CP312
Algorithm Design and Analysis I

LECTURE 10: GREEDY ALGORITHMS

Optimization Problems

- Unlike some of the sorting problems that we have seen so far, **optimization problems** are those where we have to find the **best** solution out of **several possible solutions**.
- How "**best**" is defined is determined by the **problem statement**.

Greedy Algorithms

- **A technique for algorithm design**

- Typically used to create solutions for **optimization problems.**
  - Such problems have many possible solutions and we **wish to find an optimal one**.

- Problems need to have **certain properties** in order to be solvable using the "**greedy**" approach.

Greedy Algorithms

- Problems for which we will demonstrate the greedy strategy:
  - Coin-changing
  - Fractional knapsack
  - Task-scheduling
  - Huffman Encoding

Coin Changing Problem

- Suppose that you have a shop that deals with the following currency notes:

  \$1, **\$5**, **\$10**, **\$20**

- You want to design an algorithm such that you can produce a certain amount of money V using the **minimum** number of notes.

- E.g. To get **\$100**, the best (minimum) solution is to use **five** **\$20** notes.

Coin Changing Problem

- **Idea**: always take the next big note that does not exceed V

CoinChange(V)**:**

1. Initialize set S
2. While $(V > 0)$ {

    Find the largest bill b at most V

    Add b to S

    $V = V - b$

   }
3. Return S

Coin Changing Problem

- It is easy to check that the algorithm always returns a set of bills whose sum is V.
- At each step, the algorithm makes a **greedy choice** (by including the *largest coin*) which looks best to come up with an optimal solution (the fewest number of bills)
- This is an example of a **Greedy Algorithm**.

Coin Changing Problem

- Does a greedy algorithm always work?
  - No!

- Suppose the set of bills is:

$1, $4, $5, $10

- We want to produce $8:
  - Greedy Algorithm (always choose largest): 4 bills ($5, $1, $1, $1)
  - Optimal Solution: 2 bills ($4, $4)

Greedy Algorithm

- To show a greedy algorithm **works**, we need to show that the problem has the following properties:

1. **Optimal substructure**: the optimal solution to the problem contains within it optimal solutions to subproblems

2. **Greedy-choice property**: a globally optimal solution can be arrived at by making a locally optimal solution.

Fractional Knapsack Problem

- **Problem:** Given a set of items, find a combination of items to add to a bag of some capacity that yields the most value.

- **Input:** A capacity W and a set of n weight/value pairs

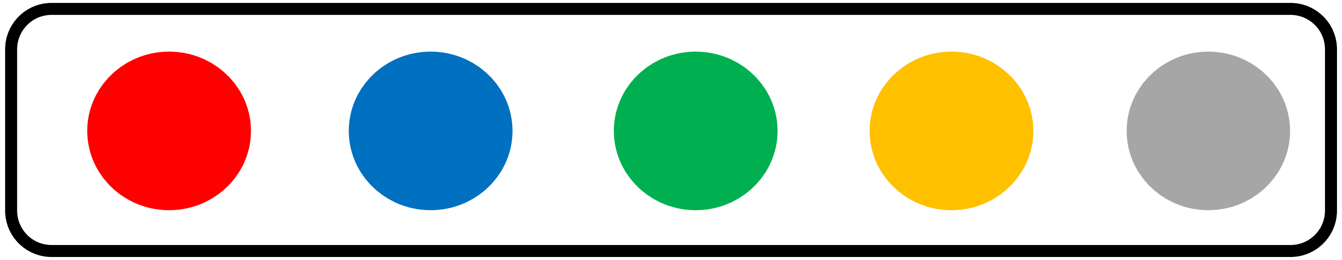
$$I = \{(1, v_1, w_1), (2, v_2, w_2), \dots, (n, v_n, w_n)\}$$

- **Output:** A set of values $P = (p_1, \dots, p_n)$ where $0 \leq p_i \leq 1$ such that:

