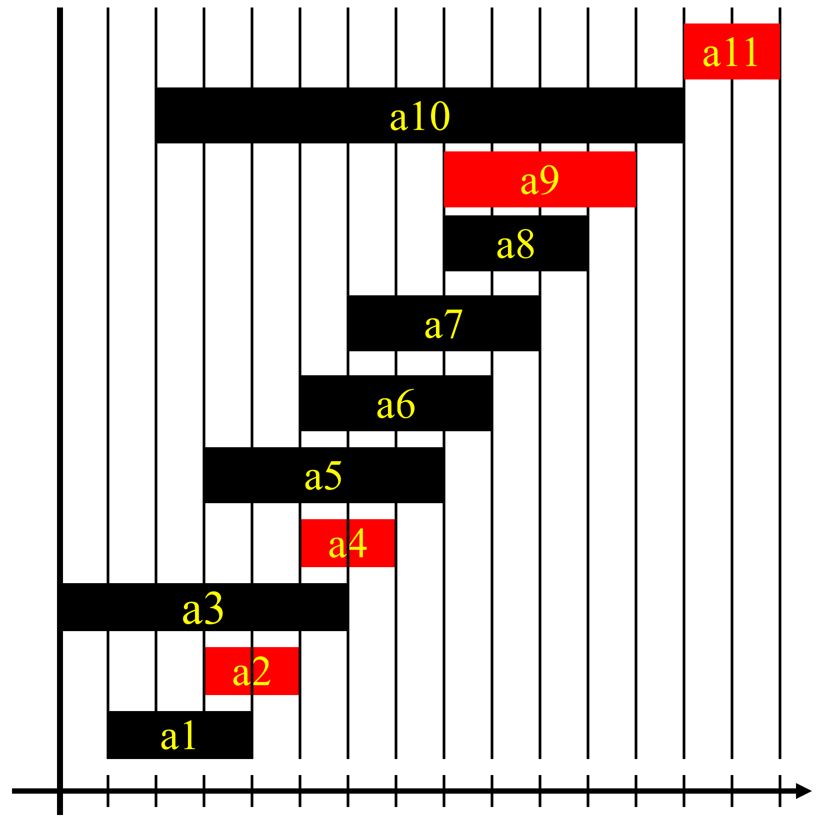
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- It is possible to choose many mutually exclusive sets of tasks:
  - $\{a_3, a_9, a_{11}\}$
  - $\{a_1, a_4, a_8, a_{11}\}$



Example: Activity Selection

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



- It is possible to choose many mutually exclusive sets of tasks:
  - $\{a_3, a_9, a_{11}\}$
  - $\{a_1, a_4, a_8, a_{11}\}$
  - $\{a_2\ a_4, a_9, a_{11}\}$

Example: Activity Selection

- Assume Activities are sorted s.t. $f_1 \leq f_2 \leq \ldots \leq f_n$
- Greedy Choice:
  - **Select Activity with lowest finish time f_k**
    - Rationale? Maximize time for remaining activities

```
function selectActivitiesGreedy(set S, set F)
  n = size[s];
  A = { 1 };
  j = 1;
  for i = 2 to n do
    if (s_i ≥ f_j) then
      A = A ∪ { i };
      j = i;
return A
```

Example: Activity Selection

- Assume Activities are sorted s.t. $f_1 \leq f_2 \leq \ldots \leq f_n$
- Greedy Choice:
  - **Select Activity with lowest finish time f_k**
    - Rationale? Maximize time for remaining activities

```
function selectActivitiesGreedy(set S, set F)
   n = size[s];      // number of activities
   A = { 1 };        // first choice – the one that ends first
    j = 1;
   for i = 2 to n do
      if (sᵢ ≥ fⱼ) then  // pick the first that begins right after
         A = A ∪ { i }; // the last one ends, and add it to sol.
         j = i;          // "advance" time to end of j
return A
```

Correctness of Greedy Algorithm

- Greedy Approach has the following Properties:
  - **Property 1.** There exists an optimal solution A starting with the greedy choice with activity 1.
    - Assume A is optimal solution starting with activity numbered k
    - Then we can define $B = A - \{ k \} \cup \{ 1 \}$
    - Since $f_1 \leq f_k$ and $f_1 \leq s_j$ for $j \neq k$
      - Activities are ordered ; Activity 1 is compatible with activities other than k
      - Activities in A and B are mutually disjoint and $|A| = |B|$ (we can trade k with 1)
      - Then B is also an optimal solution **(same number of tasks)!**
    - **Conclusion:** an optimal solution exists that begins with activity 1.
  - **Property 2.** After the first choice the problem resumes to finding a solution of acitivities compatible with activity 1.
    - Let A be an optimal solution starting with activity 1
    - Then $A' = A - \{ 1 \}$ must be an optimal solution to $S' = \{ i \in S : s_i \geq f_1 \}$
      - Otherwise there would be a solution $|B'| > |A'|$ for S' that would allow is to obtain solution B for S with more activities than A; a contradiction **!**
  - **Apply Induction to the Number of Greedy Choices**
- **Conclusion:** Greedy Algorithm computes optimal solution **!**

Correctness of Greedy Algorithm

- Property 1 - The Greedy-Choice Property
  - A globally optimal solution that can be arrived at by making a locally optimal (greedy) choice.

