Greedy Algorithms

Christian Wulff-Nilsen

Fifth lecture

Algorithms and Data Structures

DIKU

February 20, 2023

Overview for today

- What is a greedy algorithm?
- Greedy choice property
- Optimal substructure
- Activity selection problem
- Huffman codes

What is a greedy algorithm?

- A greedy algorithm solves a problem by repeatedly making a choice that looks best at the moment
- Dynamic programming makes a choice after having solved subproblems
- A greedy algorithm first makes a choice and then solves subproblems
- After each greedy choice, only one subproblem remains
- Greedy algorithms are usually more efficient than DP algorithms
- However, some problems are solvable by DP but not using a greedy algorithm (e.g., the $0/1$ knapsack problem)

Greedy choice property

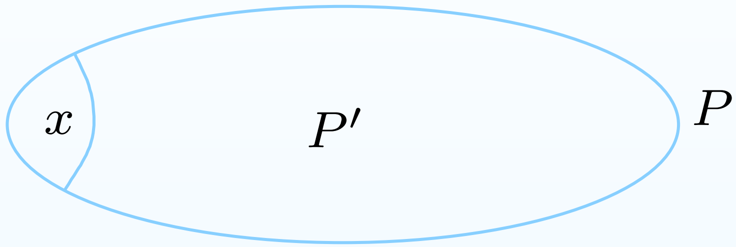
- Consider a greedy algorithm for a problem P
- After making a greedy choice, we need to be sure that we can extend it to an optimal solution to P
- We have the *greedy choice property* if **there exists an optimal solution to P which includes the greedy choice**
- Without the greedy choice property, the greedy algorithm could make a wrong greedy choice which excludes all optimal solutions
- Showing the greedy choice property is part of the correctness proof for a greedy algorithm

Optimal substructure for greedy algorithms

- Consider a problem P:

Optimal substructure for greedy algorithms

- Consider a problem P:



- Making a greedy choice x leaves us with a smaller problem P'
- Optimal substructure states that **if greedy choice x is in an optimal solution to P then this optimal solution consists of x and an optimal solution to P':**

$$\mathrm{OPT}(P) = \mathrm{OPT}(P') + x$$

- Note that optimal substructure does *not* require us to prove that x is in an optimal solution to P (we prove this when showing the greedy choice property)
- Also note that if we have optimal substructure then x combined with *any* optimal solution to P' yields an optimal solution to P

Optimal substructure and greedy choice property combined

- Suppose we can show both optimal substructure and the greedy choice property for a problem P
- Then we can solve P as follows:
  - Make a greedy choice x
  - x is in an optimal solution to P (by the greedy choice property)
  - Recursively find an optimal solution to remaining subproblem P' (making additional greedy choices along the way)
  - Combine x with optimal solution to P'
  - The combined solution is an optimal solution to P (by optimal substructure)
- Conclusion: greedy choice property + optimal substructure implies that a greedy algorithm finds an optimal solution to P

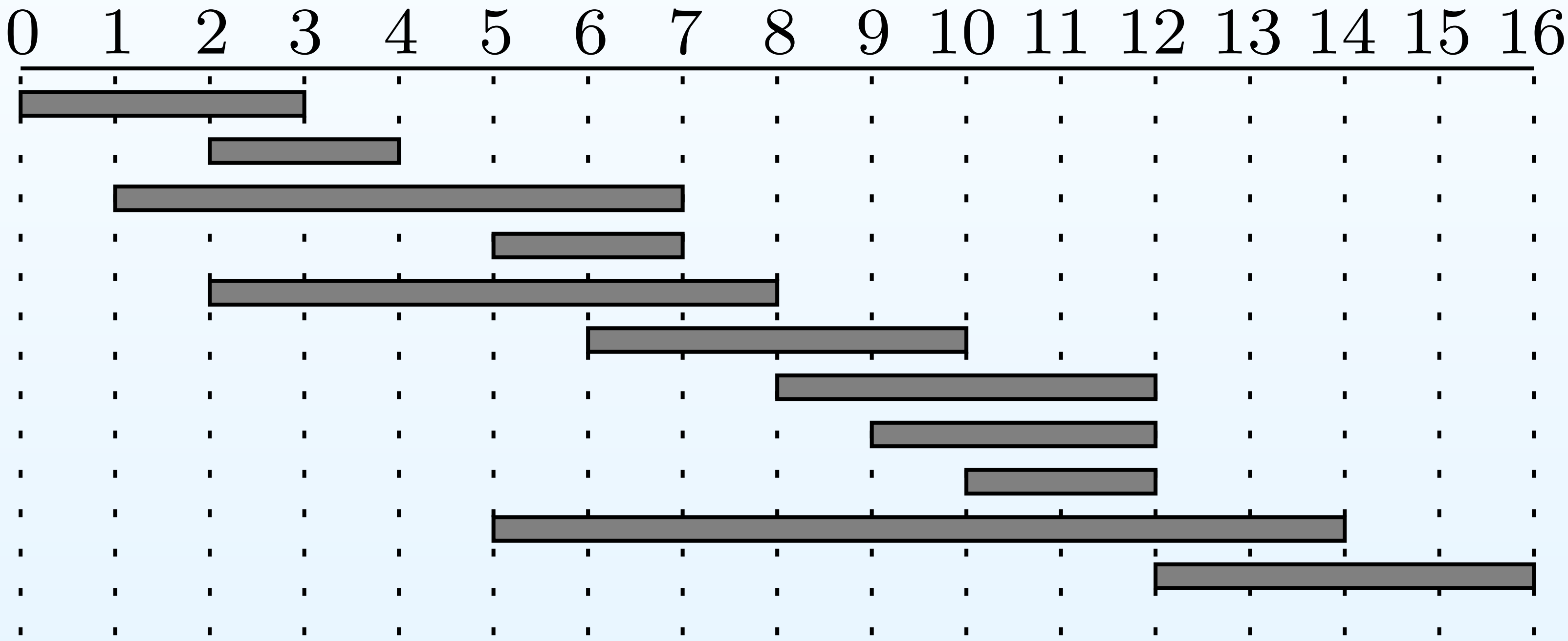
Activity selection

- We are given a set $S = \{a_1, \ldots, a_n\}$ of n *activities*
- Associated with each activity a_i is a *start time* s_i and a *finish time* f_i, $0 \leq s_i < f_i < \infty$
- If a_i takes place, it does so in the time interval $[s_i, f_i)$
- Two activities are *compatible* if their time intervals are disjoint
- Problem: find a maximum size subset of S of mutually compatible activities
- We assume that activities are sorted by finish time:

$$f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$$

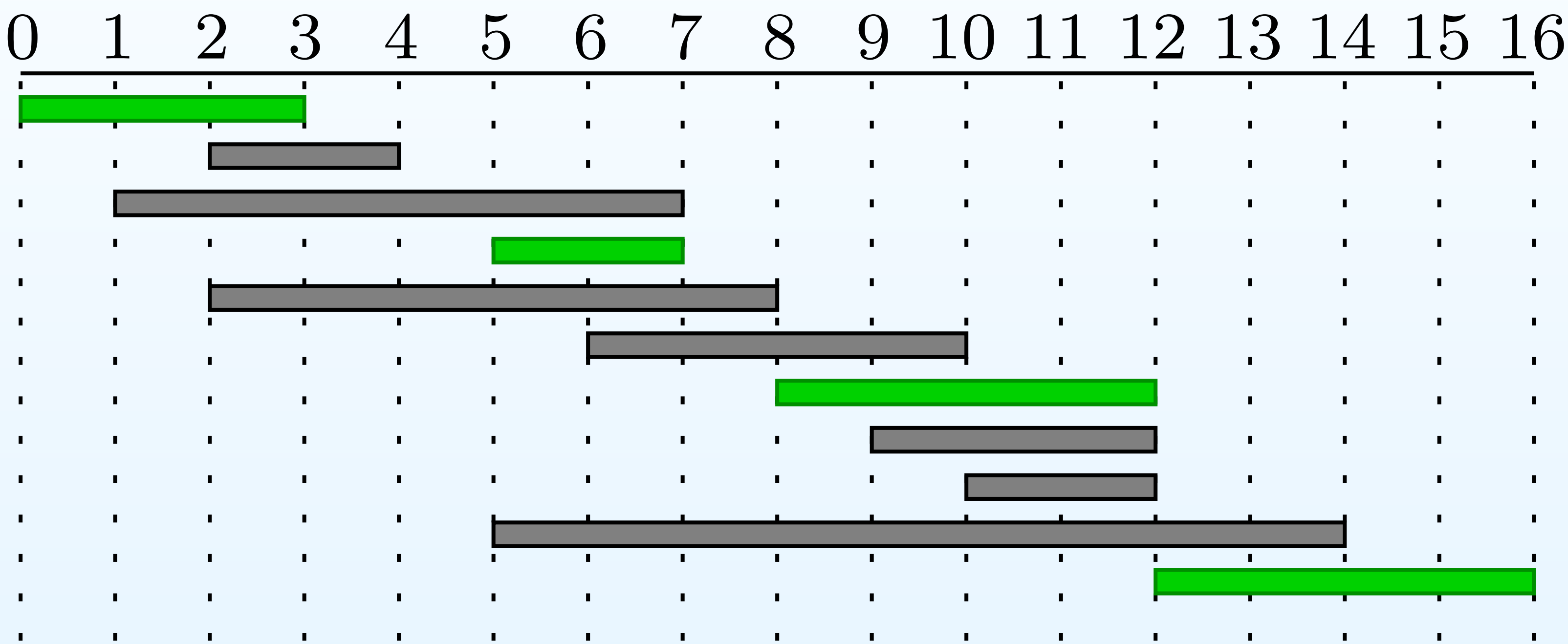
Activity selection

- Example:



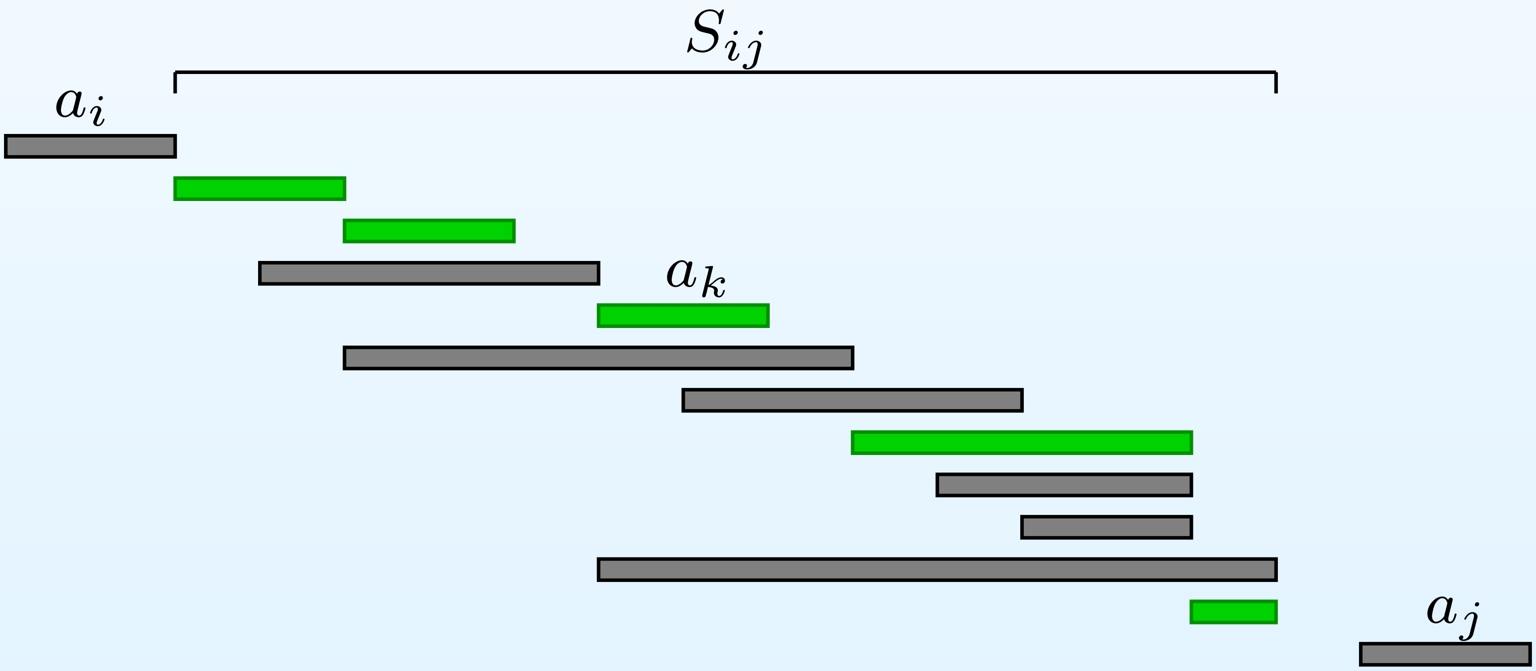
Activity selection

- Example:



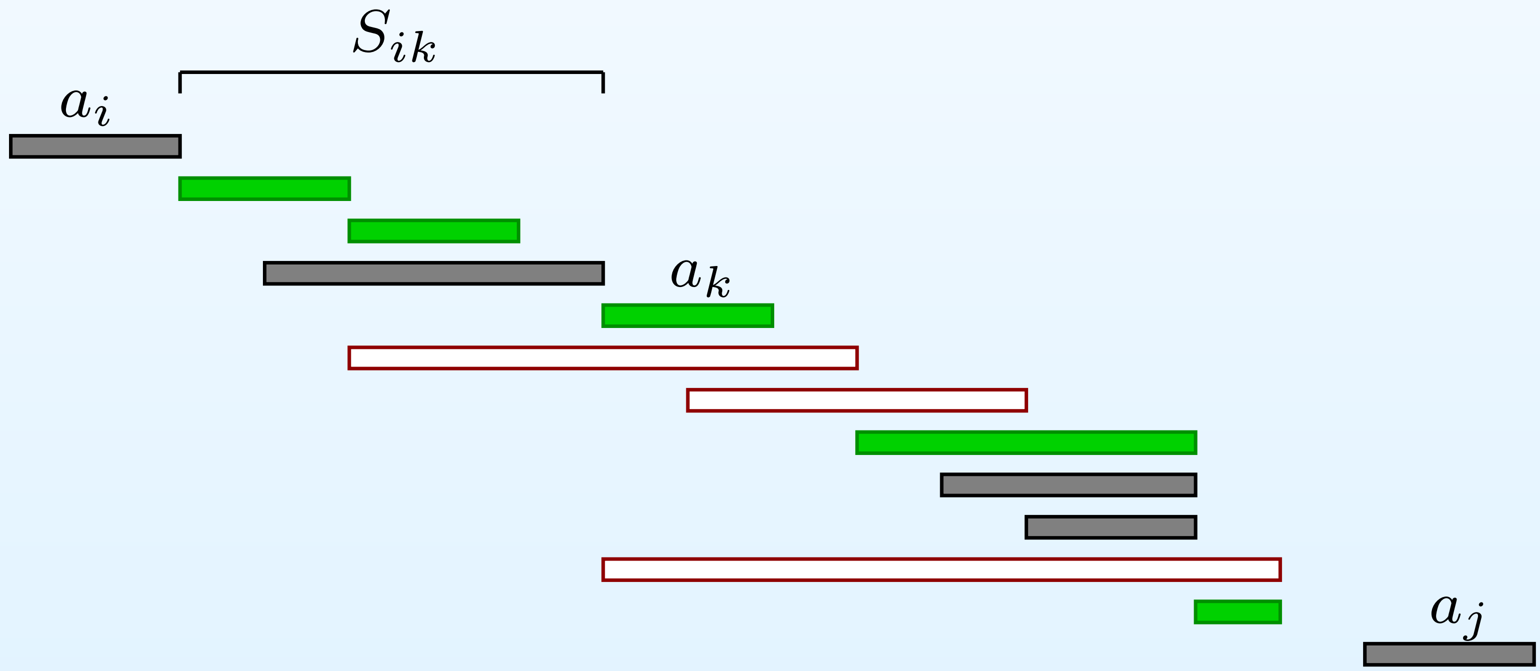
Optimal substructure (dynamic programming) for activity selection