Maximizes $\sum_{i=1}^{n} p_i v_i$

Subject to $\sum_{i=1}^{n} p_i w_i \leq W$

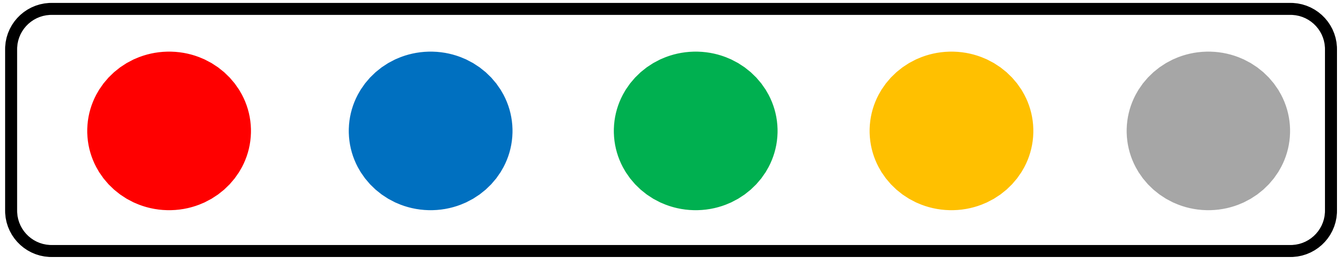
The Fractional Knapsack Problem



Weight (kg)	2	1	3	6	11
Value ($)	10	6	12	12	33

Capacity: 5 kg

The Fractional Knapsack Problem

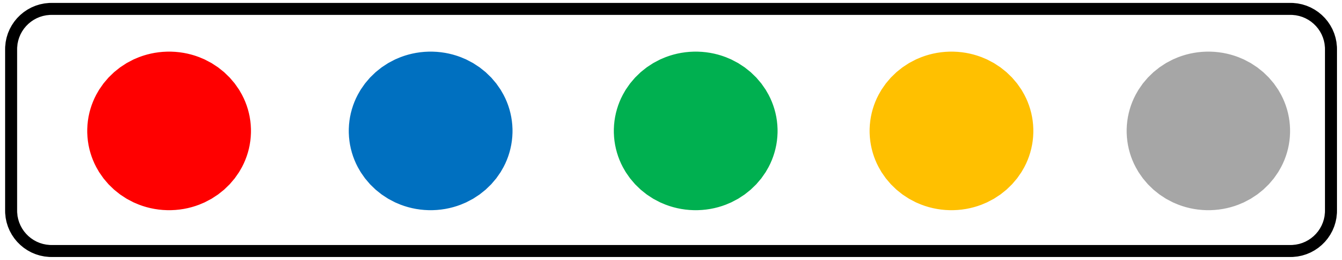


Weight (kg)	2	1	3	6	11
Value ($)	10	6	12	12	33
Density (V/W)	5	6	4	2	3

Greedy Approach: At every step, choose the item that has the highest density.

Capacity: 5 kg

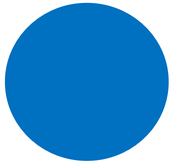
The Fractional Knapsack Problem



Weight (kg)	2	1	3	6	11
Value ($)	10	6	12	12	33
Density (V/W)	5	6	4	2	3

Total Value: $6

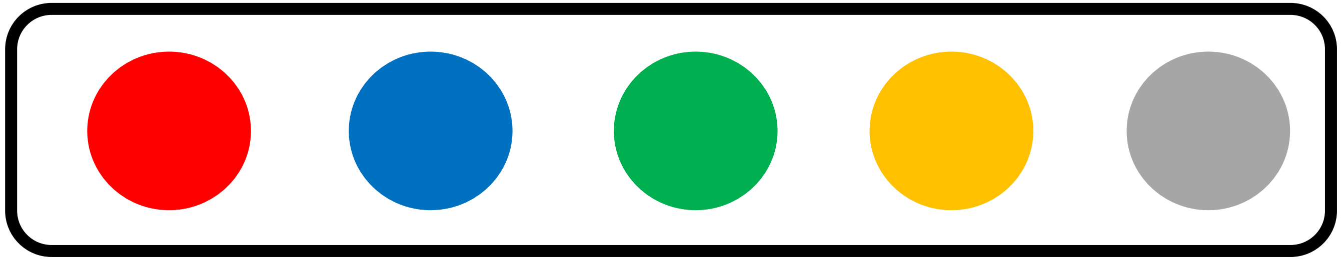
Weight so far: 1 kg



1 kg
$6

Capacity: 5 kg

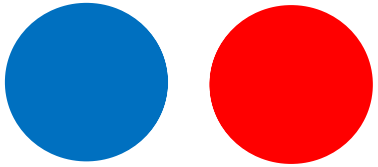
The Fractional Knapsack Problem



Weight (kg)	2	1	3	6	11
Value ($)	10	6	12	12	33
Density (V/W)	5	6	4	2	3

Total Value: $16

Weight so far: 3 kg

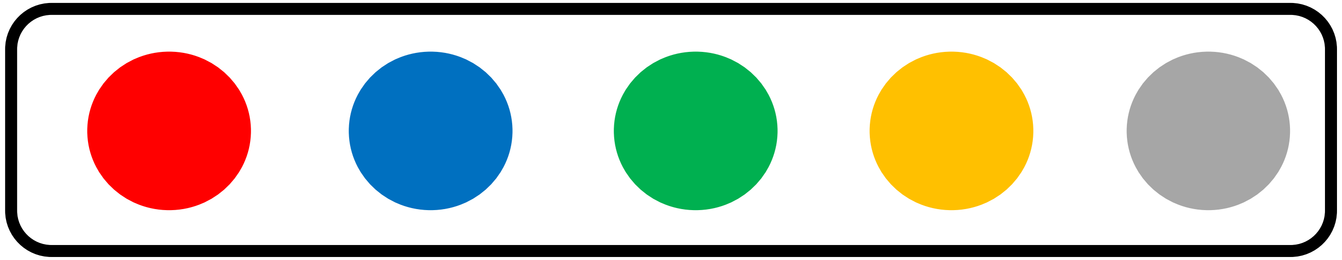


1 kg
$6

2 kg
$10

Capacity: 5 kg

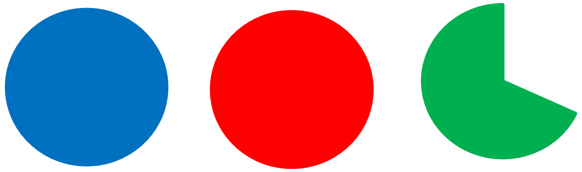
The Fractional Knapsack Problem



Weight (kg)	2	1	3	6	11
Value ($)	10	6	12	12	33
Density (V/W)	5	6	4	2	3

Total Value: $24

Weight so far: 5 kg



1 kg
$6

2 kg
$10

$2/3 \times 3$ kg
$2/3 \times \$12$

Capacity: 5 kg

Fractional Knapsack Problem

Greedy-FracKnapsack(I, W):
For each $i \in [1, n]$: $d_i = v_i / w_i$
Let $S = \{(s_i, v_{s_i}, w_{s_i})\}_{i \in [1,n]}$ be the list of items sorted based on d_i in descending order.
Initialize $w_{left} = W, V = 0, i = 0, p_1, \ldots p_n = 0$
While $w_{left} > 0$ do:
    $(s_j, v_{s_j}, w_{s_j}) \leftarrow S[i]$
    If $w_j \leq w_{left}$ then
        $p_{s_j} = 1$
    If $w_j > w_{left}$ then
        $p_{s_j} = \frac{w_{left}}{w_{s_j}}$
    $V = V + p_{s_j} v_{s_j}$ // p_{s_j} is the fraction of item s_j selected
    $w_{left} = w_{left} - p_{s_j} w_{s_j}$
    $i = i + 1$
Output $P = (p_1, \ldots, p_n)$

Fractional Knapsack Problem

- Running time of the greedy approach?
  1. Sort the n items based on density
  2. Add each item to fill the bag starting with largest density (possibly using the last item partially) until the bag is exactly full.

$$T(n) = \Theta(n \lg n)$$

Fractional Knapsack Problem

- Why did the greedy approach work here?