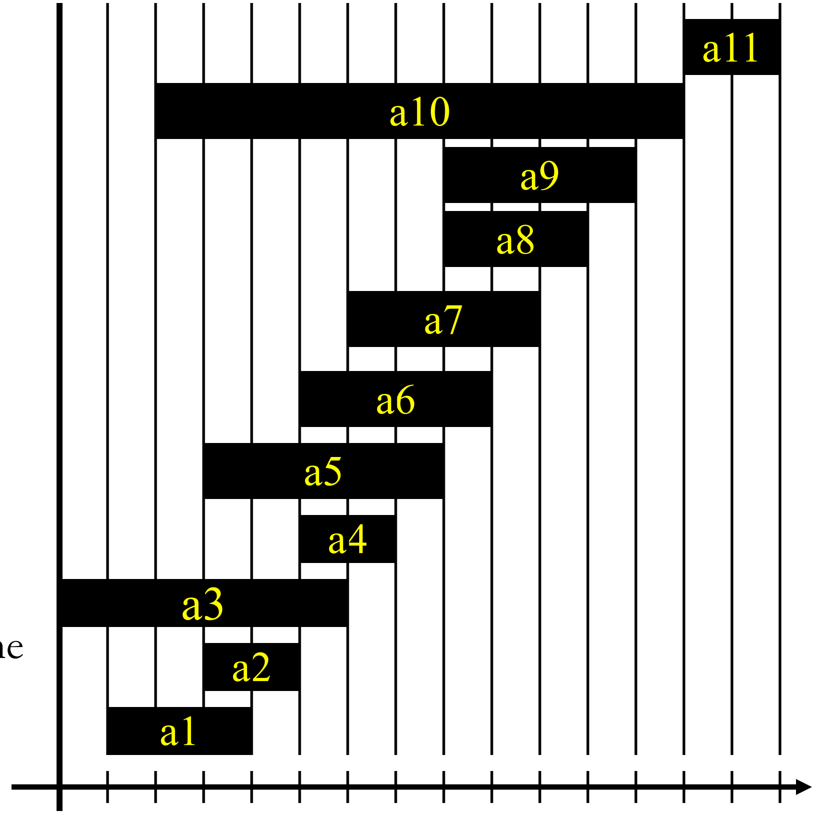
- Property 2 – The Optimal substructure Property
  - An optimal solution to the problem contains within it optimal solutions to subproblems.

Features of Greedy Approaches

- Properties of Greedy Choices
  - Global Optimal choice can be made by making optimal local choices.
- Optimal Sub-structure
  - Optimal solution to problem includes optimal solutions to sub-problems
  - Matroid Structure...
- Similar to Dynamic Programming
  - Choices can be made entirely based on local criteria and without exploring multiple local solutions.

Example: Activity Selection

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



- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time

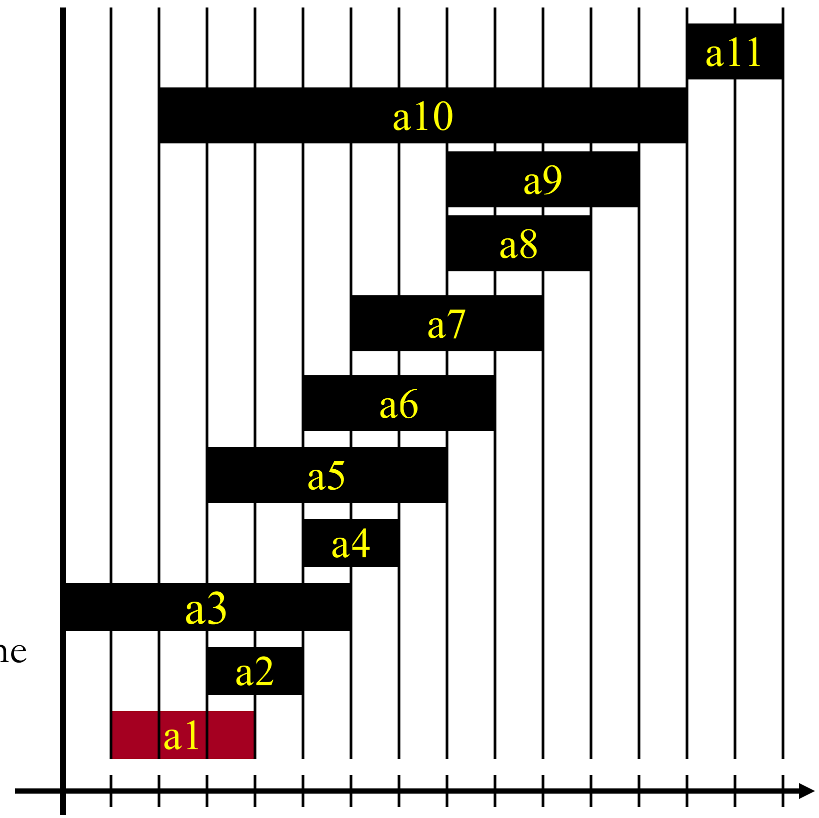
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Example: Activity Selection

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

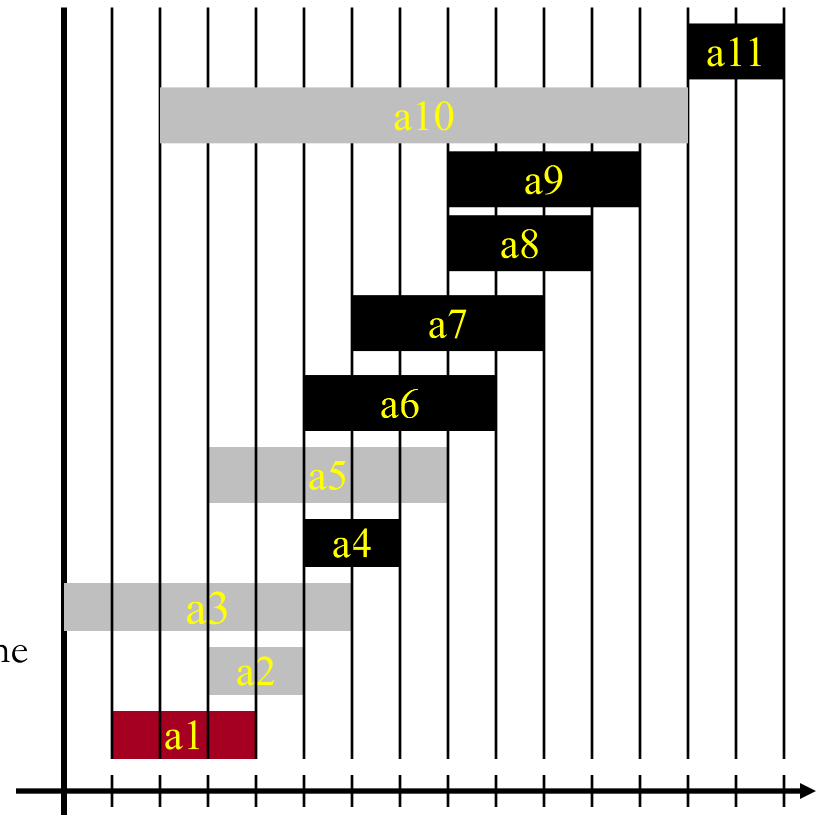
- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- { a_1 }