- S_{ij}: activities starting after f_i and ending before s_j
- Assume a_k belongs to an optimal solution for S_{ij}
- Restricting this solution to S_{ik} resp. S_{kj} gives an optimal solution to S_{ik} resp. S_{kj}:



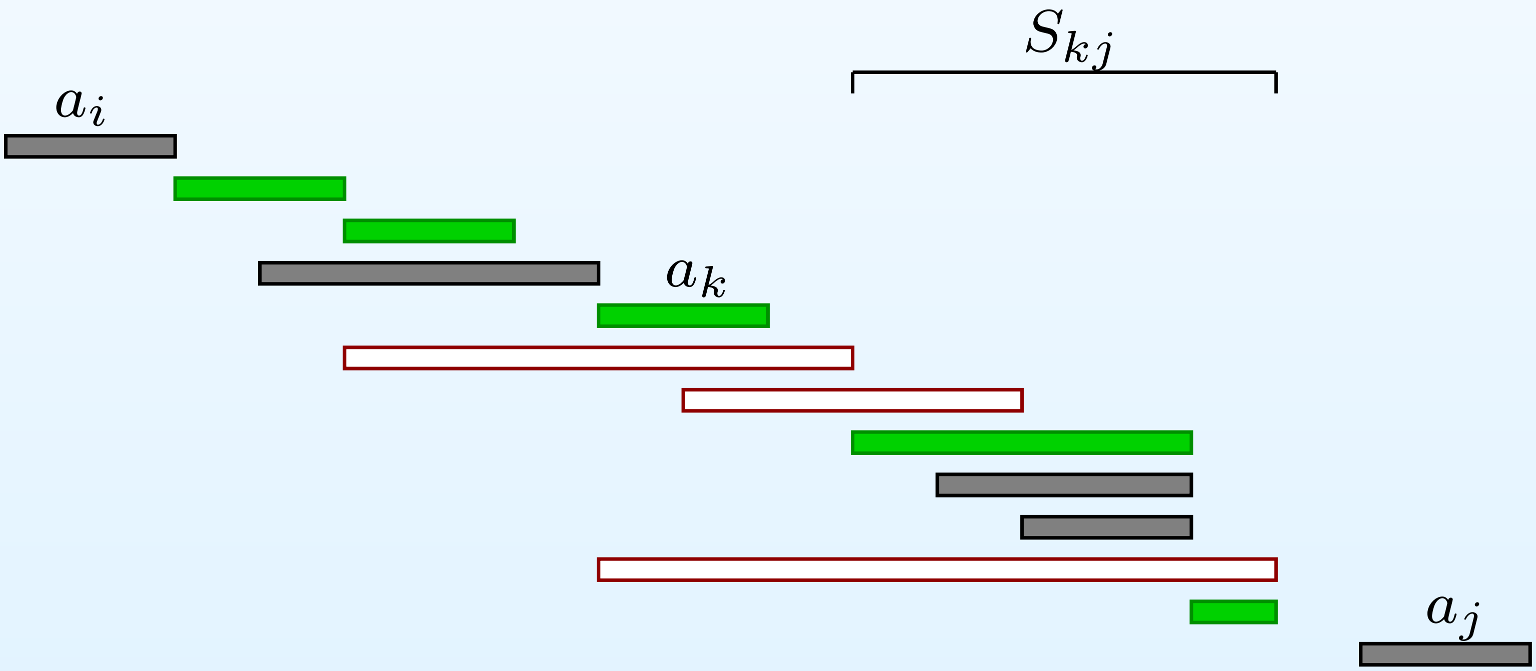
Optimal substructure (dynamic programming) for activity selection

- S_{ij}: activities starting after f_i and ending before s_j
- Assume a_k belongs to an optimal solution for S_{ij}
- Restricting this solution to S_{ik} resp. S_{kj} gives an optimal solution to S_{ik} resp. S_{kj}:



Optimal substructure (dynamic programming) for activity selection

- S_{ij}: activities starting after f_i and ending before s_j
- Assume a_k belongs to an optimal solution for S_{ij}
- Restricting this solution to S_{ik} resp. S_{kj} gives an optimal solution to S_{ik} resp. S_{kj}:



Recurrence for activity selection

- $c[i,j]$ denotes the size of an optimal solution for S_{ij}
- Assuming again that a_k belongs to an optimal solution for S_{ij}, we have:

$$c[i,j] = c[i,k] + c[k,j] + 1$$

- Since we do not know k, we try all choices:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

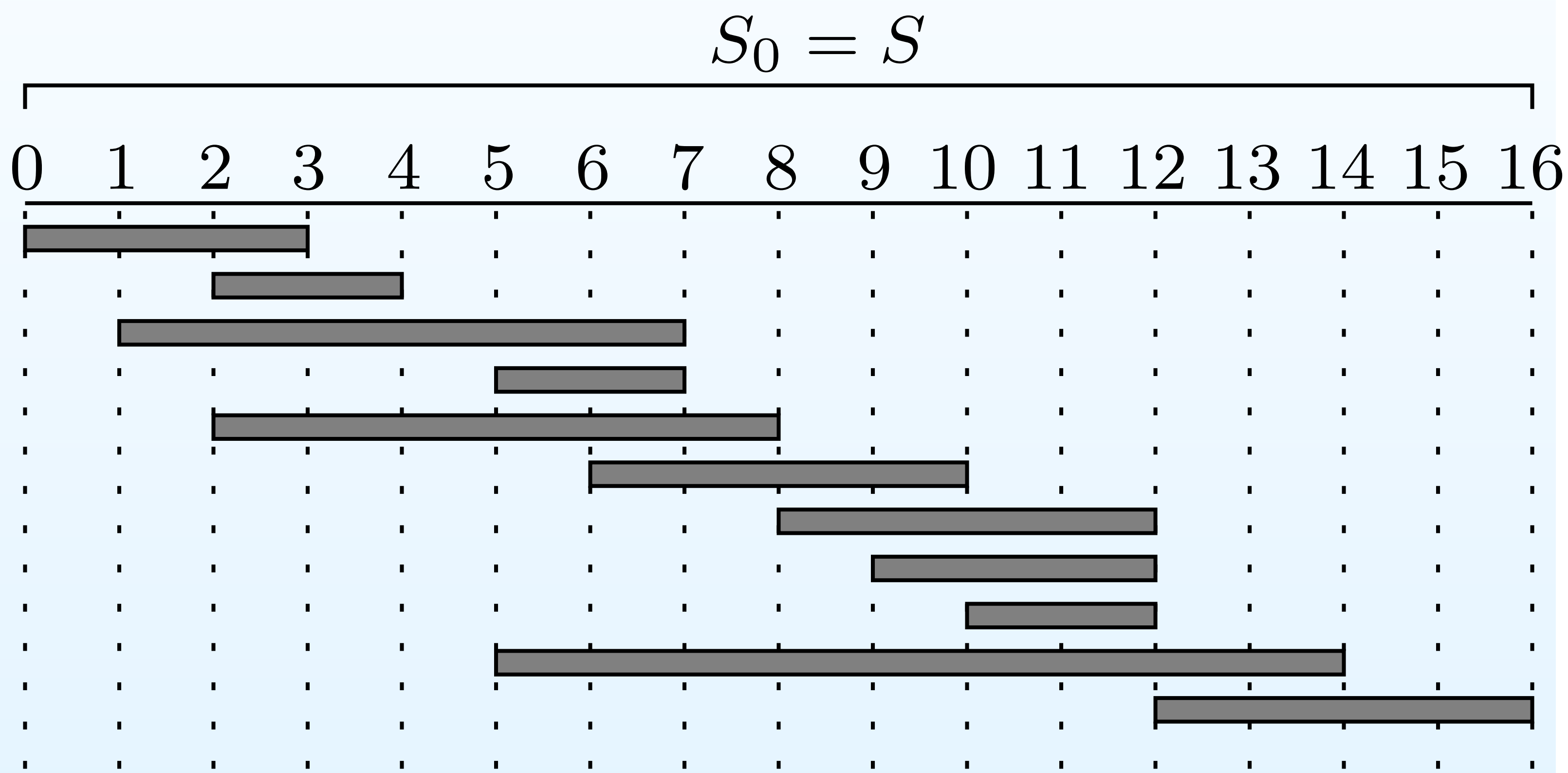
- Solvable in $\Theta(n^3)$ time with dynamic programming
- We will obtain a $\Theta(n)$ bound with a greedy approach (excluding time to sort finish times)

The greedy choice

- For a given subproblem, we would like to make a choice before solving any sub-subproblems
- For $1 \leq k \leq n$, let $S_k = \{a_i \in S | s_i \geq f_k\}$
- Also, let $S_0 = S$
- To solve S_0, we make a greedy choice, i.e., a choice that looks best at the moment
- We greedily choose a_1 as part of our optimal solution (the activity with the earliest finish time)
- Intuitively, the greedy choice a_1 leaves most space for choosing additional activities
- Then we recursively (or iteratively) solve S_1

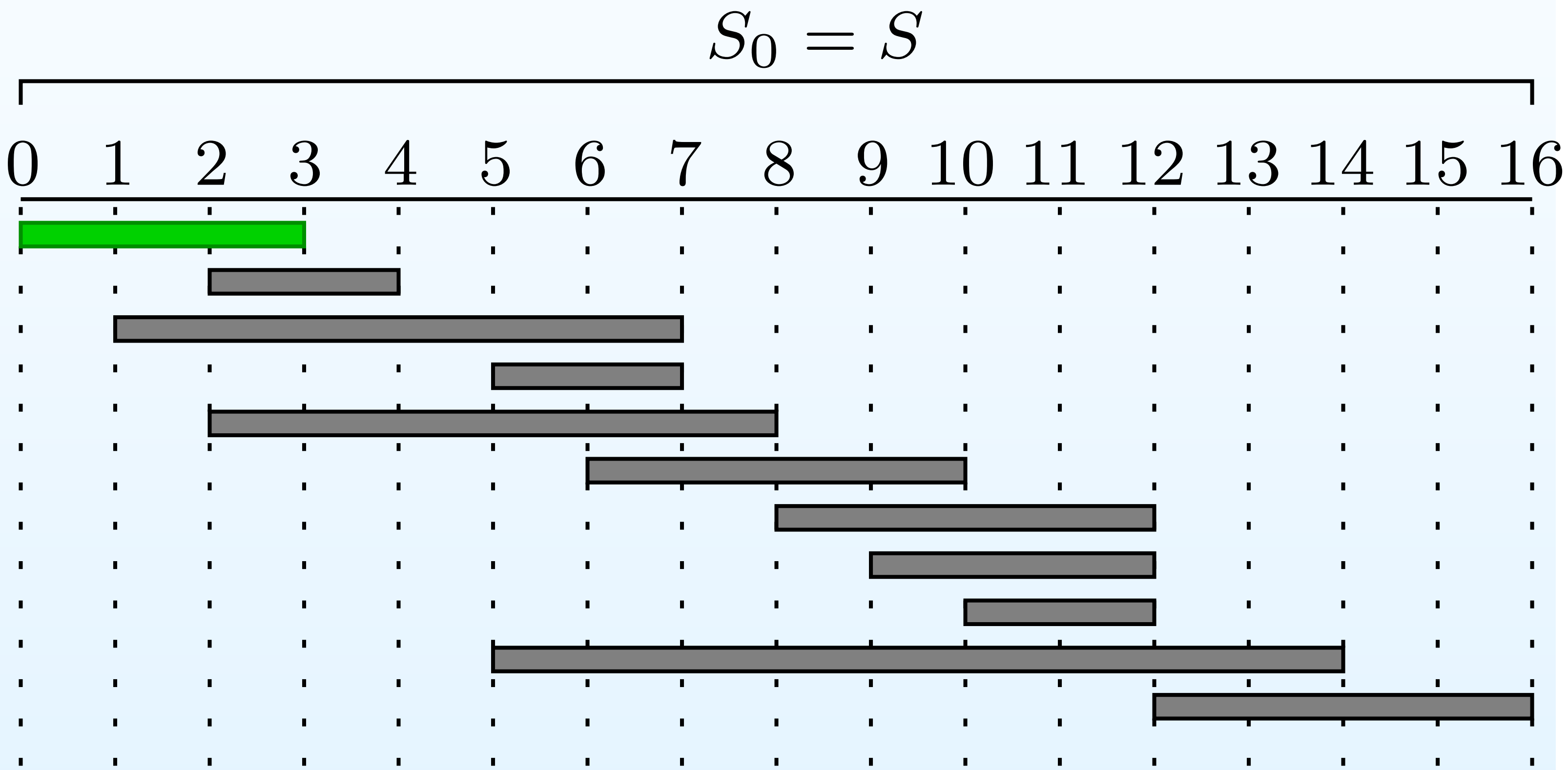
The greedy choice

- Example:



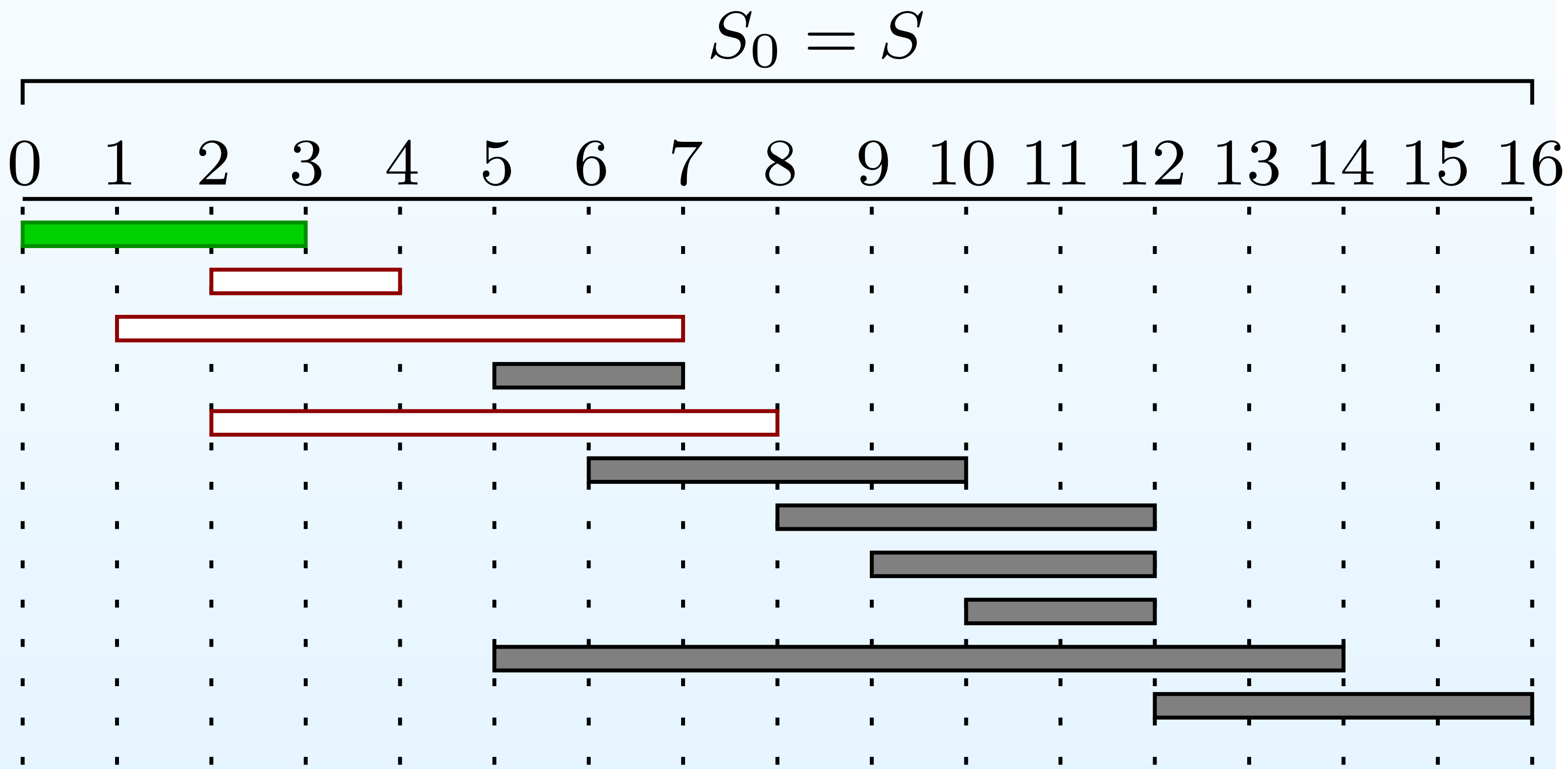
The greedy choice

- Example:



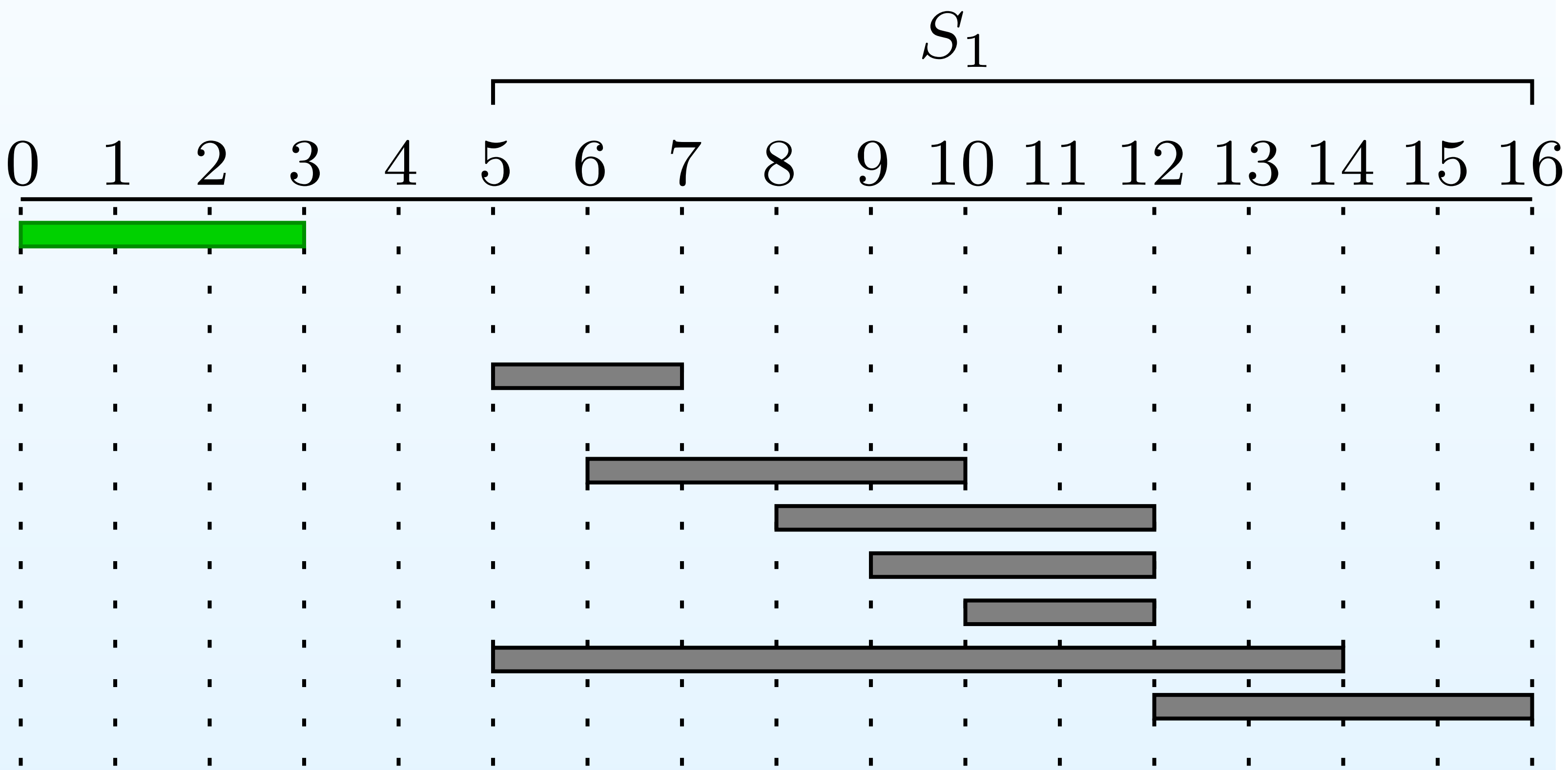
The greedy choice

- Example:



The greedy choice

- Example:



The greedy choice property for activity selection

- Consider the greedy choice a_1 for problem S
- a_1 has earliest finishing time in S
- We show the greedy choice property by showing that a_1 belongs to an optimal solution to S:
    - Consider an optimal solution not containing a_1
    - Let a_j be the activity in this optimal solution with minimum finishing time
    - Then a_j can be replaced by a_1:

The greedy choice property for activity selection

- Consider the greedy choice a_1 for problem S
- a_1 has earliest finishing time in S
- We show the greedy choice property by showing that a_1 belongs to an optimal solution to S:
  - Consider an optimal solution not containing a_1
  - Let a_j be the activity in this optimal solution with minimum finishing time
  - Then a_j can be replaced by a_1:

The greedy choice property for activity selection

- Consider the greedy choice a_1 for problem S
- a_1 has earliest finishing time in S
- We show the greedy choice property by showing that a_1 belongs to an optimal solution to S:
  - Consider an optimal solution not containing a_1
  - Let a_j be the activity in this optimal solution with minimum finishing time
  - Then a_j can be replaced by a_1:

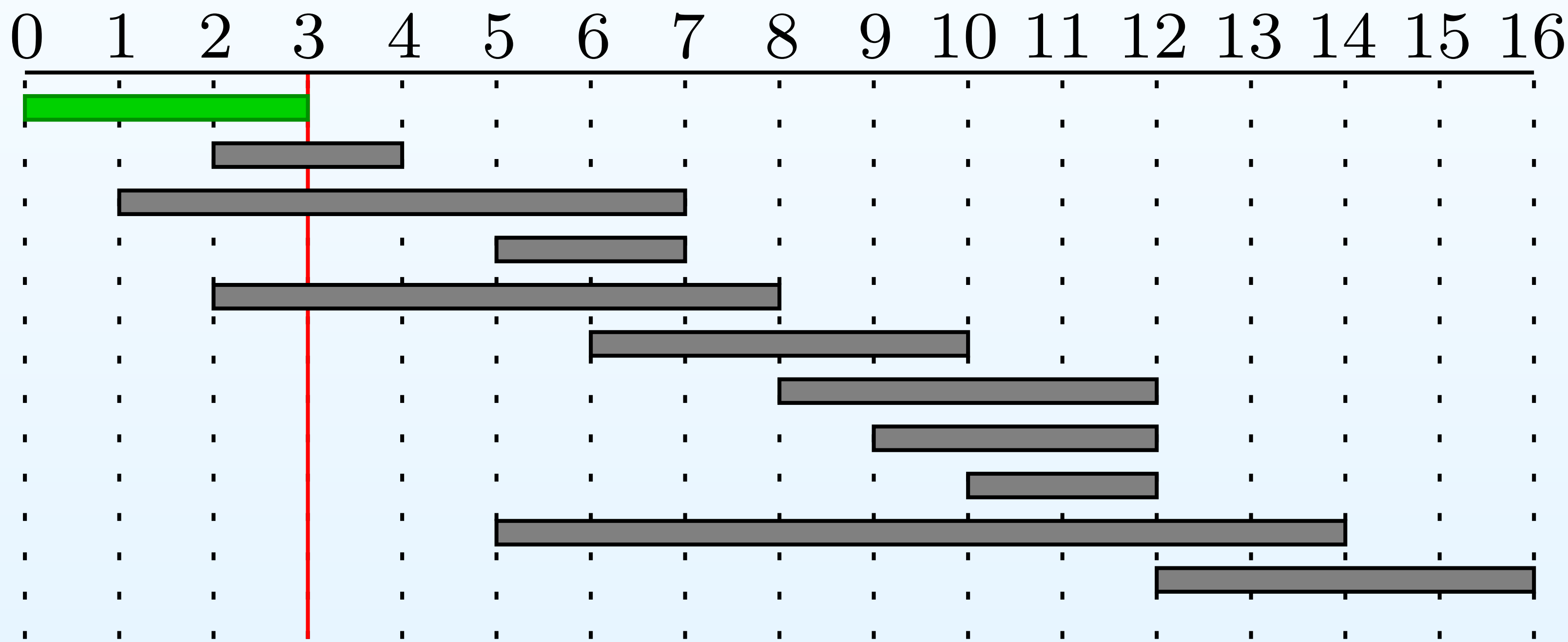
- Hence we have the greedy choice property

Optimal substructure for activity selection

- We have already shown the dynamic programming formulation of optimal substructure:
  - If a_k is in an optimal solution to S_{ij} then this optimal solution must consist of a_k and optimal solutions to subproblems S_{ik} and S_{kj}
- To show the greedy algorithm formulation of optimal substructure, we need:
  - If greedy choice a_1 is in an optimal solution to $S_0 = S$ then this optimal solution consists of a_1 and an optimal solution to S_1
  - Follows from the above with a_1 instead of a_k and S instead of S_{ij} since a_1 leaves only one non-empty subproblem (S_1)

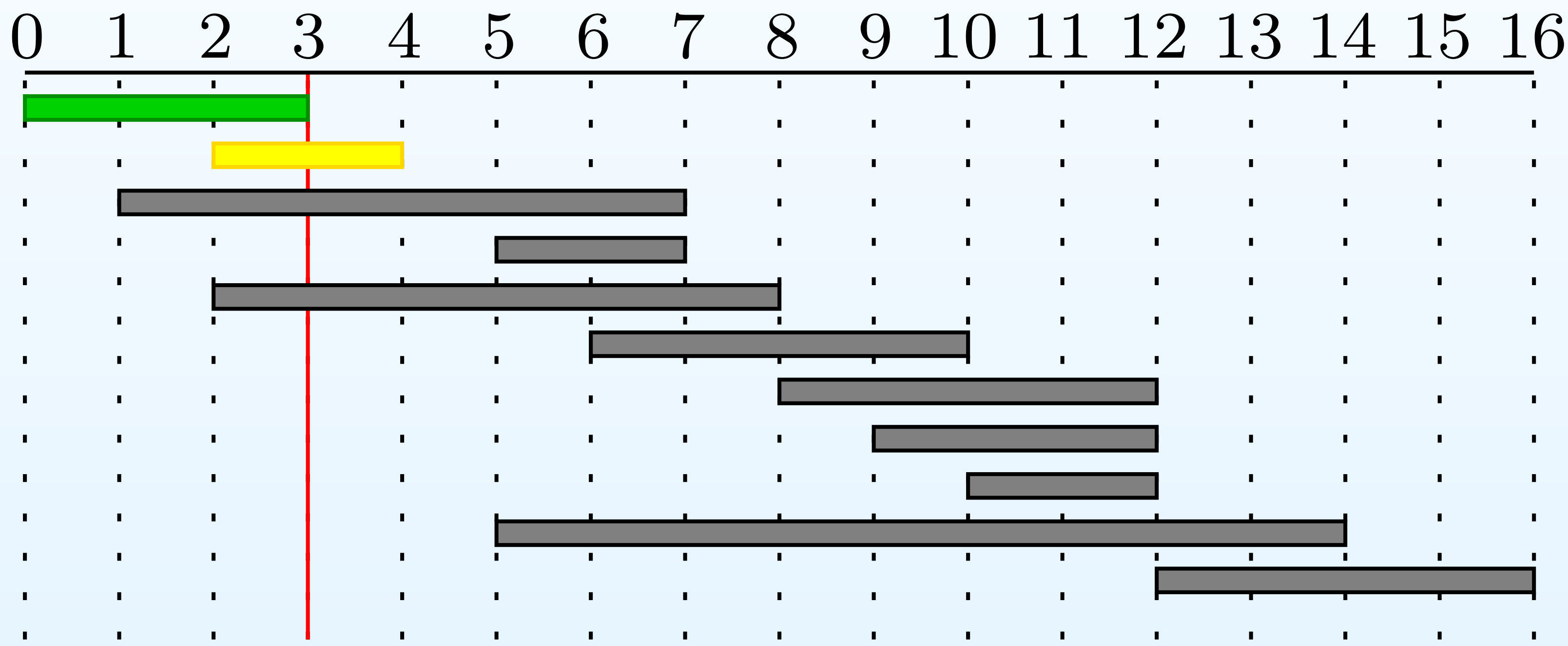
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