1. **Optimal substructure**: the final optimal solution contains optimal solutions for subproblems.

2. **Greedy-choice property**: choosing the best candidate at every step leads to an optimal solution later.

Fractional Knapsack Problem

Theorem: Algorithm **`Greedy-FracKnapsack`** leads to an optimal solution to the fractional knapsack problem

Proof of correctness:

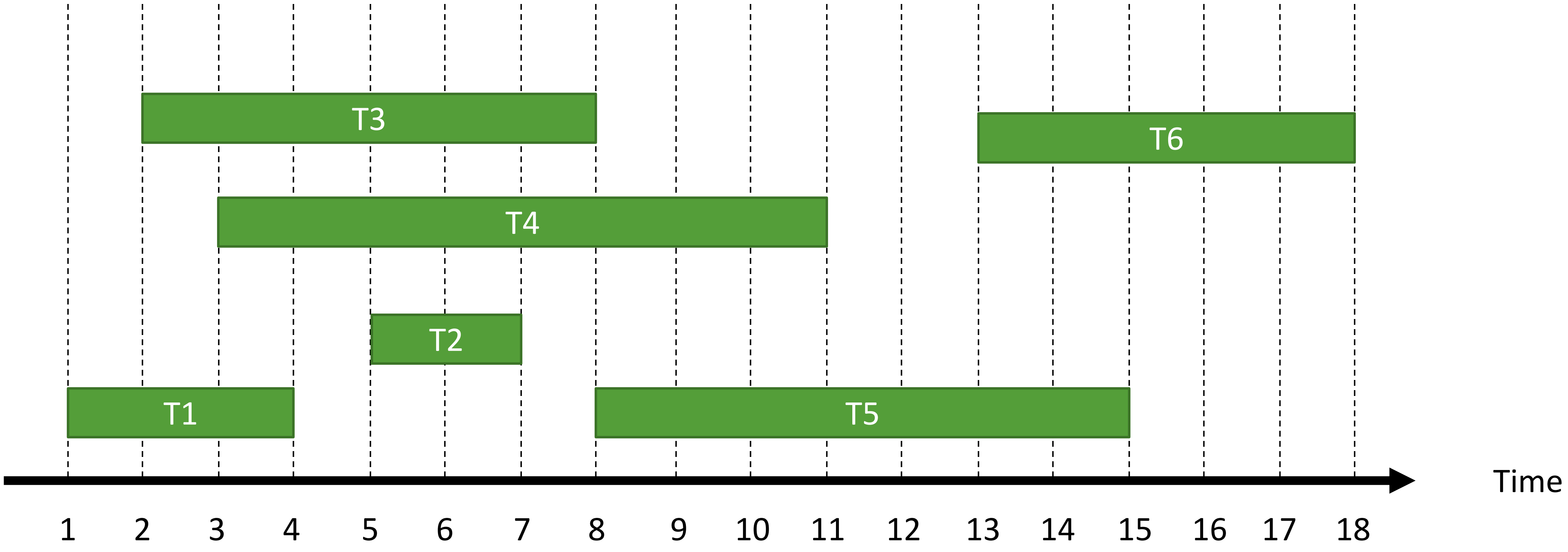
1. Consider a globally optimal solution Q
2. Show that Q can be modified so that:
   a. The **greedy choice** is made at the **first** step
   b. And this choice reduces the problem to a **smaller** fractional knapsack subproblem
3. Apply **induction** to show that a greedy choice can be taken at every step by proving that the problem exhibits optimal substructure

