TDT4121

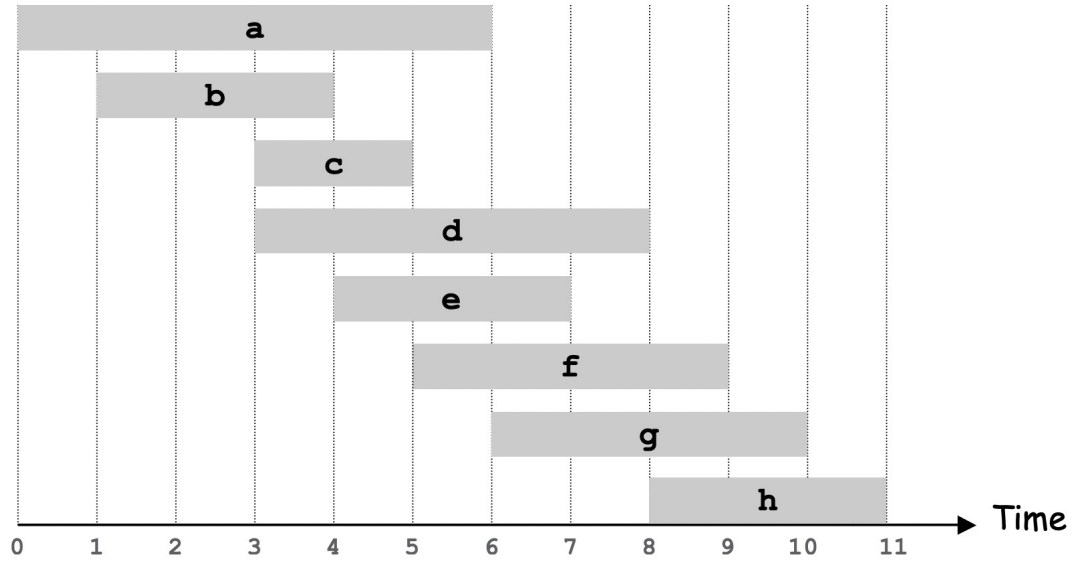
Assignment Lecture Week 41

Exercise 4

Interval scheduling

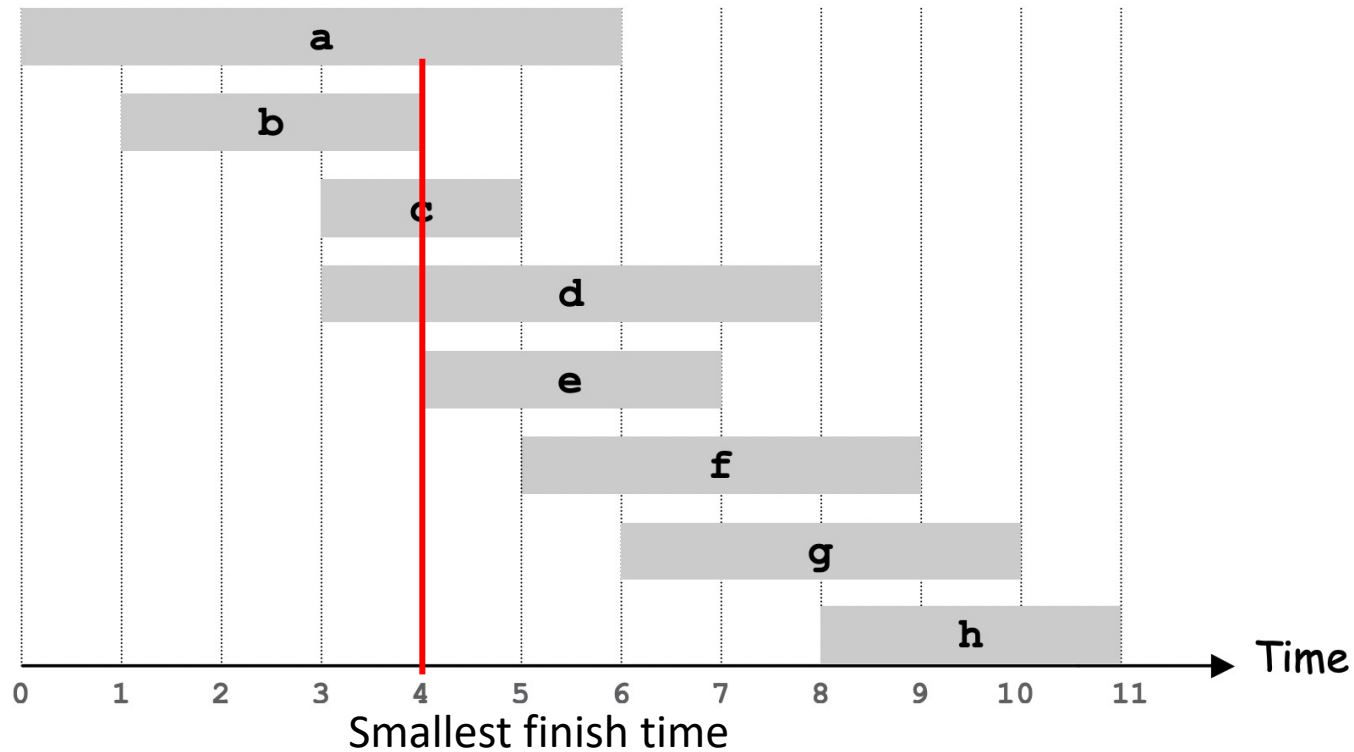
Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval scheduling.