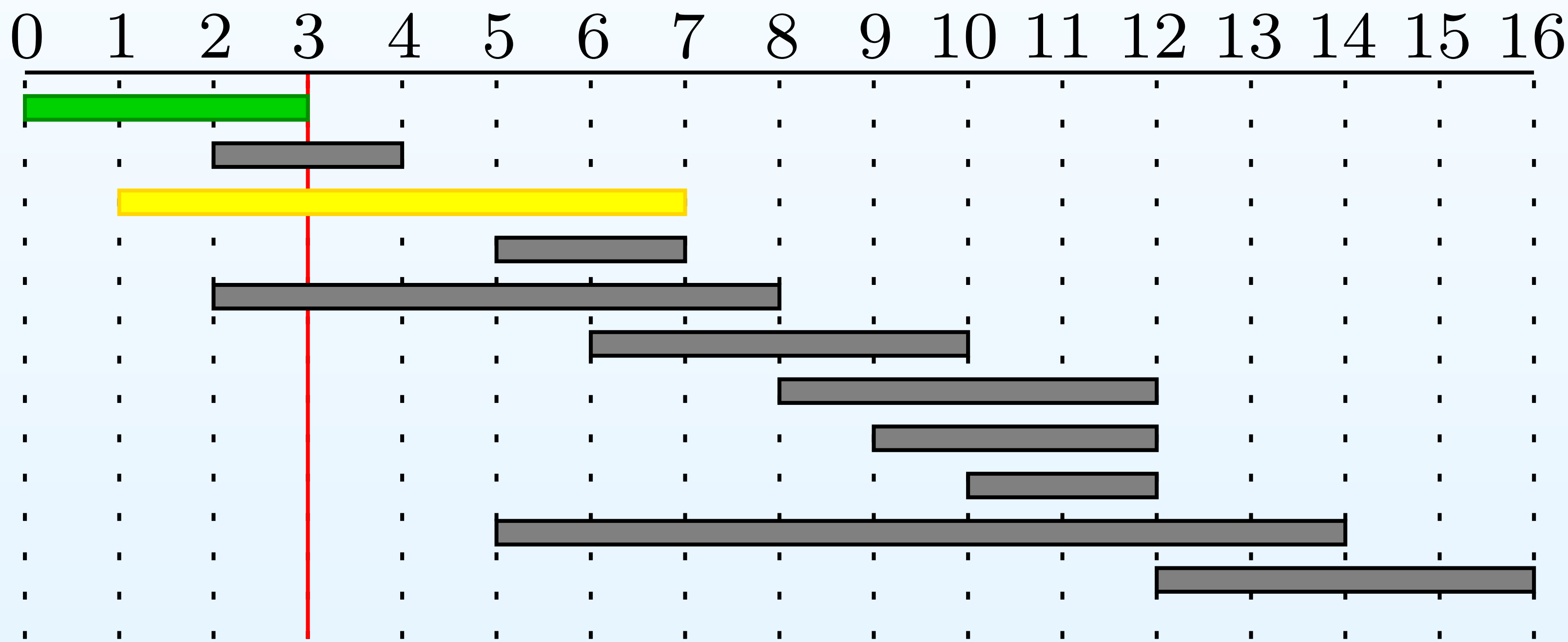
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



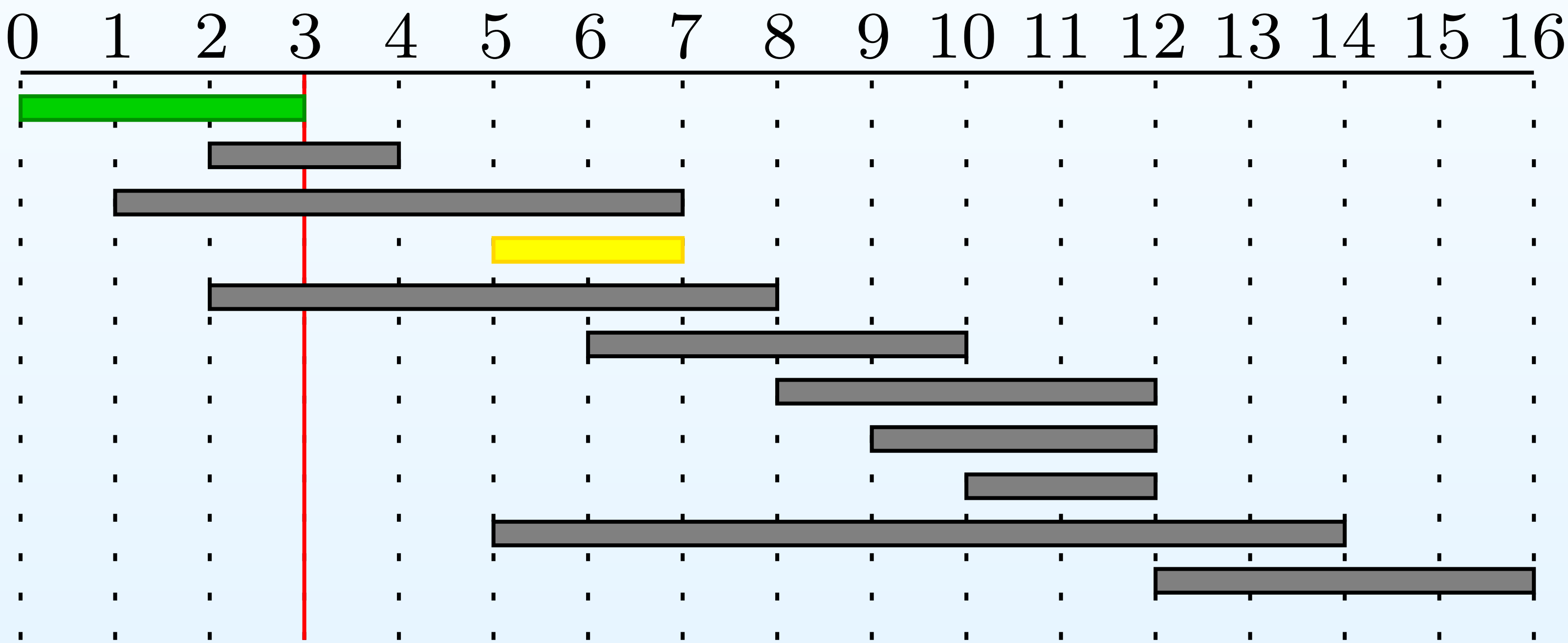
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



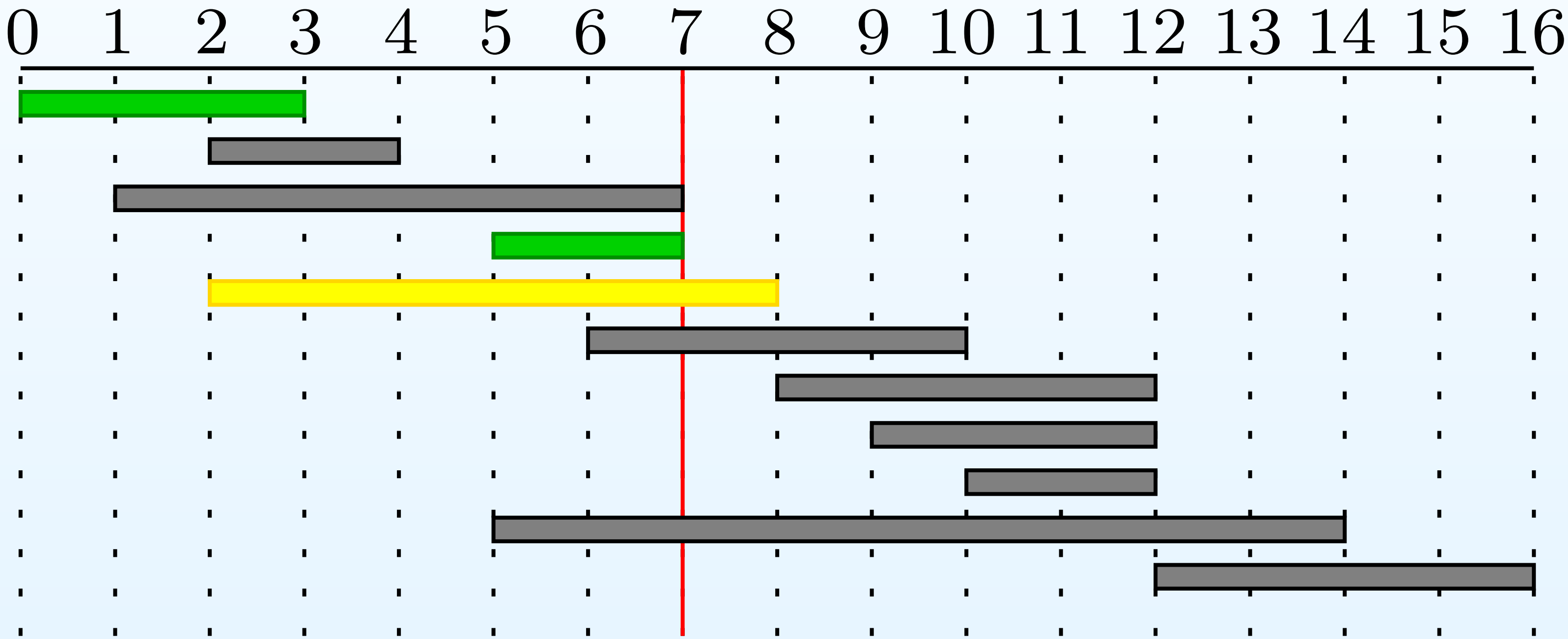
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



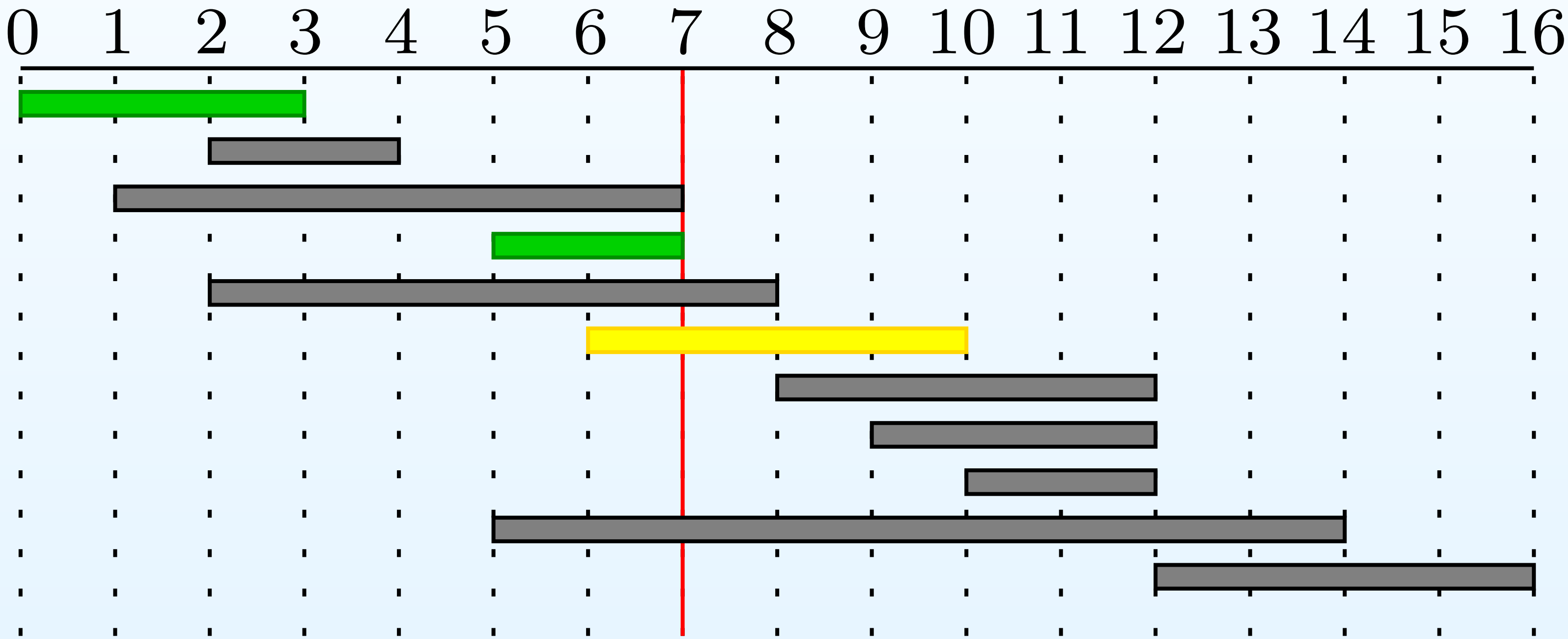
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