Task Scheduling Problem

Task No.	Start Time	End Time
1	1	4
2	5	7
3	2	8
4	3	11
5	8	15
6	13	18

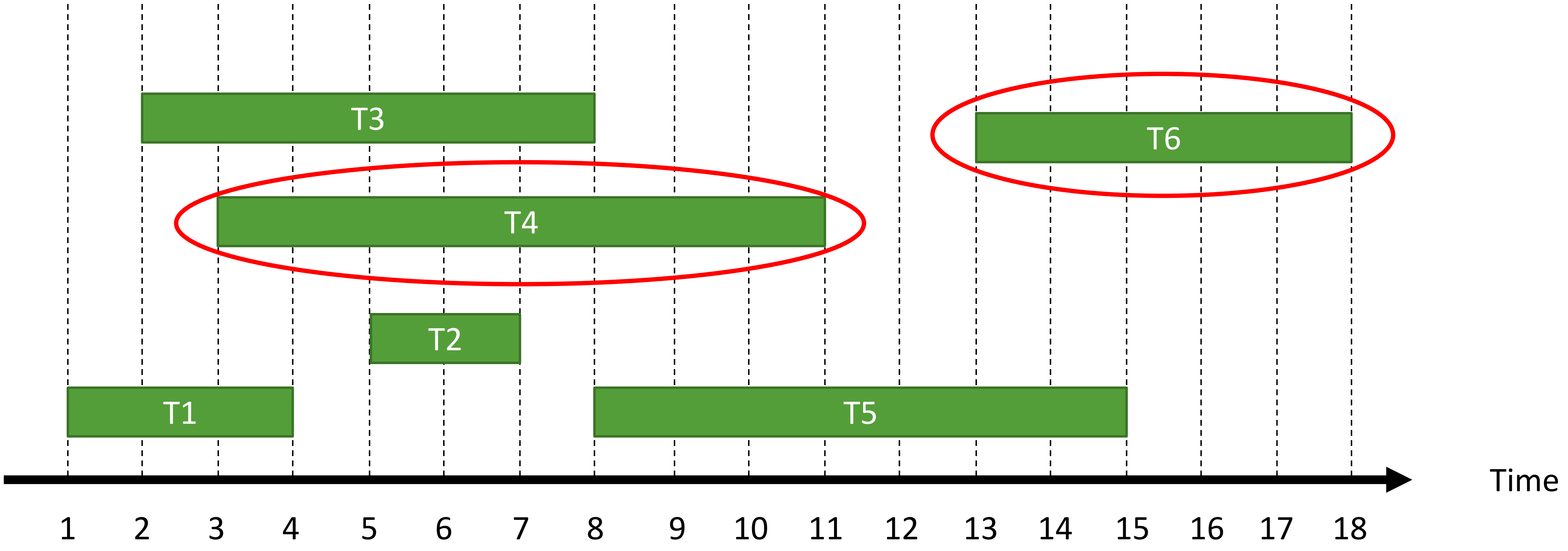
Goal: Schedule the maximum number of activities without having any two overlapping

Task Scheduling Problem



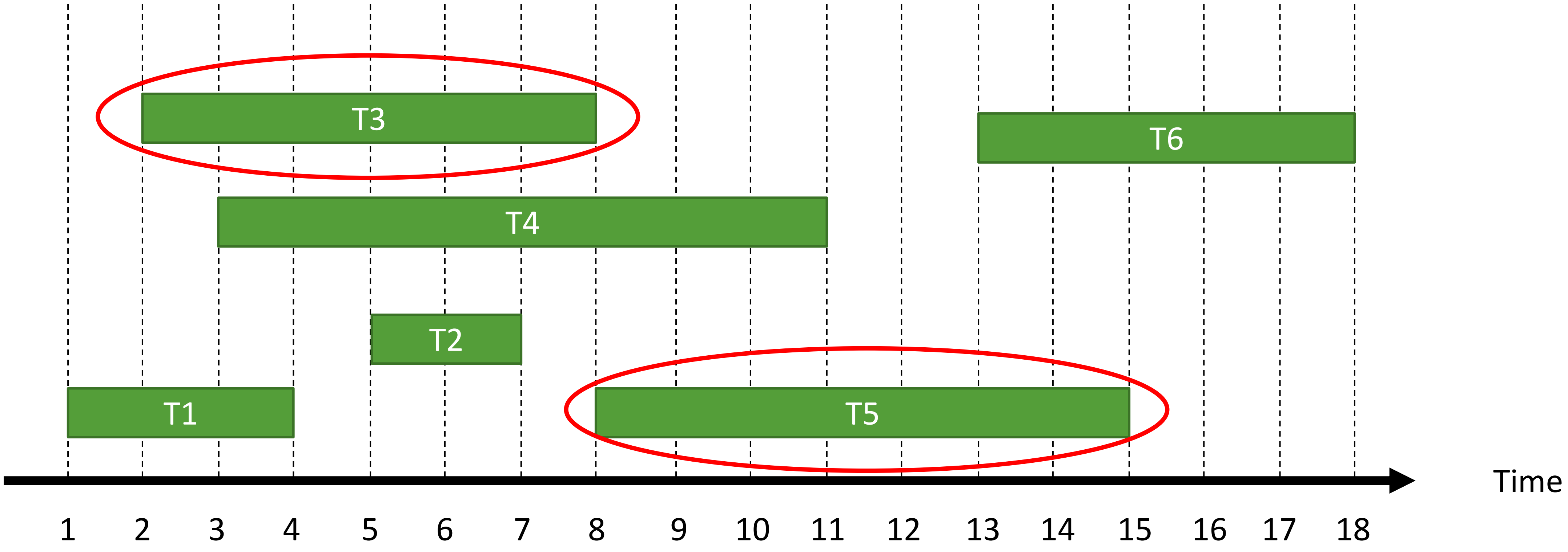
Task Scheduling Problem

Solution 1 = $\{T_4, T_6\}$



Task Scheduling Problem

Solution 2 = $\{T_3, T_5\}$



Task Scheduling Problem

- **Problem:** Given a set of activities with starting and ending times, schedule the maximum number of non-overlapping activities.
- **Input:** A set of n activities with start/finish times

$$T = \{(1, s_1, f_1), (2, s_2, f_2), \dots, (n, s_n, f_n)\}$$

- **Output:** A subset $P \subseteq T$ such that:

$$\text{Maximizes } |P|$$

$$\text{Subject to } \forall (i, s_i, f_i) \in P \text{ and } \forall \left(j \neq i, s_j, f_j\right) \in P : s_i \geq f_j \text{ or } s_j \geq f_i$$

Task Scheduling Problem

- Will a greedy approach here work?
  - We need to ask ourselves the following questions?

1. Does the problem exhibit **optimal substructure**?

2. Is there a globally optimal solution that contains greedy choices?
   - And if there is, what is the greedy choice property?