Example: Activity Selection

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

- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- { a_1 }

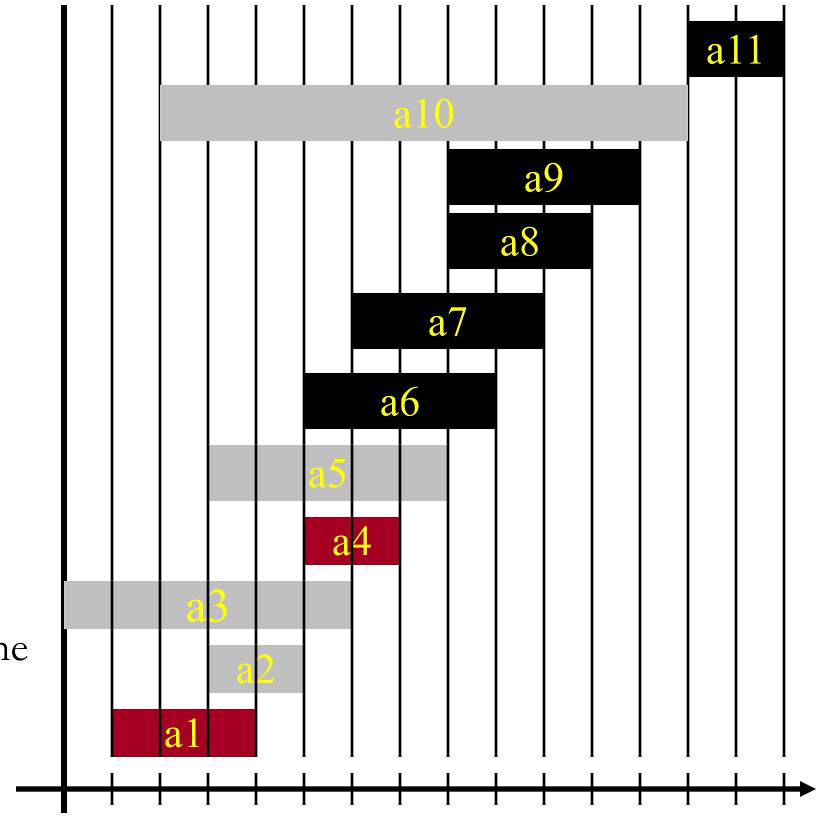


Example: Activity Selection

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

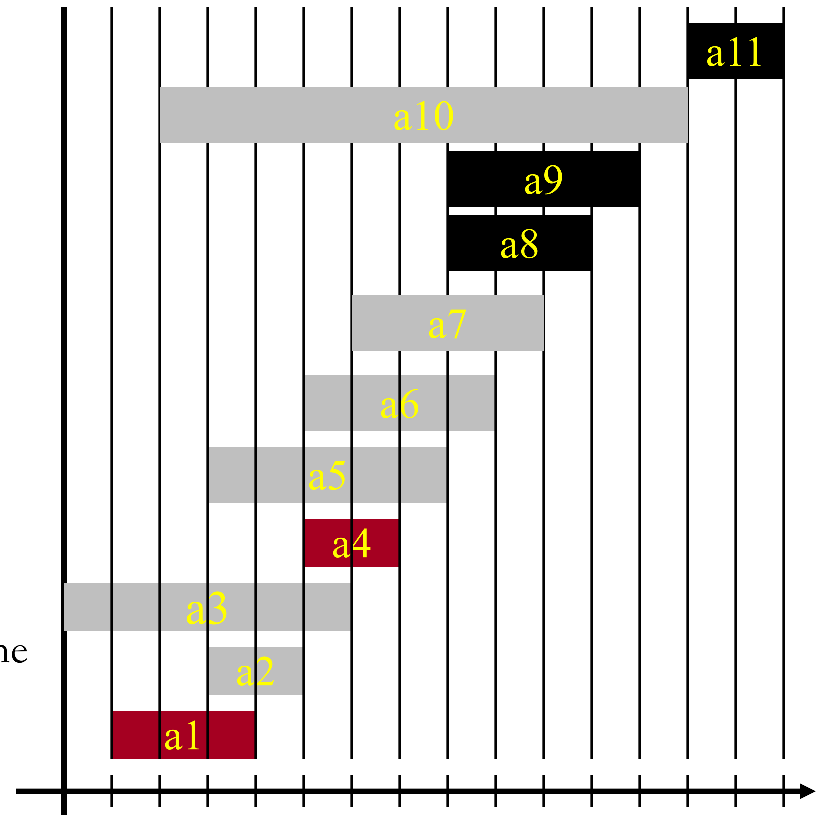
- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time

- $\{ a_1, a_4 \}$



Example: Activity Selection

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



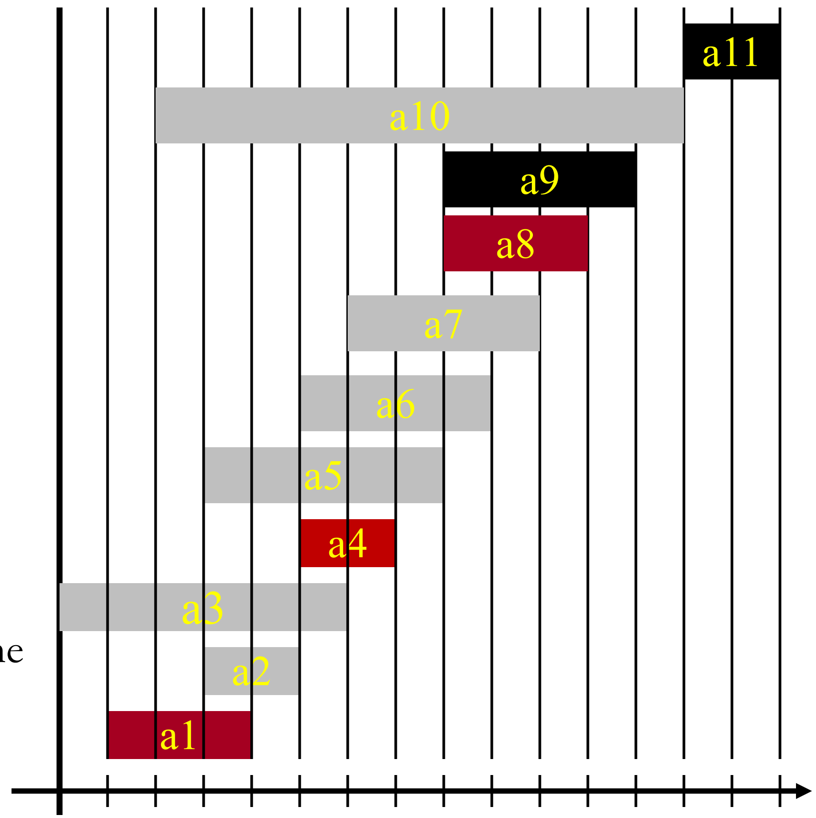
- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- $\{ a_1, a_4 \}$

