Algorithmen & Datenstrukturen essentials

Simon Sure

January 3, 2023

Contents

1 $\mathcal{O}$-notation & asymtotic growth 3
1.1 asymtotic growth . . . 3
1.2 cost model . . . 3
1.3 $\mathcal{O}$-notation . . . 4
1.4 Ω-notation . . . 4
1.5 Θ-notation . . . 4
1.6 Masther theorem . . . 4

2 Recursion, Recurrences, Induction, ... 4

3 Data Structures 5
3.1 Stack . . . 5
3.2 Queue . . . 5
3.3 Priority Queue/Heap . . . 6
3.4 Dictionaries . . . 6
3.4.1 search tree . . . 6
3.4.2 AVL tree . . . 7
3.5 Union Find . . . 8

4 Basic Algorithms 9
4.1 written multiplication . . . 9
4.2 pasture break . . . 9
4.3 star search . . . 9
4.4 Maximum Subarray Sum (MSS) . . . 9

5 Dynamic Programming 10
5.1 Fibonacci sequence . . . 11
5.2 longest increasing subsequence . . . 11
5.3 longest common subsequence . . . 12
5.4 Levenshtein distance . . . 12
5.5 subset sum . . . 12
5.6 knapsack problem . . . 13
5.7 chain matrix multiplication . . . 14

6 Sort and Search 15
6.1 search . . . 15
6.1.1 binary search . . . 15
6.1.2 Linear Search . . . 15
6.2 sort . . . 16
6.2.1 check whether sorted . . . 16
6.2.2 bubble sort . . . 16
6.2.3 selection sort . . . 16
6.2.4 insertion sort . . . 17
6.2.5 Heap Sort . . . 17

6.2.6 Merge Sort 18
6.2.7 Quicksort 18
6.2.8 algorithmic limit 19

7 Graphs **19**
7.1 Eulerian path/cycle 21
7.2 Hamiltonian path 22
7.3 topological sorting 22
7.4 depth first search 23
7.5 breadth first search 24
7.6 cheapest walks 25
7.6.1 topological order-based 26
7.6.2 Dijkstra 26
7.6.3 Bellman-Ford 27
7.7 shortest paths 28
7.7.1 Floyd-Warshall 29
7.7.2 Johnson 30
7.8 minimum span tree 31
7.8.1 Boruvka 31
7.8.2 Prim 32
7.8.3 Kruskal 32

1 $\mathcal{O}$-notation & asymtotic growth

1.1 asymtotic growth

Let $f, g : \mathbb{R}^+ \to \mathbb{R}^+$. We say that f grows asymtotically faster than g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

L'Hôpital's rule Assume:

- $f, g : \mathbb{R}^+ \to \mathbb{R}^+$ are differentiable
- $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$.
- for all $x \in \mathbb{R}^+ : g'(x) \neq 0$????????

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

1.2 cost model

To compare algorithms we consider correctness and runtime. Correctness can be (more or less) clearly identified. Runtime is impacted by

- input dependence (larger inputs - longer runtime)
- machine dependence (faster computer - shorter runtime)
- implementation dependence (choice of programming language)

To abstract, we only consider the asmyptotic growth of the runtime of an algorithm. We consider this abstract model of a computer:

- storage
  - input stored in first n memory cells
  - cells can be empty or storage an integer $\leq n^{100}$
  - cells are freely addressable
- processor
  - read/write memory cells
  - comparison of memory cells $(<, >, =, ...)$
  - basic calculations $(+, -, \cdot, ...)$

With this abstraction we treat $3 \cdot 3$ the same as $n^{100} \cdot n^{100}$.

And the runtime is understood as the number of elementary operations: runtime of the model := #elementary operations. The runtiem is a function of the input size.

Flaws with this method are:

- various memory storage units with different speeds exist
- reading/writing takes different time
- a processor may read/write multiple things simultaneously
- ...

For comparison with reality we consider: $C_1 \leq \frac{\text{runtime in the model}}{\text{runtiem in reality}} \leq C_2$. Those constants are unique for each machine and implementation.

1.3 $\mathcal{O}$-notation

We define $N := \{n_0, n_0+1, n_0+2, ...\} \subseteq \mathbb{N}, n_0 \in \mathbb{N}$. For $f : N \to \mathbb{R}^+$ we then define:

$$\mathcal{O}(f) := \{g : N \to \mathbb{R}^+ | \exists C > 0, \forall n \in N, g(n) \leq C \cdot f(n)\}$$

$\mathcal{O}$ is the degree/order of f. Instead of $g \in \mathcal{O}(f)$ we also write $g(n) \leq \mathcal{O}(f(n))$. So $g \in \mathcal{O}(f) \Leftrightarrow$ "$\frac{g}{f}$ is limited over N".

Consider:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \Longrightarrow g \notin \mathcal{O}(f) \text{ and } f \in \mathcal{O}(g)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R}^+ \Longrightarrow g \in \mathcal{O}(f) \text{ and } f \in \mathcal{O}(g)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Longrightarrow g \in \mathcal{O}(f) \text{ and } f \in \mathcal{O}(g)$$

For $f, g, h : N \to \mathbb{R}^+$, $f \leq \mathcal{O}(h)$ and $g \leq \mathcal{O}(h)$. Then:

- $c \cdot f \leq \mathcal{O}(h)$, $c \in \mathbb{R}$
- $f + g \leq \mathcal{O}(h)$

In asymptotic notation one can neglect the basis of a logarithm. Insetad of $\log_a b$ one only considers $\log b$.

1.4 Ω-notation

Let $f : N \to \mathbb{R}^+$.

Then: $\Omega(f) := \{g : N \to \mathbb{R}^+ | \exists C > 0, \forall n \in N, f(n) \leq C \cdot g(n)\}$

1.5 Θ-notation

Let $f : N \to \mathbb{R}^+$.

$\Theta(f) := \{g : N \to \mathbb{R}^+ | \exists C > 0, \exists D > 0, \forall n \in N, f(n) \leq C \cdot g(n) \leq D \cdot f(n)\}$.

This corresponds to $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$.

1.6 Masther theorem

Let $a, C > 0$ and $b \geq 0$ be constants, $T : \mathbb{N} \to \mathbb{R}^+$ such that for all $n \in \mathbb{N}$: $T(n) \leq a \cdot T(\frac{n}{2}) + C \cdot n^b$.

Then, for all $n = 2^k, k \in \mathbb{N}$:

- $b > \log_2 a \Rightarrow T(n) \leq \mathcal{O}(n^b)$
- $b = \log_2 a \Rightarrow T(n) \leq \mathcal{O}(n^{\log_2 a} \cdot \log n)$
- $b < \log_2 a \Rightarrow T(n) \leq \mathcal{O}(n^{\log_2 a})$

If T is increasing, $n = 2^k$ can be dropped. If the definition of $T(n)$ holds with an equality, we can replace $\mathcal{O}$ with Θ.

2 Recursion, Recurrences, Induction, ...

recurrence (runtime analysis) We often consider recursions (divide-and-conquer, ...). When considering the runtime of a recusrive algorithm, one always get a recurrence. First, one always needs to know a base case (mostly $T(1)$). Then, one can telescope to get to a final expression. Notice, that telescoping is our way to get to a non-recurrent formula, but it is not a proof.

Once, such a non-recurrent formula for the runtime has been found, one still needs to prove it by mathematical induction if a proof is required.

invariants (correctness) Furthermore, when proving the correctness of an algorithm, we may also encounter induction. When the algorithm terminates we want some statement (postcondition) to validate what the algorithm is supposed to do. To achieve such a statement after termination, we can consider an invariant. Such an invariant holds before and after each step of the algorithm (not necessarily/likely not during the step). This invariant then needs to imply the postcondition/intended semantics of the algorithm once all steps have been completed. For an invariant $A[i]$ one must prove:

1. $A[1]$ is valid at the beginning.
2. $A[i] \Rightarrow A[i+1]$
3. The invariance for some $i = n$ with the remaining element(s) make up the intended solution of the algorithm.

This is very similar/almost identical to a general proof by mathematical induction.

Mathematical induction We intend to prove that some statement $A(k)$ is valid for all $k \in \mathbb{N}$. We proceed with three steps:

1. *Base Case*: We must show that $A(1)$ holds.
2. *Induction Hypothesis*: We assume that the property $A(k)$ holds for some $k \in \mathbb{N}$: $A(k)$ is valid.
3. *Induction Step*: We show using the induction hypothesis that $A(k+1)$ must hold.

Finally, we might give a short conclusion.

3 Data Structures

name	runtime	storage
stack	$\mathcal{O}(1)$	$\mathcal{O}(n)$
queue	$\mathcal{O}(1)$	$\mathcal{O}(n)$
priority queue/heap	$\mathcal{O}(\log n)$ (creation: $\mathcal{O}(n \log n)$)	$\mathcal{O}(n)$
dictionary (AVL)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Figure 1: data structures, runtime and storage

3.1 Stack

Operations:

- push(x,S) - puts x on the stack S
- pop(S) - removes the top element of the stack S and returns it
- top(S) - returns the top element of the stack S

A linked linked list could be used as the data structure for this ADT. All operations in $\Theta(1)$. Storage for n elements is $\Theta(n)$.