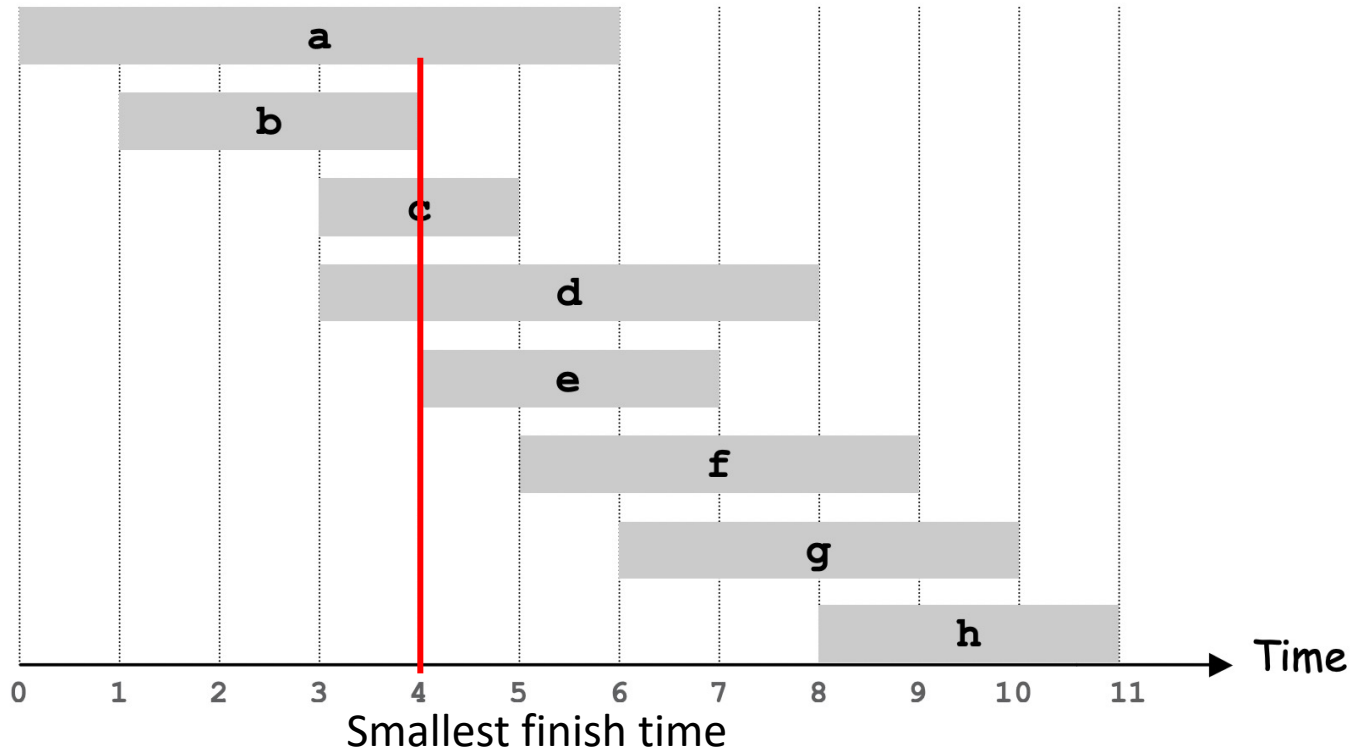
- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

