NTNU
Kunnskap for en bedre verden

Assignment Lecture 05/10

Agenda

- Assignment 3 Solution
- Assignment 4 Introduction
- Some Greedy Algorithms

NTNU

TDT4121 Introduction to Algorithms

Solution - Assignment 3
Divide and Conquer

Part 1 Theory

Task 1 Divide and Conquer

a) Divide and Conquer is not only a widely used algorithm design paradigm, but also a sociological and political strategy. Give a brief example how divide and conquer has been used outside of the scope of algorithms.

Solution:
This is an open task so no solution is given, but the answer should clearly state the main ideas of the divide and conquer strategy.

b) Bank Security

among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another?

Solution:
Divide the set of cards into two equal piles of $n/2$ cards. The algorithm will be used recursively on both piles. If the algorithm finds a majority equivalence class with more than half of a pile, a card from that class will be returned.

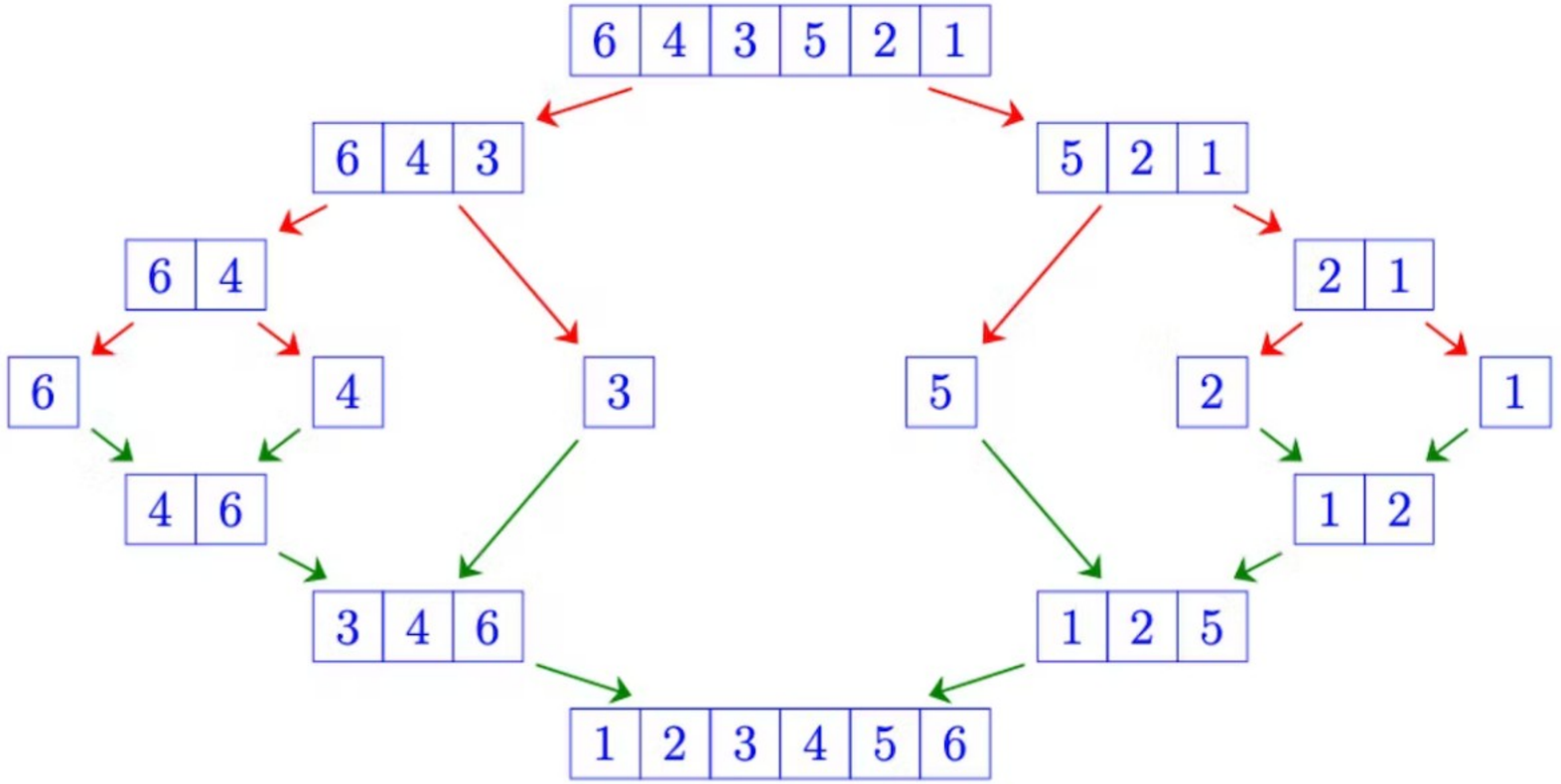
Its worth noting that if there are more than $n/2$ cards that are equivalent in the whole set then alteast one of the piles will have to have more than half of its cards from this equivalence class. For the algorithm this means that if none of the two piles return a card, then there is no equivalence class with more than $n/2$ cards. However it is not guaranteed that more than $n/2$ cards belong to an equivalence class if a card is returned, we hence need to check the returned card against all other cards.

```
1  algorithm takes a set of cards S
2      if |S| = 1 return the one card
3      if |S| = 2
4          if the two cards are equvalent
5              return either card
6          else
7              return nothing
8      let S1 be the first S/2 cards
9      let S2 be the set of the remaining cards
10     call algorithm recursively for S1
11     If a card is returned
12         test it against all other cards
13     If no card with majority equivalence is found
14         call algorithm recursively for S2
15         If a card is returned
16             test it against all other cards
17     Return card from majority equvalence class if one is found
```

Task 2 Merge Sort

a) Visualize how the array will be divided and then put back together during a merge sort. Recursion should stop when there is only 1 element left in an array.

Solution:
One possible solution. When splitting the subarrays for the second time this solution split them differently, thats simply for symmetrical reasons. The red arrows shows dividing, the green arrow shows merging.



b) How many comparisons are done during this merge sort?

Solution:
When dividing the array no comparisons are done, we only need to look at the merging operations.

When merging two subarrays, we compare the "front" of each subarray, and insert the smallest value. For instance when merging [4,6] and [3], 4 and 3 are compared, and 3 is inserted. Since one subarray is now empty, no more comparisons need to happen (4 and 6 are then inserted without any new comparisons).

If we do this for all the merging operations we get:

$$
\begin{aligned}
[6] + [4] &= 1 \text{ comparison} \\
[4,6] + [3] &= 1 \text{ comparison} \\
[2] + [1] &= 1 \text{ comparison} \\
[5] + [1,2] &= 2 \text{ comparison} \\
[3,4,6] + [1,2,5] &= 5 \text{ comparison} \\
&= \mathbf{10 \text{ comparions}}
\end{aligned}
$$

Task 3 Recurrence Relations

a) Solve the following recurrences. Provide exact answers.

i. $T(n) = T(n+1) - 2^n \quad ; \quad T(1) = 1$

Solution:
$\boldsymbol{T(n) = 2^n - 1}$

$$
\begin{aligned}
T(n+1) &= T(n) + 2^n \\
T(n) &= T(n-1) + 2^{n-1} \\
&= \sum_{k=1}^{n} 2^{k-1} \\
&= \mathbf{2^n - 1}
\end{aligned}
$$

Task 3 Recurrence Relations

ii. $T(n) = 2 \cdot T(\lfloor n/2 \rfloor) + 2$ for $n > 1$; $T(1) = 0$

> **Tip:**
> The *floor* operator $\lfloor x \rfloor$ (or `floor(x)`) strips all decimals.
> For instance: $\lfloor 4.975 \rfloor = 4$ and $\lfloor 10/3 \rfloor = 3$.
>
> You also have the *ceiling* operator:
> $\lceil 4.975 \rceil = 5$ and $\lceil 10/3 \rceil = 4$