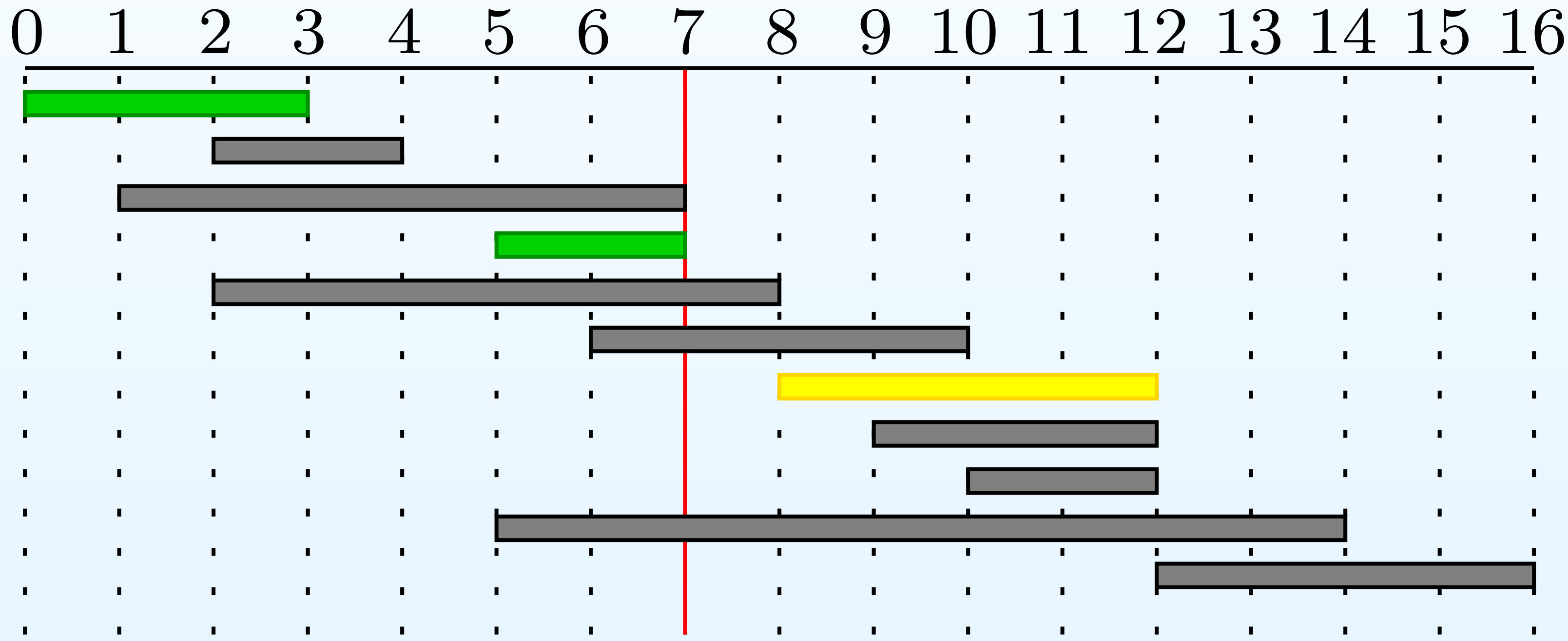
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



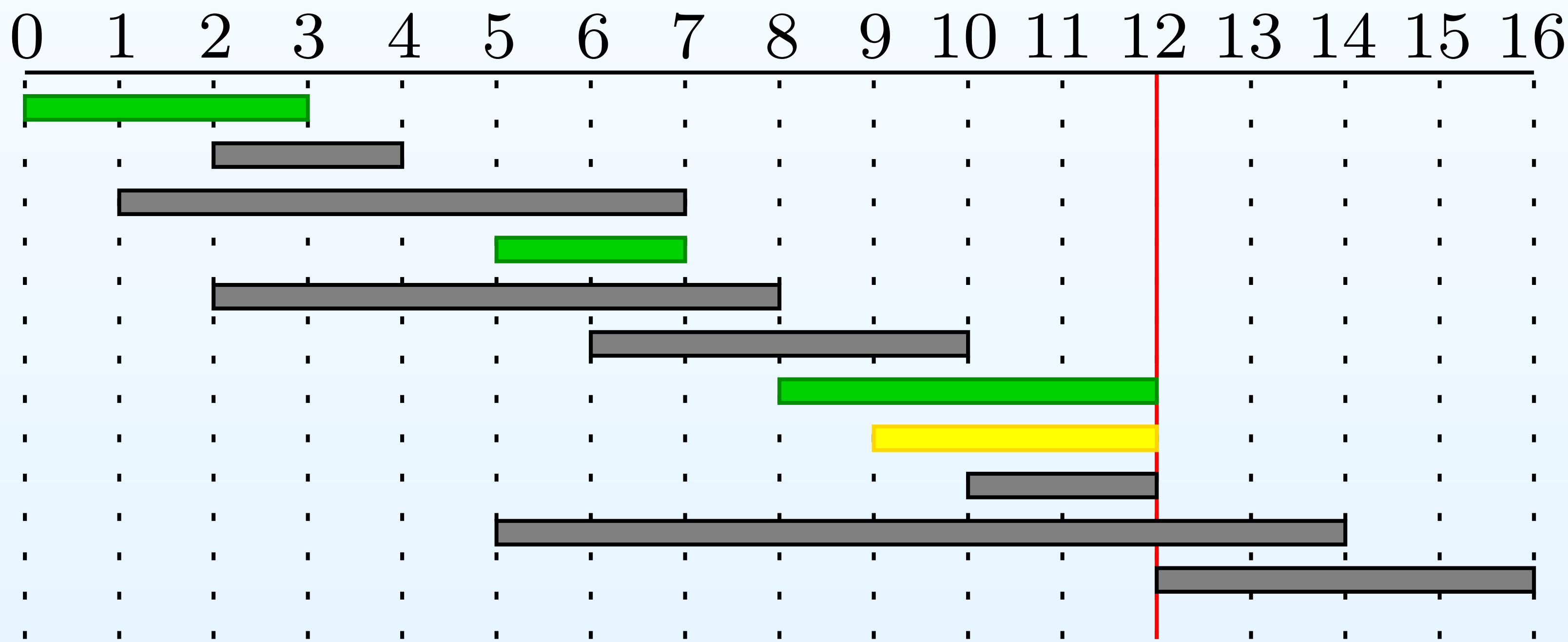
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



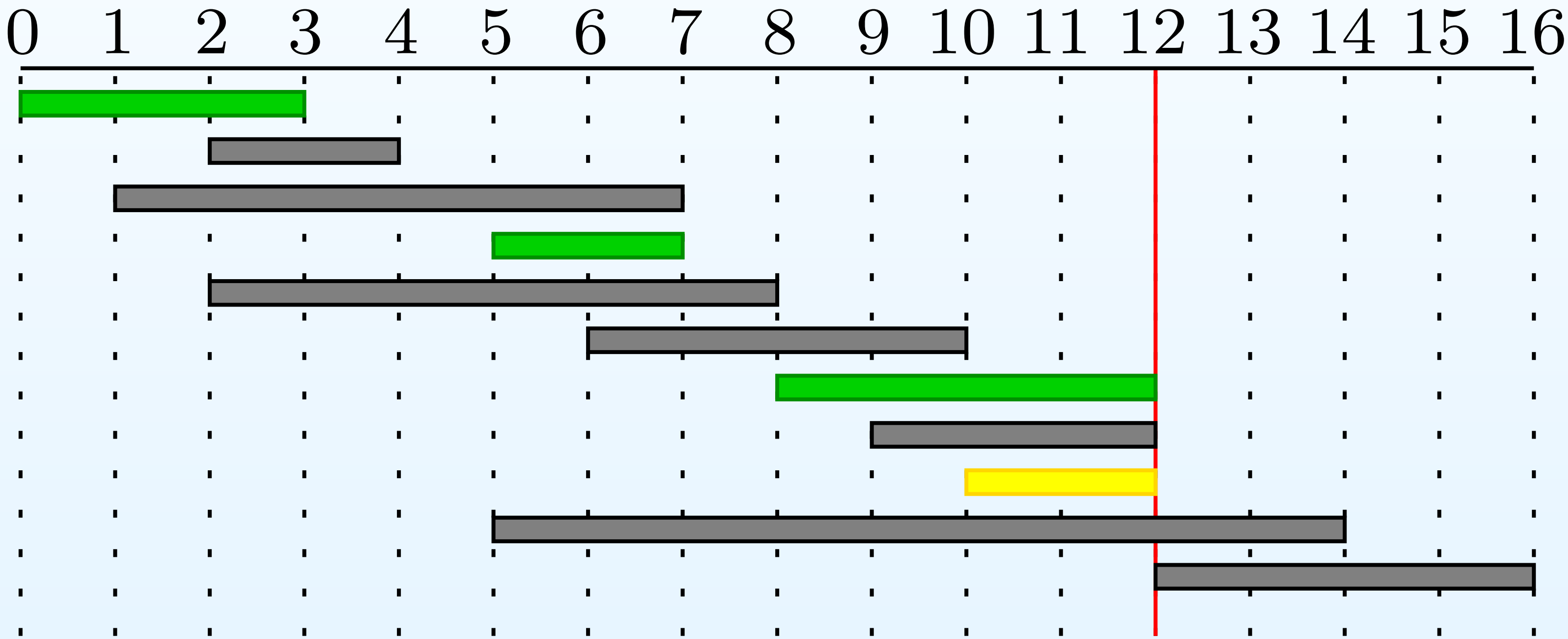
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



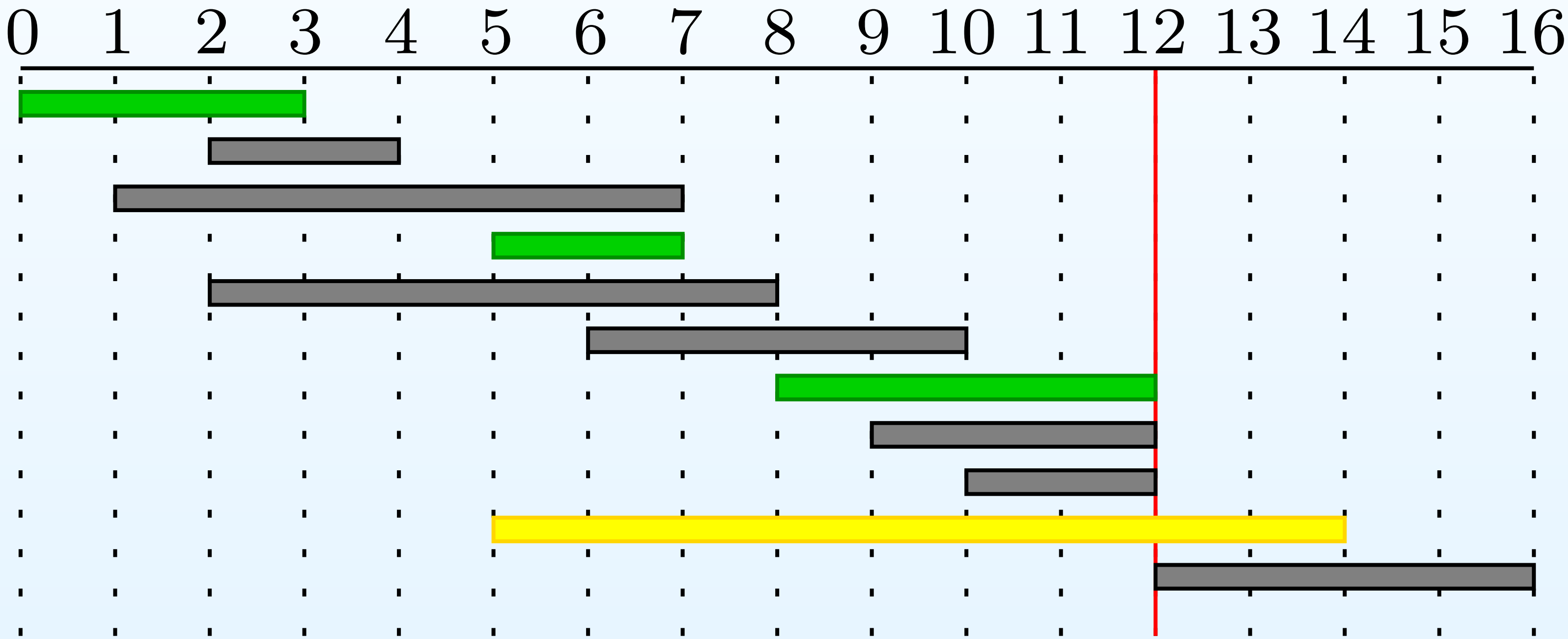
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



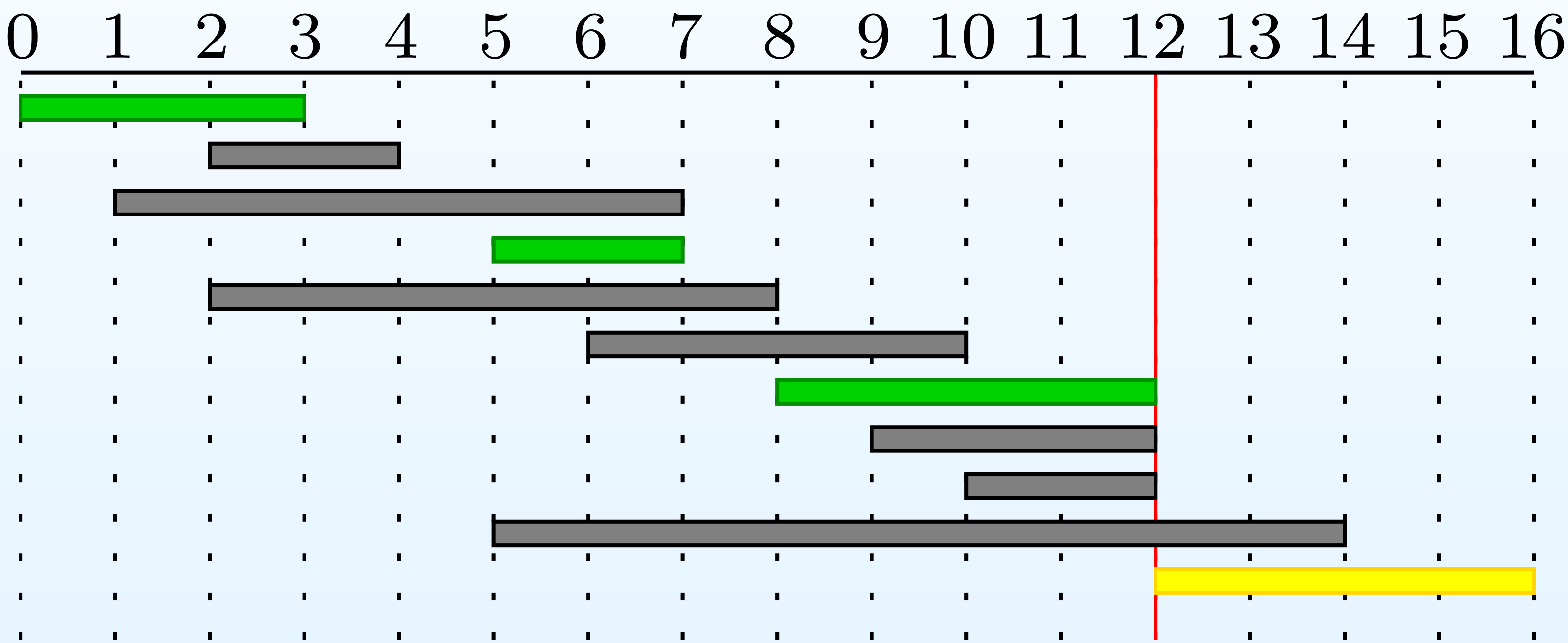
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



