Algorithms and Data structures

Janis Hutz
https://janishutz.com

September 26, 2025

$$\Omega(n) \leq \Theta(n \cdot \log(n)) \leq O(n^2)$$

"*Jetzt bleibt natürlich nur noch die Frage: Geht es besser??*"
- David Steurer, 2024

HS2024, ETHZ
Summary of all algorithms discussed

Contents

1 Introduction

1.1 Sufficiency & Necessity

Sufficiency **Definition 1.1**

A condition P is called *sufficient* for Q if knowing P is true is enough evidence to conclude that Q is true. This is equivalent to saying $Q \Rightarrow P$.

Necessity **Definition 1.2**

A condition P is called *necessary* for Q if Q cannot occur unless P is true, but doesn't imply that Q is true, only that it is false if P is false.
This is equivalent to saying $P \Rightarrow Q$

1.2 Asymptotic Growth

f grows asymptotically slower than g if $\lim_{m\to\infty} \frac{f(m)}{g(m)} = 0$. We can remark that f is upper-bounded by g, thus $f \leq \mathcal{O}(g)$ and we can say g is lower bounded by f, thus $g \geq \Omega(f)$. If two functions grow equally fast asymptotically, $\Theta(f) = g$

1.3 Runtime evaluation

Identify the basic operations (usually given by the task), then count how often they are called and express that as a function in n. It is easier to note that in sum notation, then simplify that sum notation into a formula not containing any summation symbols.

1.4 Tips for Converting Summation Notation into Summation-Free Notation

1.4.1 Identify the Pattern:

- Examine the summand.
- Look for patterns related to the index variable (usually i, j, etc.). Is it a linear function, a power of i, a combination?

1.4.2 Arithmetic Series Formula

If the summand is a simple arithmetic progression (e.g., $a + bi$ where a and b are constants), use the formula:

$$\sum_{i=m}^{n}(a+bi) = (n-m+1)\left(a + b\frac{m+n}{2}\right)$$

1.4.3 Power Rule for Sums

- For sums involving powers of i, you can use the following pattern:

$$\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k+1}$$

- Remember that this rule only applies when the index starts at 1.

1.4.4 Telescoping Series

Look for terms in consecutive elements of the summand that cancel out, leaving a simpler expression after expanding. This is particularly helpful for fractions and ratios.

1.4.5 Geometric Series Formula

For sums involving constant ratios (e.g., $a \cdot r^i$ where r is the common ratio), use:

$$\sum_{i=0}^{n} a \cdot r^i = a\frac{1 - r^{n+1}}{1 - r}$$

1.4.6 Gaussian Formula

If S is an arithmetic series with n terms, then $S = \frac{n}{2} * (a + 1)$

1.4.7 Examples

The only other way (other than learning these tips) in which you are going to get better at this is by parctising. Work through examples, starting with simpler ones and moving towards more complex expressions.

Example:

Let's convert the summation: $\sum_{i=1}^{5} i$

1. **Pattern:** The summand is simply i, which represents a linear arithmetic progression.
2. **Arithmetic Series Formula:** Applying the formula with $a = 1$, $b = 1$, $m = 1$, and $n = 5$:

$$\sum_{i=1}^{5} i = (5 - 1 + 1)\left(1 + 1 \cdot \frac{1 + 5}{2}\right) = 5 \cdot 3 = 15$$

Therefore, the summation evaluates to 15.

1.5 Specific examples

$$\frac{n}{\log(n)} \geq \Omega(\sqrt{n}) \Leftrightarrow \sqrt{n} \leq \mathcal{O}\left(\frac{n}{\log(n)}\right)$$

2 Search & Sort

2.1 Search

2.1.1 Linear search

Linear search, as the name implies, searches through the entire array and has linear runtime, i.e. $\Theta(n)$.

It works by simply iterating over an iterable object (usually array) and returns the first element (it can also be modified to return *all* elements that match the search pattern) where the search pattern is matched.

Time complexity $\Theta(n)$

2.1.2 Binary search

If we want to search in a sorted array, however, we can use what is known as binary array, improving our runtime to logarithmic, i.e. $\Theta(\log(n))$. It works using divide and conquer, hence it picks a pivot in the middle of the array (at $\lfloor \frac{n}{2} \rfloor$) and there checks if the value is bigger than our search query b, i.e. if $A[m] < b$. This is repeated, until we have homed in on b. Pseudo-Code:

Algorithm 1 BINARYSEARCH(B)

```
1 $l \leftarrow 1$, $r \leftarrow n$                              ▷ Left and right bound
2 while $l \leq r$ do
3     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
4     if $A[m] = b$ then return m                    ▷ Element found
5     else if $A[m] > b$ then $r \leftarrow m - 1$          ▷ Search to the left
6     else $l \leftarrow m + 1$                            ▷ Search to the right
7 return "Not found"
```

Time complexity $\Theta(\log(n))$

Proving runtime lower bounds (worst case runtime) for this kind of algorithm is done using a decision tree. It in fact is $\Omega(\log(n))$

2.2 Sort

Sorted data proved to be much quicker to search through, but how do we sort efficiently?

First, how to check if an array is sorted. This can be done in linear time:

Algorithm 2 SORTED(A)

```
1 for $i \leftarrow 1, 2, \ldots, n-1$ do
2     if $A[i] > A[i+1]$ then return false        ▷ Item is unsorted
3 return true
```

Time complexity $\Theta(n)$

2.2.1 Bubble Sort

Algorithm 3 BUBBLESORT(A)

```
1 for $i \leftarrow 1, 2, \ldots, n$ do
2     for $j \leftarrow 1, 2, \ldots, n$ do
3         if $A[j] > A[j+1]$ then
4             exchange $A[j]$ and $A[j+1]$        ▷ Causes the element to "bubble up"
```

Time complexity $\Theta(n^2)$

2.2.2 Selection Sort

The concept for this algorithm is selecting an element (that being the largest one for each iteration, where the iteration variable determines the upper bound for the indices of the array) and swapping it with the last item (thus moving the largest elements up, whilst moving the smallest items down). This algorithm uses a similar concept to bubble sort, but saves some runtime compared to it, by reducing the number of comparisons having to be made.