Example: Activity Selection

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

- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time

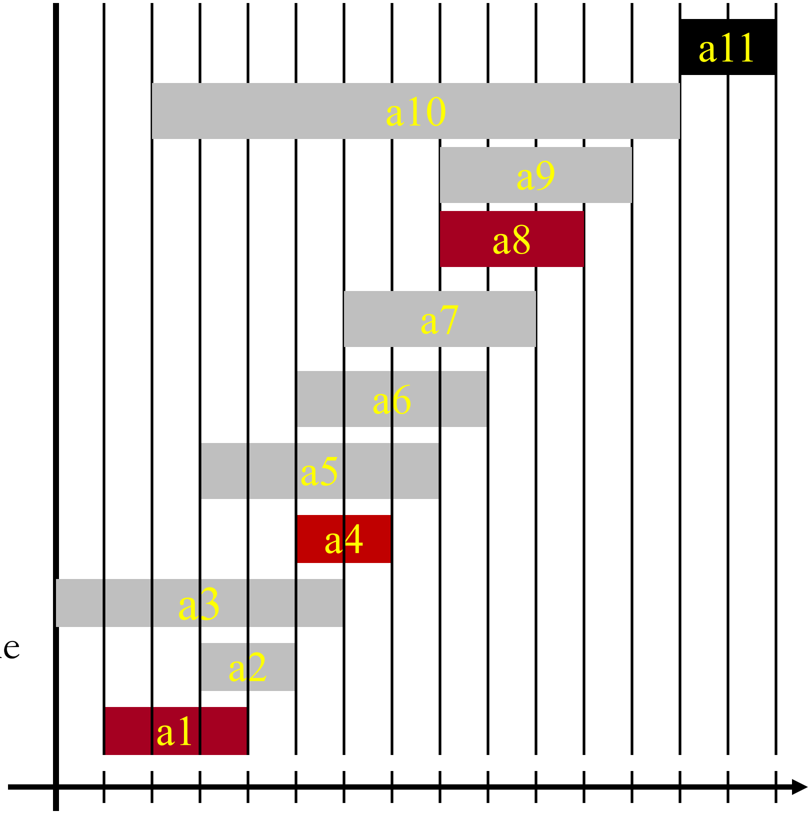
- $\{ a_1, a_4, a_8 \}$



Example: Activity Selection

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

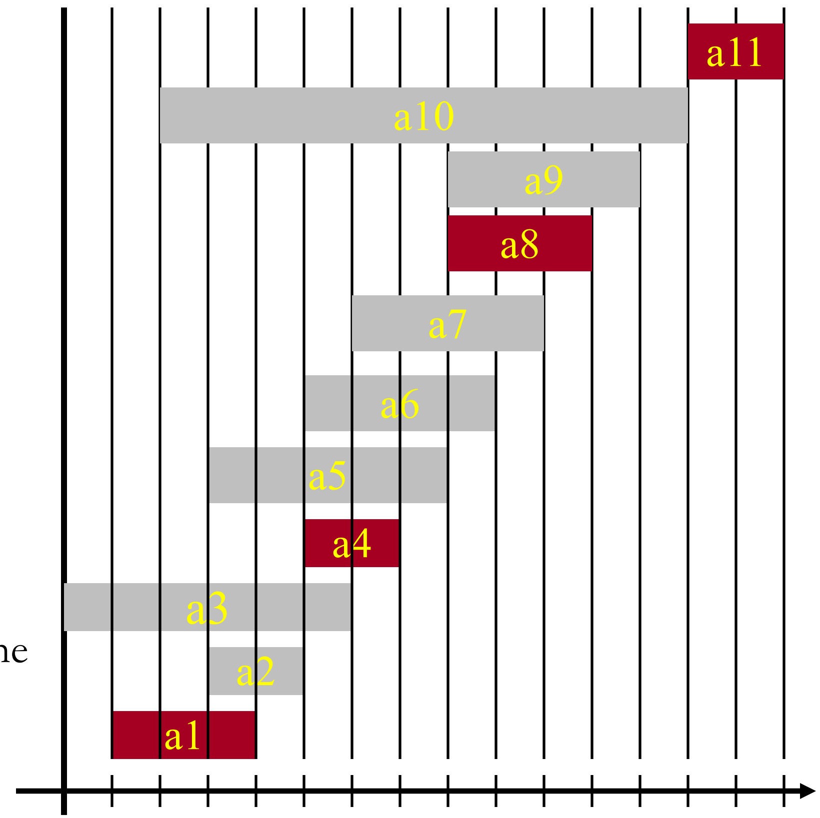
- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- $\{ a_1, a_4, a_8 \}$