3.2 Queue

Operations:

- enqueue(x,Q) - adds x to the end of the queue
- dequeue(Q) - returns and removes the front element of the queue

A doubly-linked list could be used as the data structure for this ADT. All operations in $\Theta(1)$. Storage for n elements is $\Theta(n)$.

3.3 Priority Queue/Heap

Operations:

- insert(x,P) - adds x to the priority queue P
- extractMax(P) - returns and removes the maximum element of P

A Max Heap would be used as the data structure for this ADT. The runtime for insert/extract then is $\mathcal{O}(\log n)$. Thats because adding a node requires it to trickle down. And removing the root requires 'tricke-up'.

A Max heap can be understood as a binary tree with the Heap Condition: The key of all nodes mus be greater (or equal) then the keys of its children. We also require that the tree is dense (maximum depth is $\lfloor \log_2 n \rfloor$, number of leaves $\lceil \frac{n}{2} \rceil$). When understood as an array, the Heap Condition is $\forall k \in \{1, ..., i\}$: $2k \leq i \Rightarrow A[2k] \leq A[k]$ and $2k+1 \leq i \Rightarrow A[2k+1] \leq A[k]$.

First, when inserting an element into the heap, we do so by appending it to the array or adding it to the bottom right of the tree. Then, we must 'trickle-up'. Do so so, we do a trickle-step on the parent. And we continue doing trickle-steps on the respective parents until some trickle step does not change anything or we reach the root.

Second, when removing the max value, we remove the max value/replace it with the last element in the array/the bottom right element in the tree. Then, we need to 'trickle-down' by doing trickle-steps on the root and then on the child where the respective element trickled too. We do so until we reached the bottom of the tree or a trickle step does not change anything.

Both, insert and remove take at most $\lfloor \log_2 i \rfloor - 1$ changes and $2(lfloor \log_2 i \rfloor - 1)$ comparisons, leading to $\mathcal{O}(\log n)$.

A trickle step for some node n with children l and r means, we compare n with l and r and switch it with the largest child, if larger than n.

Furthermore, we notice that we can generate a Max-Heap from an unsorted array in $\mathcal{O}(n \log n)$. That is, because we can let all elements with indices $i = \lfloor \frac{n}{2} \rfloor ...1$ trickle down.

Fibonacci HEAPS!!!!!!!!

3.4 Dictionaries

Operations:

- serach(x,W) - true if x is in W
- insert(x,W) - inserts x into W if not already containing
- remove(x,W) - removes x from W if contained

With known data types, we can get those runtimes

data type	search	insert	remove
sorted array	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n + 1 + n) = \mathcal{O}(n)$	as insert
linked list	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
max heap	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Figure 2: trivial dictionary implementaions, runtimes

The idea to improve runtime is to use trees.

3.4.1 search tree

A search tree is based on he order in which the elements would be given in an array. Generally, from some leave, all leaves must be smaller and all leaves to the right must be bigger.

A tree can be modeled as a linked data structure. In comparison to a Linked List, one now has two references - one for left and one for right.

Search is adapted Binary Search and has runtime $\mathcal{O}(h)$, h being the height of the tree. To insert an elelement not already contained, one first searches its position, which must be at the bottom of the tree. Then, one can just add a reference. This naturally has runtime $\mathcal{O}(h)$. To remove an element, we first search it and then distinguish three cases.

- Case 1: The to-be-removed element does not have any children. We may just remove the reference to that element. ($\mathcal{O}(h+1)$)
- Case 2: The to-be-removed element has one child. We may change to reference to the to-be-removed element to the one child. ($\mathcal{O}(h+1)$)
- Case 3: The to-be-removed element has two children. This element has to be replaced with the next larger or smaller element. The next larger element (considered here) is called symmetric successor. It may be found by considering the right child and then going to the left until there is a null reference. Then, one sets the current element value to the symmetric successor value and recursively removes the symmetric successor (in $\mathcal{O}(1)$, because 0 or 1 child). ($\mathcal{O}(h+h+1)$)

3.4.2 AVL tree

To improve runtime, the idea is to always keep the tree dense so that $h \leq \mathcal{O}(\log n)$. Keeping the tree perfectly dense is computationally to expensive to be feasible. Instead we consider AVL trees, which have the condition $|h(l) - h(r)| \leq 1$ - for all nodes the height of both subtrees can differ at most by one.

prove of $\log n$ height A binary tree with n keys/inner nodes has $n+1$ leaves (proofs by induction). We define "$MB(h) :=$ minimum number of leaves of some AVL tree with height h". Then: $n \geq MB(h) - 1$.

One subtree must have height $h-1$ so that we have height h. The other subtree must have height of at least $h-2$ according to the ALV condition. We get: $MB(h) = MB(h-1) + MB(h-2)$. We see that $MB(h) = Fib(h+2)$.

We get: $n \geq MB(h-1) \rightarrow n \geq Fib(h+2) - 1$. Without proof we accept $Fib(h) = \Theta((\frac{1+\sqrt{5}}{2})^h) \approx \Theta(1.6^h)$. We then get: $h \leq 1.44... \log_2 n$, which confirms that the height has an bound of $\mathcal{O}(\log n)$.

operations Each node gets the property balance: "$balance(p) =$ height right - height left". We insert r as a left (right analogously) child of u. Three cases:

- $bal(u) = -1$: impossible
- $bal(u) = 0$: set $bal(u) = -1$ and $bal(r) = 0$. Call $UPIN(u)$, because u's subtree has grown
- $bal(u) = 1$: set $bal(u) = 0$ and $bal(r) = 0$.

For $UPIN(u)$, we have three invariants:

- new element in subtree from u with increased size by 1
- $bal(u) \neq 0$
- u has a successor w

If u does not have a successor, because its the root, we are done. We assume again that u is left child of w (other case analogously). Three cases:

- $bal(w) = 1$: set $bal(w) = 0$
- $bal(w) = 0$: set $bal(w) = -1$ and call $UPIN(w)$

- $bal(w) = -1$: we must repair the AVL tree. Consider two cases:
  - $bal(u) = -1$: right-rotation of w. Right child of u becomes left whild of w. Right child of u becomes w. Parent of w is redirected to u.
  - $bal(u) = 1$: left-rotation of u. right-rotation of w.

Removing is similar. If we remove r, being the left/right chilf of u, we first check, whether the height of u changed and update the corresponding balance of u. If u's balance now would be ± 2, one must call $UPIN(x)$ with x as the right/left child of u. If u's new balance is 0, call $UPIN(0)$.

Notice that if we check $bal(u)$ during $UPIN$, it may be that $bal(u) = 0$. It then does not matter whether one does a single or double rotation.

3.5 Union Find

The union find abstract data structure is designed for use with the (efficient) Kruskal algorithm. It is supposed to track connected components for some graph $G = (V, E)$. Specifically, it models $G_u = (V, F)$. We have those operations:

- make(V) - creates the data structure for F empty
- same(u,v) - tests whether u and v are in the same connected component
- union(u,v) - unions the connected components of u and v (adding edge uv to G$_u$)

The data structure contains an array 'rep[v]', which stores a unique representative of the connected component of v: rep[u]=rep[v] $\Leftrightarrow$ ZHK(u)=ZHK(v).

For make and same we have trivial implementations:

```
make(V):
    FOR all v in V:
        rep[v] = v

same(u,v):
    test rep[u]==rep[v]
```

Trivially, we can implement union(u,v) like this:

```
union(u,v):
    FOR x in V (u last):
        IF rep[x]==rep[u]:
            rep[x]=rep[v]
```

Obviously $\mathcal{O}(n)$. To improve runtime, the ADT also maintains 'members[i]', which stores a list of all vertices in the connected component with the identifier i. We get:

```
union(u,v):
    FOR x in members[rep[u]]:
        rep[x]=rep[v]
        members[rep[v]].add(x)
```

Obviously, $\mathcal{O}(|ZHK(u)|)$. In the worst case, we have to reassign every connected component except one. For i increasing with every FOR iteration: $|ZHK(u)| = i \Rightarrow \Theta(i)$ per iteration. For all iterations: $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$.

To improve, we always chose to reassign the smaller connected component and get $\Theta(\min\{|ZHK(u)|, |ZHK(v)|\})$. For one iteration, we hence have $\mathcal{O}(\frac{n}{2}) = \mathcal{O}(n)$. This would not yield any improvement. But we do an amortized analysis for Kruskal's algorithm.

4 Basic Algorithms

name	runtime
written multiplication	$\mathcal{O}()$
pasture break	$\mathcal{O}()$
star search	$\mathcal{O}()$
maximum subarray sum	$\mathcal{O}(n)$

Figure 3: basic algorithms, runtimes

4.1 written multiplication

Usual multiplication of two numbers $a_1a_0 \cdot b_1b_0$ happens like this: $a_0b_0 + 10a_1b_0 + 100a_0b_1 + 1000a_1b_1$. We get the recurrence $T(n) = 4 \cdot T(n/2) + c \leq \mathcal{O}(n^2)$.

The idea is to compute some intermediate products and reuse those to do less calculations:

$$\begin{aligned} x &= a_0b_0 \\ y &= a_1b_1 \\ z &= -(a_1 - a_0)(b_1 - b_0) \end{aligned}$$

Then, we may compute the product as $x + 100y + 10(x + y) + 10z$. That can be easily verified. We then have the recurrence $T(n) = 3 \cdot T(n/2) + c \leq \mathcal{O}(n^{\log_2 3})$. As $\log_2 3 < 2$, this algorithm is faster.