Algorithm 4 SELECTIONSORT(A)

```
1 for $i \leftarrow n, n-1, \ldots, 1$ do
2     $k \leftarrow$ Index of maximum element in $A[1, \ldots, i]$        ▷ Runtime: $O(n)$
3     exchange $A[k]$ and $A[i]$
```

Time complexity $\Theta(n^2)$ because we have runtime $\mathcal{O}(n)$ for the search of the maximal entry and run through the loop $\mathcal{O}(n)$ times, but we have saved some runtime elsewhere, which is not visible in the asymptotic time complexity compared to bubble sort.

2.2.3 Insertion Sort

Insertion Sort **Definition 2.1**

Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one element at a time. It iteratively takes one element from the input, finds its correct position in the sorted portion, and inserts it there. The algorithm starts with the first element, assuming it is sorted. It then picks the next element and inserts it into its correct position relative to the sorted portion. This process is repeated for all elements until the entire list is sorted. At each iteration step, the algorithm moves all elements that are larger than the currently picked element to the right by one, i.e. an element $A[i]$ to $A[i+1]$

Characteristics and Performance **Properties**

- **Efficiency:** Works well for small datasets or nearly sorted arrays.
- **Time Complexity:**
  - Best case (already sorted): $\Omega\left(n \log(n)\right)$
  - Worst case (reversed order): $\mathcal{O}\left(n^2\right)$
  - Average case: $\Theta\left(n^2\right)$
- **Limitations:** Inefficient on large datasets due to its $\Theta\left(n^2\right)$ time complexity and requires additional effort for linked list implementations.

Algorithm 5 INSERTIONSORT(A)

```
procedure InsertionSort($A$)
    for $i \gets 2$ to $n$ do                          ▷ Iterate over the array
        $key \gets A[i]$                                ▷ Element to be inserted
        $j \gets i - 1$
        while $j > 0$ and $A[j] > key$ do
            $A[j+1] \gets A[j]$                        ▷ Shift elements
            $j \gets j - 1$
        $A[j+1] \gets key$                             ▷ Insert element
```

2.2.4 Merge Sort

Definition of Merge Sort **Definition 2.2**

Merge Sort is a divide-and-conquer algorithm that splits the input array into two halves, recursively sorts each half, and then merges the two sorted halves into a single sorted array. This process continues until the base case of a single element or an empty array is reached, as these are inherently sorted.

Characteristics and Performance of Merge Sort **Properties**

- **Efficiency:** Suitable for large datasets due to its predictable time complexity.
- **Time Complexity:**
  - Best case: $\Omega(n \log n)$
  - Worst case: $\mathcal{O}(n \log n)$
  - Average case: $\Theta(n \log n)$
- **Space Complexity:** Requires additional memory for temporary arrays, typically $\Theta(n)$.
- **Limitations:** Not in-place, and memory overhead can be significant for large datasets.

Algorithm 6 Merge Sort

```
procedure MergeSort($A[1..n], l, r$)
    if $l \geq r$ then
        return $A$                                          ▷ Base case: already sorted
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    MergeSort($A, l, m$)                                    ▷ Recursive sort on left half
    MergeSort($A, m+1, r$)                                  ▷ Recursive sort on right half
    Merge($A, l, m, r$)
procedure Merge($A[1..n], l, m, r$)                         ▷ Runtime: $\mathcal{O}(n)$
    $result \leftarrow$ new array of size $r - l + 1$
    $i \leftarrow l$
    $j \leftarrow m + 1$
    $k \leftarrow 1$
    while $i \leq m$ and $j \leq r$ do
        if $A[i] \leq A[j]$ then
            $result[k] \leftarrow A[i]$
            $i \leftarrow i + 1$
        else
            $result[k] \leftarrow A[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    Append remaining elements of left / right site to $result$
    Copy $result$ to $A[l, \dots, r]$
```

Algorithm	Comparisons	Operations	Space Complexity	Locality	Time complexity
Bubble-Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	good	$\mathcal{O}(n^2)$
Selection-Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	good	$\mathcal{O}(n^2)$
Insertion-Sort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	good	$\mathcal{O}(n^2)$
Merge-Sort	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n)$	good	$\mathcal{O}(n \cdot \log(n))$

Table 1: Comparison of four comparison-based sorting algorithms discussed in the lecture. Operations designates the number of write operations in RAM

2.2.5 Quick Sort

Quick Sort **Definition 2.3**

Quick Sort is a divide-and-conquer algorithm that selects a pivot element from the array, partitions the other elements into two subarrays according to whether they are less than or greater than the pivot, and then recursively sorts the subarrays. The process continues until the base case of an empty or single-element array is reached.

Characteristics and Performance **Properties**

- **Efficiency:** Performs well on average and for in-place sorting but can degrade on specific inputs.
- **Time Complexity:**
  - Best case: $\Omega(n \log n)$
  - Worst case: $\mathcal{O}(n^2)$ (when the pivot is poorly chosen)
  - Average case: $\Theta(n \log n)$
- **Space Complexity:** In-place sorting typically requires $\Theta(\log n)$ additional space for recursion.
- **Limitations:** Performance depends heavily on pivot selection.

Algorithm 7 Quick Sort

```
procedure QuickSort($A, l, r$)
    if $l < r$ then
        $k \leftarrow$ Partition($A, l, r$)
        QuickSort($A, l, k-1$)                              ▷ Sort left group
        QuickSort($A, k+1, r$)                              ▷ Sort right group
procedure Partition($A, l, r$)
    $i \leftarrow l$
    $j \leftarrow r - 1$
    $p \leftarrow A[r]$
    while $i > j$ do                                        ▷ Loop ends when i and j meet
        while $i < r$ and $A[i] \leq p$ do
            $i \leftarrow i + 1$                            ▷ Search next element for left group
        while $i > l$ and $A[j] > p$ do
            $j \leftarrow j - 1$                            ▷ Search next element for right group
        if $i \leq j$ then
            Exchange $A[i]$ and $A[j]$
    Swap $A[i]$ and $A[r]$                                  ▷ Move pivot element to correct position
    return $i$
```

The tests $i < r$ and $j > l$ in the while loops in Partition catch the cases where there are no elements that can be added to the left or right group.