Solution:
$\boldsymbol{T(n) = 2n - 2}$

$$T(n) = \sum_{k=1}^{\log_2 n} 2^k$$
$$= \boldsymbol{2n - 2}$$

Task 3 Recurrence Relations

iii. $T(n) = T(n-1) + \log(\frac{n}{n-1})$ for $n > 1$; $T(1) = 0$

Solution:
$\boldsymbol{T(n) = \log(n)}$

Lets first simplify:

$$\log\left(\frac{n}{n-1}\right) = \log(n) - \log(n-1)$$

Using induction method:

$$\begin{aligned} T(n-1) &= \log(n-1): \\ T(n) &= \log(n-1) + \log(n) - \log(n-1) \\ &= \boldsymbol{\log(n)} \end{aligned}$$

Task 3 Recurrence Relations

iii. $T(n) = T(n-1) + \log(\frac{n}{n-1})$ for $n > 1$; $T(1) = 0$

Solution:
$\boldsymbol{T(n) = \log(n)}$

Lets first simplify:

$$\log\left(\frac{n}{n-1}\right) = \log(n) - \log(n-1)$$

Using iterative method:

$$\begin{aligned}
T(n) &= \log(n) - \log(n-1) + T(n-1) \\
&= \log(n) - \log(n-1) + \log(n-1) - \log(n-2) + T(n-2) \\
&= \log(n) - \log(n-2) + T(n-2) \\
&= \log(n) - \log(n-2) + \log(n-2) - \log(n-3) + T(n-3) \\
&= \log(n) - \log(n-3) + T(n-3) \\
&\dots \\
&= \log(n) - \log(n-i) + T(n-i)
\end{aligned}$$

we let $i = n-1$ and use that $T(n-n+1) = 0$ to get:

$$\begin{aligned}
T(n) &= \log(n) - \log(n-n+1) + T(n-n+1) \\
&= \boldsymbol{\log(n)}
\end{aligned}$$

Task 4 Local Minimum of a Grid Graph

Suppose that you are given an $n \times n$ grid graph G.

> An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. A node v of G is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. The labeling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v.

Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 total nodes.)

Solution:

Let B denote the set of nodes on the *border* of the grid G - i.e. the outermost rows and columns. Say that G has *Property (*)* if it contains a noce $v \notin B$ that is adjasent to a node in B and satisfies $v \prec B$ ($\prec =$ *preceeds*). Note that in a grid G with Property (*), the *global minimum* does not occur on the border B (since the global minimum is no larger than v, which is smaller than B) - hence G has at least one local minimum that does not occur on the border. We call such a local minimum an *internal local minimum.*

We now describe a recursive algorithm that takes a grid satisfying Property (*) and returns an internal local minimum, using $O(n)$ probes. At the end, we will describe how this can be easily converted into a solution for the overall problem.

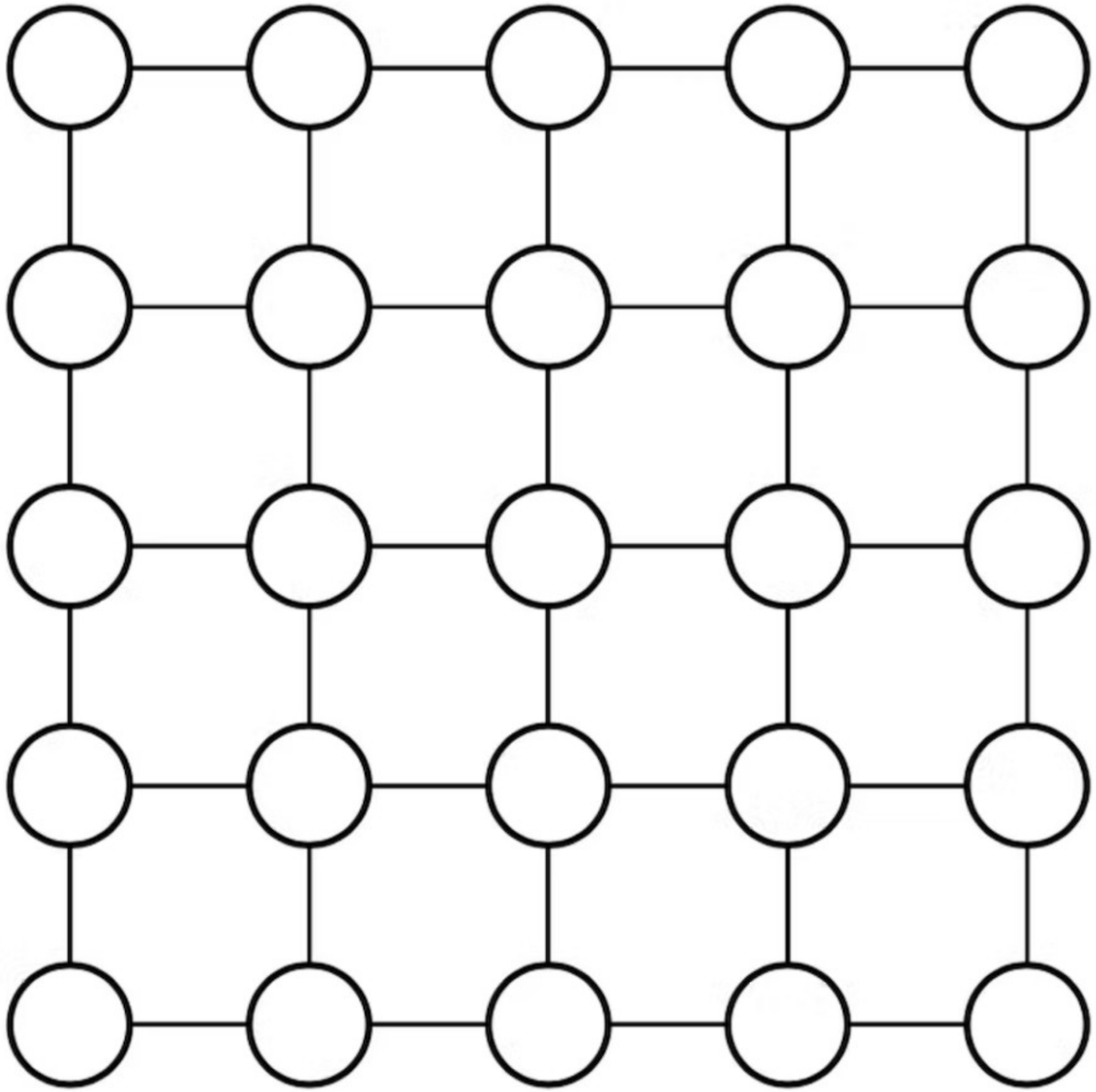
thus let G satisfy Property (*), and let $v \notin B$ be adjacent to a node in B and smaller than all nodes in B. Let C denote the union of the nodes in the middle row and middle column of G, not counting the nodes on the border. Let $S = B \cup C$; deleting S from G divides G into four sub-grids. Finally, let T be all nodes adjacent to S.

Using $O(n)$ probes, we find the node $u \in S \cup T$ of minimum value. We know that $u \notin B$, since $v \in S \cup T$ and $v \prec B$. Thus, we have two cases. If $u \in C$, then u is an internal local minimum, since all of the neighbors of u are in $S \cup T$, and u is smaller than all of them. Otherwise, $u \in T$. Let G' be the sub-grid containing u, together with the portions of S that border it. Now, G' satisfies Property (*), since u is adjacent to the border of G' and is smaller than all nodes on the border of G'. This, G' has an internal local minimum, which is also an internal local minimin of G. We call our algorithm recursively on G' to find such an internal local minimum.

If $T(n)$ denotes the number of probes needed by the algorithm to find an internal local minimum in a $n \times n$ grid, we have the recurrence $T(n) = O(n) + T(n/2)$ which solves to $T(n) = O(n)$.