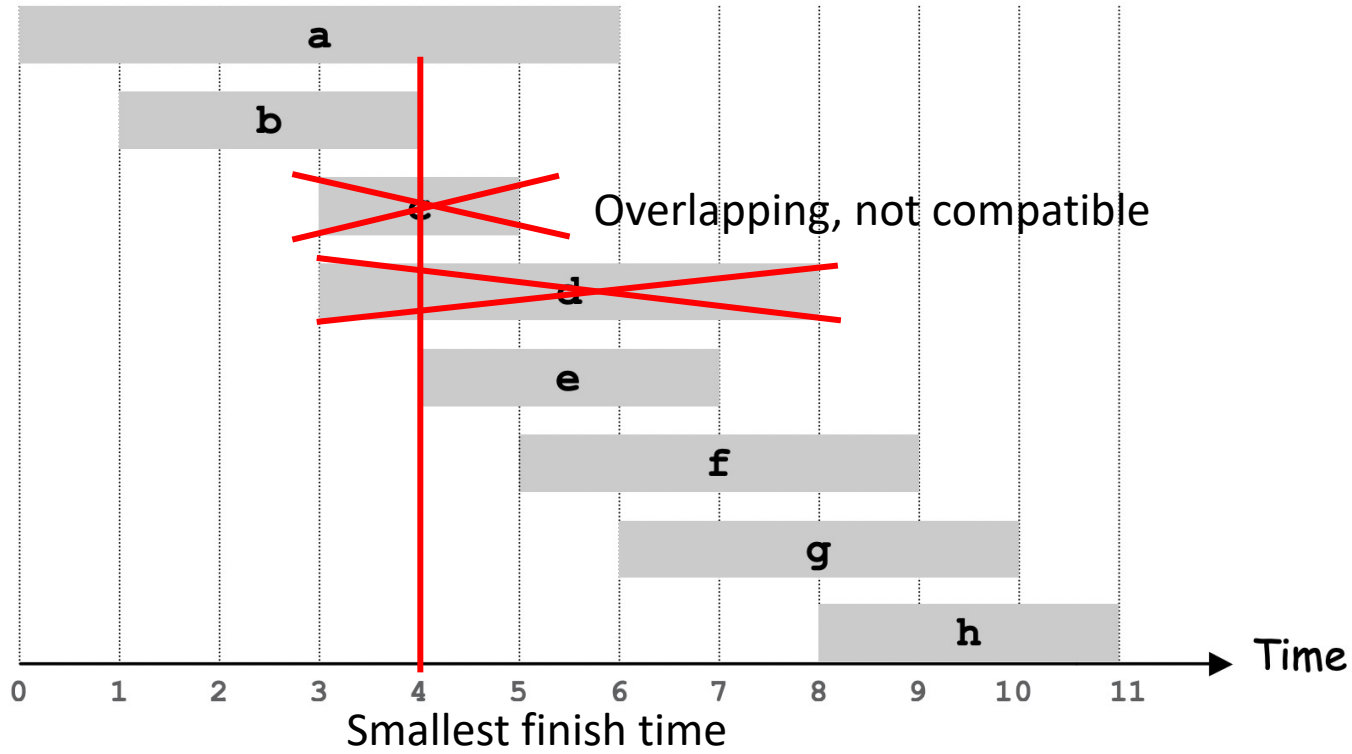
B



Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

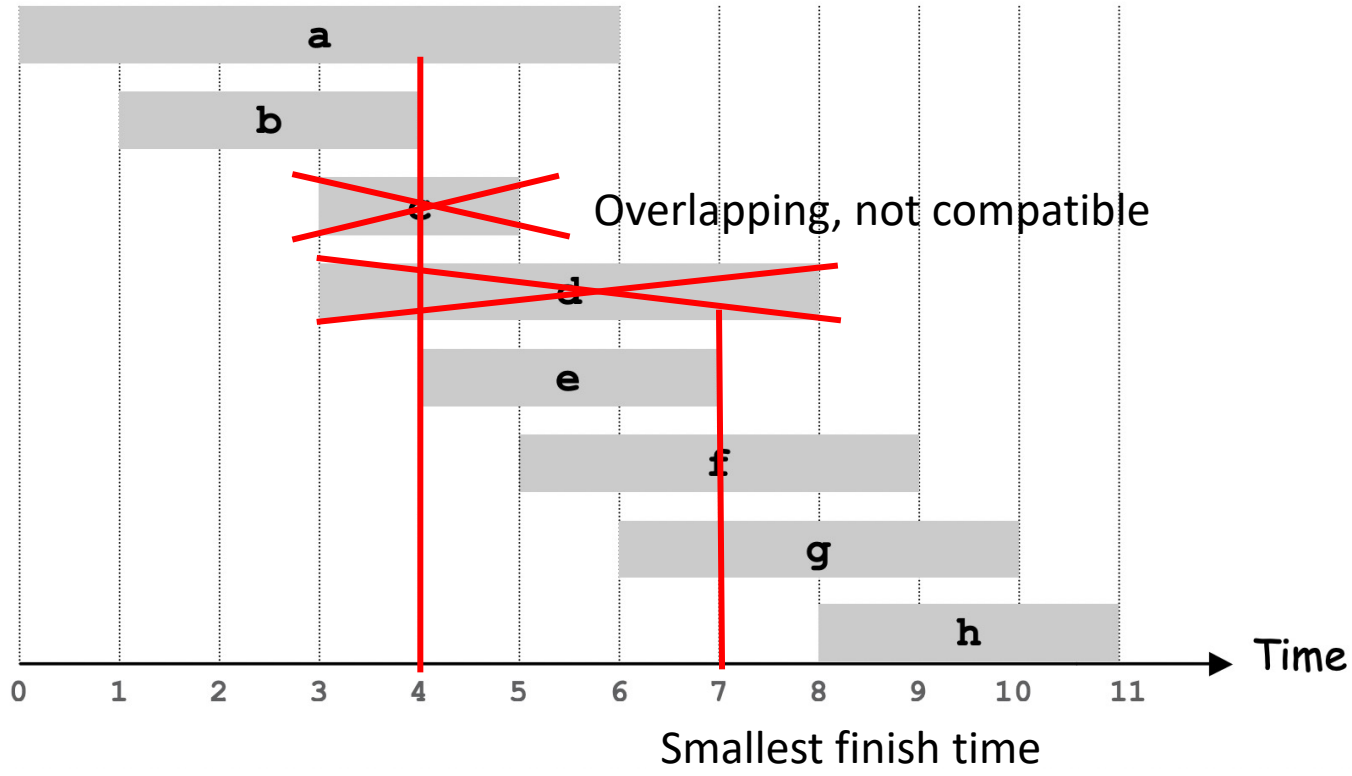
B



Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

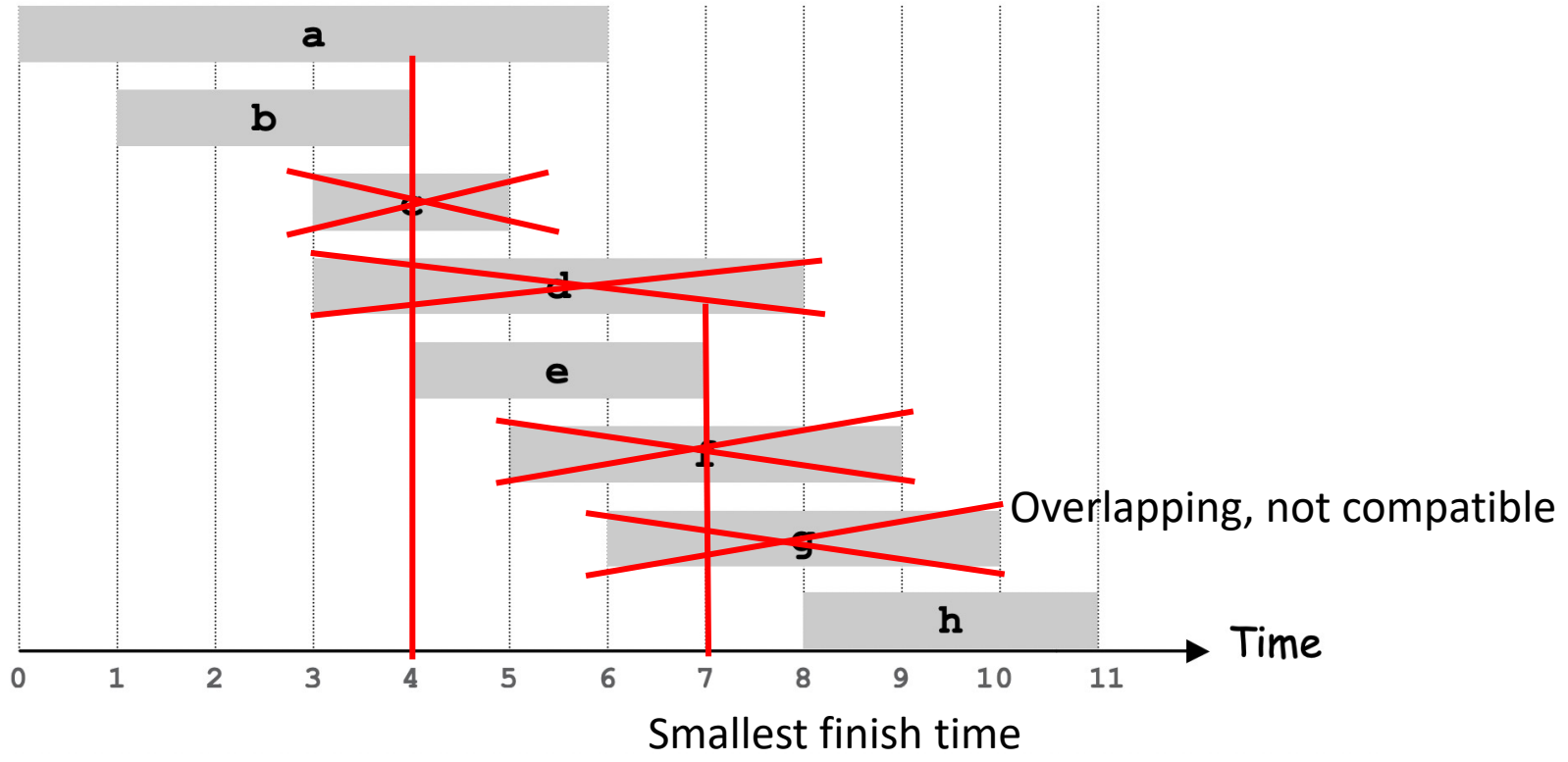
B, E



Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

B, E, H

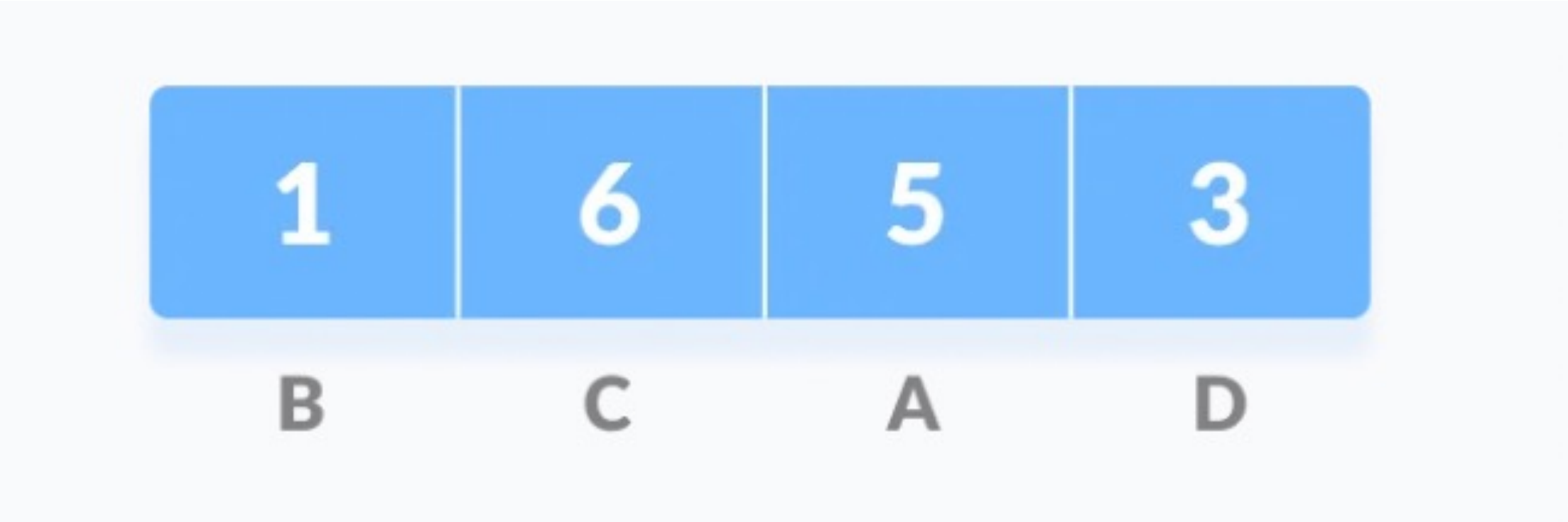


Huffman coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

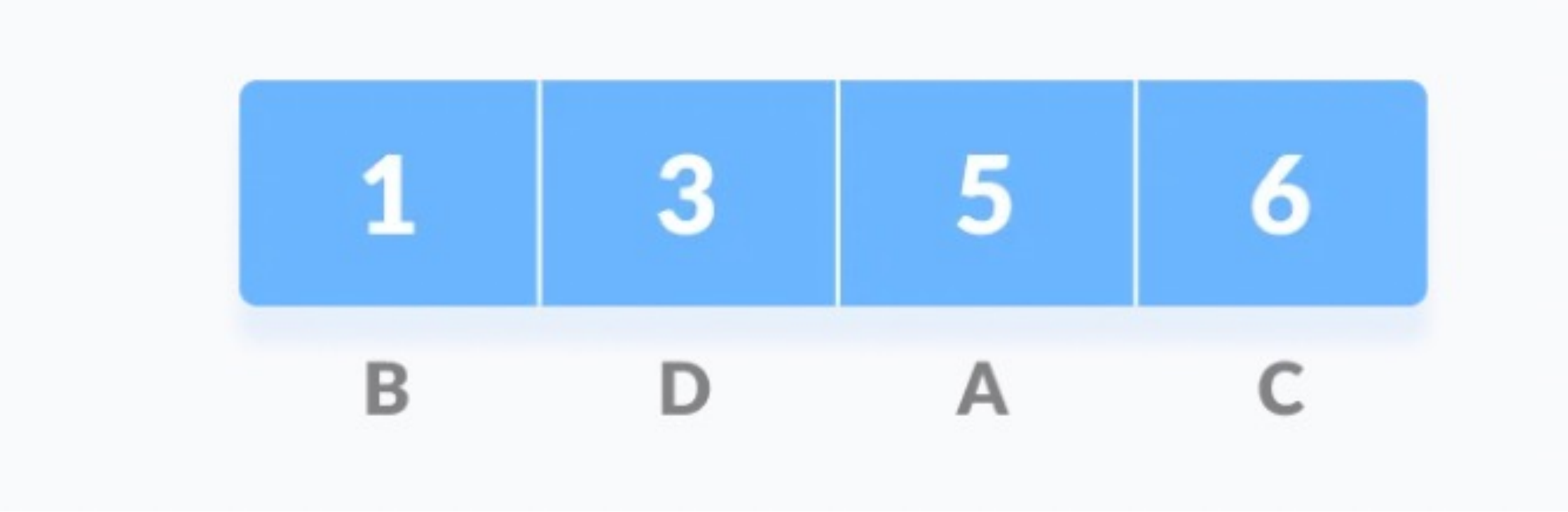
1

Calculate the frequency of each character in the string



2