The correct position for the pivot element is $k = j + 1 = i$, since all elements on the left hand side are smaller and all on the right hand side larger than p.

2.2.6 Heap Sort

Heap Sort — **Definition 2.4**

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It builds a max-heap (or min-heap) from the input array and repeatedly extracts the largest (or smallest) element to place it in the correct position in the sorted array.

Characteristics and Performance — **Properties**

- **Efficiency:** Excellent for in-place sorting with predictable performance.
- **Time Complexity:**
  - Best case: $\Omega(n \log n)$
  - Worst case: $\mathcal{O}(n \log n)$
  - Average case: $\Theta(n \log n)$
- **Space Complexity:** In-place sorting requires $\Theta(1)$ additional space.
- **Limitations:** Inefficient compared to Quick Sort for most practical datasets.

Algorithm 8 Heap Sort

```
procedure HeapSort($A$)
    $H \leftarrow$ Heapify($A$)
    for $i \leftarrow$ length($A$) to 2 do
        $A[i] \leftarrow$ ExtractMax($A$)
```

The lecture does not cover the implementation of a heap tree. See the specific section 2.3 on Heap-Trees

2.2.7 Bucket Sort

Bucket Sort **Definition 2.5**

Bucket Sort is a distribution-based sorting algorithm that divides the input into a fixed number of buckets, sorts the elements within each bucket (using another sorting algorithm, typically Insertion Sort), and then concatenates the buckets to produce the sorted array.

Characteristics and Performance **Properties**

- **Efficiency:** Performs well for uniformly distributed datasets.
- **Time Complexity:**
  - Best case: $\Omega(n + k)$ (for uniform distribution and k buckets)
  - Worst case: $\mathcal{O}(n^2)$ (when all elements fall into a single bucket)
  - Average case: $\Theta(n + k)$
- **Space Complexity:** Requires $\Theta(n + k)$ additional space.
- **Limitations:** Performance depends on the choice of bucket size and distribution of input elements.

Algorithm 9 Bucket Sort

```
procedure BucketSort($A, k$)
    $B[1..n] \leftarrow [0, 0, \dots, 0]$
    for $j \leftarrow 1, 2, \dots, n$ do
        $B[A[j]] \leftarrow B[A[j]] + 1$                          ▷ Count in $B[i]$ how many times $i$ occurs
    $k \leftarrow 1$
    for $i \leftarrow 1, 2, \dots, n$ do
        $A[k, \dots, k + B[i] - 1] \leftarrow [i, i, \dots, i]$   ▷ Write $B[i]$ times the value $i$ into $A$
        $k \leftarrow k + i$                                      ▷ $A$ is filled until position $k - 1$
```

2.3 Heap trees

2.3.1 Min/Max-Heap

Min-/Max-Heap **Definition 2.6**

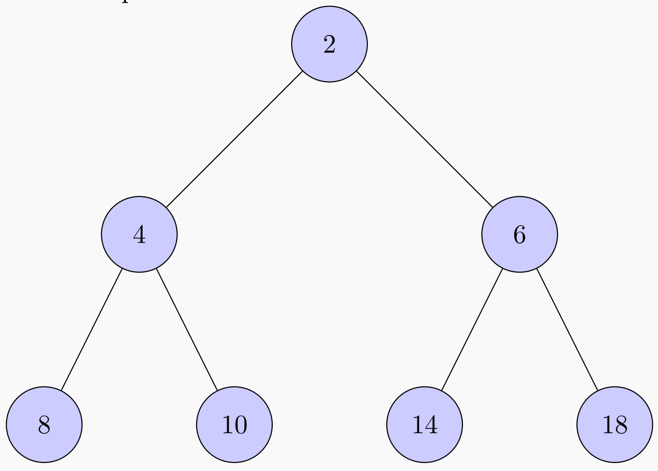
A Min-Heap is a complete binary tree where the value of each node is less than or equal to the values of its children. Conversely, a Max-Heap is a complete binary tree where the value of each node is greater than or equal to the values of its children. In the characteristics below, A is an array storing the value of a element

Characteristics **Properties**

- **Heap Property:**
  - Min-Heap: $A[parent] \leq A[child]$ for all nodes.
  - Max-Heap: $A[parent] \geq A[child]$ for all nodes.
- **Operations:** Both Min-Heaps and Max-Heaps support:
  - **Insert:** Add an element to the heap and adjust to maintain the heap property.
  - **Extract Min/Max:** Remove the root element (minimum or maximum), replace it with the last element (bottom right most element), and adjust the heap.
- **Time Complexity:**
  - Insert: $\Theta(\log n)$.
  - Extract Min/Max: $\Theta(\log n)$.
  - Build Heap: $\Theta(n)$.

Min-Heap **Example 2.1**

The following illustrates a Min-Heap with seven elements:



3 Datatypes

Abstract datatypes help with generalizing how data is stored across programming languages.

3.1 Lists

3.1.1 Arrays

An array is a data structure that stores a collection of elements of the same type in subsequent memory locations. Each element can be accessed directly using its index.

Advantages

- **Fixed Size**: Arrays have a fixed size, which makes memory allocation easy.
- **Fast Access**: Elements can be accessed in constant time $O(1)$, due to simple `base address` + `index` · `stride` calculation.
- **Cache Performance**: Neighbouring memory locations lead to better cache performance (spatial locality, see DDCA).

Disadvantages

- **Fixed Size**: The size of an array must be specified at the time of creation and cannot be changed dynamically.
- **Insertion/Deletion Overhead**: Inserting or deleting elements can be costly (if we don't allow gaps), as it requires shifting elements, resulting in a time complexity of $O(n)$.

3.1.2 Single Linked List

A linked list is a data structure consisting of nodes where each node contains data and a reference (or link) to the next node. In a single linked list, each node contains data and a pointer to the next node in the list. The last node points to `null`.

Advantages

- **Dynamic Size**: Linked lists can grow or shrink in size dynamically.
- **Efficient Insertions/Deletions**: Inserting or deleting elements at any point in the list is efficient if we already have a pointer to the desired location, with a time complexity of $O(1)$.