Finally, we convert this into an algorithm to find a local minimum (not necessarily internal) of a grid G. Using $O(n)$ probes, we find the node v on the border B of minimum value. If v is a corner node, it is a local minimum and we are done. Otherwise, v has a unique neighbor u not on B. If $v \prec u$, then v is a local minimum and again we are done. Otherwise, G satisfies Property (*) (since u is smaller than every other node on B), and we can call the above algorithm (to find the internal local minimum with $O(n)$ probes).

A key part of the solution is realising that probing *O(n)* times, is not the same as saying *maximum n probes*. This is because, as we know, ***O(n)* probes** could actually be ***5n+10 probes*** (or any other linear function)



A key part of the solution is realising that probing *O(n)* times, is not the same as saying *maximum n probes*. This is because, as we know, ***O(n)* probes** could actually be ***5n+10 probes*** (or any other linear function)

The solution given here selects the Border as B

On B we find the node with the lowest value using O(4n) => O(n) probes, we call this node v

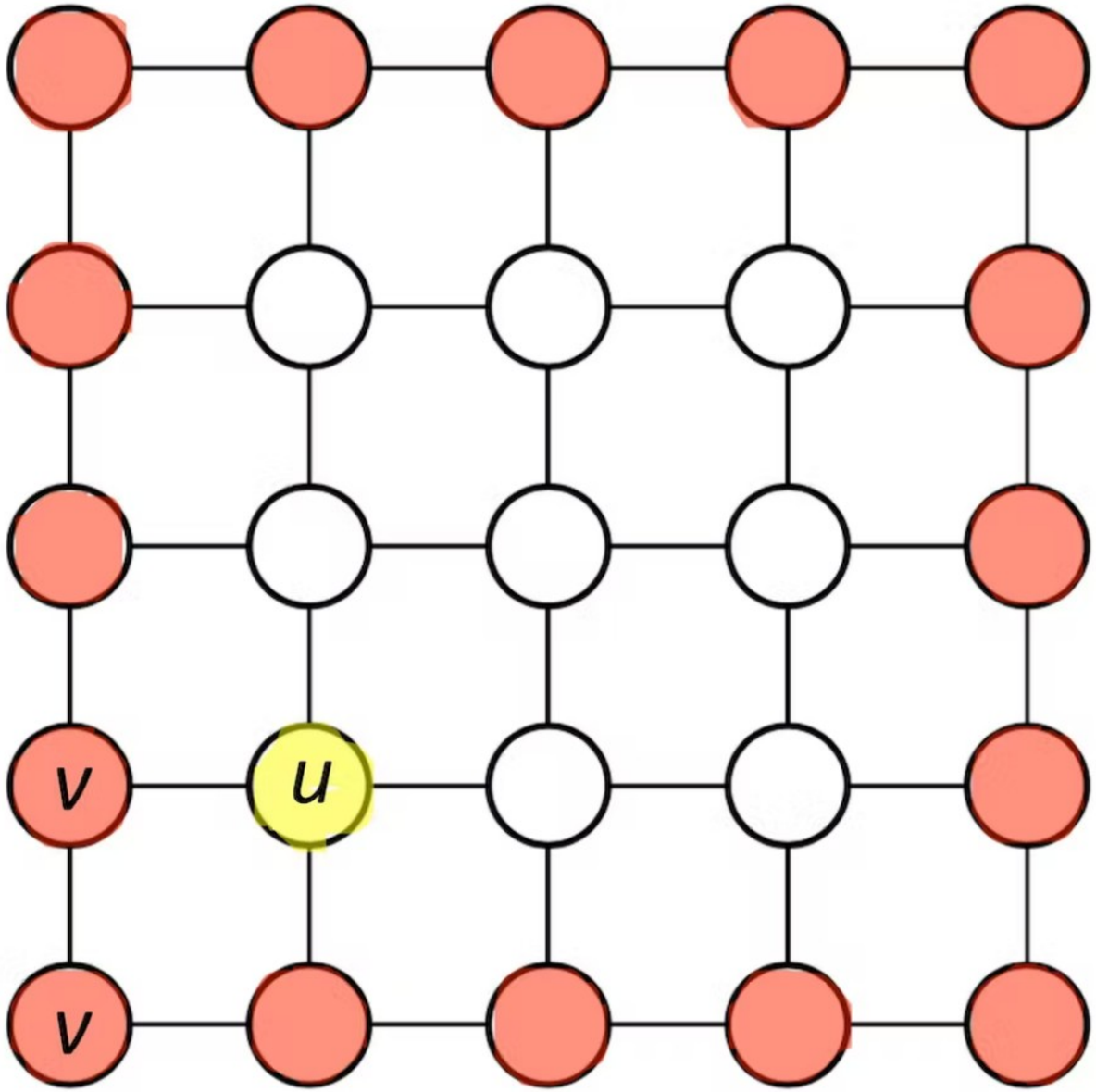
If v is a corner node, we already know it is a local minimum.

If not we have to check vs only neighbour that is not on the border.

If not we have to check vs only neighbour that is not on the border, and call this node *u*

If u is lager than v, then v is a local minimum

If u is smaller than v, we perform the algorithm described in the solution, which finds an internal local minimum with O(n) probes since we now know G satisfies Property (*)



NTNU
Kunnskap for en bedre verden

Part 2 Programming

is_smaller()

```python
def is_smaller(smaller: str, larger: str) -> bool:
    """Return True if 'smaller' is smallest, False otherwise"""
    if type(smaller) == type(larger):
        # Both are of the same type (number or letter) and can be compared directly
        return smaller < larger

    # We want numbers to count as "smaller" than letters
    # We know now they are of different types, so that means if 'smaller' is a number
    # then 'larger' must be a letter and we return 'True'.
    # And if 'smaller' is not a number, then 'larger' must be number. and we return 'False'
    return type(smaller) == int
```

merge()

```
def merge(L: list, R: list, array: list):
    l = 0  # L index
    r = 0  # R index
    for i in range(len(array)):

        R_is_empty = r == len(R)
        L_is_not_empty = l < len(L)

        if R_is_empty or (L_is_not_empty and is_smaller(L[l], R[r])):
            # Insert Element from L
            array[i] = L[l]
            l += 1
        else:
            # Insert Element from R
            array[i] = R[r]
            r += 1
```

merge_sort()

```
def merge_sort(array: list):
    if len(array) <= 1:
        # array of 0 or 1 elements is already sorted, nothing to be done
        return
    # Finding the mid of the array
    middle_index = len(array) // 2
    # Dividing the array elements into 2 halves
    L = array[:middle_index]
    R = array[middle_index:]
    # Sort each half
    merge_sort(L)
    merge_sort(R)
    # Merge the halves
    merge(L, R, array)
```

```python
def test_sorting(array: list):
    print(f"Original Array ({len(array)} elements): \n{array}")
    merge_sort(array)
    print(f"Sorted Array: \n{array}")

test_sorting(['c', 'A', 'b', 'C', 42, 'B', 123, 'a', 1])
```

```
Original Array (9 elements):
['c', 'A', 'b', 'C', 42, 'B', 123, 'a', 1]
Sorted Array:
[1, 42, 123, 'A', 'B', 'C', 'a', 'b', 'c']
```

```python
test_sorting([29, 'h', 9, 24, 'n', 'N', 'F', 58, 54, 'H', 's', 'r', 'U', 'K', 21, 56, 'r', 53, 'r', 'V', 32, 'A', 'g', 'P', 63, 'F'
```

```
Original Array (120 elements):
[29, 'h', 9, 24, 'n', 'N', 'F', 58, 54, 'H', 's', 'r', 'U', 'K', 21, 56, 'r', 53, 'r', 'V', 32, 'A', 'g', 'P', 63, 'F', 91, 65, 'j'
Sorted Array:
[1, 3, 7, 8, 9, 9, 11, 12, 12, 13, 14, 16, 21, 23, 24, 24, 28, 29, 32, 34, 36, 36, 39, 42, 46, 53, 53, 53, 54, 56, 56, 58, 59, 63,
```