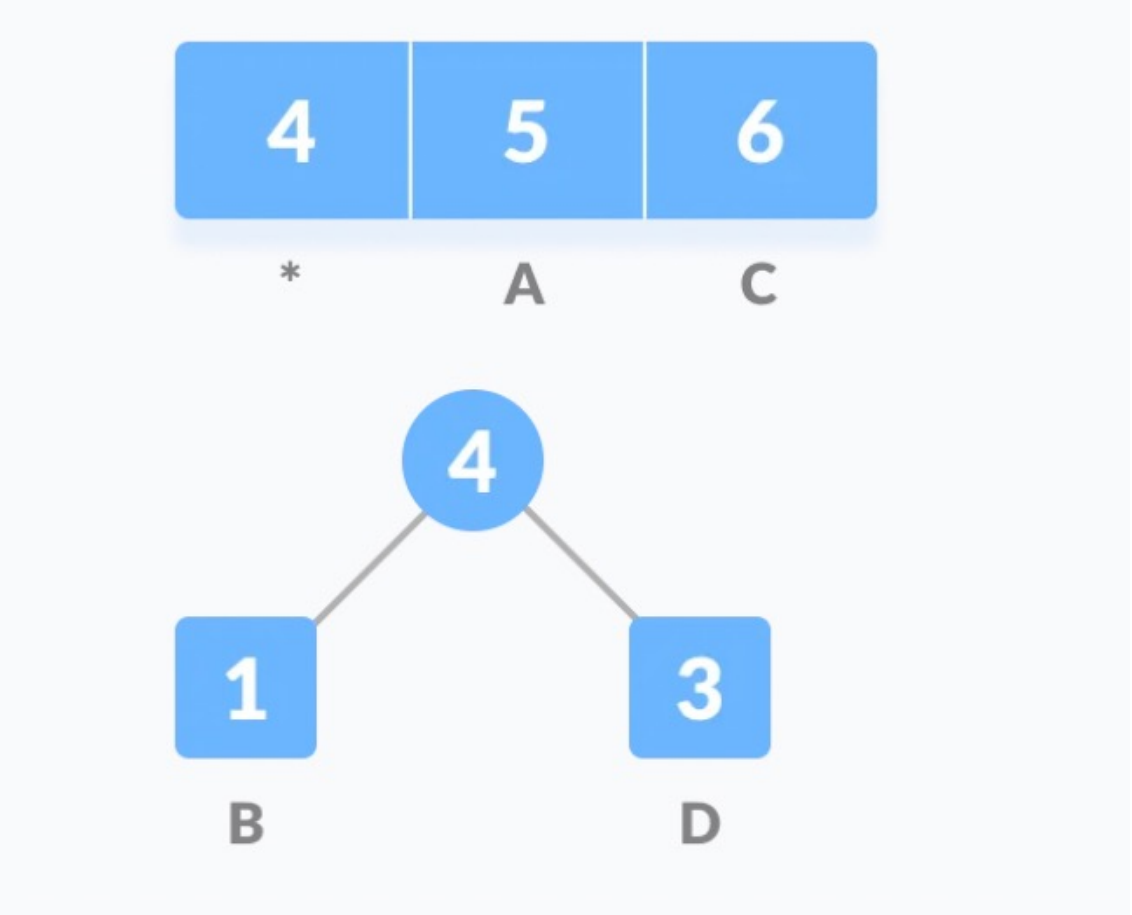
Sort after frequency



3

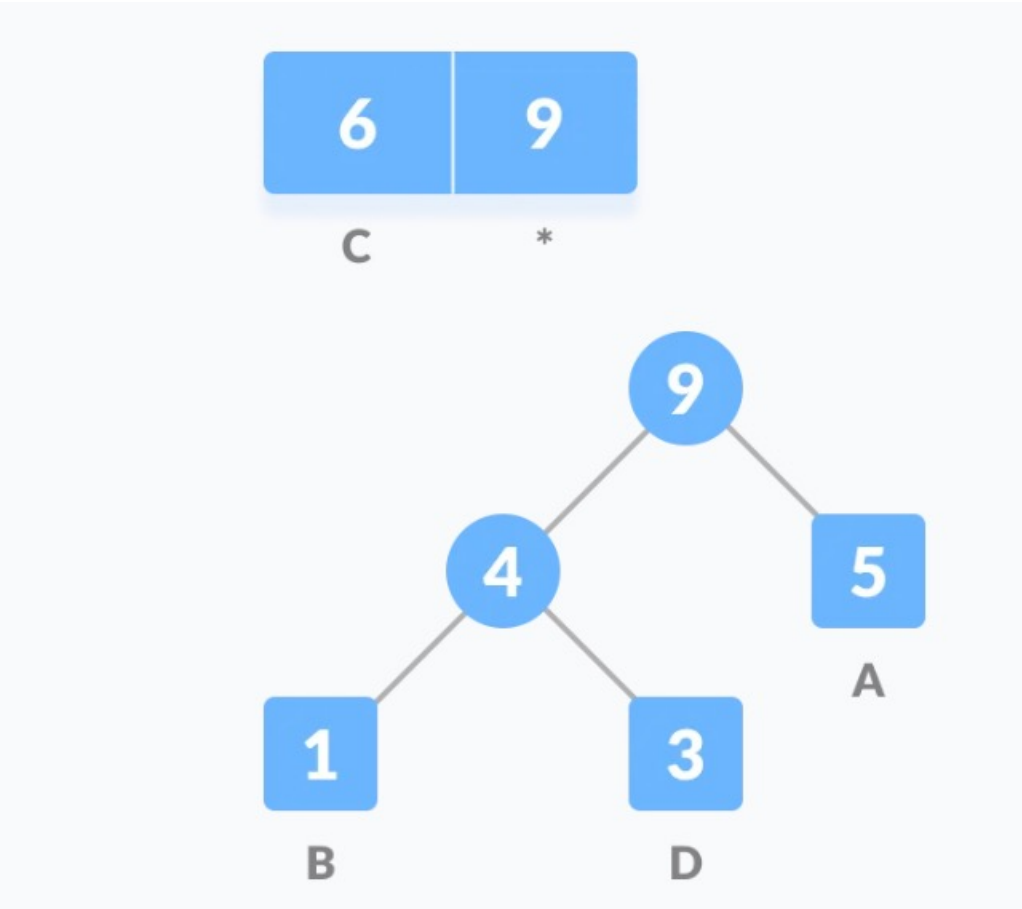
Create a node with the value of the sum of the two minimum frequencies. The left child is the least frequency and the right is the second minimum frequency. Here node B and D

Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure)



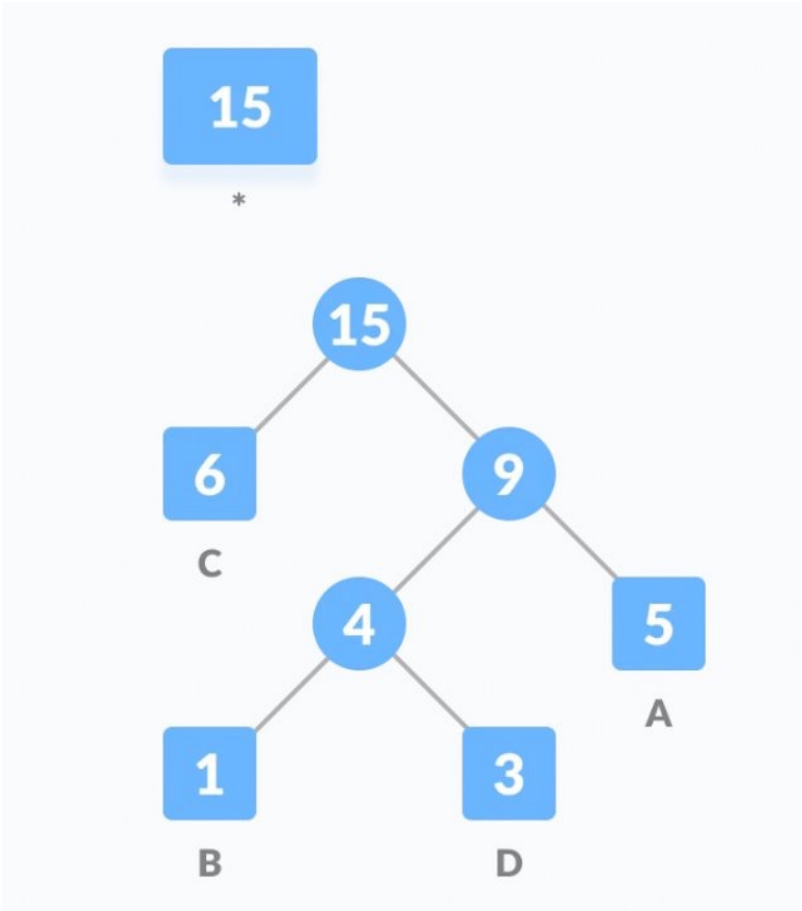
4

Insert new node and repeat step 1-3

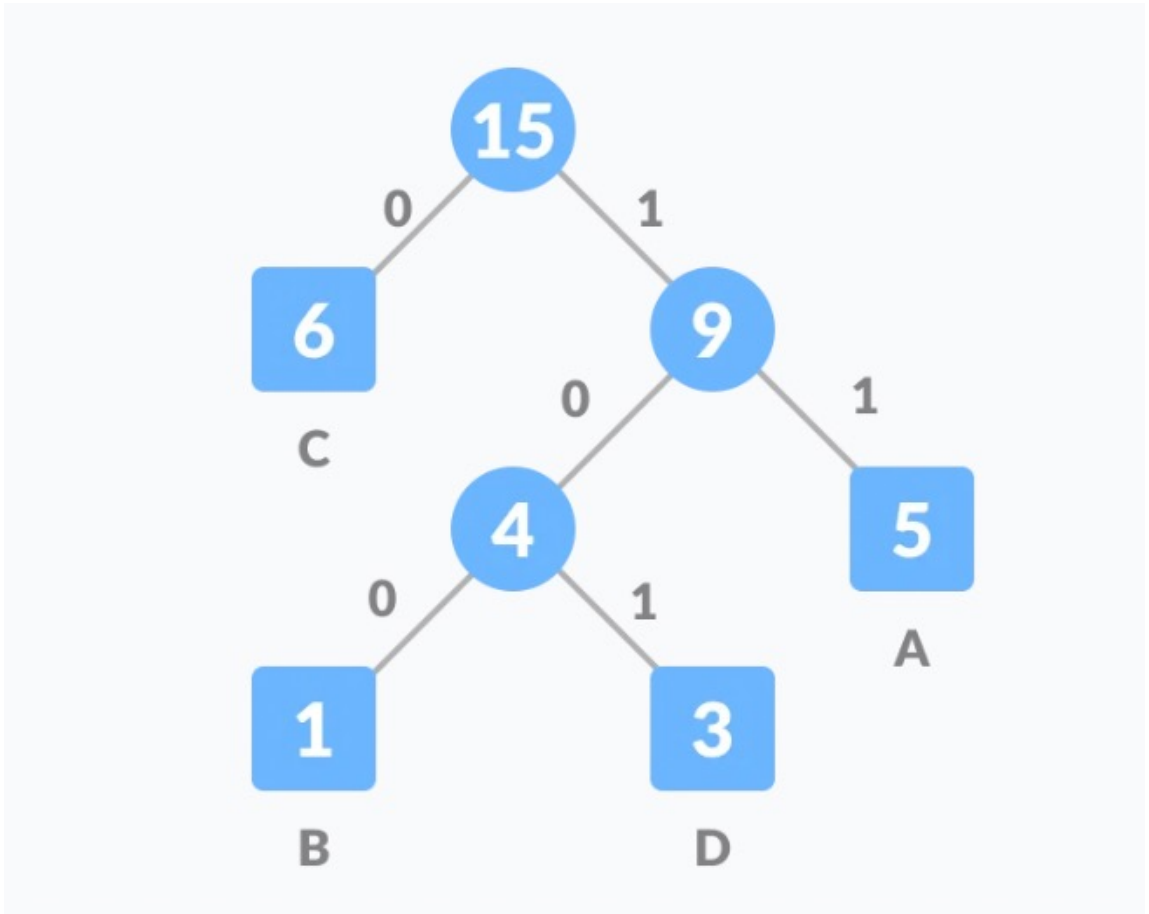


4

Insert new node and repeat step 1-3



For each non-leaf node, assign 0 to the left edge and 1 to the right edge



Exercise 5

Dynamic Programming

Task 1

Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

a) **Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?**

b) **You decide to help Jesse. Based on your observations from task a), describe an algorithm that solves the same problem in linear time**

c) **What properties must a problem have for it to be able to be solved with dynamic programming?**

What happens when you call fib(5)?

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

a) Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?

If 5 <= 1:

return 5

Return fib(5 – 2 = 3) + fib(5-1 = 4)

What happens when you call fib(5)?

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

a) Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?

If 5 <= 1:

return 5

Return fib(5 – 2 = 3) + fib(5-1 = 4)

What happens when you call fib(5)?

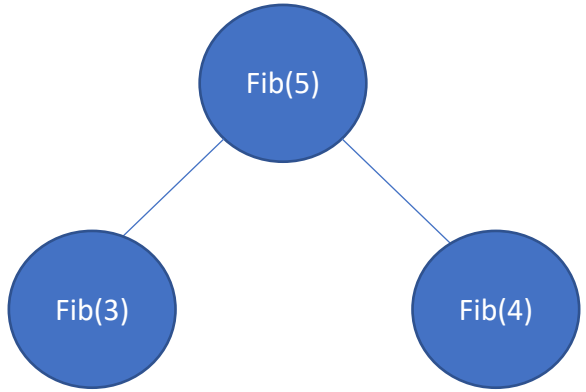
```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

a) Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?

If 5 <= 1:

return 5

Return fib(5 – 2 = 3) + fib(5-1 = 4)



What happens when you call fib(5)?

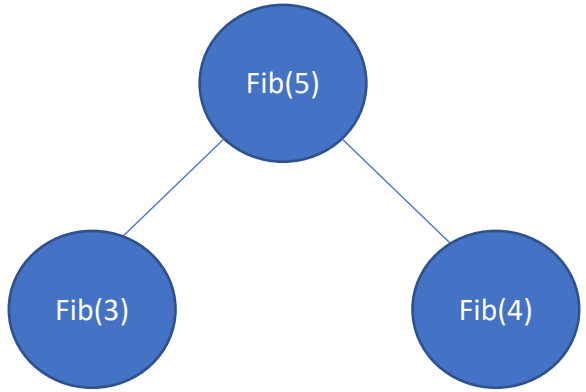
```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

a) Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?

If 5 <= 1:

return 5

Return fib(5 – 2 = 3) + fib(5-1 = 4)



How does the tree grow as n grows?

Task 1b

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

b) You decide to help Jesse. Based on your observations from task a), describe an algorithm that solves the same problem in linear time

Will there be repeated calls in the case of fib(5)? Is there a way to save and reuse results?

Task 1c

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