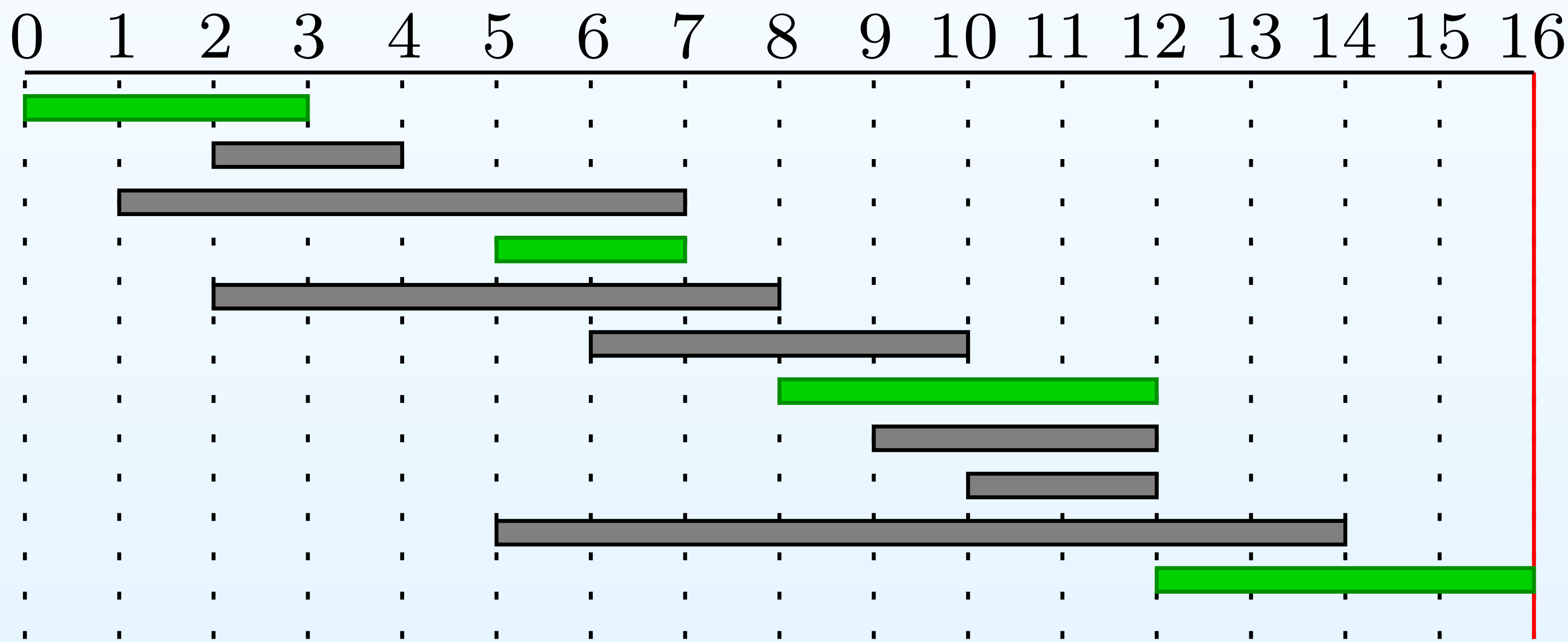
A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



A linear-time greedy algorithm

- The iterative $\Theta(n)$ time algorithm `GREEDY-ACTIVITY-SELECTOR`:



Data compression

- Suppose we have some text that we want to store on, say, a hard drive
- The text is stored as a binary string
- We can assign binary codewords to each text symbol
- For instance, using ASCII codes, the letter A is given codeword 01000001, B is given codeword 01000010, etc
- With this coding, each letter requires one byte
- In a text, some letters typically appear with higher frequency than other letters
- Can we exploit this to get a more compact representation of the text?

Data compression

- Suppose the text consists only of the six letters: a, b, c, d, e, and f
- Then three bits suffice as encoding
- Example:

Letters:	a	b	c	d	e	f
Codewords:	000	001	010	011	100	101

- The word 'badeabe' is encoded as the binary string 001000011100000001100
- If letters occur with different frequencies in the text, can we then do better?

Data compression

- Example:

Letters:	a	b	c	d	e	f
Frequency:	45	13	12	16	9	5
3 bits:	000	001	010	011	100	101
Variable length:	0	00	01	1	11	100

- Length of text with 3 bit codes (freq in 1000s):
  $(45 + 13 + 12 + 16 + 9 + 5) \cdot 1000 \cdot 3 = 300000$ bits
- Length of text with variable length codes:
  $(45 \cdot 1 + 13 \cdot 2 + 12 \cdot 2 + 16 \cdot 1 + 9 \cdot 2 + 5 \cdot 3) \cdot 1000 = 144000$ bits
- What is the problem here?
- What text does the following represent: 00?

Data compression: prefix codes

- To deal with this problem, we use *prefix codes* instead:

Letters:	a	b	c	d	e	f
Frequency:	45	13	12	16	9	5
3 bits:	000	001	010	011	100	101
Variable length:	0	101	100	111	1101	1100

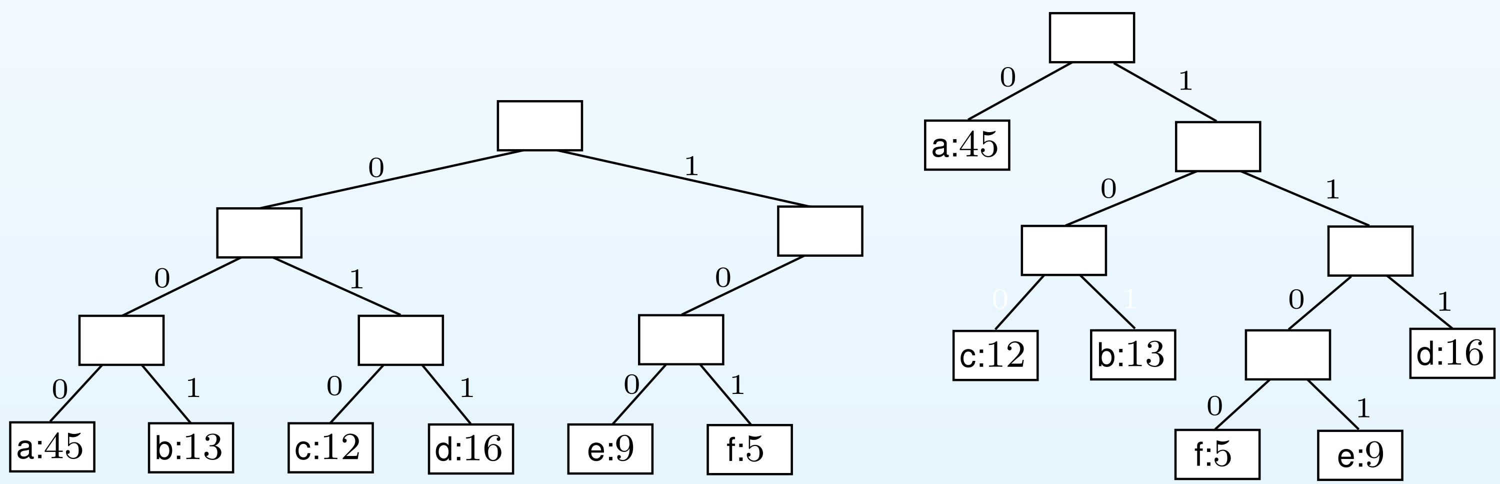
- Prefix code: no codeword is a prefix of any other codeword
- Length of text with 3 bit codes:
  $(45 + 13 + 12 + 16 + 9 + 5) \cdot 1000 \cdot 3 = 300000$ bits
- Length of text with variable length prefix codes:
  $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000$ bits

Parse trees for prefix codes

- Example:

Letters:	a	b	c	d	e	f
Frequency:	45	13	12	16	9	5
3 bits:	000	001	010	011	100	101
Variable length:	0	101	100	111	1101	1100

- Corresponding parse trees:



Cost of a parse tree

- Consider a parse tree T for some prefix code
- Let C be the set of characters (example: $C = \{a, b, c, d, e, f\}$)
- For each character $c \in C$, let f_c denote its frequency and let $d_T(c)$ denote the depth of its leaf in T (in CLRS, c.freq is used instead of f_c)
- Then we define the *cost* $B(T)$ of T to be:

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

- What does $B(T)$ indicate?
- We now present Huffman's algorithm
- It is a greedy algorithm which finds a tree T with minimum value $B(T)$

Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:

a:45

d:16

b:13

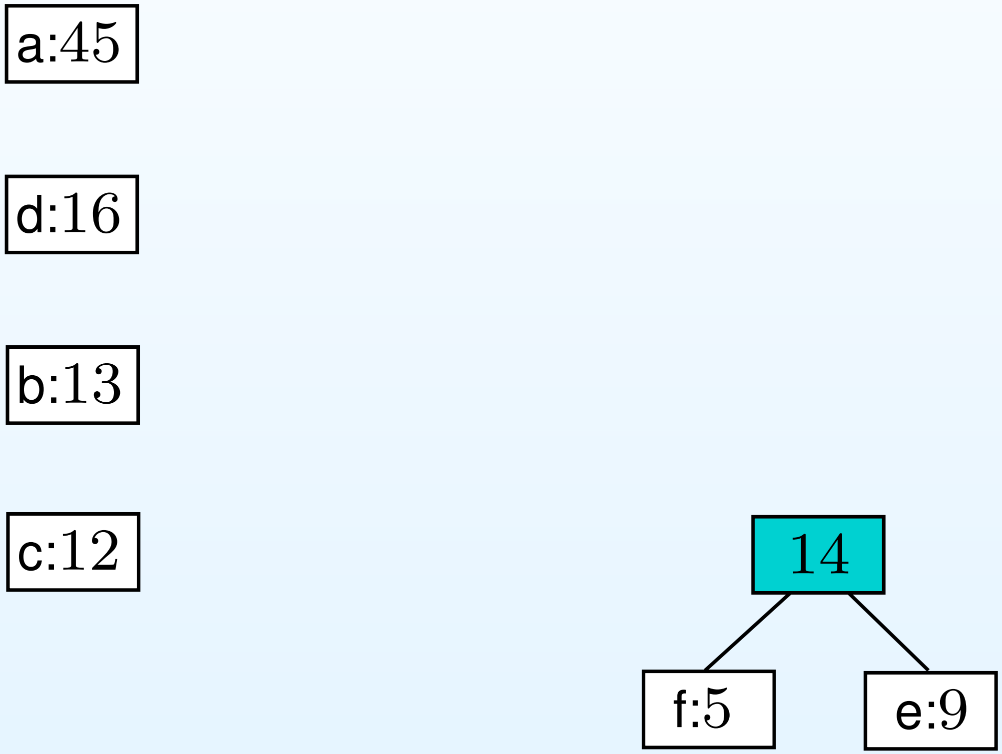
c:12

e:9

f:5

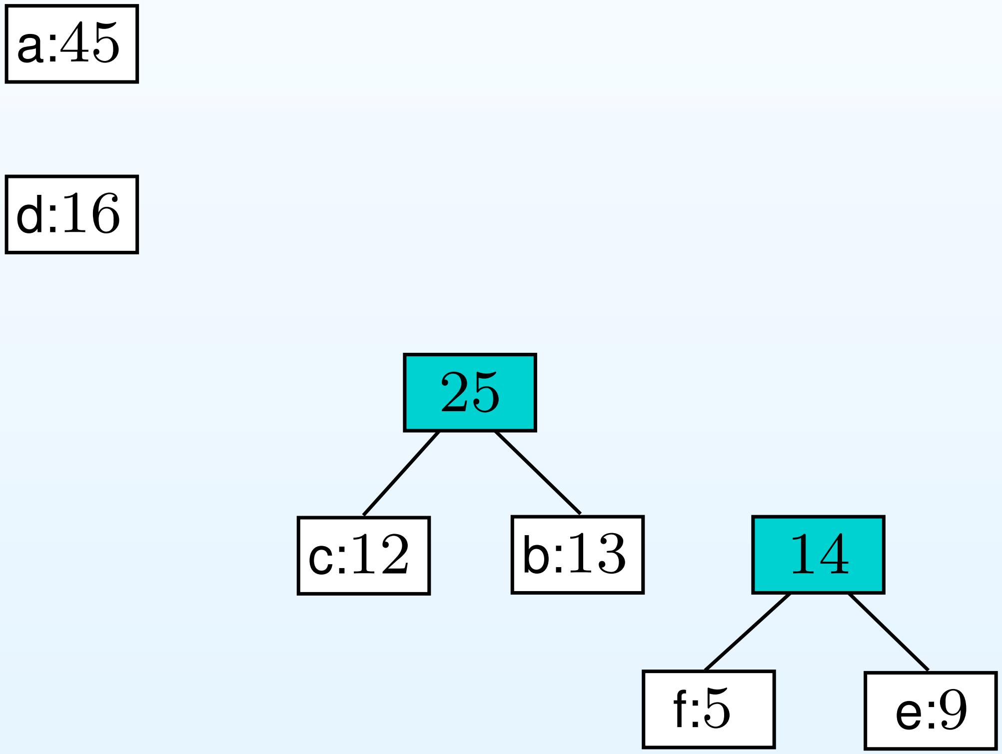
Huffman’s algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