Disadvantages

- **Slow Access**: Elements cannot be accessed directly. We must traverse from the head node to find the desired element, thus time complexity $O(n)$.
- **Memory Overhead**: Each node requires additional memory for the pointer.

3.1.3 Double Linked List

In a double linked list, each node contains data, a pointer to the next node, and a pointer to the previous node. Java's `LinkedList` is a double linked list.

Advantages

- **As above**: Same apply as above
- **Bidirectional Traversal**: Allows traversal in both forward and backward directions.

Disadvantages The same disadvantages as for single linked lists apply here as well

3.1.4 Time Complexity Comparison

Operation	Array	Single Linked List	Double Linked List
INSERT(k, L)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
GET(i, L)	$\mathcal{O}(1)$	$\mathcal{O}(l)$	$\mathcal{O}(l)$
INSERTAFTER(k, k', L)	$\mathcal{O}(l)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
DELETE(k, L)	$\mathcal{O}(l)$	$\mathcal{O}(l)$	$\mathcal{O}(1)$

We assume that for the DELETE and INSERAFTER operation we get the memory address of k. We also assume that for the linked lists, we have stored a pointer to the last element. l is the current length of the list.

Table 2: Time Complexity Comparison of Arrays and Linked Lists

The operation INSERT(k, L) appends an element at the end of the list L, GET(i, L) returns the element at index i in list L, INSERTAFTER(k, k', L) inserts element k' after element k into the list L, while DELETE(k, L) removes the element k from the list (k being either an index (in the case of array) or memory address (in the case of linked lists)).

3.1.5 Space Complexity Comparison

Data Structure	Array	Single Linked List	Double Linked List
Fixed Size	Yes	No	No
Memory Overhead	None	1 pointer per node	2 pointers per node

Table 3: Space Complexity Comparison of Arrays and Linked Lists

3.2 Stack

We stack elements in it, so it is a first in - last out data structure. The time complexity of each operation depends on the underlying data structure used. This data structure is commonly an array or linked list, where a new item is inserted at the start of said linked list.

Operations: PUSH(k, S) pushes object k to stack S. POP(S) removes and returns the top most (most recent) element of the stack. TOP(S) returns the top most (most recent) element of the stack.

3.3 Queue

A queue has similar properties like the stack, but is first in - first out. The time complexity of each operation depends again on the underlying data structure used. Commonly an array or a linked list. **Operations:** ENQUEUE(k, Q) adds the element k to the queue Q, DEQUEUE(Q) returns the first element of the queue (the least recent element) from the queue Q.

3.4 Priority Queue

A priority queue is an extension of the queue, where we can also specify a priority for an element. This can for example be used for a notification system, which sends out urgent messages first *(although that would be quite inefficient and better be solved by using a normal queue for each priority level (if there are not that many) and then for each iteration process the notifications from the highest priority queue, then, if empty, decrease priority).*

Operations: Priority queues commonly have additional operations, like REMOVE(k, Q), which removes the element from the queue and INCREASEKEY(k, p, Q), which increases the priority of k to p if p is greater than the current priority.

The basic operations are INSERT(k, p, Q), which inserts the element k with priority p into Q and EXTRACTMAX(Q), which returns the element with the highest priority. If two elements have the same priority, the order of insertion does *not* matter and a *tie-braking* function is used to determine the priority of the two. Commonly, elements with high priority have low numbers and we then have EXTRACTMIN(Q) to find the highest priority element. They are most commonly implemented using some kind of heap, whilst a fibonacci heap is the fastest in terms of time complexity.

3.5 Dictionaries

A **dictionary** stores a collection of key-value pairs.

3.5.1 Operations:

1. **Insertion (`insert(key, value)`)**:
    - Adds a new key-value pair to the dictionary. If the key already exists, it may update the existing value.
    - Time Complexity: Average case $\Theta(1)$ (with hashing), worst case $\Omega(n)$ (when all keys hash to the same bucket).
2. **Deletion (`remove(key)`)**:
    - Removes a key-value pair from the dictionary by key.
    - Time Complexity: Average case $\Theta(1)$, worst case $\mathcal{O}(n)$.
3. **Search/Access (`get(key)` or `find(key)`)**:
    - Retrieves the value associated with a given key.
    - Time Complexity: Average case $\Theta(1)$, worst case $\mathcal{O}(n)$.
4. **Contains (`containsKey(key)`)**:
    - Checks if a key is present in the dictionary.
    - Time Complexity: Average case $\Theta(1)$, worst case $\mathcal{O}(n)$.
5. **Size/Length**:
    - Returns the number of key-value pairs in the dictionary.
    - Time Complexity: $\mathcal{O}(1)$ (stored separately).
6. **Clear**:
    - Removes all key-value pairs from the dictionary.
    - Time Complexity: $\mathcal{O}(n)$ (depends on implementation, some implementations might be faster).

3.5.2 Space Complexity:

The space complexity is dependent on the underlying data structure used to implement the dictionary.

1. **Hash Table:**
    - Average case: $\Theta(n)$, where n is the number of key-value pairs.
    - Additional space can be used for maintaining hash table buckets, which may lead to higher constant factors but not asymptotic growth in complexity.
2. **Balanced Binary Search Tree (e.g., AVL tree or Red-Black Tree):**
    - Space Complexity: $\mathcal{O}(n)$.
    - This structure uses more space compared to a hash table due to the need for storing balance information at each node.
3. **Dynamic Array:**
    - This is not a common implementation for dictionaries due to inefficiencies in search and insertion operations compared to hash tables or balanced trees.
    - Space Complexity: $\mathcal{O}(n)$.
    - Time complexity for insertion, deletion, and access can degrade significantly to $\mathcal{O}(n)$ without additional optimizations.