c) What properties must a problem have for it to be able to be solved with dynamic programming?

What makes dynamic programming work? For what problems will it not work?

Task 2

Task 2 Smoothie algorithm

Your friend Walter has made an algorithm based on dynamic programming. The algorithm makes the best-tasting smoothie based on the fruit and vegetables in Walter's pantry. You look at the algorithm and observe that it only calculates and returns the taste levels of the optimal smoothie. But it does not give the actual ingredients. Walter argues it's a trivial difference, and adding said functionality would be easy. What do you think?

How does dynamic programming build solutions? Can we easily modify the technique so that the decisions made are returned as well?

Task 3

Task 3 Shortest-path

Bellman-Ford and Dijkstra's algorithms are both algorithms for finding the shortest path in a graph. Explain their differences, in both the problem they solve and how they solve it.

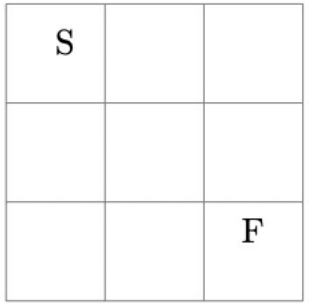
*When does Dijkstra's **not** work? When does Bellman-Ford **not** work?*

Task 4a

Task 4 Grid traveling

a) Given a $n \times m$ grid where you are only allowed to move to the right or downwards from one cell to another. In how many ways can you travel from the top left corner (S) to the bottom right corner(F) when the grid has the following sizes:

i. 3×3 Grid



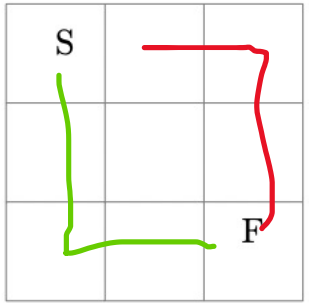
Task 4a

Task 4 Grid traveling

a) Given a $n \times m$ grid where you are only allowed to move to the right or downwards from one cell to another. In how many ways can you travel from the top left corner (S) to the bottom right corner(F) when the grid has the following sizes:

i. 3×3 Grid

Two possible paths



Task 4b

```
gridTravel(n,m):
    Array A[0...m, 0...n]
    A[1,1] = 1

    for j = 1,2,...,n:
        for i = 1,2,...,m:
            current = A[i,j]
            A[i+1][j] += current
            A[i][j+1] += current
        Endfor
    Endfor
    return A[m, n]
```

Does Saul's pseudocode work? Explain why or why not.