4.2 pasture break

Probably not relevant.

4.3 star search

Probably not relevant./Quite trivial after the complete semester.

4.4 Maximum Subarray Sum (MSS)

As input we have $a_1, ..., a_n \in \mathbb{Z}^+$. As output we want the maximum possible sum $S^* = a_i + a_{i+1} + ... + a_j$ with $i, j \in \{1, ..., n\}$ and $i < j$. If all numbers are negative, then the output should be $S^* = 0$.

naive algorithm One just computes all possible subsums. That takes $\Theta(n^3)$ additions.

naive algorithm 2 This can be improved by still computing all subsums but reusing already computed subsums insetad of recomputing everything. This brings one to $\Theta(n^2)$.

recursive algorithm We divide-and-conquer in the middle and have three options:

- MSS entirely in left half: $j \leq \frac{n}{2}$
- MSS entire in right half: $i > \frac{n}{2}$
- MSS has elements in both halfs: $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$

The algorithm computes (1) the MSS of both halfs recursively, (2) computes S_{ij} with $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$, and (3) outputs the biggest MSS.

For the second step, we have $\max_{i \leq \frac{n}{2}, j > \frac{n}{2}} S_{i,j} = \max_{i \leq \frac{n}{2}} S_{i,\frac{n}{2}} + \max_{j > \frac{n}{2}} S_{\frac{n}{2}+1,j}$. As each half takes $(\frac{n}{2} - 1)$ addtions, we totally require $n - 1$ additions.

Recurrence for the total algorithm: $A(n) = 2 \cdot A(\frac{n}{2}) + n - 1$. For analysis we know that $A(1) = 0$. From table consideration we get $A(2^k) = k \cdot 2^k - (2^k - 1) = (k-1)2^k + 1 \leq k \cdot 2^k = n \log n$. Thus, $\Theta(n \log n)$ for $n \geq 2$.

$$\mathcal{O}(n \log n)$$

recursive algorithm 2 We separate the last element and only look at those beforehand. We consider the edge maxima $R_j := S_{ij}$ $(i \leq j)$ - maximum sum of subsequent array elements up to j: $R_j = \left\{\begin{smallmatrix} R_{j-1}+a_j \text{ if } R_{j-1}>0 \\ a_j \text{ otherwise}\end{smallmatrix}\right.$. The MSS S^* of the array then is the max value of all $R_1, ..., R_n$.

$$\Theta(n)$$

algorithmic limit No algorithm can outperfrom $\mathcal{O}(n)$.

Proof. (by Contradiction)
Thus, the algorithm can only make $< n$ read operations for all inputs $a_1, ..., a_n$. $\Rightarrow \geq 1$ element a_i is not read. $\Rightarrow$ The output remains unchanged if a_i is changed. $\Rightarrow$ The algorithm may not be correct because the cases that $a_i = \pm \sum_{j \neq i} |a_j|$ can not be distinguished. (Once, a_i must (not) be contained). $\square$

5 Dynamic Programming

Dynamic programming is about efficiently computing recurrences/induction. It can often be used for probems with some sort of quality measure. The general procedure is:

1. Design of recurrence/induction
2. Defining the DP table (dimensions and initialization)
3. Filling the table bottom-up
4. Identifying the soluting (by backtracking, if required)

The hardest part is to figure out an invariant $dp()$, which can be compute for various i by relying on $dp(j)$ with $j < i$, which have been computed before.

Before, we have already seen an algorithm for the Maximum Subarray Sum. That also is a DP algorithm, as we continuously compute edge maxima for an increasing i. The solution, then can be read/extracted from all $dp(i)$/all randmaxima.

name	runtime
Fibonacci	$\mathcal{O}(n)$
longest increasing subsequence	$\mathcal{O}(n \log n)$
longest common subsequence	$\mathcal{O}(nm)$
Levenshtein distance	$\mathcal{O}(nm)$
subset sum	$\mathcal{O}(nb)$, $b = 2^n$ (exponential) $\vee\, b = n^c$ (polynomial)
knapsack	$\mathcal{O}(nW)$, $\mathcal{O}(nV)$, approximation: $\mathcal{O}(n^3 \frac{1}{\epsilon}) = \mathcal{O}(n^2 \frac{v_{\max}}{k})$
chain matrix multiplication	$\mathcal{O}(n^3)$

Figure 4: dynamic programming algorithms, runtimes

5.1 Fibonacci sequence

Recursive definition: $F_0 = 0, F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ $(n \geq 2)$. Just writing this recursion as a function will lead to exponential runtime.

Algorithm 1 FibonacciTopDownMemorization

```
FibonacciTopDownMemorization(n):
    if n-th saved: return memo[n]
    if n=0: f=0
    if n<=2: f=1
    else: f=FibonacciTopDownMemorization(n-1)+
       FibonacciTopDownMemorization(n-2)
    memo[n]=f
    return f
```

But even simpler is a bottom-up computation.

Algorithm 2 FibonacciBottomUp

```
FibonacciBottomUp(n):
    F[0]=0
    F[1]=1
    for i=2...n:
        F[i]=F[i-1]+F[i-2]
    return F[n]
```

The array F is the DP table here, which we fill $0 \to n$ with $0, 1$ being the initialization. Thus, we get runtime $\mathcal{O}(n)$, significantly better than the exponential runtime of the initial algorithm.

This algorithm could be further optimized for storage by only storing the last two numbers and, thus, only requirying $\mathcal{O}(1)$ storage.

$$\mathcal{O}(n)$$

5.2 longest increasing subsequence

As input we have some array. The output should be longest increasing subsequence in that array. As the invariant (L[i]) we have: For each subsequence length, the last element of the subsequence ending in the smallest element and its predecessor.

During each iteration, the algorithm distinguishes two cases:

- $A[i]$ continues $L[i-1]$ $(A[i] > L[i-1])$: $L[i] = A[i]$ and set $L[i]$'s predecessor to the current $L[i-1]$.
- Otherwise: Find k with $L[k] > A[i]$ and $L[k-1] < A[i]$. Set $L[k] = A[i]$ and note $L[k-1]$ as the new $L[k]$'s predecessor.

So, for each step we need to do some work. In the first case that is constant. In the second case, we need to find k in the sorted $L[]$-array and then do some constant work. Thus: $\sum_{i=0}^{n} a \cdot \log i \leq \mathcal{O}(n \log n)$.

$$\mathcal{O}(n \log n)$$

5.3 longest common subsequence

We have word A and word B as arrays of chars. We consider $LGS(n,m)$ as the longest common subsequence of $A[1...n]$ and $B[1...m]$. To compute $LGS(n,m)$, there are four cases:

- $^{X}_{-} \Rightarrow LGS(n,m) = LGS(n-1,m)$
- $^{-}_{X} \Rightarrow LGS(n,m) = LGS(n,m-1)$
- $^{X}_{Y}, X \neq Y \Rightarrow LGS(n,m) = LGS(n-1,m-1)$
- $^{X}_{X} \Rightarrow LGS(n,m) = LGS(n-1,m-1)+1$

However, as we do not know which option is valid in advance, the recurrence must consider all possibilities. Because we seek the longest common subsequence, we simply have to take the maximum value: $LGS(i,j) = \max\{LGS(i-1,j), LGS(i,j-1), LGS(i-1,j-1)(+1)\}$. The (+1) is only considered if $A[i] = B[j]$.

This recurrence can be implemented bottom up using a DP table with a DP algorithm. The DP table is 2-dimensional with size $A.length+1 \times B.length+1$. We initialize $DP[0,x] = 0 = DP[0,y]$ with x, y arbitrary values as long as still bound to the table. Then, we can fill the table from top left to bottom right.

By remembering the predecessor of each value, we can simply do backtracking to determine the actual longest common subsequence and not only its length.

The computation of each table entry terminates in constant time. But we must fill nm entries. Therefore, we have $\mathcal{O}(nm)$.

$$\mathcal{O}(nm)$$

5.4 Levenshtein distance

This is very similar to the longest common subsequence. Insetad of the longest common substring, we are interested in to minimal amount of editing steps to convert one string into another with the operations

- inserting a character
- deleting a character
- editing a character

We distinguish the same four cases as above, however we recurrence changes, because we do not count same letters but different letter (as those need to be changed) instead: $ED(i,j) = \min\{ED(i-1,j)+1, ED(i,j-1)+1, ED(i-1,j-1)(+1)\}$. The last (+1) is only to be omitted if $A[i] = B[j]$. The DP table, computation order etc. is the same.

$$\mathcal{O}(nm)$$

5.5 subset sum

We have a n-array $A[1...n]$ and $b \in \mathbb{N}$. The output should be a truth value whether b can be expressed as a sum of elements of A. Specifically, we want to compute $I \subseteq \{1,...,n\}$ so that $b = \sum_{i \in I} A[i]$.

Let's think about when some s if a subset sum of $A[1...k]$. For s to be such a subset sum, there are two options

- s is a subset sum of $A[1...k-1]$ (then, s is not relevant for the subset sum)
- $s - A[k]$ is a subset sum of $A[1...k-1]$ (then, s is relevant for the subset sum)

We get this recurrence: $SS(i,s) = SS(i-1,s) \vee SS(i-1, s-A[i])$ with $SS(a,b) :=$ b is subset sum of $A[1...a]$.