NTNU
Kunnskap for en bedre verden

Assignment 4 Introduction

Chapter 5: Greedy Algorithms

Chapter 5 - Greedy Algorithms

> *An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution*

Goals 🏁

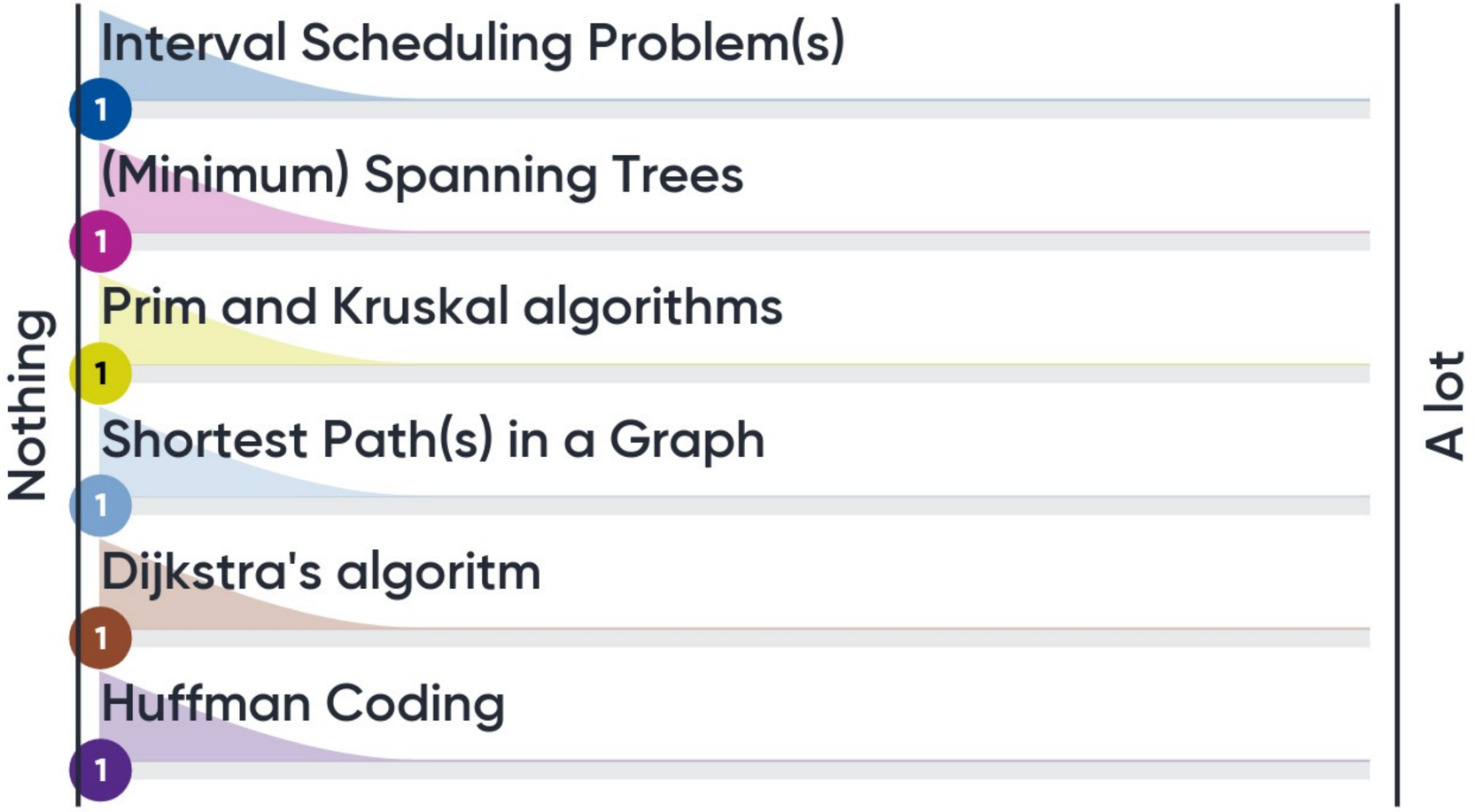
The student should be able to:

- Understand the different *Interval Scheduling* problems
- Understand the *Optimal Caching Problem*
- Understand the *Shortest Path Problem* and how *Dijkstra's algorithm* solves it
- Know what a *Minimum Spanning Tree* is and understand the *Minimum Spanning Tree Problem*
- Understand *Prim's* and *Kruskal's algorithm*
- Understand *Huffman* and *Huffman Codes*

1.1
1.3
1.2
1.5, 1.6, 2.1

How much do you know about

NTNU
Kunnskap for en bedre verden



Contents

Task 1 The interval scheduling problem

The following task will be based on the interval scheduling problem as explained in the book:

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
    Choose a request i from R that has the smallest finishing time
    Add request i to A
    Delete all requests from R that are not compatible with request
    i
EndWhile
Return the set A as the set of accepted requests
```

a) Explain in your own words how the algorithm described above returns an optimal set

b) Given the following table of activities:

Activity	U	V	W	X	Y	Z
Start time	3	2	5	5	10	1
Finishing time	6	4	8	9	12	5

After running the interval scheduling algorithm described above, which of the intervals will be number two in the return set A?

And what will the size of A be?

Why is this a Greedy algorithm?

NTNU
Kunnskap for en bedre verden

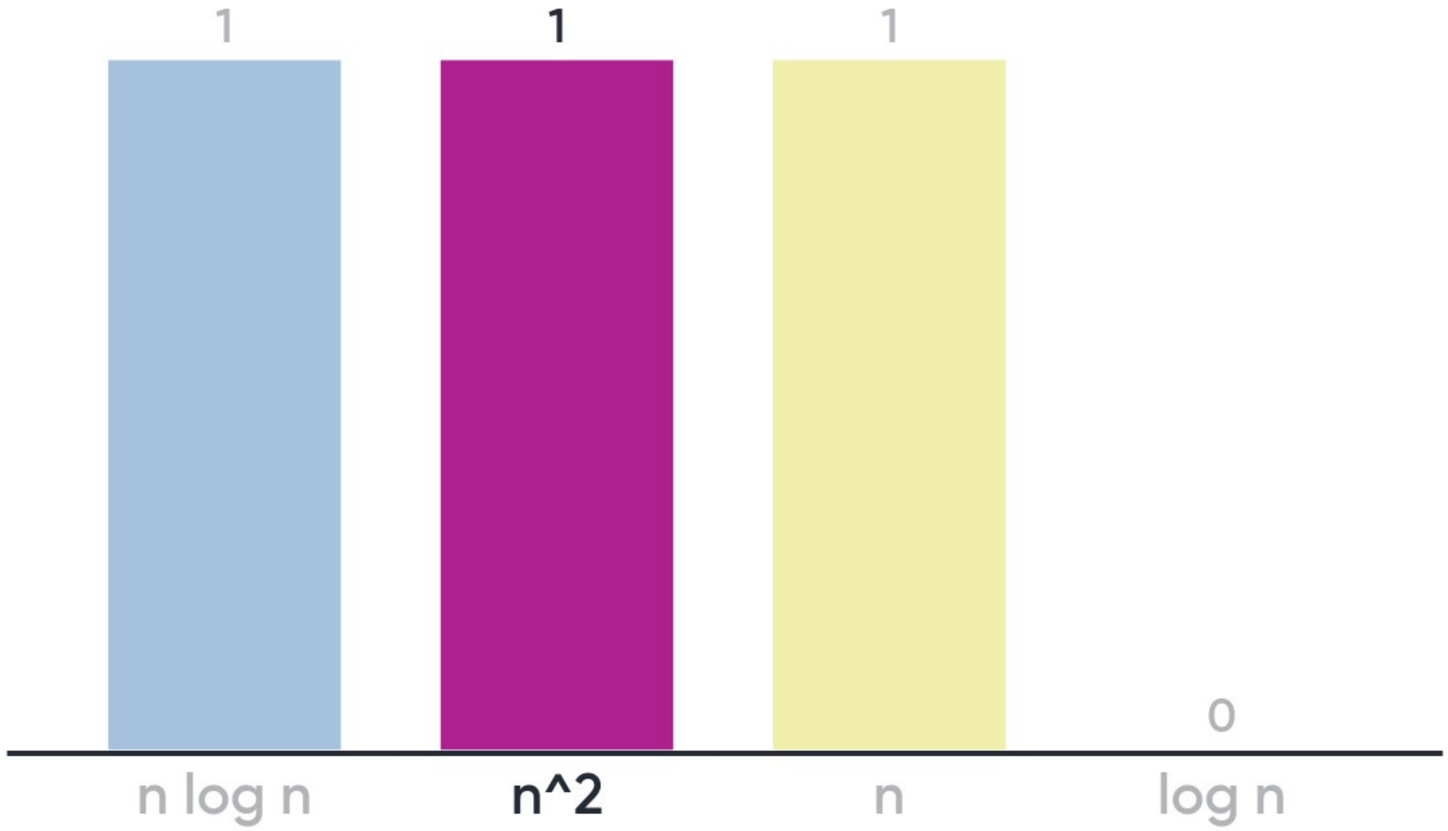
Because its greedy, chooses optimal solution at each step and builds upon that

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

NTNU
Kunnskap for en bedre verden

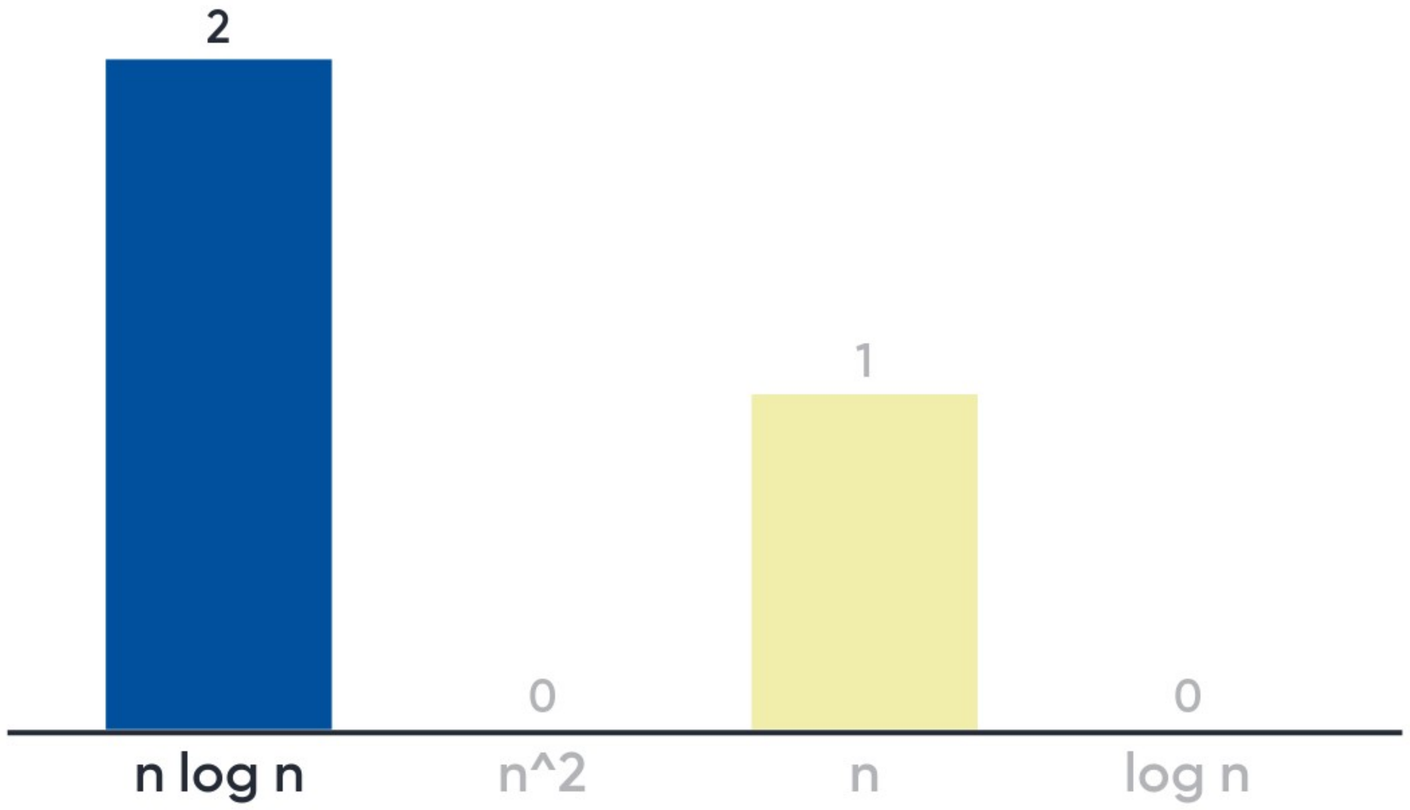
Interval Scheduling in Python

What is the complexity of my implementation of the Interval Scheduling algorithm?



What is the theoretical complexity of the Interval Scheduling algorithm?

NTNU
Kunnskap for en bedre verden



Task 2 Minimum spanning trees

a) We have an edge (u, v) that has a strictly lower weight than all the other edges in a connected graph. Will this edge be included in the minimum spanning tree? Explain why or why not.

b) Consider the following statement about Minimum Spanning Trees

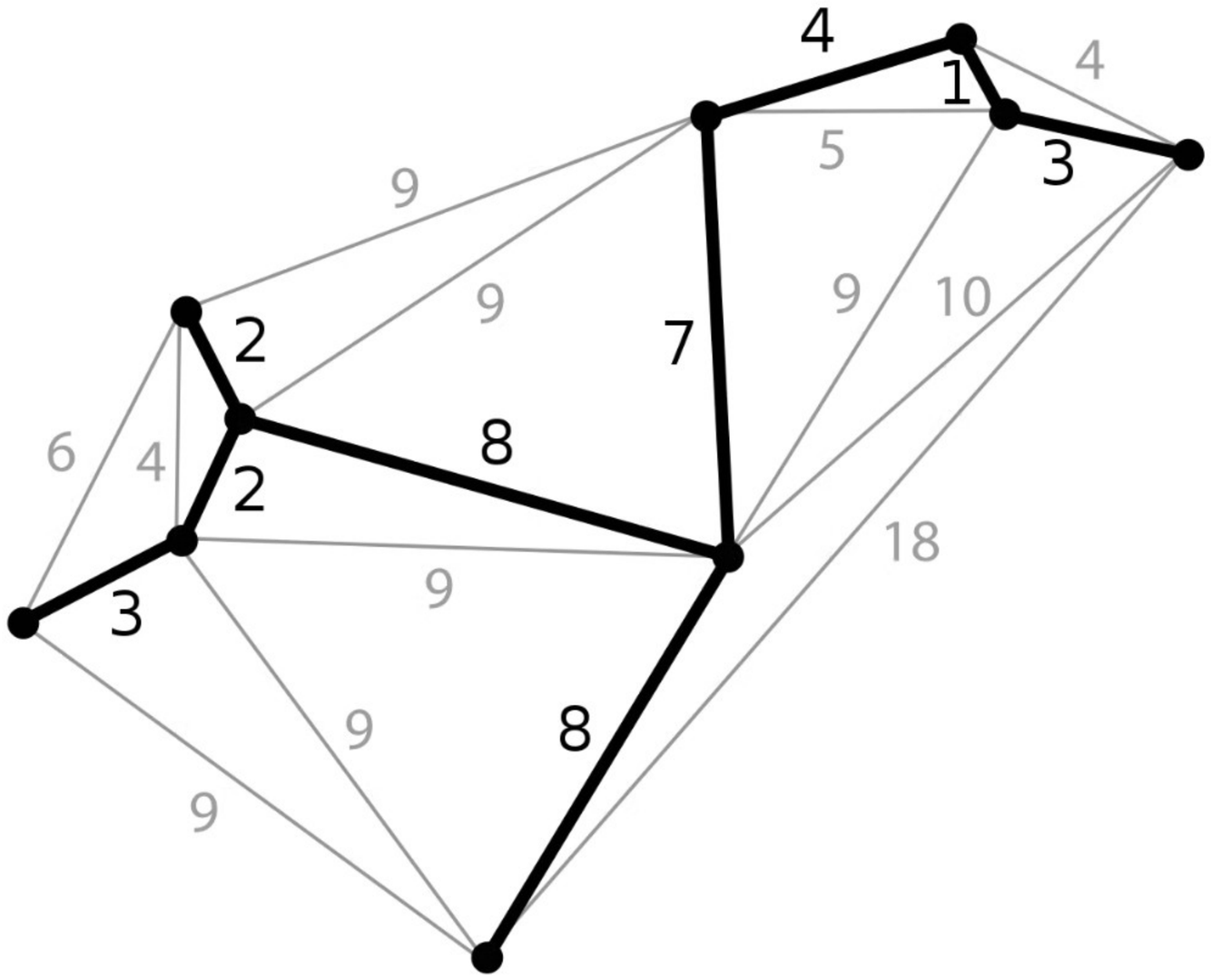
We have a weighted undirected graph $G = (V, E)$ where all of the edges are weighted differently:

If the nodes V can be divided into two disjoint sets X and Y , then the minimum spanning tree will include the edge between X and Y with the lowest weight.

Is this correct? Explain why or why not.

Minimum Spanning Tree