Task Scheduling Problem

1. Does the problem exhibit **optimal substructure**?

Theorem: Let k be a task in an optimal solution $Q \subseteq T$ then $Q - \{(k, s_k, f_k)\}$ is an optimal solution to the subproblem $T' = \{(i, s_i, f_i) \in T : s_i \geq f_k \mid\mid f_i \leq s_k\}$

Task Scheduling Problem

- **Proof:** (by contradiction)
  - Let Q be the optimal solution to the original problem T
  - Let $Q' = Q - \{(k, s_k, f_k)\}$ be the solution to the subproblem $T' = \{(i, s_i, f_i) \in T : s_i \geq f_k\}$
  - Suppose, for the sake of contradiction, that Q' is not optimal
    - Let $\hat{Q}$ be the optimal solution instead (i.e. it contains more tasks than Q')
    - Then $|\hat{Q}| > |Q'| = |Q - \{(k, s_k, f_k)\}| = |Q| - 1$
    - But this means that there exists a solution $\hat{Q} + \{(k, s_k, f_k)\}$ for the original problem T that is better than Q
    - So Q is NOT optimal (contradiction)
    - Hence, Q' is optimal as well

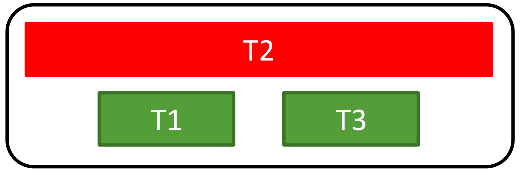
Task Scheduling Problem

2. Is there a globally optimal solution that contains greedy choices?
   - And if there is, what is the **greedy choice** property?

None of these greedy choices work!

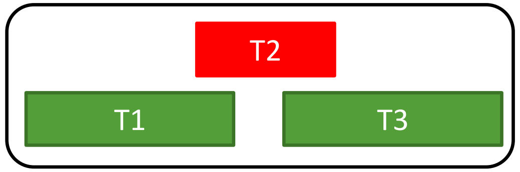
Greedy Choice 1: Earliest start time

Counterexample



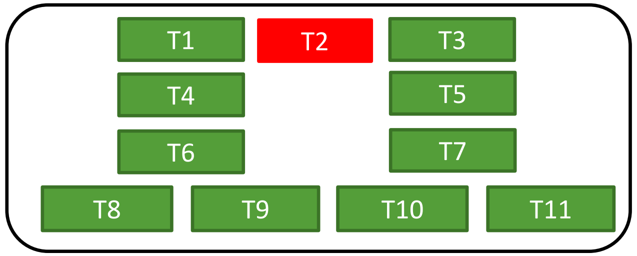
Greedy Choice 2: Shortest duration

Counterexample



Greedy Choice 3: Fewest conflicts

Counterexample



Task Scheduling Problem

2. Is there a globally optimal solution that contains greedy choices?
   - And if there is, what is the **greedy choice** property?

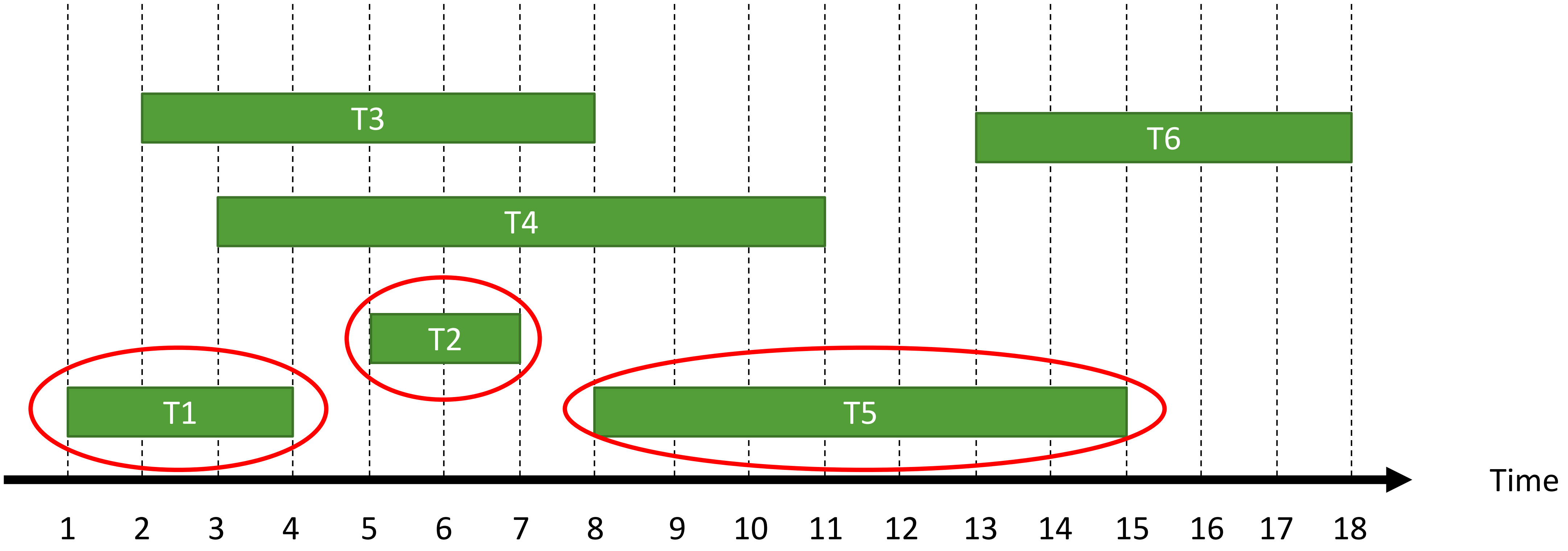
Correct Greedy Choice: At every step, choose the activity that finishes first

Greedy Approach: At every step, choose the activity that finishes first

Task Scheduling Problem

Is this optimal?