Example: Activity Selection

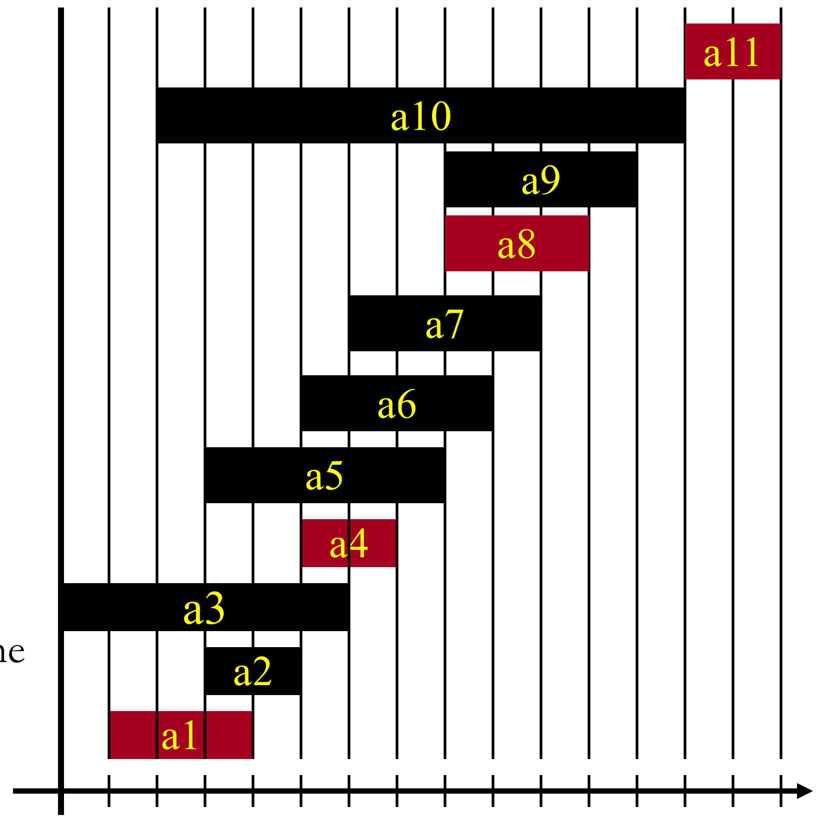
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- $\{ a_1, a_4, a_8, a_{11} \}$



Example: Activity Selection

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



- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- { a_1, a_4, a_8, a_{11} } optimal solution
- **But not unique !**

Execution Time Complexity:

O(n log n) + O(n) = O(n log n)

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA DEI
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Example: Knapsack Problem

- Problem Definition:
  - Given n objects (1, …, n) and a Knapsack of capacity W
  - Each object has value v_i and weight w_i
  - **It is possible to transport a fraction x_i of an object: $0 \leq x_i \leq 1$**
  - Transported weight cannot exceed W
  - Objective:
    - Maximize the transported value of objects while meeting the Knapsack's weight constraint
- Formalization:

$$\max \quad \sum_{i=1}^{n} x_i v_i$$

$$\textit{such that} \quad \sum_{i=1}^{n} x_i w_i \leq W$$

$$v_i \geq 0,\ w_i \geq 0,\ 0 \leq x_i \leq 1,\ 1 \leq i \leq n$$

Example: Knapsack Problem

- Observations:
  - Sum of the selected objects cannot execeed weight limit W
  - Optimal solution must fill up knapsack entirely, $\sum x_i w_i = W$
    - Otherwise we could transport more ***fractional*** items, thus with larger aggregate value **!**

- Algorithm:

```
function fillUpKnapsackGreedy(v, w, W)
   weight = 0;
   while weight < W do
      select element i with maximal vi/wi
      if (wi + weight ≤ W) then
         xi = 1; weight += wi
      else
         xi = (W-weight)/ wi; weight = W
```

Execution Time:
O(n), or O(n log n)

Example: Knapsack Problem

- Observations:
  - Sum of the selected objects cannot execeed weight limit W
  - Optimal solution must fill up knapsack entirely, $\sum x_i w_i = W$
    - Otherwise we could transport more ***fractional*** items, thus with larger aggregate value **!**

- Algorithm:

function fillUpKnapsackGreedy(v, w, W)
  weight = 0;
  while weight < W **do**
    select element i with maximal $\mathbf{v_i/w_i}$
    if (w_i + weight ≤ W) **then**
      x_i = 1; weight += w_i
    else
      x_i = (W-weight)/ w_i; weight = W

fraction of last object to fit into Knapsack

Execution Time:
O(n), or O(n log n)

Knapsack Greedy Algorithm Optimality

- Proof by Contradiction:
  - Let item i be the item with the maximum value to weight ratio (v/w). We want to show that the optimal solution contains as much of item i as possible.
  - We prove that this statement is true by contradiction. We start by assuming that there is an optimal solution where we did not take as much of item i as possible and we also assume that our knapsack is full (If it is not full, just add more of item i!).
  - Since item i has the highest value to weight ratio, there must v_j exist an item j in our knapsack such that $v_j/w_j < v_i/w_i$.
  - We can take item j of weight x from our knapsack and we can add item i of weight x to our knapsack (Since we take out x weight and put in x weight, we are stilll within capacity.).
  - The change $x\,(v_i/w_i) - x\,(v_j/w_j) = x\,((v_i/w_i) - (v_j/w_j)) > 0$ since $v_j/w_j < v_i/w_i$
  - Therefore, we arrive at a contradiction because the ”so-called” optimal solution in our starting assumption, can in fact be improved by taking out some of item j and adding more of item i.

Knapsack Greedy Algorithm Optimality

- **Greedy Choice Property:** The optimal solution contains the best item according to the algorithm's greedy criterion.
- **Optimal Substructure:** The optimal solution to problem S contains an optimal to subproblems of S.

Optimality: Greedy Choice Property

- Let item i be the item with the maximum value to weight ratio (v_i/w_i).
- **Goal:** Show that the optimal solution contains as much of item i as possible.

- **Proof:** (by contradiction – as sketched before)
  - Optimal solution X takes as much of item i as possible, say x_i
  - Solution Y has $y_i < x_i$ and is also "optimal"
  - Since v_i/w_i has the highest ratio, there must exist an item j in Y with $v_j/w_j < v_i/w_i$ then, we can take k weight of item j and assign it to item i, yielding a net value improvement of k (v_i/w_i - j_i/w_j) > 0.
  - Therefore, Y was not optimal after all (the contradiction).

Optimality Proof: Optimal Subproblem

- Assume that X is the Optimal solution to problem S with value V and knapsack capacity W.
- Then, $X' = X - x_j$ is an Optimal solution to subproblem $S' = S - \{j\}$ and knapsack capacity $W' = W - w_j$
- **Proof** (by contradiction):
  - Assume X' is not optimal to S' and that we have another solution X'' to S' that has a higher total value $V'' > V'$.
  - Then, $X'' \cup \{x_j\}$ is a solution to S with value $V''+v_j > V' + v_j = V$.
  - This is a contradiction as V is assumed to be optimal.