- **A tree** is an acyclic undirected graph
- Consider a connected undirected graph G
- **A spanning tree** $T \subseteq G$ is a tree which includes all nodes of G
- **A minimum spanning tree** is a spanning tree with the minimum possible total edge weight



Minimum Spanning Tree (black) of a graph G (gray)

How many edges E in a spanning tree $T \subseteq G$, where G has N nodes?

1

E = N

0

E = N log N

0

E = 2N

2

E = N - 1

What algorithms are used to find minimum spanning trees?

NTNU
Kunnskap for en bedre verden

Intervall scheduling?

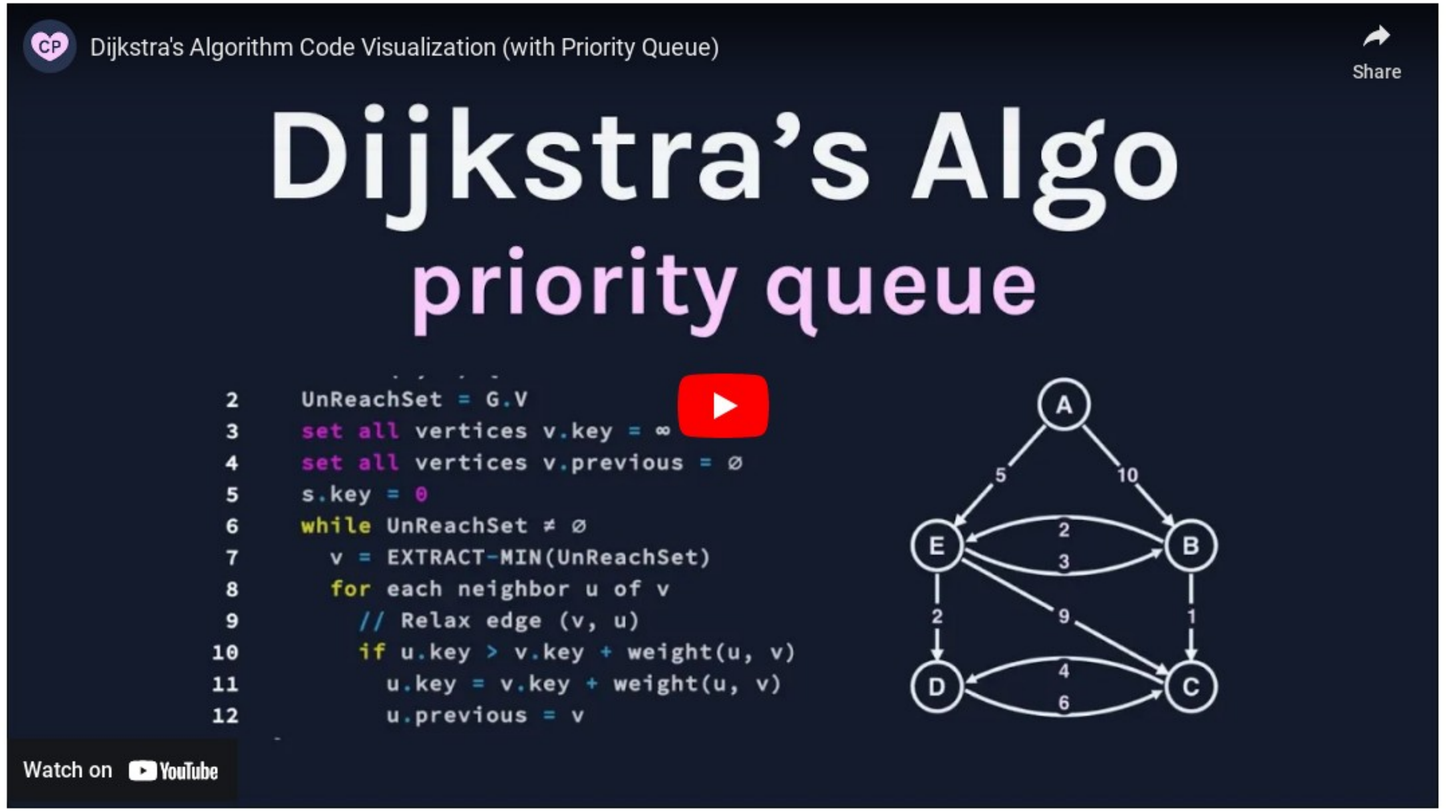
prims and kruskal

2

Task 3 Shortest path in a graph

a) Can we use Dijkstra's algorithm to solve the shortest path problem on an undirected graph? Explain why or why not.

b) Explain, in your own words, why Dijsktra's algorithm works


Dijkstra's Algorithm Code Visualization (with Priority Queue)
Share
Dijkstra's Algo
priority queue
UnReachSet = G.V
set all vertices v.key = ∞
set all vertices v.previous = ∅
s.key = 0
while UnReachSet ≠ ∅
v = EXTRACT-MIN(UnReachSet)
for each neighbor u of v
// Relax edge (v, u)
if u.key > v.key + weight(u, v)
u.key = v.key + weight(u, v)
u.previous = v
A
B
C
D
E
5
10
2
3
2
9
1
4
6
Watch on YouTube

Dijkstra finds the shortest path from

NTNU
Kunnskap for en bedre verden

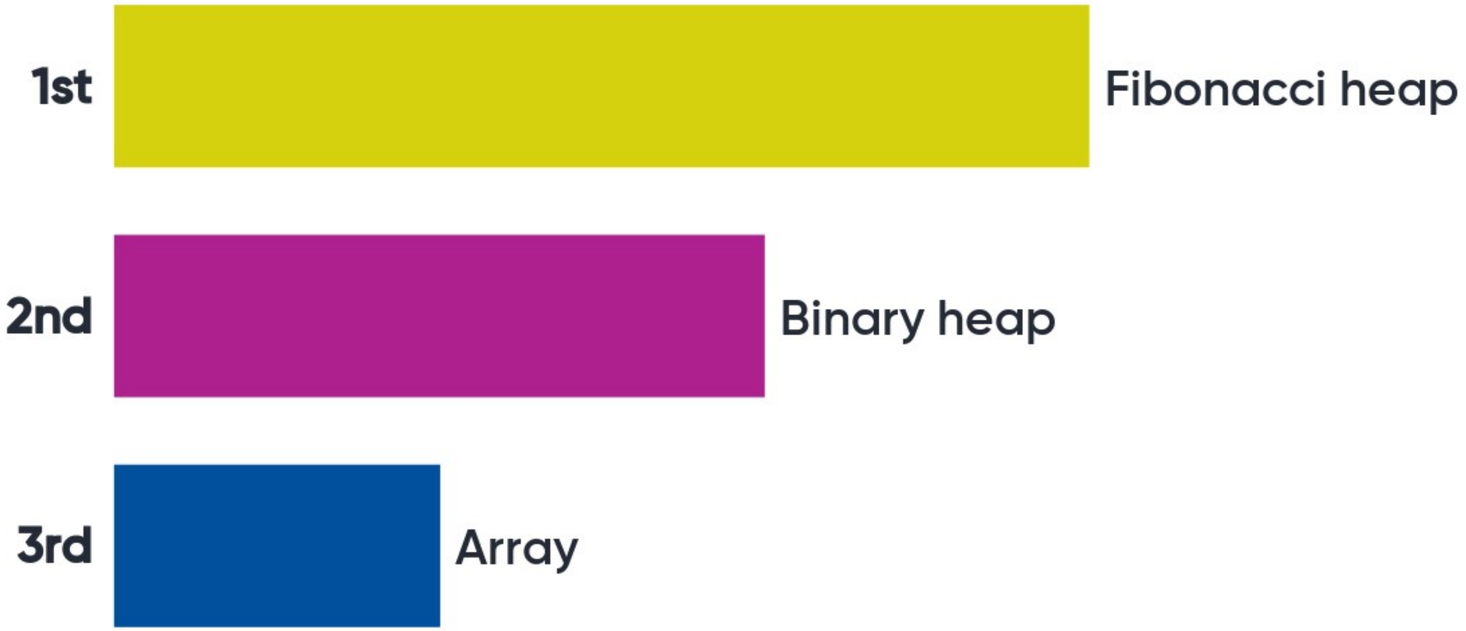
0
all nodes to a single end node

0
a single source to a single end node

2
a single source to all other nodes

0
all nodes to all other nodes

The complexity of Dijkstra depends on the data structure of the internal priority queue. Sort by complexity from low (1) to high (3)



Task 4 Video game design

A video game developer has approached you since they heard rumors that you are taking the course TDT4121, and you know a thing or two about algorithms. They are making a side-scrolling video