Solution $= \{T_1, T_2, T_3\}$



Task Scheduling Problem

`Greedy-TaskSchedule`(T):
Let $T' = \{(t_i, s_{t_i}, f_{t_i})\}_{i\in[1,n]}$ be the list of items sorted based on f_i in ascending order.
Initialize $P = \{(t_1, s_{t_1}, f_{t_1})\}$ and $j = 1$
For $i = 2$ to n:
    if $s_{t_i} \geq f_{t_j}$ then
        $P = P \cup (t_i, s_{t_i}, f_{t_i})$
        $j = i$ // j is the index of the most recent activity added to P
Output P

Task Scheduling Problem

- Running time of the greedy approach?
  1. Ascendingly sort the n activities based on finish time
  2. Add each activity (in the sorted order) to the schedule starting with the first activity in the list as long as that activity does not overlap with any previously chosen activities.

$$T(n) = \Theta(n \lg n)$$

Task Scheduling Problem

Theorem: Algorithm **`Greedy-TaskSchedule`** leads to an optimal solution to the task scheduling problem

Proof of correctness:

1. Consider a globally optimal solution Q
2. Show that Q can be modified so that:
   a. The **greedy choice** is made at the **first** step
   b. And this choice reduces the problem to a **smaller** task scheduling subproblem
3. Apply **induction** to show that a greedy choice can be taken at every step by proving that the problem exhibits optimal substructure (already done!)

Task Scheduling Problem

- **Proof:**
  - Let $Q = (q_1, q_2, \ldots)$ be the optimal solution ordered by finish time (i.e. $f_{q_1} \leq f_{q_2} \leq \cdots$)
  - Let $P = (p_{j_1}, p_{j_2}, \ldots)$ be the greedy solution sorted by selection order (i.e. finish time)

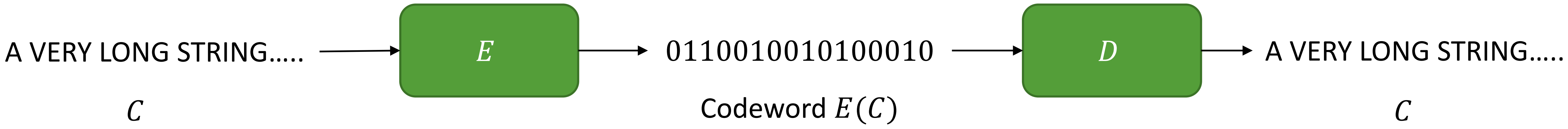
  - We show that there exists an optimal solution where the first choice is greedy:
    - Case 1: If $q_1 = p_{j_1}$ then we are done, the optimal solution already includes the first greedy choice
    - Case 2: if $q_1 \neq p_{j_1}$ then we will show how to construct a different optimal solution where $q_1 = p_{j_1}$:
      - Remove task q_1 from Q
      - Add p_{j_1} to Q to create a new solution Q'.
        - Note that p_{j_1} will not overlap with $q_2, q_3, \ldots$ because $f_{p_{j_1}} \leq f_{q_1}$ and we already know that q_1 did not overlap with them.
      - Since we have not decreased the number of tasks in the new solution, Q' is an optimal solution (which has the first choice as greedy)

The Data Compression Problem

- We would like to **represent data** using **less space** than the original representation.
- **Encoding data** using **fewer bits** than the original encoding

The Data Compression Problem

- **Problem:** Given an input file with a sequence of characters, output an encoding of the input that is lossless, decodable, and of smaller size.
- **Input:** A sequence of characters $C = (c_1, \dots, c_n)$
- **Output:** An encoder E and decoder D such that $D\big(E(C)\big) = C$ and the length of the codeword $|E(C)|$ is **minimized**.



The Data Compression Problem

- Example: Let the input string be

BCAADDDCCACACAC

- Without compression: using **ASCII**, the number of bits needed to store this string is $8 \times 15 = \mathbf{120}$ **bits** since each character is 1 byte.
- How can we reduce the number of bits needed to represent this string?

The Data Compression Problem

- Example: Let the input string be

BCAADDDCCACACAC

- **Naive Method** (**Fixed length code**):
  - Assign unique equal-length codes for each letter in the input string.
  - So now the input string is now encoded as follows:

$$01\ 10\ 00\ 00\ 11\ 11\ 11\ 10\ 10\ 00\ 10\ 00\ 10\ 00\ 10$$

  - We reduced the length of the representation to $\mathbf{2 \times 15 = 30}$ **bits**
  - But, is this the shortest encoding possible?

Character	Encoding
A	00
B	01
C	10
D	11

The Data Compression Problem

Character	Encoding
A	00
B	01
C	10
D	11

- Example: Let the input string be

  BCAADDDCCACACAC

- **Naive Method** (**Fixed length code**):
  - **This method is wasteful**
  - **Some characters might appear more often than others**, but we are giving the same encoding length of more frequent characters as those do not appear a lot.