3.5.3 Operation time complexity

Operation	**Unsorted Arrays**	**Sorted Arrays**	**Doubly Linked Lists**	**Binary Trees**	**AVL Trees**
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(n)$	$\mathcal{O}((n))$	$\mathcal{O}(n)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$
Search	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$

Table 4: Time Complexities of Dictionary Operations for Various Data Structures

3.6 Binary tree

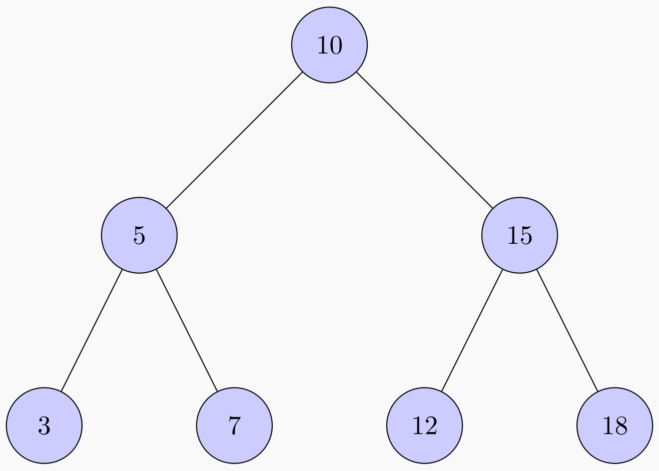
Binary Trees — **Definition 3.1**

A Binary Tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is widely used in applications such as searching, sorting, and hierarchical data representation.

Characteristics — **Properties**

- **Nodes:** Each node contains a value, and at most two child pointers (left and right).
- **Tree Types:**
  - **Full Binary Tree:** Each node has 0 or 2 children.
  - **Complete Binary Tree:** All levels except possibly the last are fully filled, and the last level is filled from left to right.
  - **Perfect Binary Tree:** All internal nodes have 2 children, and all leaves are at the same level.
- **Time Complexity (basic operations):**
  - Search: $\Theta(h)$, where h is the height of the tree.
  - Insert: $\Theta(h)$.
  - Delete: $\Theta(h)$.
- **Height:** The height of a binary tree is the longest path from the root to a leaf node.

Binary Tree — **Example 3.1**



3.6.1 Operations

Basic Operations on Binary Trees — **Properties**

- **Traversal:** Visit nodes in a specific order:
  - **Inorder (LNR):** Left, Node, Right.
  - **Preorder (NLR):** Node, Left, Right.
  - **Postorder (LRN):** Left, Right, Node.
  - **Level-order:** Breadth-first traversal using a queue.
- **Insertion:** Add a new node to the tree, ensuring binary tree properties are preserved.
- **Deletion:** Remove a node and reorganize the tree to maintain structure:
  - Replace with the deepest node (binary tree property).
  - If deleting the root in a binary search tree, replace it with the smallest node in the right subtree.
- **Searching:** Locate a node with a specific value. For binary search trees, use the property that left child $<$ parent $<$ right child.
- **Height Calculation:** Compute the maximum depth from the root to any leaf node.

3.6.2 Construction / Adding

Manual Construction of a Binary Tree **Tutorial**

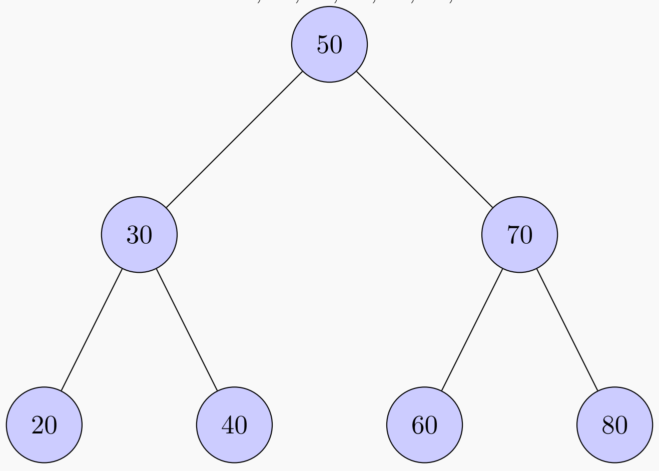
A binary tree can be created by manually following these steps:

1. Start with an empty tree.
2. Insert the root node.
3. For each subsequent value:
    - Compare it to the current node.
    - Place it as the left child if it is smaller or as the right child if it is larger.

This method results in a binary search tree if values are inserted in order.

Manual Construction of a Binary Tree **Example 3.2**

We construct a binary tree from the values: $50, 30, 70, 20, 40, 60, 80$.



Steps for Construction (Visualized Above) **Remarks**

- Insert 50 as the root.
- Insert 30: Smaller than 50, placed as the left child.
- Insert 70: Larger than 50, placed as the right child.
- Insert 20: Smaller than 50 and 30, placed as the left child of 30.
- Insert 40: Smaller than 50, larger than 30, placed as the right child of 30.
- Insert 60: Smaller than 70, larger than 50, placed as the left child of 70.
- Insert 80: Larger than 70, placed as the right child of 70.

Key Notes **Properties**

- The structure depends on the order of insertion.
- Duplicate values are either not allowed or placed in a consistent direction (e.g., always left).
- Balancing may be required for large datasets to avoid degeneration into a linked list. (AVL-Trees)

3.6.3 Deletion

Deletion in Binary Trees — **Definition 3.2**

Deleting a node from a binary tree involves removing the node and reorganizing the tree to maintain the binary tree properties. The process depends on the number of children the node has.

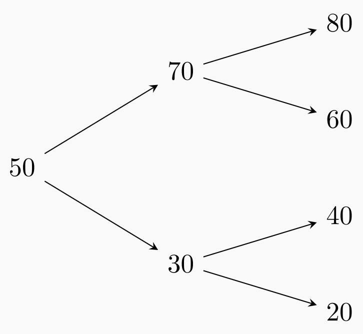
Cases for Deletion in Binary Trees — **Properties**

- **Leaf Node** (No children): Simply remove the node.
- **One Child:** Replace the node with its child.
- **Two Children:** Replace the node with the smallest node in its right subtree (inorder successor).

Deletion of a Node in a Binary Tree — **Example 3.3**

We consider the binary search tree below and perform the following operations:
- Delete 20 (leaf node).
- Delete 30 (one child).
- Delete 50 (two children).

Initial Tree:



Steps for Deletion — **Remarks**

- **Delete 20:**
  - 20 is a leaf node. Remove it directly.
- **Delete 30:**
  - 30 has one child (40). Replace 30 with 40.
- **Delete 50:**
  - 50 has two children. Find the inorder successor (smallest element in right subtree) (60).
  - Replace 50 with 60 and recursively delete 60.