Problem: Minimize System Processing Time

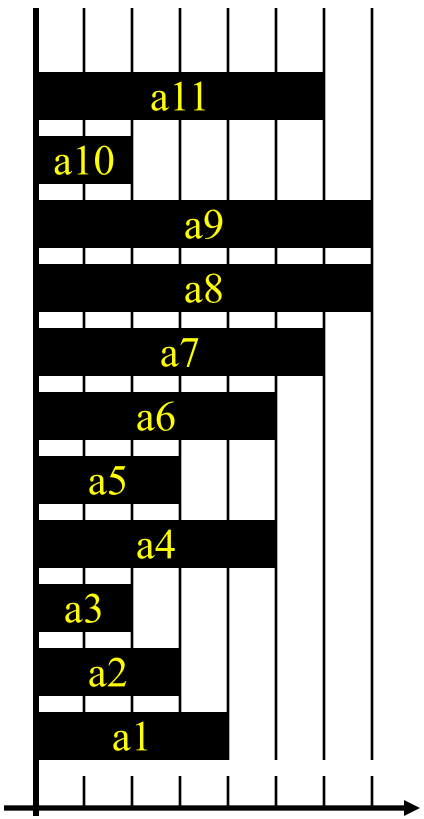
- Given a Server with **n** clients, each with known service time (*i.e.* client **i** takes time $\mathbf{t_i}$), minimize the total time taken in the system serving all clients

$$\min imize\ T = \sum_{i=1}^{n} \left(\text{total time in the system by client i}\right)$$

- Greedy Solution:
  - Process Clients by Increasing Order of Service Time
  - **Rationale:** Take care of fast orders first, leads to lower aggregate wait time for others – fewer people waiting in line...

Example: Minimize System Processing Time

i	1	2	3	4	5	6	7	8	9	10	11
s_i	4	3	2	5	3	5	6	8	8	2	7



- Strategy 1: Process longest jobs first !
  - Order of service = { 9, 8, 7, 11, 4, 6, 1, 2, 3, 10 }
  - Total Service Time = $10 \times 8 + 9 \times 8 + 8 \times 7 + 7 \times 6 + 6 \times 5 + 5 \times 4 + 4 \times 3 + 3 \times 3 + 2 \times 2 + 1 \times 2 = 315$

- Strategy 2: Process shortest jobs first !
  - Order of service = { 10, 3, 2, 1, 6, 4, 11, 7, 8, 9 }
  - Total Service Time = $10 \times 2 + 9 \times 2 + 8 \times 3 + 7 \times 3 + 6 \times 4 + 5 \times 5 + 4 \times 6 + 3 \times 7 + 2 \times 8 + 1 \times 8 = 191$

Example (Cont.)

- Greedy Algorithm finds Optimal Solution
  - $P = p_1 p_2 \ldots p_n$, is a permutation of the integer from 1 to n
    - Let $s_i = t_{p_i}$
      - *e.g.*, $s_1 = t_{p_1} = t_5$
  - Given the client to be processed by order P, the service time for the client in position i is s_i
  - Total time spent by all clients in the system is:

$$T(P) = \sum_{k=1}^{n} (n-k+1) s_k$$

s_1 shows up n times, and s_n only once

  - Assume clients are sorted by increasing order of service time in P
    - If there are indeces a and b, with a < b, and $s_a > s_b$

Example (Cont.)

- We can swap the order of the clients a and b, to get the order P'
  - Same as P with integers p_a and p_b swapped

$$T(P') = (n-a+1)\, s_b + (n-b+1)\, s_a + \sum_{\substack{k=1 \\ k\neq a,b}}^{n} (n-k+1)\, s_k$$

- Yielding,

$$\begin{aligned} T(P) - T(P') &= (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) \\ &= (b-a)(s_a - s_b) > 0 \end{aligned}$$

- That is P' is a better order of service (with lower total service time)

- Algorithm finds the Optimal Solution !

Example: Huffman Codes

- Applications in data compression
- Example:
  - File with 100,000 characters
  - Fixed-length encoding: each symbol gets a code of the same length

	a	b	c	d	e	f
Frequency (x1000)	45	13	12	16	9	5
Code	000	001	010	011	100	101

  - Compressed file size: $3 \times 100{,}000 = 300{,}000$ bits
  - Variable-Length code may be better than Fixed-Length code
    - Associate shorter codes to more frequent characters

Example: Huffman Codes

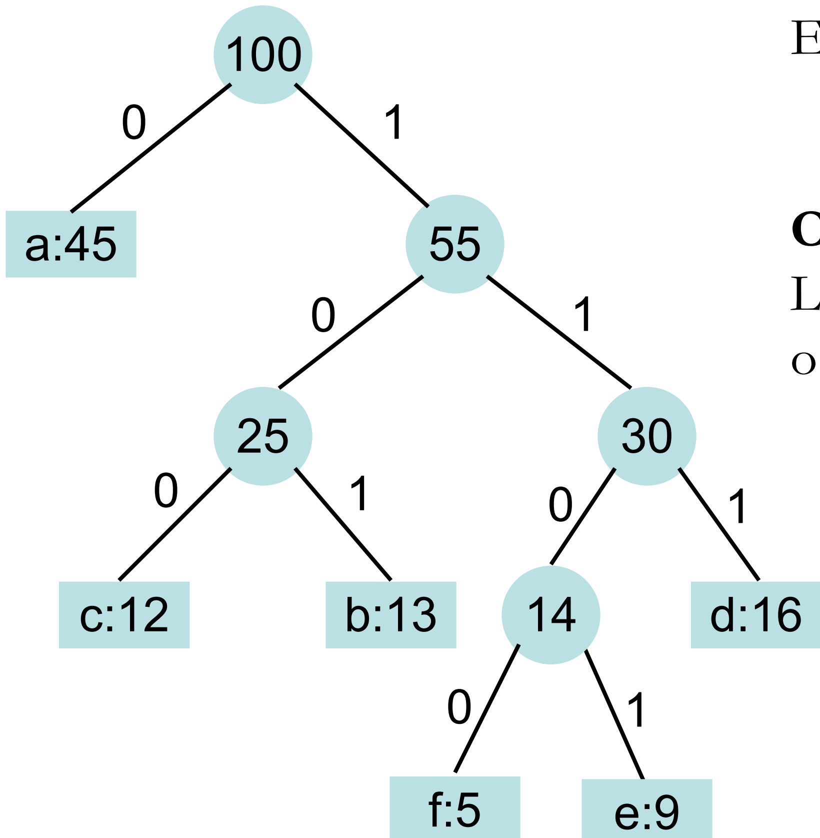
- Variable-Length encoding:

	a	b	c	d	e	f
Frequency (x1000)	45	13	12	16	9	5
Variable Code	0	101	100	111	1101	1100