The DP table then has dimensions $d+1 \times n+1$. As initialization we set $DP[0,0] = 1$ and $DP[0,x] = 0$ with $n \geq x > 0$. Afterwards, if $DP[d+1, n+1] = 1$, then such a subset sum exists.

$$\mathcal{O}(nb)$$

This is different than other considerations, because b is not the input size, but a number. For a binary representation, the input size for b is $\log_2 b$. Total input size: $n + \log_2 b \leq \mathcal{O}(n + \log b)$.

We must consider how b grows in dependence of n:

- $b = 2^n$ (exponentially), then:
  input size: $\Theta(n)$
  runtime: $\mathcal{O}(2^n n)$ (exponential!)
- $b = n^c$ (polynomial), then:
  input size: $\Theta(n)$
  runtime: $\mathcal{O}(n^{c+1})$ (polynomial!)

One says: The runtime is pseudo-polynomial.

5.6 knapsack problem

Version 1 Input is a weight limit W. Also, we have n pairs (v_i, w_i), v being the value and w the weight of an item. The goal is to compute $I \subseteq \{1, ..., n\}$ so that $\sum_{\in I} w_i \leq W$ and $\sum_{i \in I} v_i$ maximal.

The naive algorithm just tries all combinations in $\mathcal{O}(2^n n)$. Additinoally, the greedy algorithm following the value density $\frac{v_i}{w_i}$ does not in this case.

To get a DP algorithm, we must think about the invariant. How can we learn whether I is the best solution for n item with weight limit W? Let $MV(i,w)$ be the max value of $I \subseteq \{1, ...i\}$ with weight limit w. If MV is the best solution for the i items with weight limit w, then there are two cases:

- $MV - v_i$ is the best solution for $i-1$ item with weight limit $w - w_n$
- MV is the best solution for $i-1$ items with weight limit w

From that we get this recurrence: $MV(i,w) = max\{MV(i-1, w-w_i)+v_i, MV(i-1,w)\}$. The DP table has dimensions $n+1 \times W+1$ and the table is initialized with $DP[0,x] = 0$ for arbitrary bound x. Each cell may be computed in constant time and fixed storage. Thus, for runtime and storage: $\mathcal{O}(nW)$. This is pseudo-polynomial.

$$\mathcal{O}(nW)$$

Alternative Instead of the maximum value, we might consider the minimum weight. That leads to $MinW(i,v) =$ minimum weight to reach value v with i item. Specifically: $MinW(i,v) = min\{MinW(i-1,v), MinW(i-1, v-v_i) + w_i\}$. The size of the DP table is $n+1 \times V+1$ with $V = \sum_{i=1}^{n} v_i$. It is initialized with $DP[0,0] = 0$ and $DP[0,x] = \infty$ with $x > 0$ but still bound according to the DP table size.

$$\mathcal{O}(nV)$$

Approximation There also is an approximized version with a slight error margin. We demonstrate the fully polynomial approximation scheme. So far, we had w_i, v_i, W with the optimal solution $OPT \subseteq \{1, ..., n\}$. Now, we consider $w_i, \lfloor \frac{v_i}{k} \rfloor, W$. For that we have the optimal solution $\overline{OPT} \subseteq \{1, ..., n\}$. This new option is obviously k times faster.

We know $V \leq n \cdot v_{\max}$ and, newly, $\overline{V} \leq \frac{n \cdot v_{\max}}{k}$. Thus, to compute $\overline{OPT}$, we require $\mathcal{O}(n\overline{V}) \leq \mathcal{O}(n^2 \frac{v_{\max}}{k})$ time. Below, we will further show that $k = \frac{\epsilon}{n} v_{\max}$, where ϵ is defined by the relation $Value(\overline{OPT}) \geq (1-\epsilon) Value(OPT)$, being the error margin. For a given error margin ϵ with the associated k, we the get the polynomial runtime $\mathcal{O}(n^3 \frac{1}{\epsilon})$.

$$\mathcal{O}(n^3 \frac{1}{\epsilon})$$

$$\mathcal{O}(n^2 \frac{v_{\max}}{k})$$

Now, we show why $k = \frac{\epsilon}{n} v_{\max}$.

We can naturally write:

$$\frac{v_i}{k} - 1 \leq \lfloor \frac{v_i}{k} \rfloor \leq \frac{v_i}{k}$$

$$\Leftrightarrow v_i - k \leq k \lfloor \frac{v_i}{k} \rfloor \leq v_i$$

$$\Rightarrow \sum_{i \in OPT} (v_i - k) \leq \sum_{i \in OPT} k \lfloor \frac{v_i}{k} \rfloor = k \sum_{i \in OPT} \lfloor \frac{v_i}{k} \rfloor \leq k \sum_{i \in \overline{OPT}} \lfloor \frac{v_i}{k} \rfloor \leq \sum_{i \in \overline{OPT}} v_i = Value(\overline{OPT})$$

Also:

$$\sum_{i \in OPT} (v_i - k) \geq \sum_{i \in OPT} v_i - nk = Value(OPT) - nk$$

Thus, we get:

$$Value(\overline{OPT}) \geq Value(OPT) - nk$$

We now demand $Value(OPT) - nk \overset{!}{\geq} (1-\epsilon) Value(OPT)$ and thereby define ϵ (as the error margin). That can be transformed to $k \leq \frac{\epsilon}{n} Value(OPT)$. And with $Value(OPT) \geq v_{\max}$ to $k = \frac{\epsilon}{n} v_{\max}$.

5.7 chain matrix multiplication

We have $A_1 \cdot A_2 \cdot ... \cdot A_n$ - a product of matrices with suitable dimensions. This algorithm exploits associativity of matrix multiplication to compute the fasts way of multiplication. We define $M(p, q) =$ minimal number of operations to compute $A_p \cdot ... \cdot A_q$.

We have the invariant: $M(p, q)_{p \leq q} = \min_{p \leq i < q}(M(p, i) + M(i+1, q) +$ cost for $(A_p \cdot ...A_i)(A_i \cdot ...A_q))$. We thus, have a $n \times n$ DP table of which we use the upper right triangle, which is filled from the diagonal to the upper right corner. The solution then can be extracted from the upper right corner.

Storage requirement: $\mathcal{O}(n^2)$. Runtime: $\mathcal{O}(n^3)$.

$$\mathcal{O}(n^3)$$

6 Sort and Search

name	runtime
binary search	$\mathcal{O}(\log n)$
search in unsorted array	$\mathcal{O}(n)$
isSorted	$\mathcal{O}(n)$
bubble sort	$\mathcal{O}(n^2)$
selection sort	$\mathcal{O}(n^2)$
insertion sort	$\mathcal{O}(n^2)$
heap sort	$\mathcal{O}(n \log n)$, but bad locality
merge sort	$\mathcal{O}(n \log n)$, but $\mathcal{O}(n) storage$
quicksort	$\mathcal{O}(n^2)$, but $\mathcal{O}(n \log n)$ on average

Figure 5: sort and search algorithms, runtimes

6.1 search

6.1.1 binary search

As input we have a n-array sorted in increasing order ($A[0] \leq A[1] \leq ... \leq A[n-1]$) and an element b which is to be found.

The ouput is either k with $A[k] = b$ or that b is not contained in A.

Algorithm 3 Binary Search (iteratively)

```
BinarySearch(A,b):
    left = 0
    right = n-1
    while left <= right:
        middle = floor( left/2 + right/2 )
        if b=A[middle]: return middle
        if b<A[middle]: right=middle-1
        else: left=middle+1
    return "not found"
```

For runtime we have $T(n) \leq T(\frac{n}{2}) + d$. TElescoping with $n = 2^k$ leads us to $T(n) \leq T(\frac{n}{2}) + d \leq T(\frac{n}{4}) + 2d \leq T(\frac{n}{8}) + 3d \leq ... \leq T(\frac{n}{n}) + \log_2 n \cdot d$.

$$\mathcal{O}(\log n)$$

This is the best runtime possible for a comparison-based algorithm. Consider a tree: Array elements are vertices, and a path through the tree corresponds to the behavior of a comparison-based algorithm. Either finds, goes to the right or to the left. Thus, the height determines the maximum number of comparisons. For some optimal algorithm, we would have a most dense tree. But for the height of such a most dense tree we have $h > \log_2 n$. Therefore, there need to be at least $\log n$ operations/comparisons.

6.1.2 Linear Search

We simply have to look at all elements in order.

$$\Theta(n)$$

This is optimal for comparison-based algorithms. One has to make r comparisons within A and s comparisons with b. For r comparisons, one can partition A in at most $g \geq n - r$

Algorithm 4 Linear Search

```
LinearSearch(A,b):
    for i=1...n:
        if A[i]=b: return i
    return "not found"
```

groups (best cae). Then, one still has to compare b to every group. Thus: $r + s \geq r + g \geq r + (n - r) = n$.

6.2 sort

We generally consider as input a n-array A. The targeted output is a permutation of A, which is sorted in increasing order.

Sorting inplace means that no additional storage like a copy of A (besides $\mathcal{O}(1)$ for variables etc.) is required/used. Sorting happens based on some key. That key usually is a number to be put in increasing order.

For the runtime, the number of comparisons and exchanges are counted.