The Data Compression Problem

Character	Frequency	Encoding
C	6	0
A	5	10
D	3	110
B	1	1110

- Example: Let the input string be

  BCAADDDCCACACAC

- **Method 2**: Unary variable-length **prefix code**
  - **Characters that appear more frequently are given a shorter encoding**
  - Prefix code: only use codes for which no code word is a prefix of another one
  - So now, the input string is now encoded as follows:

$$1110\ 0\ 10\ 10\ 110\ 110\ 110\ 0\ 0\ 10\ 0\ 10\ 0\ 10\ 0$$

  - We reduced the length of the representation to:

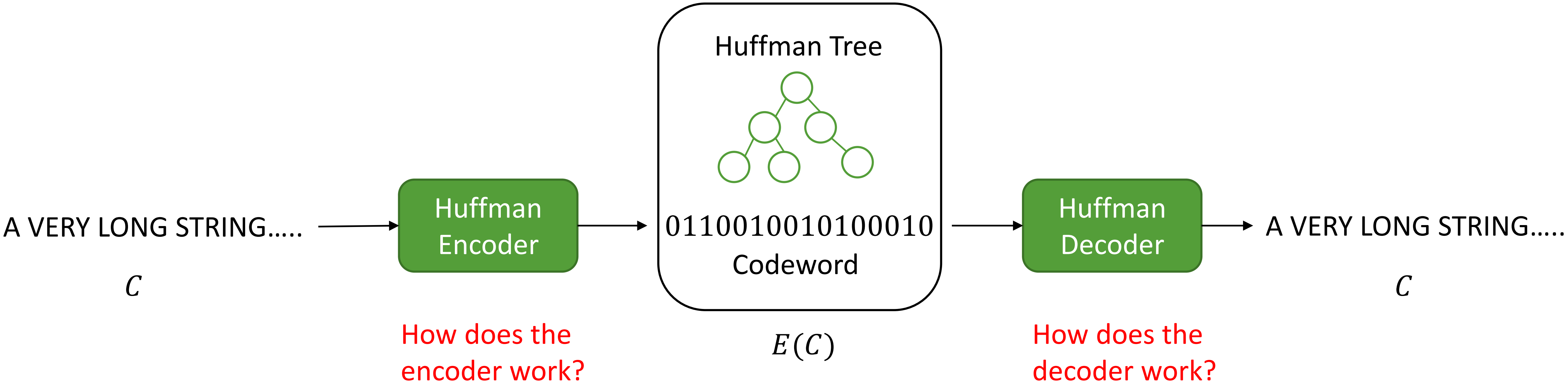
$$(6 \times 1) + (5 \times 2) + (3 \times 3) + (1 \times 4) = \textbf{29 bits}$$

  - Can we do better?

Huffman Encoding

- Creates variable length encodings for each distinct character in the input
- Encodings are created so characters that appear **more frequently** are given **shorter** **prefix encodings** compared to characters that occur less frequently
- The **greedy choice** here is: to assign the **shorter encoding** to the character with the **higher frequency**.

Huffman Encoding



Huffman Encoding

- **Encoding** algorithm steps:

1. Find the frequency of each character in the input file
2. Sort the characters in increasing order of frequency
3. Build a Huffman tree from the frequency data
4. Traverse the Huffman tree and build the encodings for each character found in the input file
5. For each character in the input file, write the bits of the Huffman encoding to the output file
6. Output the codeword and the Huffman Tree

Huffman Encoding: Example

BCAADDDCCACACAC

1. **Find the frequency** of each character in the input file

Character	Frequency
C	6
A	5
D	3
B	1

Huffman Encoding: Example

BCAADDDCCACACAC

2. Sort the characters in increasing order of frequency

Character	Frequency
C	6
A	5
D	3
B	1

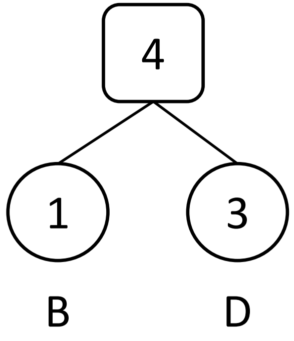
Character	Frequency
B	1
D	3
A	5
C	6

Huffman Encoding: Example

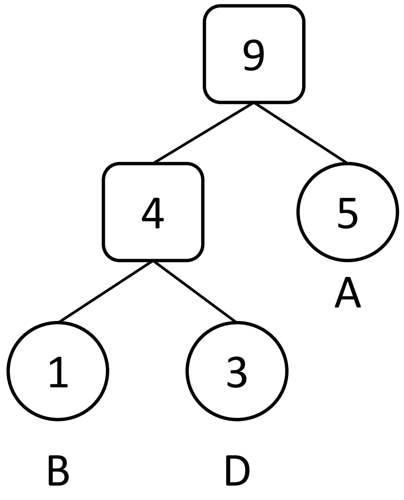
BCAADDDCCACACAC

Character	Frequency
B	1
D	3
A	5
C	6

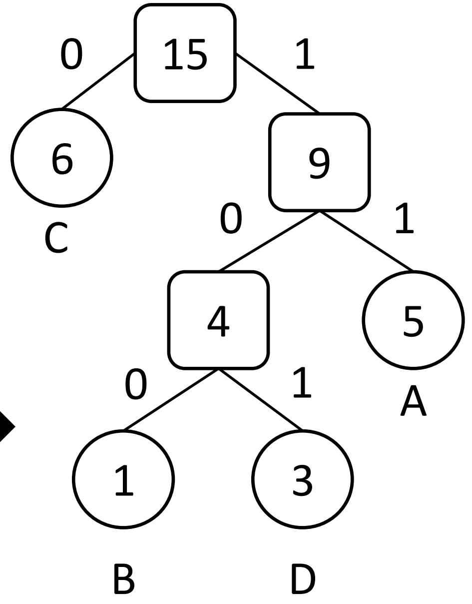
3. **Build a** **Huffman tree** from the frequency data