  – Number of required bits:
    - $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224{,}000$ bits

- Prefix-free codes:
  – No code is prefix of another code
  – $001011101 \rightarrow 0.0.101.1101$
  – Codes represented by a complete binary tree

Prefix-Free Codes



Complete Binary Tree:
Each internal node with two children

Observation:
Length of code for character = depth of character in the tree

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Example: Huffman Codes

- Given a tree T associated with a prefix-free code
  - f(c): frequency (occurrency) of character c in a file/stream
  - $d_T(c)$: depth of leave c in the tree

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

Number of required bits to represent file

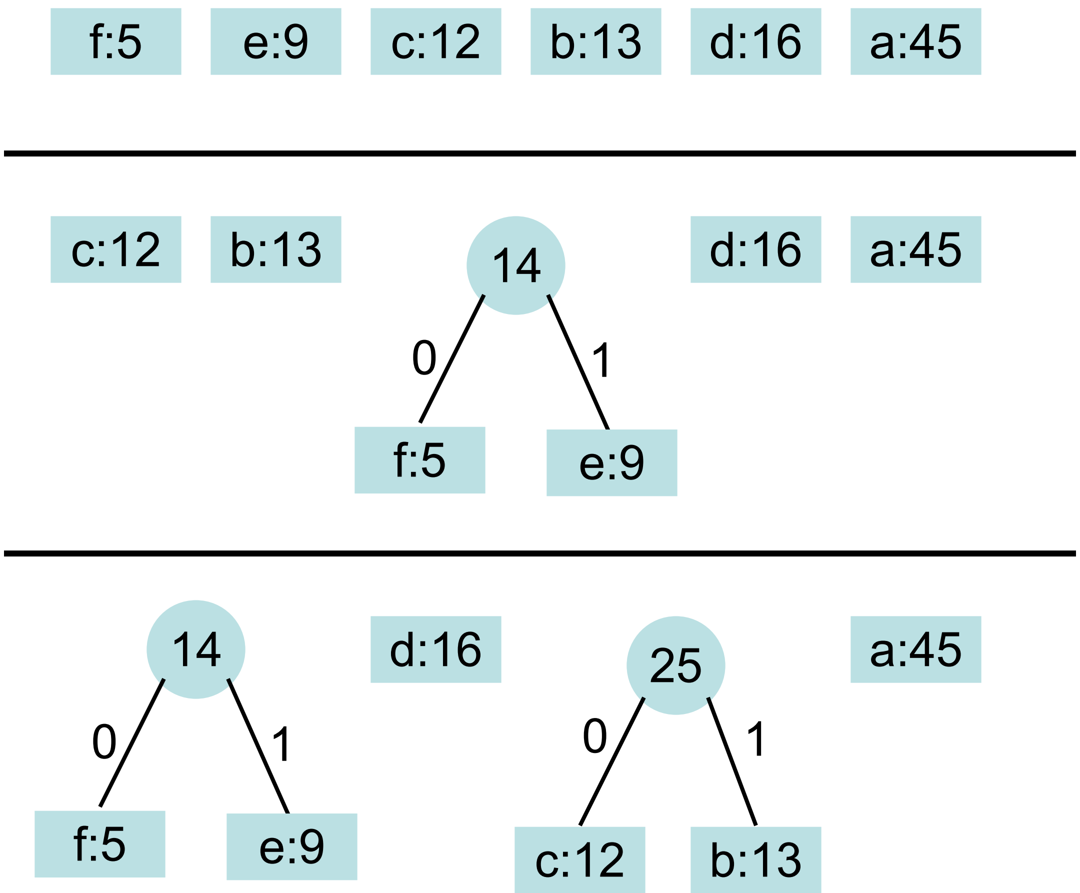
- Huffman Code:
  - Begin with each character $c \in C$ with frequency f[c]
  - Develop a prefix-free code for C represented by a binary tree T
  - Begin with |C| leaves (for each character in the file) and perform |C| - 1 merge operation to obtain final tree
  - How?
    - Aggregating x, y characters in C with the least frequencies into a “symbol” with aggregate frequency
    - More frequent symbols will be closer to root of tree and thus with shorter code lengths.

Example: Huffman Codes

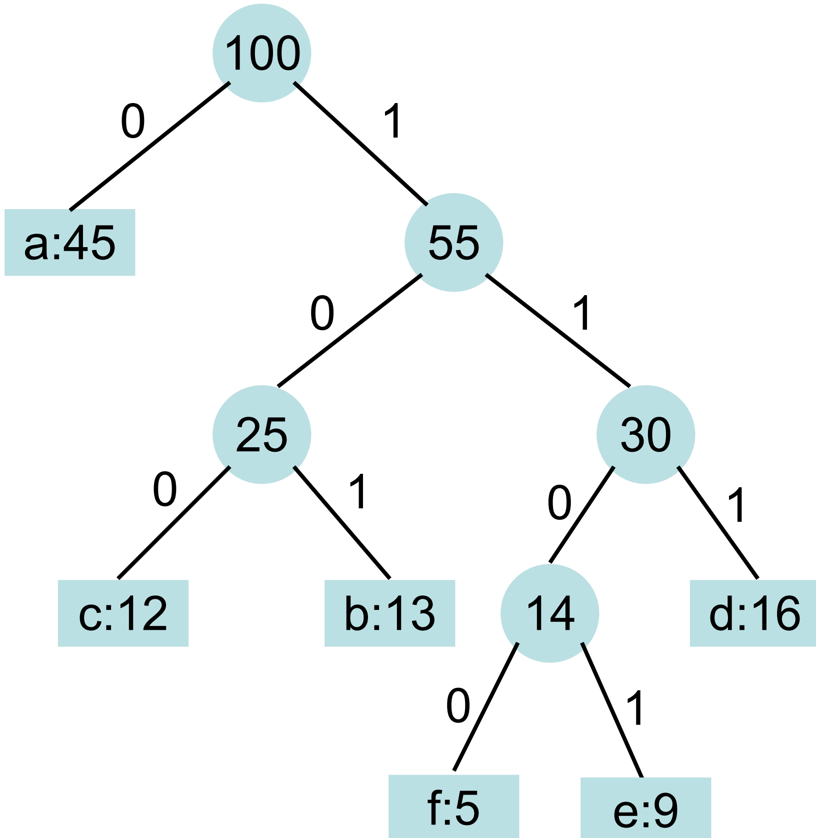
```
function Huffman(C)
  n = |C|;
  Q = C ;                          // Constructs priority queue
  for i = 1 to n - 1 do
     z = AllocateNode();
     x = left[z] = ExtractMin(Q);
     y = right[z] = ExtractMin(Q);
     f[z] = f[x] + f[y];
     Insert(Q, z);
 return
```