6.2.1 check whether sorted

One can trivially check, whether A is sorted.

Algorithm 5 IsSorted

```
IsSorted(A):
    for i: 1...n-1:
        if A[i]>A[i+1]: return false
    return true
```

$$\Theta(n)$$

6.2.2 bubble sort

Algorithm 6 BubbleSort

```
BubbleSort(A):
    for j=1...n-1:
        for i=1...n-j:
            if A[i]>A[i+1]: change A[i] and A[i+1]
```

Correctness is given when considering the invariant in j that after the loop terminated for some j, the last j elements are in the correct position.

$$\mathcal{O}(n^2)$$

6.2.3 selection sort

Inductive approach. After the ith step, the first i elements are in the correct position. This is achieved by (in the ith step) considering $A[i...n]$ and putting the smallest element to the left at $A[i]$.

Algorithm 7 SelectionSort

```
SelectionSort(A):
    for i=1...n-1:
        j=index(minimum(A[i...n]))
        change A[i],A[j]
```

There are $\mathcal{O}(n)$ position switches. For the number of comparisons we have $\sum_{i=1}^{n-1}(n-i) \leq \mathcal{O}(n^2)$.

$$\mathcal{O}(n^2)$$

6.2.4 insertion sort

Inductive appraoch. We have the invariant $Inv(i)$ = the first i elements are sorted, which is valid after the ith step. During step i, we must insert $A[i]$ at the correct position in $A[1...i-1]$.

Algorithm 8 InsertionSort

```
InsertionSort(A):
    for i=2...n:
        k=binarySearch(A[1...i-1],A[i])
        x=A[i]
        push A[k...i-1]->A[k+1...i]
        A[k]=x
```

We have $\leq \sum_{i=1}^{n} a \cdot \log_2 i$ comparisons (in binary search). This equals to $\mathcal{O}(\log n!) = \mathcal{O}(n \log n)$ comparisons. Furthermore, we have $\mathcal{O}(n^2)$ changes.

$$\mathcal{O}(n^2)$$

6.2.5 Heap Sort

Inductive approach with the invariant: $INV(i)$ = before the ith step, the first $n-i$ elements from the right are sorted and in the correct position. To achieve that (during the ith step) one needs to find the biggest element of $A[1...i]$ and change its position wih $A[i]$.

Hence, one requires a data structure in which finding the max value is cheap. Specifically, we consider a Max-Heap in which finding the max value (and repairing the heap) is $\mathcal{O}(\log n)$.

Algorithm 9 Heap Sort

```
HeapSort(A):
    for i=floor(n/2)...1:
        RestoreHeapCondition(A,i,n)
    for i=n...2:
        change A[1], A[i]
        RestoreHeapCondition(A,1,i-1)
```

The function RestoreHeapCondition(A,k,i) lets the kth element trickle down in $A[1...i]$.

Heap creation takes $\mathcal{O}(n \log n)$. Changing and restoring the heap takes $\leq \sum_{i=n}^{2} a \cdot \log_2 i \leq \mathcal{O}(n \log n)$.

$$\mathcal{O}(n \log n)$$

Good is inplace. But it has bad locality - a real world concenr.

6.2.6 Merge Sort

Recursived/divide-and-conquer approach. An array is split into two parts, both sorted recurisvely, and then both parts are merged to one sorted array. To merge, a copy of the array will be created.

Algorithm 10 Merge Sort

```
Merge(A,l,m,r):
    i=l
    j=m+1
    k=l
    while i<=m && j<=r:
        if A[i]<A[j]:
            B[k]=A[i]
            i++
            k++
        else:
            B[b]=A[j]
            j++
            k++
    add remaining of non-empty half
    copy B to A

MergeSort(A,left,right):
    if left<right:
        middle = lower(left/2+right/2)
        MergeSort(A,left,middle)
        MergeSort(A,middle+1,right)
        Merge(left,middle,right)
```

The recurrence for the runtime is $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n \leq \mathcal{O}(n \log n)$.

$$\mathcal{O}(n \log n)$$

The major disadvantage of this algorithm is the extra $\mathcal{O}(n)$ storage.

Anyway, a real world improvement is 'natural merge sort'. One searches for already sorted parts, on which one does not have to recursively compute MergeSort.

6.2.7 Quicksort

Recursive approach. During one call, some pivot p is chosen and the array structured so that all smaller elements are on the left and all greater elements on the right, without being fully sorted (separate function). Thus, p is at its correct position. Then, both subarrays are sorted recursively.

Algorithm 11 Quick Sort

```
separate(A,l,r):
    i=l
    j=r-1
    p=A[r]

    while i<r and A[i]<p: i++
    while j>l and A[j]>p: j--
    while i<j:
        switch A[i],A[j]
        while i<r and A[i]<p: i++
        while j>l and A[j]>p: j--
    switch A[i],A[j]
    return i


QuickSort(A,l,r):
    k = separate(A,l,r)
    QuickSort(A,l,k-1)
    QuickSort(A,k+1,r)
```

In the best case, the pivot element always ends up in the middle. Runtime: $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n \leq \mathcal{O}(n \log n)$.

In the worst case, the pivot element is always directly at one edge. Runtime: $T(n) = T(n-1) + c \cdot n \leq \mathcal{O}(n^2)$.

However, this does not disqualify Quicksort, becaue it is (a) inplace and (b) the worst case is very rare. An analysis beyond the scope of this course shows that the average case is $\mathcal{O}(n \log n)$.

$$\mathcal{O}(n^2 - (\textbf{on average: } n \log n))$$

6.2.8 algorithmic limit

There does not exist a comparison-based algorithm sorting faster than $\mathcal{O}(n \log n)$. Consider a binary tree. At each branch, an algorithm does a comparison of two elements and then some action. If we just permute the input array, the algorithm will have a different comparison at some point and, thus, chose a different path. Hence, for every permutation there will be a different path. Accordingly, the number of leaves must be at least the number of permutations $n!$. For a dense binary tree of height h, we know that there are 2^h leaves. Thus: $2^h \geq n! \Leftrightarrow h \geq \log_2 n!$.

The height h of the binary tree determines the runtime. Becuase $\log_2 n! = \Theta(n \log n)$, we have $h = T(n) \geq \Omega(n \log n)$.

7 Graphs

undirected graphs An undirected graph G is the tupel $G = (V, E)$. V is the set of vertices, E is the set of edges. An edge is an unordered pair of two vertices $\{u, v\}$ (usually $u \neq v$ is mandatory). We write $e = \{u, v\} \in E$ or $e = uv$ with $u, v \in V$. Generally, duplicate entries in E are not allowed.

Furthermore, we define those general terms:

- u, v are adjacent if $\exists e \in E : e = uv$
- e is incident o to v if $\exists u \in V : e = uv$

- $deg(u)$ (degree): number of incident edges

Also, urgently remember the handshaking lemma: $\sum_{v\in V} deg(v) = 2|E|$ (proven with vertices handing a \$ to each incident edge $\rightarrow$ each edge receives 2\$).

directed graphs A directed graph $G = (V, E)$ is different in the sense that its edges are ordered pairs. If there is a connection from u to v with e: $e = (u, v) \in \mathbb{E}$. v is successor and u is predecessor.

- $deg_{in}(u) = \sum_{(v,u)\in E} 1$ with $v \in V$: number of edges that point towards u (incoming degree)
- $deg_{out}(u) = \sum_{(u,v)\in E} 1$ with $v \in V$: number of edges that start at u (outgoing degree)

The equivalence to the handshaking lemma is: $\sum_{u\in V} deg_{in}(u) = \sum_{u\in V} deg_{out}(u) = |E|$. A node u is called a source if $deg_{in}(u) = 0$ and a sink if $deg_{out}(u) = 0$.

general properties Furthermore, those definitions are also of utmost importance:

- walk: series of adjacent vertices
- directed walk: sequence of edges directed in the same direction, which join a series of vertices
- trail: walk in which all edges (between to neighboring vertices) are distinct
- directed trail: directed walk with distinct (neighboring) edges
- path: walk without repeating vertices
- directed path: directed walk with distinct vertices
- cycle: trail with $v_0 = v_l$ and $l \geq 2$
- directed cycle: directed trail with $v_0 = v_l$ and $l \geq 2$

The length of any kind of walk is the number of vertices - 1. We say the u reaches v if there exists a walk between $v_0 = u$ and $v_l = v$.

For undirected graphs, the reaches relation is an equiavlence realtion (symmetric, reflexive, transitive). An equivalence class is called connected component. Some graph is connected if exactly one connecte component exists. A trail is a cycle if and only if the end vertices are incident to an even number of vertices of the walk.

representation of graphs for/in computers Generally, $V = \{1, 2, ..., n\}, G = (V, E), m = |E|$.