Intermediate Steps and Final Tree:

Delete elements **Usage**

1. After deleting 20:

$$\begin{tikzcd}
 & & 80 \\
 & 70 \arrow[ur] \arrow[dr] & \\
50 \arrow[ur] \arrow[ddr] & & 60 \\
 & & 40 \\
 & 30 \arrow[ur] \arrow[dr] & \\
 & & {}
\end{tikzcd}$$

2. After deleting 30:

$$\begin{tikzcd}
 & & 80 \\
 & 70 \arrow[ur] \arrow[dr] & \\
50 \arrow[ur] \arrow[dr] & & 60 \\
 & 40 &
\end{tikzcd}$$

3. After deleting 50:

$$\begin{tikzcd}
 & & 80 \\
 & 70 \arrow[ur] \arrow[dr] & \\
60 \arrow[ur] \arrow[dr] & & {} \\
 & 40 &
\end{tikzcd}$$

Key Points to Remember **Properties**

- For leaf nodes, deletion is straightforward.
- For nodes with one child, bypass the node by connecting its parent to its child.
- For nodes with two children, replacing with the inorder successor ensures the binary search tree property is preserved.

3.7 AVL Tree

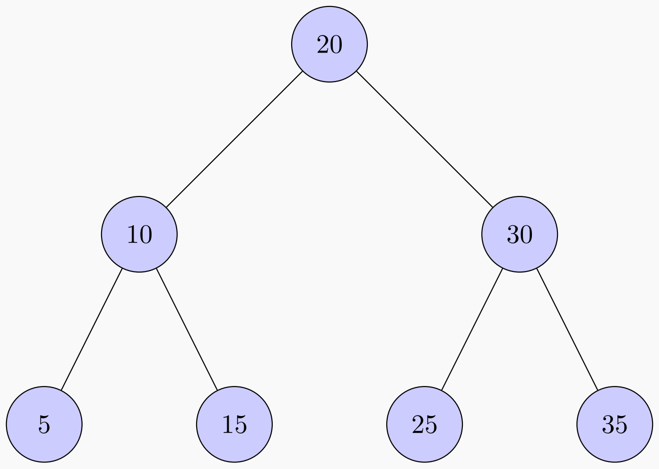
AVL Trees **Definition 3.3**

An AVL Tree is a self-balancing binary search tree in which the difference in heights of the left and right subtrees (called the balance factor) of any node is at most 1. Named after its inventors, Adelson-Velsky and Landis, it ensures logarithmic height for efficient operations. The operations in core work the same as in Binary Trees, but might require rebalancing of the tree after a BST operation is performed on the tree

Characteristics **Properties**

- **Balance Factor:** For any node, Balance Factor = Height of Left Subtree − Height of Right Subtree. The balance factor is always $-1, 0, \text{or } 1$.
- **Rotations:**
  - **Single Rotation:** Left or right rotation to restore balance.
  - **Double Rotation:** A combination of left-right or right-left rotations for more complex imbalance cases.
- **Time Complexity:**
  - Search: $\Theta(\log n)$.
  - Insert: $\Theta(\log n)$.
  - Delete: $\Theta(\log n)$.
- **Height:** The height of an AVL Tree with n nodes is $\Theta(\log n)$.

AVL-Tree **Example 3.4**



3.7.1 In more detail

Balance Factor

The key to maintaining balance in an AVL tree is the concept of the **balance factor**. For each node, the balance factor is calculated as the height of its left subtree minus the height of its right subtree:

$$\text{Balance Factor} = \text{height(left subtree)} - \text{height(right subtree)}$$

A node with a balance factor of 0, 1, or -1 is considered balanced. If the balance factor becomes 2 or -2, the tree is unbalanced and requires rebalancing.

Rotations

Rotations in AVL Trees **Definition 3.4**

Rotations in AVL Trees are operations used to restore balance when the balance factor of a node exceeds 1 or goes below -1. They are classified into single and double rotations based on the imbalance type.

Types of Rotations and Their Characteristics

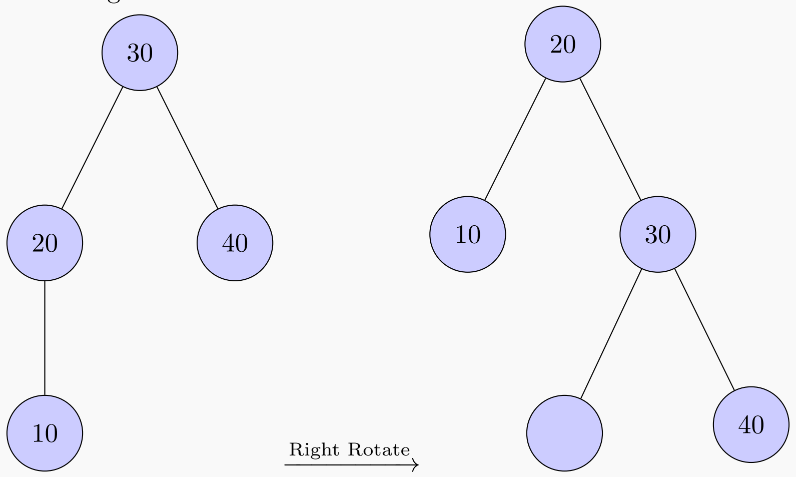
Properties

- **Single Rotations:**
  - **Right Rotation (RR):** Used to fix a left-heavy subtree.
  - **Left Rotation (LL):** Used to fix a right-heavy subtree.
- **Double Rotations:**
  - **Left-Right Rotation (LR):** A combination of a left rotation followed by a right rotation.
  - **Right-Left Rotation (RL):** A combination of a right rotation followed by a left rotation.
- **Restoring Balance:** After a rotation, the balance factors of affected nodes are recalculated to ensure the AVL property is restored.