Character	Frequency
B+D	4
A	5
C	6



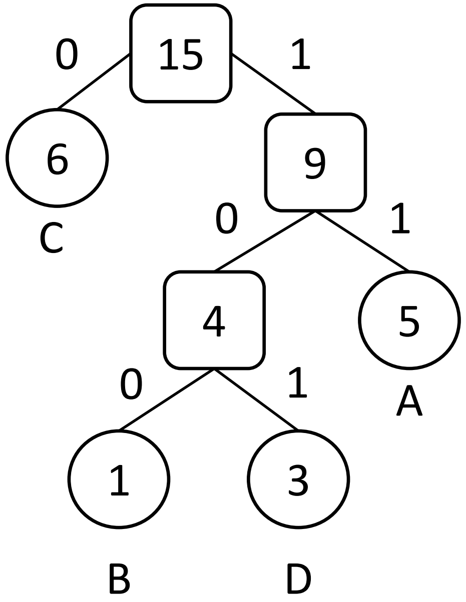
Character	Frequency
C	6
B+D+A	9



Huffman Encoding: Example

BCAADDDCCACACAC

4. Traverse the Huffman tree and **build the encodings** for each character found in the input file



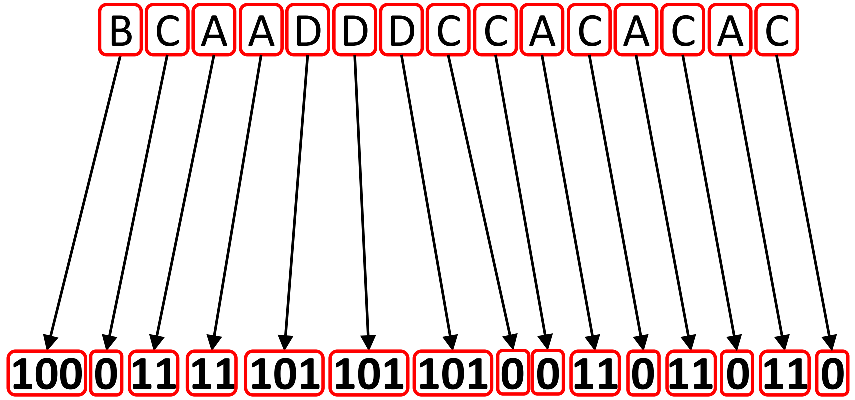
Character	Frequency	Encoding
C	6	0
A	5	11
D	3	101
B	1	100

Huffman Encoding: Example

BCAADDDCCACACAC

5. For each character in the input file, **write the bits** of the Huffman encoding to the output file

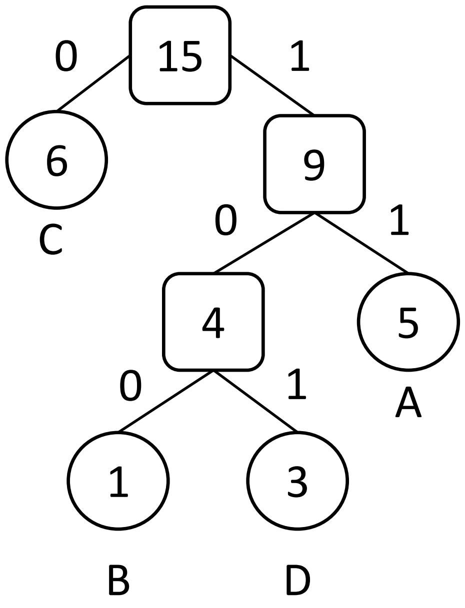
Character	Frequency	Encoding
C	6	0
A	5	11
D	3	101
B	1	100



Huffman Encoding: Example

BCAADDDCCACACAC

6. **Output the codeword** and the Huffman Tree



100 0 11 11 101 101 101 0 0 11 0 11 0 11 0

28 bits

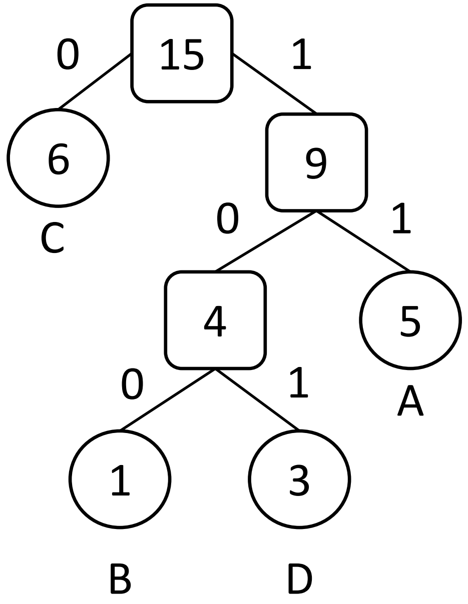
Huffman Decoding

- **Decoding** algorithm steps:

1. Read the representation of the Huffman tree from the input file and rebuild the Huffman tree
2. Read the bits from the input file
3. Traverse the Huffman tree using the bits from the input file and output the character whenever a leaf is reached

Huffman Decoding

- Decode the following bit string:



$$1101011000$$

$$\downarrow$$

$$ACDBC$$

$$\boldsymbol{T(n) = \Theta(n \lg n)}$$

Huffman Encoding: Running Time

1. Find the frequency of each character in the input file $\Theta(n)$
2. Sort the characters in increasing order of frequency $\Theta(n \lg n)$
3. Build a Huffman tree from the frequency data $\Theta(n \lg n)$
4. Traverse the Huffman tree and build the encodings for each character found in the input file $\Theta(n \lg n)$
5. For each character in the input file, write the bits of the Huffman encoding to the output file $\Theta(n \lg n)$
6. Output the codeword and the Huffman Tree $\Theta(n)$

Summary

- When are greedy algorithms useful?
  - For problems that exhibit greedy property and optimal substructure
- Procedure for greedy algorithm design
  - Identify optimal substructure
  - Define the greedy choice property
  - Write your algorithm based on the greedy choice
- How do we know if a greedy algorithm does not work?
  - Show a counterexample