- adjacency matrix:
  $A = (A_{uv})$, $A_{uv} = \left\{ \begin{smallmatrix} 1,(u,v)\in E \\ 0,\text{ otherwise} \end{smallmatrix} \right.$
  storage: $\mathcal{O}(n^2)$, even if $m \ll n$ (suboptimal)
- adjacency list:
  list of linkedLists, at position i of root list ist the list of all successors of i

relevant operations	runtime adjacency matrix	runtim adjacency list
test $(u, v) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(1 + deg_{out}(u))$
list all successors of u	$\mathcal{O}(n)$	$\mathcal{O}(1 + deg_{out}(u))$

Figure 6: graph representations, runtime comparison

name	runtime
Eulerian path/cycle	$\mathcal{O}(n+m)$
Hamiltonian path	$\Omega(n!)$
topological sorting	$\mathcal{O}(n+m)$ (DFS, after all)
depth first search	$\mathcal{O}(n+m)$
breadth first search	$\mathcal{O}(\|V\|+\|E\|)$
topological order-based/BFS	$\mathcal{O}()$
Dijkstra	$\mathcal{O}((n+m)\log n)$/Fibonacci-Heap: $\mathcal{O}(m+n\log n)$
Bellman-Ford	$\mathcal{O}(n(n+m))$
Floyd-Warshall	$\mathcal{O}(n^3)$
Johnson	$\mathcal{O}(mn+n^2\log n)$
Boruvka	$\mathcal{O}((n+m)\log n)$
Prim	$\mathcal{O}((n+m)\log n)$, Fibonacci-Heaps: $\mathcal{O}(m+n\log n)$
Kruskal	$\mathcal{O}(m\log m+n\log n)$, trivial: $\mathcal{O}(nm)$

Figure 7: graph algorithms, runtimes

7.1 Eulerian path/cycle

An Eulerian path is a trail, which contains all edges of a graph.

For all vertices not start/end: $\#to = \#from$. For start: $\#to + 1 = \#from$. For end: $\#to = \#from + 1$. If star = end: For start/and: $\#to = \#from$. Hence, for not start/end vertices, we have $deg(u) = \#to + \#from = 2\#to = 2\#from$ - those degrees are all even. Thus, only the start/end vertices may be unenve. Accordinly, there may only be two uneven vertices in total for an Eulerian path to exist.

Can only exist if $m \geq n-1$. Naive algorithm: $\Omega(n!)$.

Now, we only consider Eulerian cycles (cycles, which contain all edges of a graph exactly once). Eulerian paths can be reduced to this by connecting the two uneven start and end vertices.

$\exists$ Eulerian cycle $\Leftrightarrow$ all degrees even and all edges in one connected component The direction to the right is trivial. The direction to the left is proven with an algorithm.

1. iteratively compute cycles until all edges used
2. merge those cycles to one cycle

Algorithm 12 Finding Cycles

```
walk(u):
    if v with uv in E exists and unmarked:
        mark uv
        walk(v)
```

Note:

- $walk(u)$ marks the start u
- every edge is marked at most once
- the end vertex of W has all edges marked

We formulate the invariant: $\forall v \in V$, amount of unmarked edges to v is even. We claim:

1. The invariant is maintained by $walk(u)$.
2. If the invariant is valid before $walk(u)$, then W is a cycle.

Notice: that u has at least one unmarked edge.

To prove $2/W$ being a cycle: Amount of edges incident to end vertex in W are even. Unmarked edged to end vertex before $walk(u)$ are even according to the invariant/after $walk(u)$ are zero according to characteristic 3 above. Thus, $walk(u)$ marks an even number of edges incident to u.

Prove of 1 follows from 2: Even number of incident edges in cycle W.

integrated algorithm Now, the entire algorithm is implemented recurisvely: We find a cycle and then directly integrate it with anothe cycle.

Algorithm 13 Euler cycles

```
Euler(G): // finds an Eulerian cycle in G if exists
    empty list Z
    all edges unmarked
    EulerWalk(u) // u being arbitrary in V with unmarked edge
    return Z

EulerWalk(u):
    for all uv in E, not makred:
        mark uv
        EulerWalk(v)
    append u to Z (Z=(Z,u))
```

$$\mathcal{O}(n+m)$$

But as $m \geq n-1$, basically $\mathcal{O}(m)$.

7.2 Hamiltonian path

A Hamiltonian path is a path that visits each vertex exactly once. Can only exist if $m \geq n-1$. Naive algorithm: $\Omega(n!)$.

Hamiltonian paths can not be computed (identified) in polynomial time if $P \neq NP$ holds.

7.3 topological sorting

A topological order in a directed graph is an order of the vertices such that all vertices, which have edges to some other vertex must come before that other vertex in the topological order.

Generally: $\exists$ topological order $\iff \nexists$ cycle. The direction to the right is quite obvious. Direction is considered proven with below algorithm.

Approach:

1. find a sink v
2. put v at the front of the topological order
3. remove v from the graph and continue recursively

Algorithm 14 Find sink

```
path(u):
    mark u
    if unmarked successor v of u exists:
        path(v)
```

- $path(u)$ marks path p with start u
- all successors of the end vertex v of P are marked

We easily see that: P has no sink $\Rightarrow \exists$ directed cycle. This proves $\nexists$ directed cycle $\Rightarrow P$ has a sink

Consider DFS for an optimized (appropriate) implementation.

7.4 depth first search

Algorithm 15 Topological Order - Depth First Search

```
visit(u):
    mark u
    for each unmarked successor v of u:
        visit(v)
    add u to the topological order

DFS(G):
    for each unmarked u in V: // order does not matter, by
       convention lexicographic
        visit(u)
```

Runtime for adjacency matrices: Besides constants, $visit(u)$ is executed for each u and takes $\mathcal{O}(n)$ for each. In total: $\mathcal{O}(n^2)$.

Runtime for adjacency lists: Besides constants, $visit(u)$ is executed for each u and takes $\mathcal{O}(1 + deg_{out}(u))$. In total: $\mathcal{O}(\sum_u 1 + deg_{out}(u)) = \mathcal{O}(\sum_u 1 + \sum_u deg_{out}(u)) = \mathcal{O}(n+m)$.

$$\mathcal{O}(n+m)$$

Algorithm 16 extended DFS

```
visit(u):
    pre[u] <- T
    T <- T+1
    mark u
    FOR all unmarked successors v of u:
        visit(v)
    post[u] <- T
    T <- T+1

DFS(G):
    T <- 1
    all vertices unmarked
    FOR unmarked u in V:
        visit(u)
```

Notice!!! Iterative implementation with Stack possible/recommended.

For each vertex, we have the interval, in which it has been run: $I_u = \{pre[u], ..., post[u]\}$. All intervals can be represented graphically as horizontal stacked interval lines. Alternatively, one can construct the depth first search tree. It may be that one gets a depth first search tree forst if the DFS routine must vall the visit routine on multiple vertices.

All edges naturally in a depth first search tree are called tree edges. However, not all edges muts be presented in a depth first search tree. Those can be classified. Consider $(u, v) \in E$ with I_u and I_v:

- back edges: $I_u \subset I_v$ (indentical with fordward edges for undirected graphs)
- forward edges: $I_v \subset I_u$ (indentical with back edges for undirected graphs)
- cross edges: I_u comes after I_v (impossible for undirected graphs)
- impossible edges: I_v comes after I_u & I_v starts after u begings but before it ends but ends after u ends and other direction (intersecting but not enclosed)

We observe:

- $\nexists$ back-edge $\exists$ cycle.
- $\forall$ non-back-edges $(u, v) \in E \Rightarrow \mathit{post}[u] > \mathit{post}[v]$

We conclude: $\nexists$ cycle $\Rightarrow \nexists$ back-edge $\Rightarrow$ the inverse post roder is a topological order

7.5 breadth first search

Starting at some vertex, we try to identify the walks with the least amount of edges to any other vertex. So we search for $d(s, v)$ forall $v \in V$ (d being the distance/shortest walk length function).

We observe that the distance/shortest walk mus talways be a path.

The solution can be represented as a shortest path tree (reordered graph): Each level k is described by $S_k := \{v|d(s, v) = k\}$.

We think about how to recursively compute S_k from $S_{k-1}, ..., S_0$: $v \in S_k \overset{def}{\Longleftrightarrow} (u, v) \in E$ and $u \in S_{k-1}$, $0 \leq i < k$.

Algorithm 17 trivial BFS

```
S0 = {s}
S1={}, S2={}, ..., Sn-1={}

FOR k=1...n-1:
    FOR u in Sk-1:
        FOR (u,v) in E with v not in S0+...+Sk-1
            Sk=Sk+{v}
```

To improve this algorithm there are two ideas:

- mark vertices as soon as their distance is known (simplify v not in $S0 + ...Sk - 1$)
- use a queue which allows, $Sk - 1$ to be processed and Sk to be extended/build

Algorithm 18 Breadth First Search

```
BFS(s):
    Q <- {s}
    enter[s] <- 0
    // distance[s] <- 0
    T <- 1
    WHILE Q not empty:
        u <- dequeue(Q)
        leave[u] <- T
        T <- T+1
        FOR (u,v) in E, whith enter[v] not assigned
            enqueue(Q,v)
            enter[v] <- T
            T <- T+1
            // distance[v] <- distance[u] + 1
```

Proof. We define $t_k := \min\{leave[v] | d(s,v) \geq k\}$. Obviously $1 = t_0 < t_1 < ... < t_n = \infty$ (∞ covering unreachable vertices).

We also define $R_k := \{v | t_k \leq leave[v] < t_{k+1}\}$.

To prove correctness of the algorithm we must prove: $\forall k \in \mathbb{N}_0$, $S_k = R_k = \{v | t_k \in I_v\}$, which we do with induction.