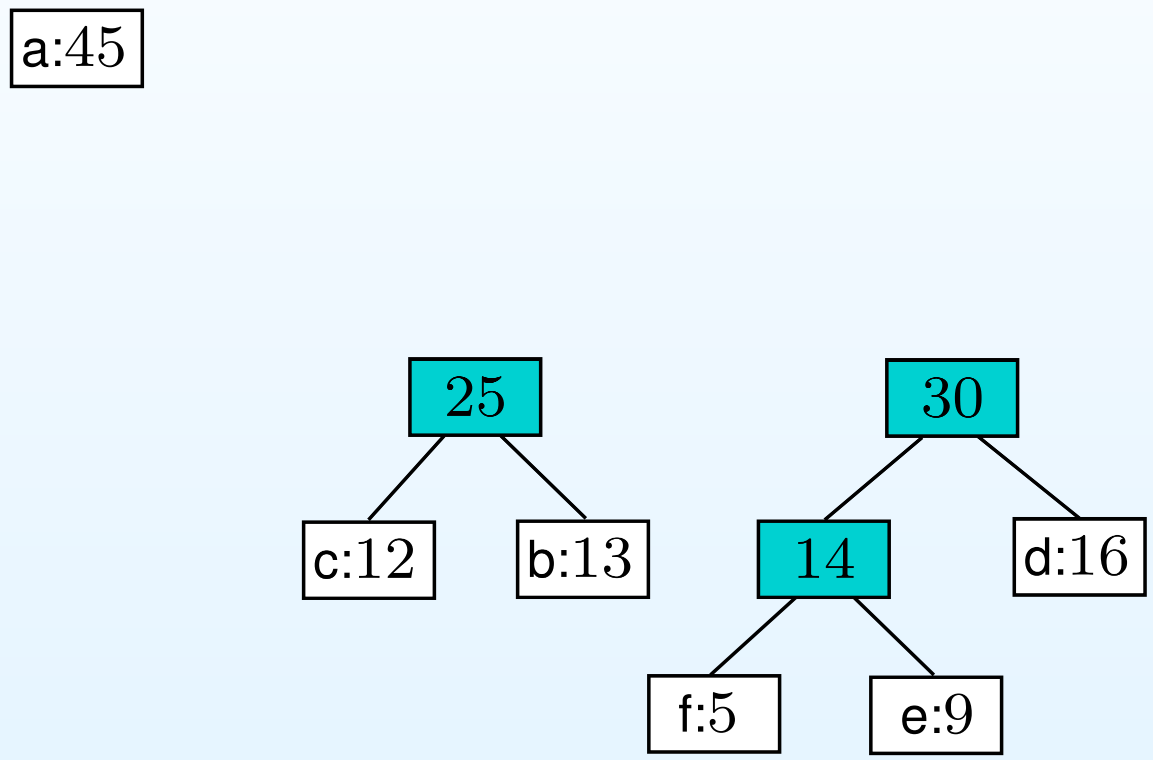
Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



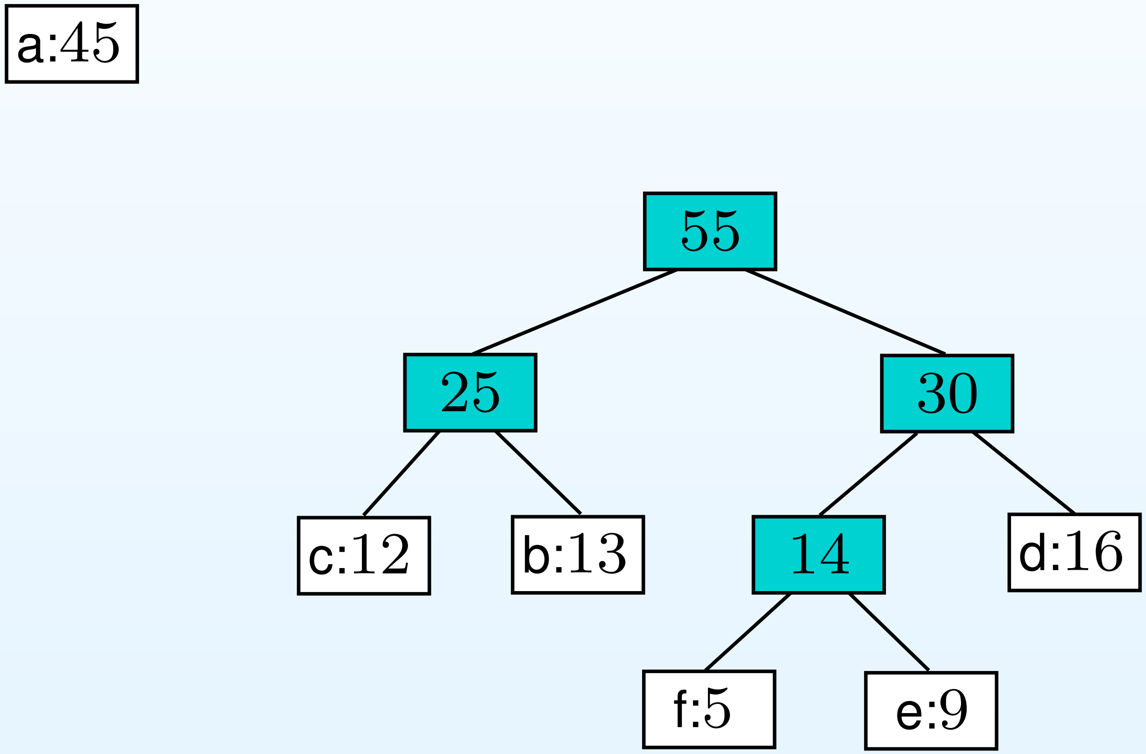
Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



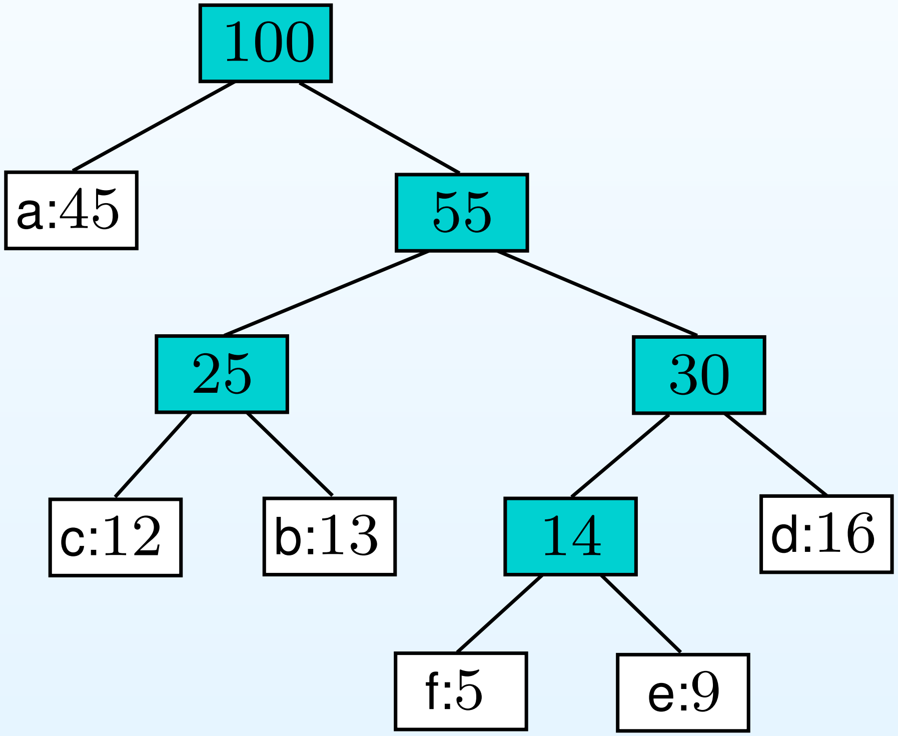
Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



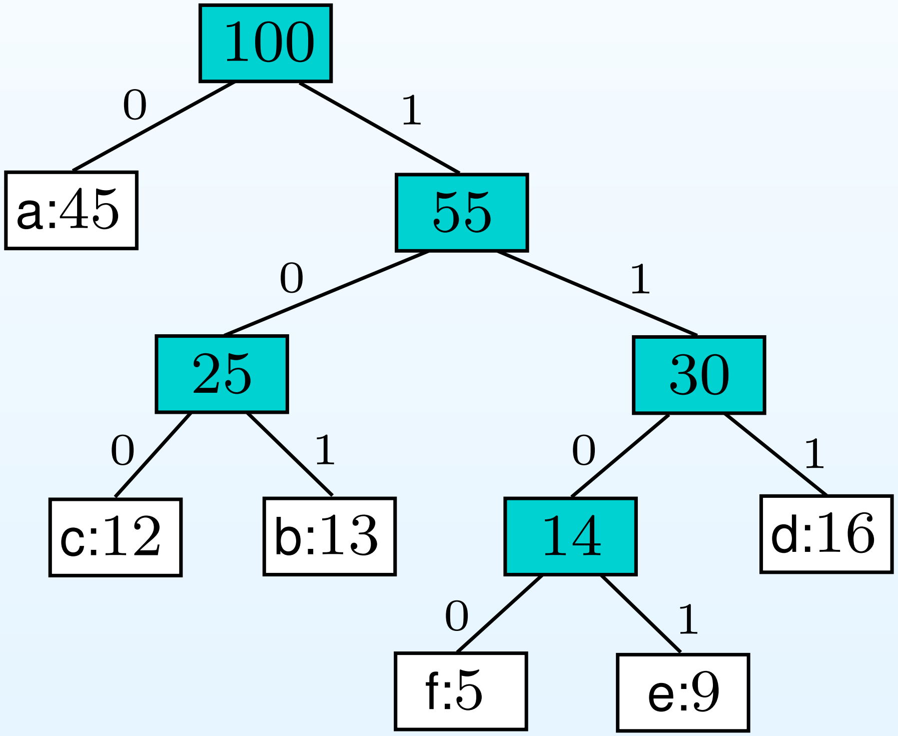
Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



Huffman's algorithm

- Keep pairing up nodes with minimum freq and set freq of parent node to the sum of freqs of its children:



Running time of Huffman's algorithm

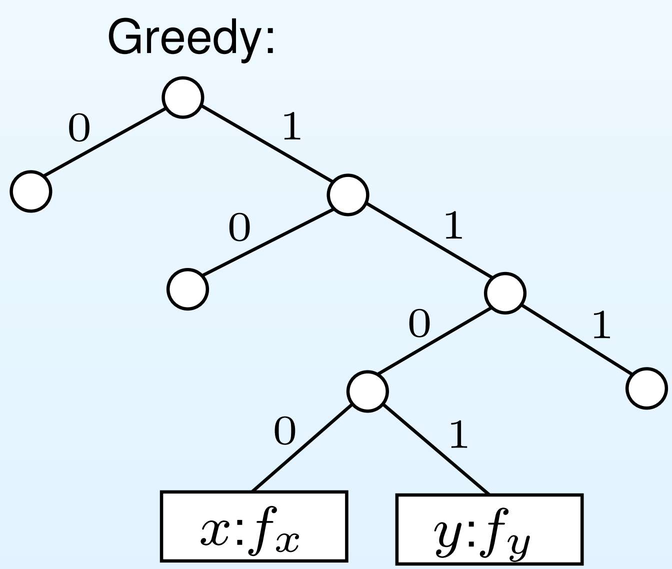
- We keep characters in a min-heap H where keys are the frequencies
- H can be constructed in $\Theta(n)$ time
- It allows us to extract characters of minimum frequency and to add new elements to H in $O(\log n)$ time per operation
- In total, there are $\Theta(n)$ operations (why?)
- Hence, Huffman's algorithm runs in $O(n \log n)$ time

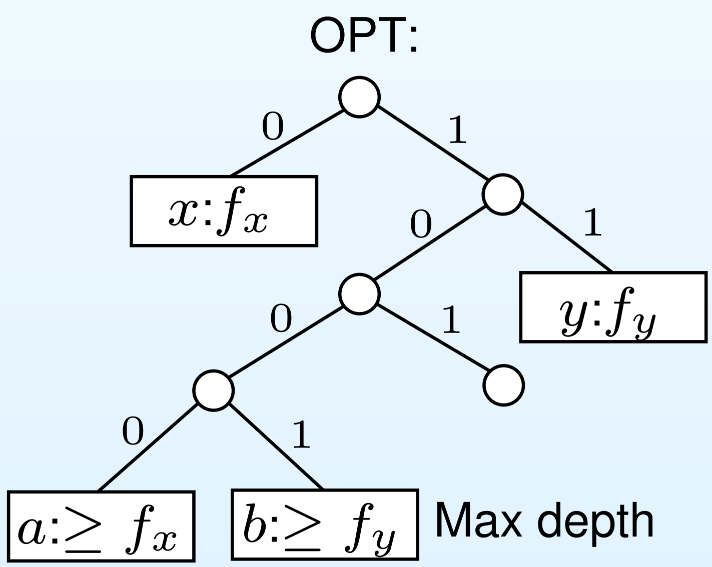
Huffman codes

- The tree output by Huffman's algorithm is a parse tree T
- The corresponding prefix code is called a *Huffman code*
- We will show that $B(T)$ is minimal
- In other words, we will show that a Huffman code is an optimal prefix code

Greedy choice property

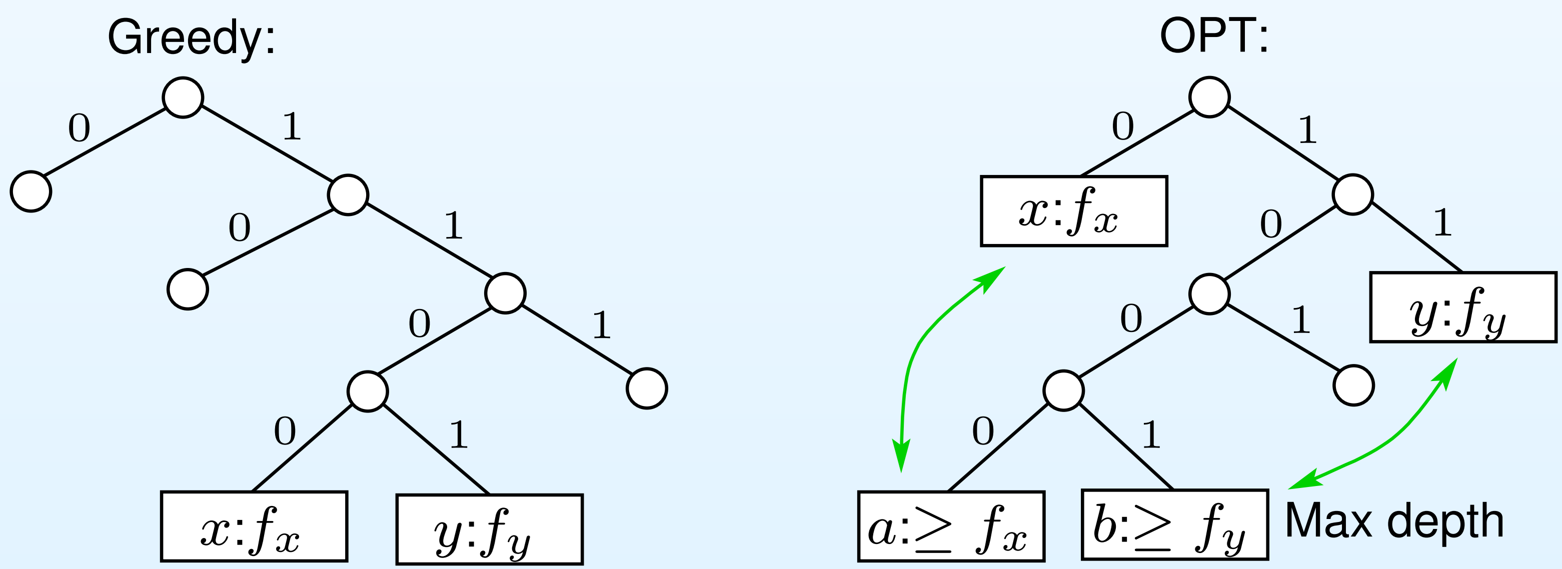
- Suppose Huffman's algorithm starts by pairing up characters x and y, $f_x \leq f_y$ (the first greedy choice)
- Consider an optimal tree OPT
- The greedy choice property follows from the fact that OPT can be transformed into another optimal tree OPT' containing the greedy choice (x and y are sibling leaves):