Right Rotation (RR)

Example 3.5

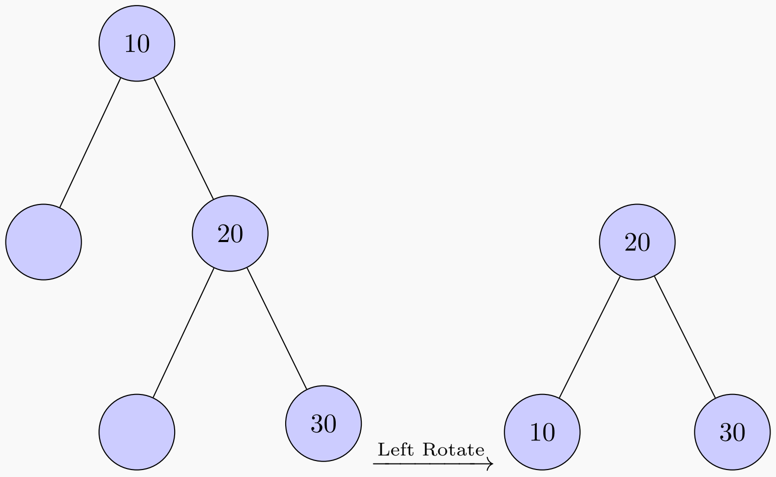
The following illustrates a right rotation around node 30:



Left Rotation (LL)

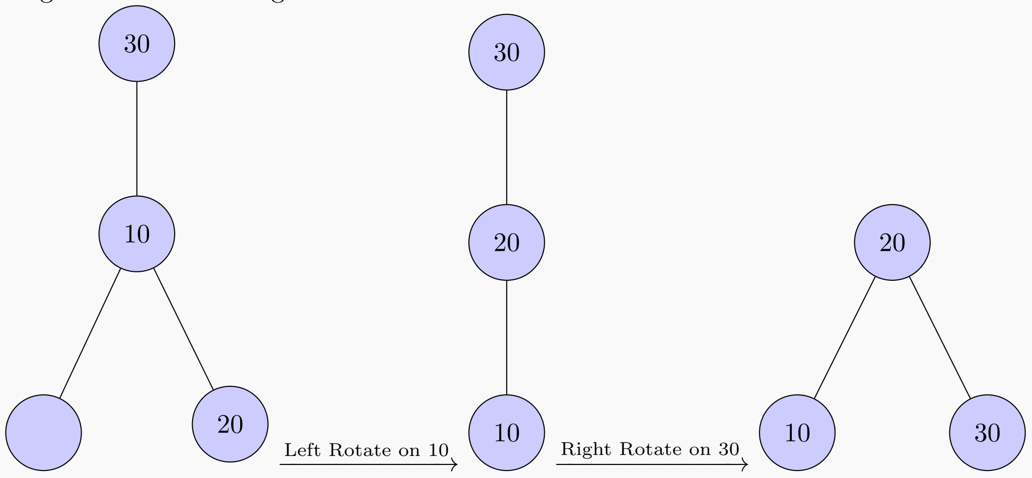
Example 3.6

The following illustrates a left rotation around node 10:



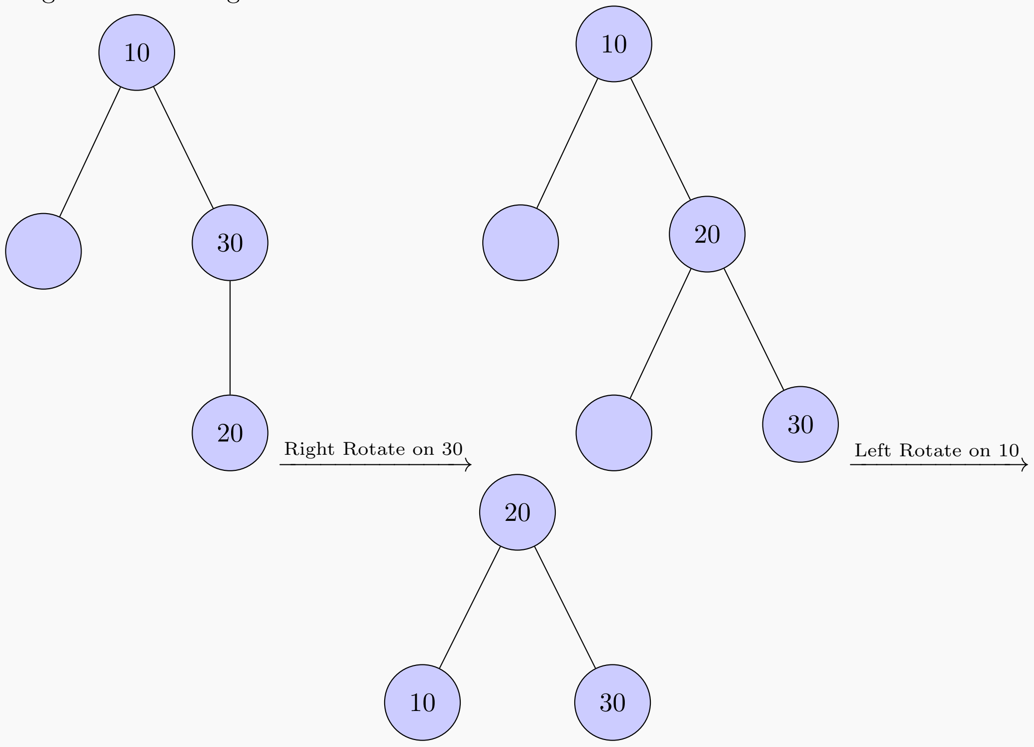
Left-Right Rotation (LR) **Example 3.7**

The following illustrates a left-right rotation:



Right-Left Rotation (RL) **Example 3.8**

The following illustrates a right-left rotation:



4 Dynamic Programming

4.1 Algorithm design

We focus on these six crucial steps when creating an algorithm in DP (for the exams at least):

Dynamic Programming Algorithm design **Usage**

I *Dimension of the DP table*: What is the sizing of the DP table and how many dimensions does it have?
II *Subproblems*: What is the meaning of each entry in the DP table? (Usually the hardest part)
III *Recursion / Recurrence relation*: How to calculate an entry of the DP table from previously computed entries? Also justify why it is correct, specifying base cases.
IV *Calculation order*: In which order do the entries of the table have to calculated to ensure that all entries needed to compute an entry have been computed. Usually to be specified as either top-down or bottom up, but in tricky cases also specify edge cases and elements that need to be specifically computed in advance and ones that can be ignored.
V *Extracting the solution*: How do we find the solution in the table after the algorithm has finished processing? Often it's the last element, but sometimes can be more complex than that
VI *Time & space complexity*: Analyze how much storage and time is required for this algorithm (often though only time needed) and note that down in big-O notation

4.2 Examples

4.2.1 Maximum Subarray Sum

The maximum subarray sum is the problem where we need to find the maximum sum of a subarray of length k

Here, the concept is to either choose an element or not to do so, i.e. the recurrence relation is $R_j = \max\{A[j], R_{j-1} + A[j]\}$, where the base case is simply $R_1 = A[1]$. Then, using simple bottom up calculation, we get the algorithm.