Base Case $S_0 = \{0\}$, $enter[s] = 0$, $leave[s] = 1$

$\forall v \neq s$: $leave[v], enter[v] > 1$

Thus, $t_1 > 1$, $R_0 = \{0\}$, $\{v | t_0 \in I_v\} = \{s\}$.

Induction hypothesis Have left Q before t_k: $S_0 \cup ... \cup S_{k-1}$

In Q at time t_k: S_k

Leaving Q in time t_k till t_{k+1}: S_k

Induction Step Enter Q in time t_k till t_{k+1}: all successors of S_k not in $S_0 \cup ... \cup S_k$. This is S_{k+1} according to the formula for recursive S_k computation.

In Q at time t_{k+1}: S_{k+1}

Leaving Q in time t_{k+1} till t_{k+2}: S_{k+1} □

Runtime: The WHILE loop is called at most once for element of the raph. For some u, the runtime of the WHILE loop is $\mathcal{O}(1 + det_{out}(u))$. For all u we get: $\mathcal{O}(\sum_{u \in V}(1 + deg_{out}(u))) = \mathcal{O}(|V| + |E|)$.

$$\mathcal{O}(|V| + |E|)$$

7.6 cheapest walks

To select the best algorithm, consider this:

name	case
BFS	no weights
topological order-based	no cycles
Dijkstra	only positive weights
Bellman-Ford basic	no negative cycles
Bellman-Ford extended	everything

Figure 8: cheapest walk, algorithm selection guide

7.6.1 topological order-based

Assumptions:

- $\nexists$ cycles ($\Rightarrow \nexists$ negative cycles)

$d(s,s) = 0$ holds. The triangle inequality $d(u,w) \leq d(u,v) + d(v,w)$ holds.
If $v_0, ..., v_l$ is the cheapest path, then so is $v_0, ..., v_{l-1}$. This leads to the recursion: $\forall v \neq s, d(s,v) = min_{u \to v} d(s,u) + c(u,v)$.
To compute this, the base case and computation order for acyclic graphs is given by the topological order.

Algorithm 19 Topological Order Based Sheapest Walk

```
FOR v in V (iteration following the topological order):
    if v==s:
        d[v] = 0
    else if degreeIn(v)==0:
        d[v] = infty
    else:
        d[v] = min( d[u]+c(u,v) ) // over all u->v
```

When using an adjacency lists:

$$\mathcal{O}(|V| + |E|)$$

7.6.2 Dijkstra

Assumptions:

- only positive weights ($\Rightarrow \nexists$ negative cycles)

We consider the vertices $v_1, ..., v_n$ ordered by their distance $d(s,v_1) < d(s,v_2) < ... < d(s,v_n)$ (assuming all distances distinct, but not relevant for algorithm correctness). Because the triangle equation and direct consequences hold again, we get this updated recursion: $d(s,v_k) = \min_{\substack{v_i \to v_k \\ i<k}} d(s,v_i) + c(v_i,v_k)$. This is justified by $d(s,v_k) > d(s,v_i)$. To compute v_k from $v_1, ..., v_{k-1}$:

```
chose edge u*->v*, u* in S, v* not in S
search for minimal of d(s,u*)+c(u*,v*)
v_k=v*
```

We claim correctness with $d(s,v*) = d(s,u*) + c(u*,v*)$. We prove that with:
$\forall W = s$ to $v*, c(W) \geq d(s,u) + c(u,v) + d(v,v*) \geq d(s,u) + c(u,v) \geq d(s,u*) + c(u*,v*)$. (This defines our choice of $u*, v*$.)
To get fast runtime, we think about how to quickly find $v*$ (not in S, edge from S, minimal edge):

- $d[]$ to manage upper bounds
- $d[v] = min_{\substack{u \to v \\ u \in S}} d(s,u) + c(u,v)$
- choice of $v*$: vertex of $V \backslash S$ with minimal upper bound
- adding $v*$ to S and changing bounds of successors of $v*$

Algorithm 20 basic Dijkstra

```
Dijkstra(s):
    d[s] = 0
    for all v in V\{s}:
        d[v] = infinity
    S = empty set
    WHILE S not V:
        chose v* in V\S with minimal d[v*]
        add v* to S
        FOR (v*,v) in E, v not in S:
            d[v] = min( d[v],d[v*]+c(v*,v) )
```

To efficiently chose $v*$, we use a max heap to arrive at a proper Dijkstra implementation:

Algorithm 21 Dijkstra

```
Dijkstra(s):
    d[s] = 0
    for all v in V\{s}:
        d[v] = infinity
    S = empty set
    H = make_heap(V)
    decrease_key(H,s,0)
    WHILE S not V:
        v* = extract_min(H)
        add v* to S
        FOR (v*,v) in E, v not in S:
            d[v] = min( d[v],d[v*]+c(v*,v) )
            decrease_key(H,v,d[v])
```

Runtime: $\mathcal{O}(n + \#extractMin \cdot \log n + \#decreaseKey \cdot \log n) = \mathcal{O}((n + m) \log n)$.

$$\mathcal{O}((n + m) \log n)$$

With Fibonacci heaps, one might even get $\mathcal{O}(n + m \log n)$ (???).

7.6.3 Bellman-Ford

Assumptions

- $\nexists$ negative cycles

Consider $S_{\leq l} := \{v \in V | \exists$ cheapest walk to v with ≤ 1 edges $\}$ - being the vertices v sorted according to the amount of edges in the shortest path. We have: $S_{\leq 0} = \{0\}$ and $S_{\leq n-1} = V$.

We get the recurrence $\forall v \in S_{\leq l} \backslash \{s\}$: $d(s, v) = \min_{\substack{u \to v \\ u \in S_{\leq l-1}}} d(s, u) + c(u, v)$. The upper boundary $d[]$ is l-good, when considering the recurrence till $S_{\leq l}$. Upper boundaries may be improved like this:

```
FOR v in V:
    d[v] = min( d[v], min( d[u]+c(u,v) for u->v ) )
```

We get the Bellman-Ford algorithm by iterating the above $n - 1$ times to reach $v \in S_{\leq n-1} = V$, which implies $d[v] = d(s, v)$.

Algorithm 22 Bellman Ford

```
boundImprovement():
    FOR v in V:
        d[v] = min( d[v], min( d[u]+c(u,v) for u->v ) )

BellanFord(s):
    d[] = n-array
    d[s] = 0
    for v in V\{s}:
        d[s] = infinity
    repeat n-1 times:
        boundImprovement()
```

Runtime: Each boundImprovement() takes $\mathcal{O}(n+m)$. The total algorithm thus takes $\mathcal{O}(n(n+m))$. But as usually $m \geq n-1$, we often consider $\mathcal{O}(nm)$.

$$\mathcal{O}(n(n+m))$$

Adjusted assumptions

- $\exists$ negative cycles

To detect (and process) negative, we just run Bellman-Ford for one additional iteration (n instead of $n-1$ times) and claim "$\exists$ negative cycle from s reachible $\Leftrightarrow$ boundaries change in last iteration".

$\Leftarrow$: " $\nexists$ negative cycle $\Rightarrow$ boundaries do not change". Consider the correctness of Bellman Ford.

$\Rightarrow$: $d'[v_i] \leq d[v_{i-1}] + c(v, v_i)$ (after bound improvements). We consider $1, ..., l$ as enuemration of the negative cycle.

$$\Rightarrow \sum_{i=1}^{l} d'[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + c(Z) < \sum_{i=1}^{l} d[v_{i-1}] = \sum_{i=1}^{l} d[v_i]$$

$$\Rightarrow \sum_{i=1}^{l} d'[v_i] < \sum_{i=1}^{l} d[v_i]$$

Hence, at least one bound needs to be lower.

7.7 shortest paths

The task is to compute the minimum distance between any two vertices, not only between one vertex and all others. In contrast to the previous computations, which were on-to-all, this is considered an all-pairs problem.

A trivial approaches is to just execute the already known algorithms for each vertex n.

- $n\times$adjusted topological order-based: $\mathcal{O}(mn + n^2)$
- $n\times$Dijkstra: $\mathcal{O}(n(n+m)\log n)$, Fibonacci Heaps $\mathcal{O}(mn + n^2 \log n)$
- $n\times$Bellman Ford: $\mathcal{O}(n^2 m)$

However, the newly introduced algorithms (both tackle the negative cycles problem) perform better:

name	runtime
Floyd-Warshall	$\mathcal{O}(n^3)$
Johnson	$\mathcal{O}(mn + n^2 \log n)$

Figure 9: shortest path algorithms, runtimes

7.7.1 Floyd-Warshall

This is a DP algorithm. We assign each vertex a unique number $1...n$ and define d_{uv} as the cost of the cheapest path from u to v with all vertices in between having a number $\leq i$.

To compute d_{uv}^i from d_{uv}^{i-1}, we notice two cases:

- i is part of the path described by d_{uv}^i
- i is not part of the path described by d_{uv}^i

We get the recurrence $d_{uv}^i = \min\{d_{uv}^{i-1}, d_{ui}^{i-1} + d_{iv}^{i-1}\}$. For the initialization we have:

- $d_{uu}^0 = 0$
- $d_{uv}^0 = c(u,v)$ for $(u,v) \in E$ with $u \neq v$
- $d_{uu}^0 = \infty$ for $(u,v) \notin E$ with $u \neq v$

Algorithm 23 Floyd-Warshall

```
FloydWarshall(G):
    // initialization
    for all u in V: duu0 = 0
    for all (u,v) in E: duv0 = c(u,v)
    for all (u,v) not in E: duv0 = infinity

    // DP
    FOR i=1...n:
        FOR u=1...n:
            FOR v=1...n:
                duvi = min( duv(i-1), dui(i-1)+div(i-1) )
                // inplace: duv = min( duv, dui+div )
```

Negative cycles may be identified: v is in a negative cycle $\Leftrightarrow d_{vv}^i < 0$ as soon as i at least the largest index of vertices in the cycle.