game where the main character moves from left to right, and he wants to add checkpoints to the game using an algorithm.

You can imagine the game as a flat straight line going from left to right, with obstacles placed along the way with varying distances between them. You want to add checkpoints so that each obstacle is within 400 pixels of a checkpoint, and you also want to add as few checkpoints as possible. Construct an efficient algorithm that places as few checkpoints as possible. Prove the correctness of your algorithm.

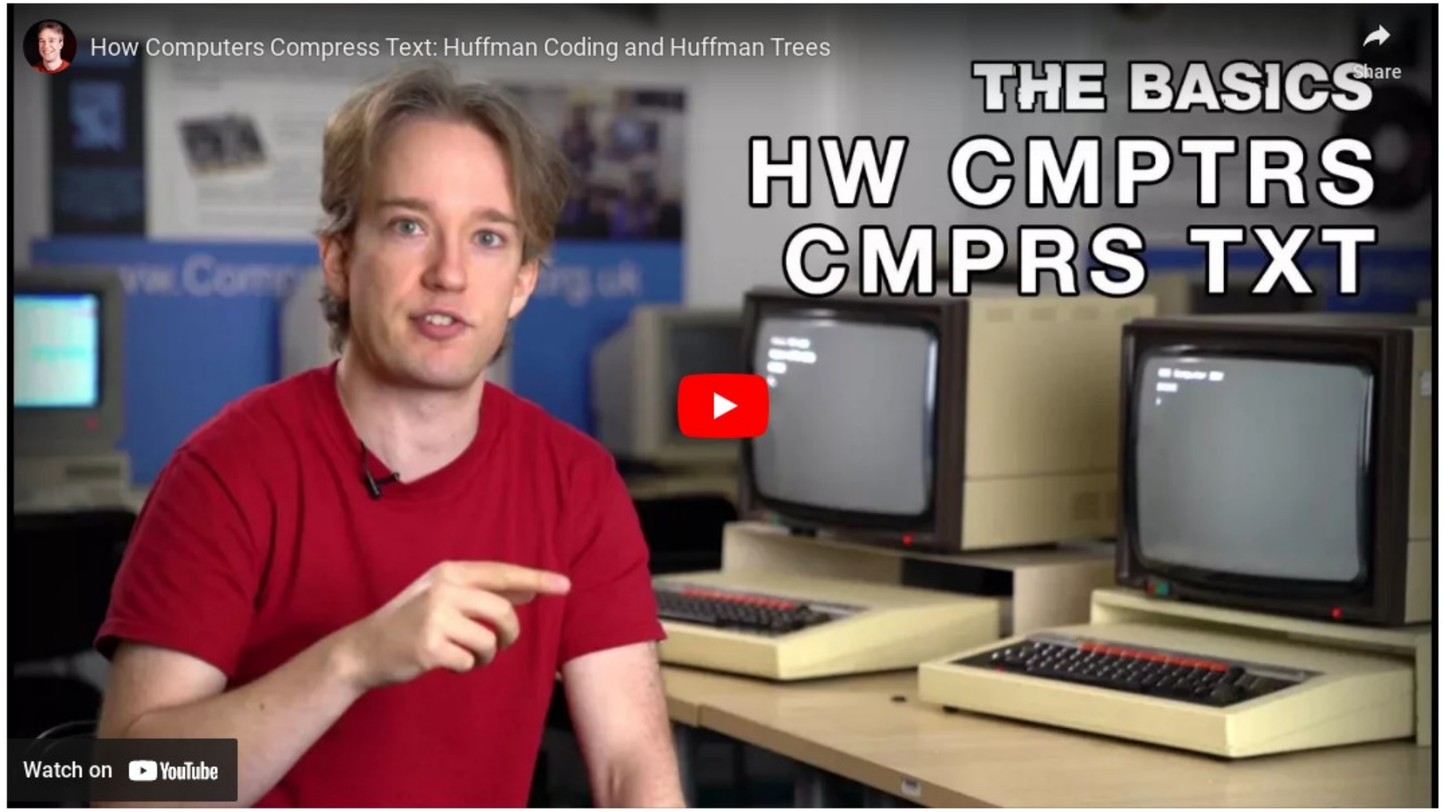
Task 5 Huffman coding

a) You are given a text file with the following frequencies:

Letter	W	X	Y	Z
Frequency	10	4	2	5

What will the Huffman code for Y be in this example? Show your steps

b) Provide a Huffman encoding for the phrase: "SHE SELLS SEASHELLS SHE SEES SEASHELLS THE SHELLS SHE SELLS ARE SEASHELLS SHE SEES THEM". Explain your answer by making a frequency table and drawing a Huffman tree, and then creating the codes based on the tree. You can ignore spaces.

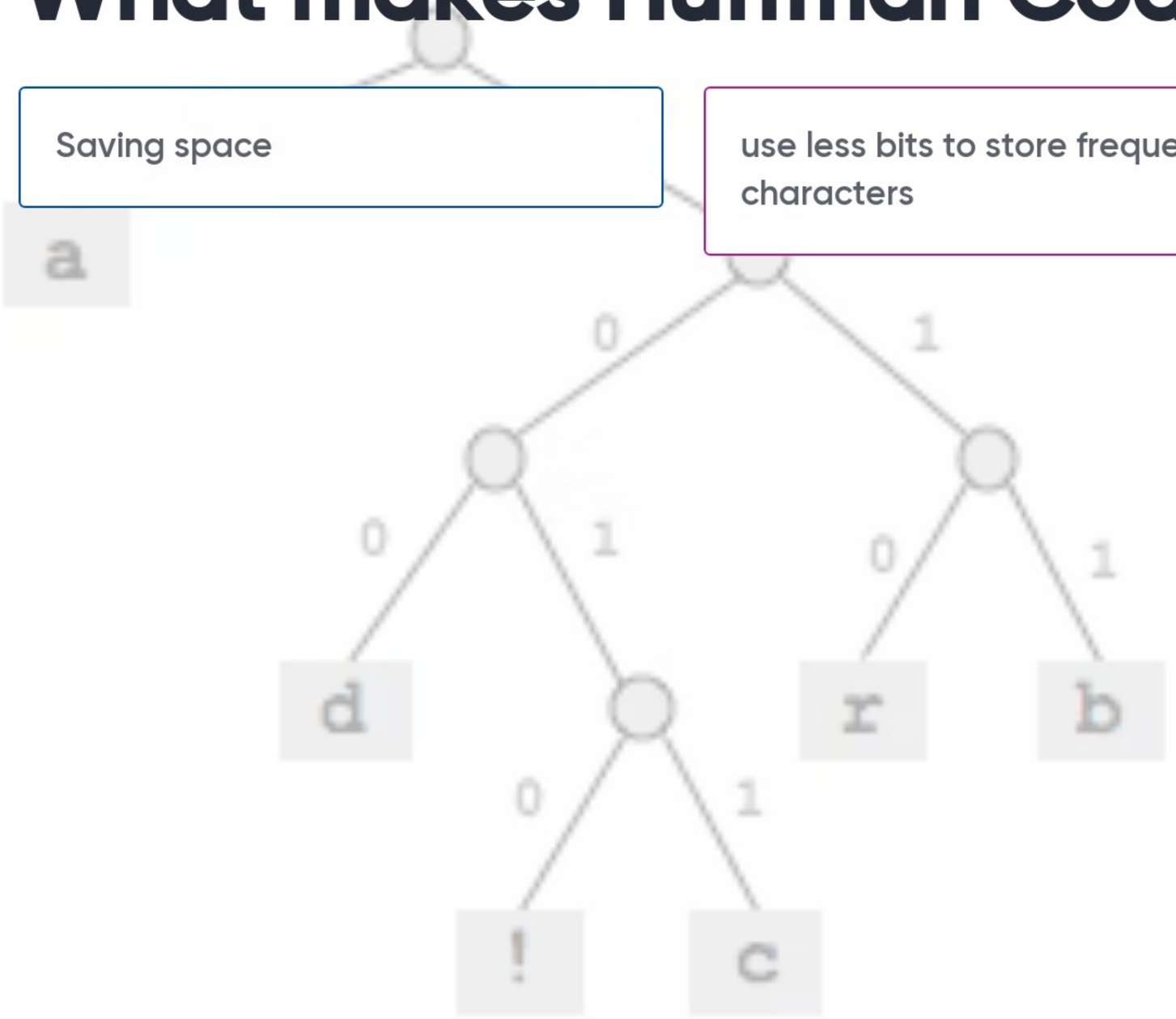

How Computers Compress Text: Huffman Coding and Huffman Trees
Share
THE BASICS
HW CMPTRS
CMPRS TXT
Watch on YouTube

What makes Huffman Coding efficient?

NTNU
Kunnskap for en bedre verden

Saving space

use less bits to store frequent characters



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

1 - Data structures set up

We want to sort our characters using a binary heap. You can implement a binary heap in Python using the heapq library. We want to create a HeapNode class that will work as the basis for storing our Huffman tree. To use our HeapNode class with the heapq library, we need to modify some of its built-in arithmetic operations. We want to change the < and = operations to compare the values of the nodes.