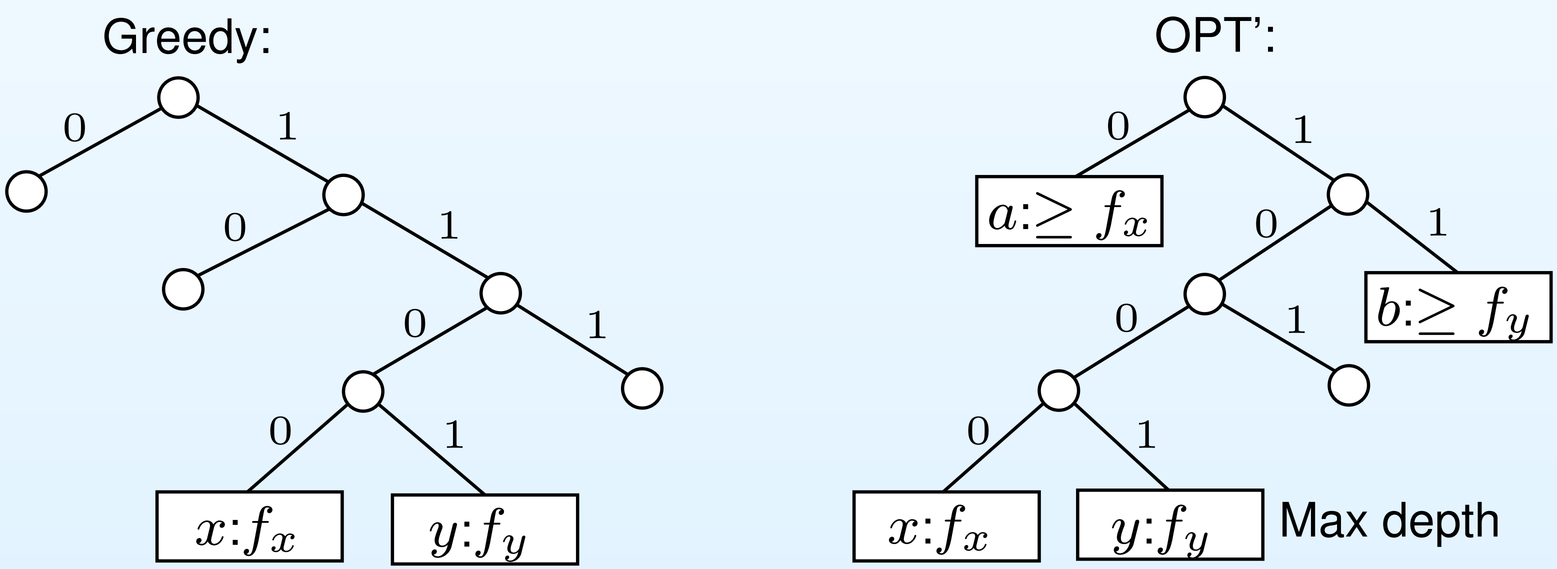
Greedy choice property

- Suppose Huffman's algorithm starts by pairing up characters x and y, $f_x \leq f_y$ (the first greedy choice)
- Consider an optimal tree OPT
- The greedy choice property follows from the fact that OPT can be transformed into another optimal tree OPT' containing the greedy choice (x and y are sibling leaves):



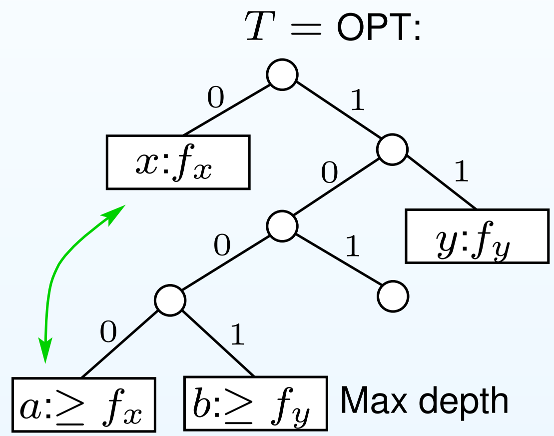
Greedy choice property

- Suppose Huffman's algorithm starts by pairing up characters x and y, $f_x \leq f_y$ (the first greedy choice)
- Consider an optimal tree OPT
- The greedy choice property follows from the fact that OPT can be transformed into another optimal tree OPT' containing the greedy choice (x and y are sibling leaves):



Proving the Greedy choice property for Huffman

- Let $T = \mathrm{OPT}$ and let T' be OPT after a and x have been swapped:



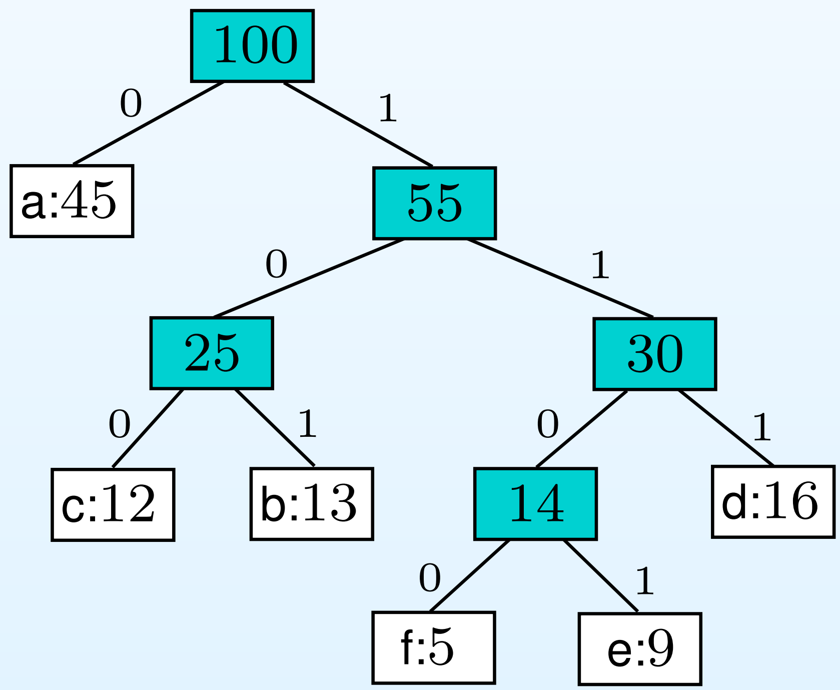
- Need to show $B(T') \leq B(T)$, i.e., that $B(T') - B(T) \leq 0$:

$$
\begin{aligned}
&B(T') - B(T) \\
&\quad = f_x \cdot d_{T'}(x) + f_a \cdot d_{T'}(a) - (f_x \cdot d_T(x) + f_a \cdot d_T(a)) \\
&\quad = f_x \cdot d_T(a) + f_a \cdot d_T(x) - (f_x \cdot d_T(x) + f_a \cdot d_T(a)) \\
&\quad = \underbrace{(f_x - f_a)}_{\leq 0} \underbrace{(d_T(a) - d_T(x))}_{\geq 0} \leq 0
\end{aligned}
$$

A lemma (essentially exercise 16.3-4)

- Letting W be the set of non-root nodes of any parse tree T,

$$B(T) \stackrel{\text{def}}{=} \sum_{c \in C} f_c \cdot d_T(c) = \sum_{v \in W} f_v$$



What is the subproblem after making the greedy choice?

- The first greedy choice in Huffman's algorithm reduces the problem to a smaller one
- We can regard the reduction as replacing x and y by a new character z with $f_z = f_x + f_y$
- Alphabet for reduced problem: $C' = (C \setminus \{x, y\}) \cup \{z\}$

$\boxed{\text{a:}45}$

$\boxed{\text{d:}16}$

$\boxed{\text{b:}13}$

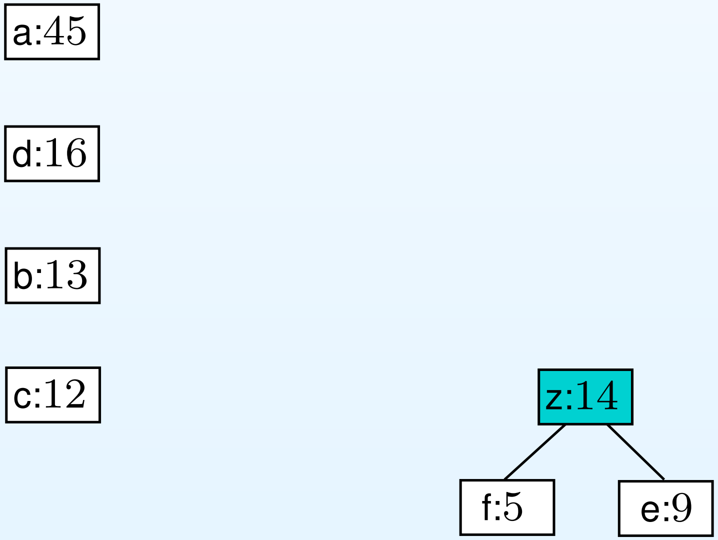
$\boxed{\text{c:}12}$

$\boxed{\text{e:}9}$

$\boxed{\text{f:}5}$

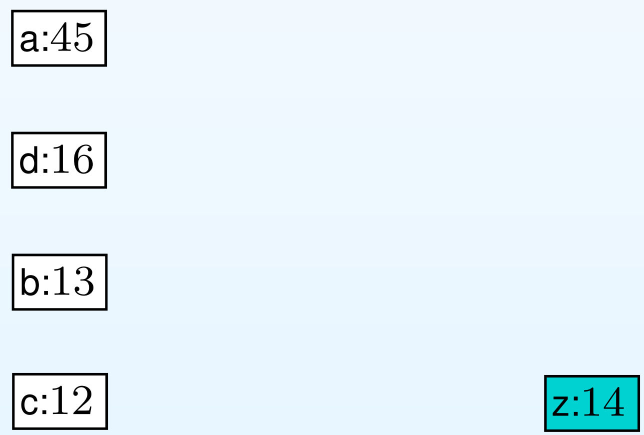
What is the subproblem after making the greedy choice?

- The first greedy choice in Huffman's algorithm reduces the problem to a smaller one
- We can regard the reduction as replacing x and y by a new character z with $f_z = f_x + f_y$
- Alphabet for reduced problem: $C' = (C \setminus \{x, y\}) \cup \{z\}$



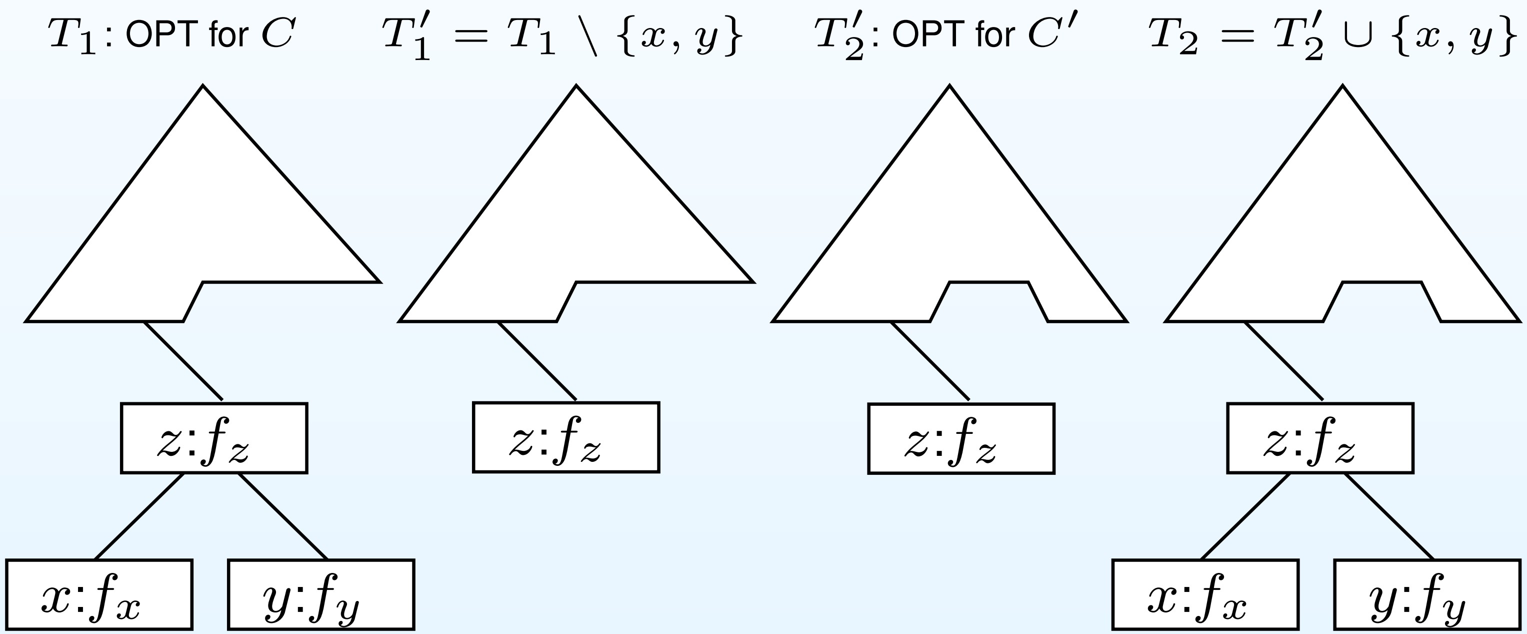
What is the subproblem after making the greedy choice?

- The first greedy choice in Huffman's algorithm reduces the problem to a smaller one
- We can regard the reduction as replacing x and y by a new character z with $f_z = f_x + f_y$
- Alphabet for reduced problem: $C' = (C \setminus \{x, y\}) \cup \{z\}$



Optimal substructure property (see notes on Absalon)

- We need to show that if the greedy choice is in an optimal tree T_1 for C then $T_1' = T_1 \setminus \{x, y\}$ is an optimal tree for C':



- $B(T_1') \overset{\text{lemma}}{=} B(T_1) - f_x - f_y \leq B(T_2) - f_x - f_y \overset{\text{lemma}}{=} B(T_2')$
- Since also $B(T_1') \geq B(T_2')$, we have $B(T_1') = B(T_2')$

Plan for the lecture on February 22

- Amortized Analysis
  - Aggregate analysis
  - The accounting method
  - The potential method
  - Dynamic tables