Task 4b

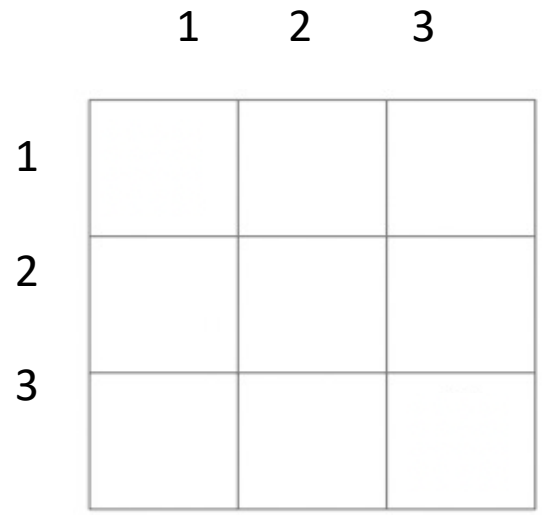
```
gridTravel(n,m):
    Array A[0...m, 0...n]
    A[1,1] = 1

    for j = 1,2,...,n:
        for i = 1,2,...,m:
            current = A[i,j]
            A[i+1][j] += current
            A[i][j+1] += current
        Endfor
    Endfor
    return A[m, n]
```

Does Saul's pseudocode work? Explain why or why not.

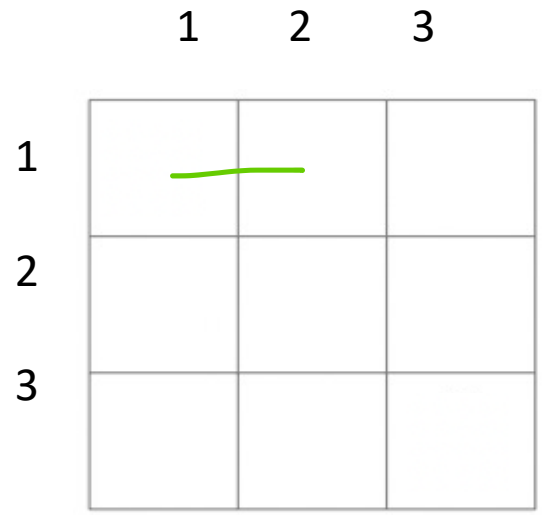
Why do we set A[1,1] = 1?

Task 4b



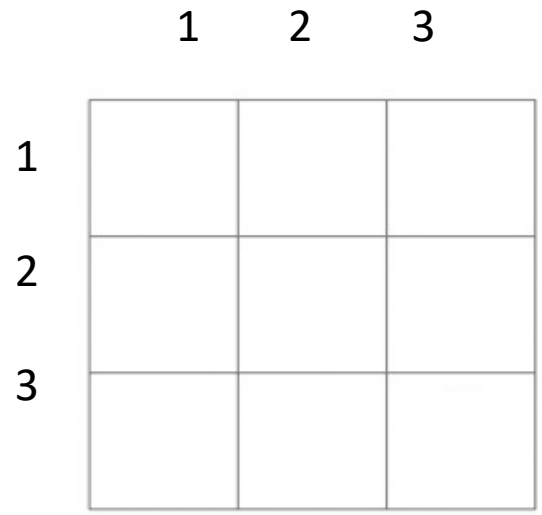
How many paths are there from (1,1) to (2, 1)?

Task 4b



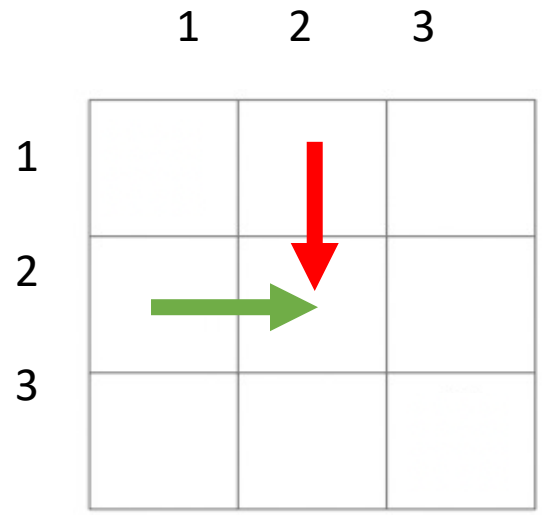
Only 1 path, so we set A[1,2] = 1

Task 4b



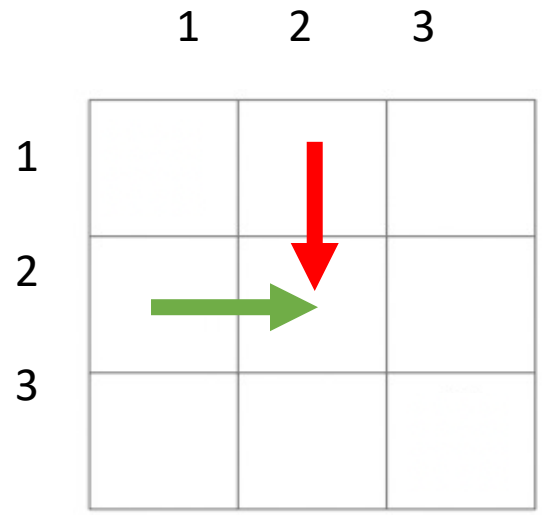
How many from (1, 1) to (2,2)?

Task 4b



We can enter (2,2) either from (2,1) or from (1,2)

Task 4b



So the number of paths to (2, 2) equals:

The number of paths to (2, 1)

+

The number of paths to (1,2)

Task 5b - Sequence alignment

b) Which of the following alignments of PALETTE and PALATE, has the lowest cost? Explain your answer.

Assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# Allignment 1
PALETTE
PALATE-

# Allignment 2
PALETTE
PALAT-E

# Allignment 3
P-ALETTE
-PALAT-E
```

How many gaps are there? How many misalignments? What is the cost of each?