```python
class HeapNode:
    def __init__(self, character, value):
        # TODO: Initialize the HeapNode with a character and value
        # We also want to keep track of left_child and right_child,
        # so these should be initialized as None
        print("Example code")

    def __lt__(self, other):
        # TODO: check if other is a HeapNode object (return false if it isn't)
        # and return a lesser than (<) comparision between the two objects
        values
        print("Example code")

    def __eq__(self, other):
        # TODO: check if other is a HeapNode object (return false if it isn't)
        # and return an equals (==) comparision between the two objects values
        print("Example code")

    def __str__(self):
        # TODO Add a fitting __str__ for printing the Node, could be usefull
        # for testing along the way
        return "example code"

    def __repr__(self) -> str:
        return self.__str__()
```

[1] Python

NTNU
Kunnskap for en bedre verden

2 - Making the frequency heap

Let's start by constructing our frequency heap, we want to take a string as input and count the characters using a dictionary, we then want to input the characters into our heap using the heapq library.

```python
def make_frequency_heap(string: str) -> list:
    # Initialize freq dictionary and heap array
    freq = {}
    heap = []
    # TODO loop through the characters in the string and
    # insert them into the freq dictionary,
    # with the character being they key and the value being the frequency

    # TODO insert the values from the freq dictionary into the heap
    # using the HeapNode object and the heapq library

    return heap


teststring = "ABBCCCDDDD"
heap = make_frequency_heap(teststring)
for node in heap:
    print(node)
```

[2] Python

3 - Merging the codes

Next, we want to merge the characters with their frequencies together. Follow the instructions in the pseudocode bellow to merge the heap into a single Huffman tree

```python
def merge_code(heap: list) -> HeapNode:
    # TODO
    # While there is more than one node in the heap
    # Extract the two nodes with the lowest frequency letters from the heap
    (remember that the letter with the lowest frequency will always be at the
    top of the heap)
    # Create a new node that has the sum of the values of the two nodes as its
    value, and the two nodes as left and right child respectively
    # Push this new node into the heap

    # Return the root of the tree
    return heap[0]
```

[3] Python

4 - Traversing the tree

We've now made a program that can construct a Huffman tree using Huffman's algorithm. Now we want to traverse said tree and find out what the Huffman encoding is for each letter. We will do this recursively. We have created the main function `traverse_huffman()` to set up the variables for you. Your job is to finish the implementation of `traverse_huffman_recursive()`.

```python
def traverse_huffman(root: HeapNode) -> dict:
    # Stores the codes for each letter
    codes = {}
    # Keeps track of the current code
    current_code = ""
    # traverse recursively
    traverse_huffman_recursive(root, current_code, codes)
    # return finished encoding
    return codes


def traverse_huffman_recursive(node: HeapNode, current_code: str, codes: dict)
-> None:
    # TODO if there exists a character in the node,
    # append current_code as the value and the character
    # as the key in codes and return

    # TODO make the recursive calls, there should be two,
    # one for the left side of the tree and one for the right
    # When you traverse to the left, append 0 to the current code,
    # and 1 if you traverse to the right
    return
```

[4] Python

Running the program

Here we have a main function to run the whole program, use it to test if you get the correct output:

```
def main():
    text = "ABBBBCCCDDEEEEAAAEEBBBCC"
    heap = make_frequency_heap(text)
    root = merge_code(heap)
    encoding = traverse_huffman(root)
    print(encoding)


main()
```

[1] ✓ 0.4s Python

Expected output: `{'C': '00', 'E': '01', 'D': '100', 'A': '101', 'B': '11'}`

Ask me anything

NTNU
Kunnskap for en bedre verden

0 questions
0 upvotes