Due to the the nested loop for computation, runtime obviously $\mathcal{O}(n^3)$.

$$\mathcal{O}(n^3)$$

Notice that this algorithm has special general meaning/use cases.

Transitive Closure If one is not interested about the distance of the connection between two vertices but just whether they are connected, one can simply replace the invariant with $d_{uv}^i = d_{uv}^{i-1} \vee (d_{ui}^{i-1} \wedge d_{iv}^{i-1})$. In such a graph, $u \circ v \Leftrightarrow (u,v) \in E$.

Matrix multiplication Consider the updated (inplace) invariant $d_{uv} = d_{uv} + d_{ui}d_{iv}$. Consider matrix potentiation. $A^2 = [b_{ij}]$ with $b_{ij} = \sum_{k=1}^n a_{ik}a_{kj}$. With Floyd-Marshall we compute $d_{uu} + d_{ui}d_{iv}$, which captures exactly the fact that we sum up all combinations of $a_{ik}a_{ki}$ for $k \in (i,j)$.

Considering the matrix as an adjacency matrix/graph, this can also be understood as counting the number of paths from i to j with length 2 if we consider the 2nd power of A. This has interesting applications. Computing the number of triangles in a graph would be to compute $A^3/3$. Alternatively, one could compute the shortes path $i \rightsquigarrow j$ by continuously taking powers of A until $(i,j) \neq 0$. The power then gives the length of the shortest path. But as each potentiatino takes $\mathcal{O}(n^3)$, we have the uncompetitive runtime $\mathcal{O}(n^4)$.

So, can one do matrix multiplication faster than $\mathcal{O}(n^3)$? Consider splitting both matrices in a four-segmented block matrix each. Each block is assigned a letter in mathematical rotation from top left in each matrix from left to right. So, we must compute

$$\begin{aligned} u &= ae + bg \\ v &= af + bh \\ w &= ce + dg \\ x &= cf + dh \end{aligned}$$

This has runtime $T(n) = 8 \cdot T(n/2) + \Theta(n^2) \Rightarrow \Theta(n^3)$. However, analogously to Karatsubas algorithm, we splitthe computation and compute

$$\begin{aligned} t_1 &= (a+d)(e+h) \\ t_2 &= (c+d)e \\ t_3 &= a(f-h) \\ t_4 &= d(g-e) \\ t_5 &= (a+b)h \\ t_6 &= (c-a)(e+f) \\ t_7 &= (b-d)(g+h) \\ \\ u &= t_1 + t_4 - t_5 + t_7 \\ v &= t_3 + t_5 \\ w &= t_2 + t_4 \\ x &= t_1 - t_2 + t_3 + t_6 \end{aligned}$$

This has runtime $T(n) = 7 \cdot T(n/2) + \Theta(n^2) \Rightarrow \Theta(n^{\log_2 7}) \approx \Theta(n^{2.8\cdots})$. Unfortunately, this is mostly useless in practise. Because of a huge constant, we require large inputs. But when we chose large inputs, the numeric stability of this algorithm is too worse to be of any good. Additionally, computers are optimized for (nested) loops (used in the original approach).

7.7.2 Johnson

The idea of this algorithm is to use Dijkstra, thus the similar runtime. To use Dijkstra, all edge weights must be increased without changing the shortest paths. To do so:

- add a new vertex to the graph, which has a directed edge to every other edge with weight 0
- add a height $h(v)$ to every vertex $v \in V$

Given that, we define the new weights as $\hat{c}(u,v) = c(u,v) + h(u) - h(v)$.

We must show two things: Shortest paths remain shortest paths and all weights are positive. So far, for shortest paths we had $c(s \rightsquigarrow t) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$. Now, we have $\hat{c}(s \rightsquigarrow t) = \sum_{i=1}^{k-1} \hat{c}(v_i, v_{i+1}) = \sum_{i=1}^{k-1} (c(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})) = c(s \rightsquigarrow t) + h(s) - h(t)$. Thus, the shortest path only depends on the start and end and, hence, does not change.

Now we consider how to chose the assignment $h(v)$ so that $\hat{c} \geq 0$. We define $h(u) :=$ length of teh shortest path from the new vertex to the vertex u. Thus, $h(u) \leq 0$ always holds. Then, for all $(u,v) \in E$: $h(v) \leq h(u) + c(u,v) \Leftrightarrow c(u,v) + h(u) - h(w) \geq 0 \Leftrightarrow \hat{c}(u,v) \geq 0$.

Runtime analysis:

- Integrating the new vertex and edges: $\mathcal{O}(n)$
- Computing $h()$ values, given the new vertex as start vertex, can be done with Bellman-Ford: $\mathcal{O}(nm)$.

- Running Dijkstra n times: $\mathcal{O}(nm + n^2 \log n)$.

$$\mathcal{O}(nm + n^2 \log n)$$

In the worstcase, the runtime is $\mathcal{O}(n^3)$. However, usually this is noticably faster than Floyd-Warshall.

7.8 minimum span tree

We consdier a connected graph $G = (V, E)$ with the following restrictions/definitions:

- $w(e) \geq 0, e \in E$
- $A \subseteq E$ is spanning if (V, A) is connected
- weight of A: $w(A) := \sum_{e \in A} w(e)$
- span tree: spanning $T \subseteq E$ without cycles
- minimum span tree (MST): span tree with minimal weight (without cycles always possible, because one edge of cycle removed only at possible benefit)

For simplification only, we consider all weights to be distinct. But that does not impact the algorithms general correctness. Then, the MST will be unique.

We (will) consider safe edges, which are edges which must be part of the MST. We state that the smallest/cheapest edge to any connected component (thus, also single vertices) is a safe edge: $\forall S \subset V, S \neq \varnothing$, the minimal edge $e = uv$ to S ($u \in S, v \notin S$) is safe.

Proof. (indirect)
Consider span tree $T \subseteq E$ with $uv \notin T$. Consider $xy \in T$ edging S (being part of the cycle $U \cup \{u, v\}$). By definition of uv, we must have $w(xy) > w(uv)$. We than may replace xy with uv in T to get T'. Then, $w(T') < w(T)$ and T can not be a MST. $\square$

7.8.1 Boruvka

Algorithm 24 Boruvka

```
Boruvka(G=(V,E)):
    F = initially empty set of safe edges
    WHILE f not span tree:
        (S1,...,Sk) = connected components of (V,F)
        (e1,...,ek) = minimal edges to (S1,...,Sk)
        F.add(e1,...,ek)
```

The connected components can be identified as different trees with DFS in $\mathcal{O}(n + m)$. Finding the minimal edges also be done in $\mathcal{O}(n + m)$ (constant time per edge). The number of iterations is $\mathcal{O}(\log n)$, because an edge may be selected by 2 connected components at most (always reducing the number by at least half).

$$\mathcal{O}((n + m) \log n)$$

7.8.2 Prim

Algorithm 25 trivial Prim

```
Prim(G,s):
    F = empty set of safe edges
    S = {s} // set of connected component
    WHILE F not span tree
        u*v* = minimal edge to S (u* in S, v* not in S)
        F.add((u*,v*))
        S.add(v*)
```

Algorithm 26 Prim

```
Prim(G,s):
    H = make_heap(V,infinity)
    S = empty set of connected component
    d[s] = 0
    for v in V\{s}:
        d[v] = infinity
    decrease_key(H,s,0)
    WHILE H not empty:
        v* = extract_min(H)
        S.add(v*)
        FOR v*v in E, v not in S:
            d[v] = w(v*v)
            decrease_key(H,v,d[v])
```

$$\mathcal{O}((n+m)\log n)$$

With Fibonacci-Heap:

$$\mathcal{O}(m + n\log n)$$

7.8.3 Kruskal

Algorithm 27 Kruskal

```
Kruskal(G):
    F = empty set
    FOR uv in E - sorted in increasing order by weight:
        IF u,v in different connected components of (V,F):
            F.add(uv)
```

With a naive interpretation we get runtime $\mathcal{O}(nm)$ - m loop iterations, n each.

But we may use the uninon find data structure instead. For that we do an amortized analysis considering the total number of 'rep[]' reassignments for some u. 'rep[u]' is reassigned if $|ZHK(u)| \leq |ZHK(v)|$. After the union call, we hence have $|ZHK'(u)| \geq |ZHK(u)| + |ZHK(v)| \geq 2|ZHK(u)|$. Therefore, there may only be '$\log n$' reassignments of 'rep[u]' for some $u \in V$. With that, for $n = |V|$: $\mathcal{O}(n \log n)$ for all loop iteraions. However, before, we must sort all edges in increasing order, leading to $\mathcal{O}(m \log m + n \log n)$.

$$\mathcal{O}(m \log m + n \log n)$$