Execution Time: $O(n \log n)$

Example


f:5
e:9
c:12
b:13
d:16
a:45
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45

Example



- Codes:
  - a: 0
  - b: 101
  - c: 100
  - d: 111
  - e: 1101
  - f: 1100

	a	b	c	d	e	f
Frequency (x1000)	45	13	12	16	9	5
Variable Code	0	101	100	111	1101	1100

Huffman Greedy Algorithm Optimality

- **Greedy Choice Property:**

  Theorem: There exists a prefix-free code for C such that the codes x and y (with the least frequencies) have the same length and difer only in the last bit

  Proof: (by contradiction)

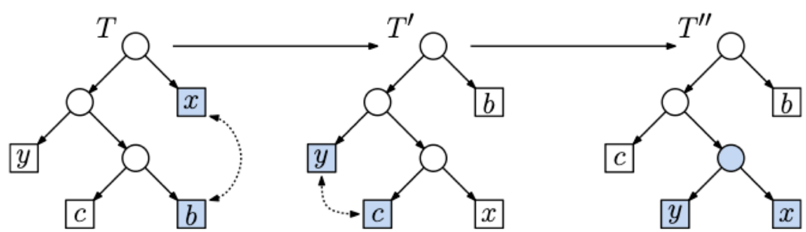
  – Assume that T is optimal but x and y **do not have** the same code length, that is their depths in the tree are diferent.

  – Then, in T there must be two symbols b and c (siblings) both at maximum depth (one of which may be either x or y but not both, by assumption).

  – Assume, $f[b] \leq f[c]$, and $f[x] \leq f[y]$. Note that since x and y have the smallest frequencies, if follows that $f[x] \leq f[b]$, and $f[y] \leq f[c]$ (some maby be identical pairwise).

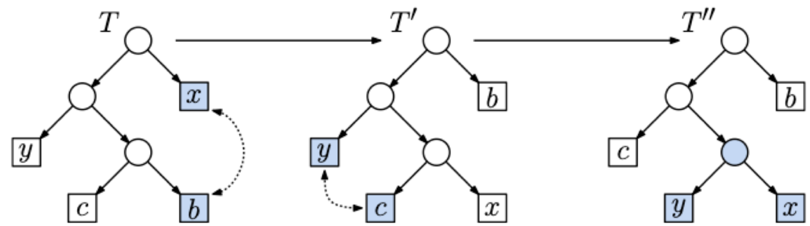
  – Because b and c are at the maximum depth, $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$ and we have that $f[b] - f[x] \geq 0$ and $d_T(b) - d_T(x) \geq 0$ and hence their product is negative.

  – Then, we can create another trees T' by swapping positions of x and b in T and then in T'' by swapping positions of c and y .



Huffman Greedy Algorithm Optimality

- **Greedy Choice Property: (cont)**
  – In these new trees B(T) ≥ B(T') ∧ B(T') ≥ B(T'') which is a contradiction since T is optimal ⇒ B(T'), B(T'') ≥ B(T)



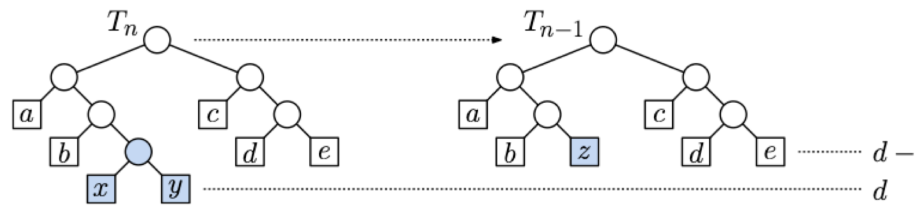
$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

  – This proof applies to just a pair of nodes, those with the lowest frequencies.
  – Induction, requires we convert the problem of n to n-1 characters.

Huffman Greedy Algorithm Optimality

- **Optimal Sub-struture Property:**

Theorem: Let z be an internal node of T, and x and y leaf nodes, then the tree T' = T - {x, y} is an optimal prefix tree for C' = C - {x, y} ∪ { z } where z has f[z] = f[x] + f[y].



Proof:

– B(T) = B(T') + f[x] + f[y], as z is placed at a higher tree level and hence, its code lenght is smaller.

– If T' Is not optimal, then there exists T'' such that B[T''] < B[T']

– But z is a leaf node in T'' (see Greedy choice property)

  - Adding x and y as children of z in T''
  - We get a prefix-free code for C with cost: B[T''] + f[x] + f[y] < B[T]
  - But T is optimal (B[T''] + f[x] + f[y] ≥ B[T]); and so T' is also optimal

$$
\begin{aligned}
f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\
&= f[z]d_{T'}(z) + (f[x] + f[y])
\end{aligned}
$$

The Huffman algorithm produces an optimal prefix-free code

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Summary

- Greedy Algorithms:
  - Selection of Activities
  - Knapsack Problem
  - Minimization of System Tasks
  - Huffman Codes
- Features of Greedy Algorithms
  - Optimality of Greedy Choice
  - Optimality of Subproblems
  - Theory: Matroids