Algorithm 10 Maximum Subarray Sum

```
$R[1 \ldots n] \gets$ new array
$R[1] \gets A[1]$
for $j \gets 2, 3, \ldots, n$ do
    $R[j] \gets \max\{A[j], R[j-1] + A[j]\}$
```

The same algorithm can also be adapted for minimum subarray sum, or other problems using the same idea.

Time complexity $\Theta(n)$ (Polynomial)

4.2.2 Jump Game

We want to return the minimum number of jumps to get to a position n. Each field at an index i has a number $A[i]$ that tells us how far we can jump at most.

A somewhat efficient way to solve this problem is the recurrence relation $M[k] = \max\{i + A[i] | 1 \leq i \leq M[k-1]\}$, but an even more efficient one is based on $M[k] = \max\{i + A[i] | M[k-2] < i \leq M[k-1]\}$, which essentially uses the fact that we only need to look at all i that can be reached with *exactly* $l - 1$ jumps, since $i \leq M[k-2]$ can already be reached with $k-2$ jumps. While the first one has time complexity $\mathcal{O}(n^2)$, the new one has $\mathcal{O}(n)$

4.2.3 Longest common subsequence

Algorithm 11 Longest common subsequence

```
1 $L[0..n, 0..m] \leftarrow$ new table
2 for $i \leftarrow 0, 1, \dots, n$ do $L[i, 0] \leftarrow 0$
3 for $j \leftarrow 0, 1, \dots, m$ do $L[0, j] \leftarrow 0$
4 for $i \leftarrow 1, 2, \dots, n$ do
5     for $j \leftarrow 1, 2, \dots, m$ do
6         if $a_i = b_j$ then $L[i, j] \leftarrow 1 + L[i-1, j-1]$
7         else $L[i, j] = \max\{L[i, j-1], L[i-1, j]\}$
```

To find the actual solution (in the sense of which letters are in the longest common subsequence), we need to use backtracking, i.e. finding which letters we picked.

Time complexity $\Theta(n \cdot m)$ (Polynomial)

4.2.4 Editing distance

This problem is based on the LCS problem, where we want to insert, modify or delete characters to change a sequence A into a sequence B. (Application: Spell checker).

The recurrence relation is $ED(i, j) = \min \begin{cases} ED(i-1, j) + 1 \\ ED(i, j-1) + 1 \\ ED(i-1, j-1) + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{if } a_i = b_j \end{cases} \end{cases}$

Time complexity $\Theta(n \cdot m)$ (Polynomial)

4.2.5 Subset sum

We want to find a subset of a set $A[1], \dots, A[n]$ such that the sum of them equals a number b. Its recurrence relation is $T(i, s) = T(i-1, s) \vee T(i-1, s - A[i])$, where i is the i-th entry in the array and s the current sum. Base cases are $T(0, s) = \mathit{false}$ and $T(0, 0) = \mathit{true}$. In our DP-Table, we store if the subset sum can be constructed up to this element. Therefore, the DP table is a boolean table and the value $T(n, b)$ only tells us if we have a solution or not. To find the solution, we need to backtrack again.

Time complexity $\Theta(n \cdot b)$ (Pseudopolynomial)

4.2.6 Knapsack problem

We have the element i with weight $W[i]$ and profit $P[i]$.

The recurrence relation is $DP(i, w) = \begin{cases} DP(i-1, w) & \text{if } w < W[i] \\ \max\{DP(i-1, w), P[i] + DP(i-1, w - W[i])\} & \text{else} \end{cases}$. The solution can be found in $P(n, W)$, where W is the weight limit.

Time complexity $\Theta(n \cdot W)$ (Pseudopolynomial)

4.3 Polynomial vs non-polynomial

An interesting theorem from theoretical computer science is this: If the subset sum problem is solveable in polynomial time, then all non-polynomial problems are solveable in polynomial time. The same goes for the Knapsack problem and many many more.

Pseudopolynomial: The efficiency of the algorithm is dependent on the input given.

4.4 Knapsack with approximation

We can use approximation to solve the Knapsack problem in polynomial time. For that, we round the profits of the items and define $\overline{p_i} := K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor$, meaning we round down to the next multiple of K. As such, we reduce the time and space complexity by a factor of K, whilst also reducing the accuracy of the output, but only slightly lower (which is good, because overshooting would be less than ideal in most circumstances)

4.5 Longest ascending subsequence

Algorithm 12 Longest ascending subsequence

```
T[1..n] ← new table
T[1] ← A[1]
for $l \leftarrow 2, 3, \dots, n$ do $T[l] \leftarrow \infty$
for $i \leftarrow 2, 3, \dots, n$ do
    $l \leftarrow$ smallest index with $A[i+1] \leq T[l]$
    $T[l] \leftarrow A[i+1]$
return $\max\{l : T[l] \leq \infty\}$
```

Time complexity $\mathcal{O}(n \cdot \log(n))$

4.6 Matrix chain multiplication

Multiplying matrices can be more efficient if done in a certain order (because associativity exists). The problem to solve is finding the optimal parenthesis placement for the minimal number of operations.

The recurrence relation for this problem is $M(i,j) = \begin{cases} 0 & \text{falls } i = j \\ \min_{i \leq s < j}\{M(i,s) + M(s+1,j) + k_{i-1} \cdot k_s \cdot k_j\} & \text{else} \end{cases}$

Time complexity $\mathcal{O}\left(n^3\right)$

5 Graph theory

Graphs: Directed and Undirected — **Definition 5.1**

A **graph** $G = (V, E)$ consists of:

- A set of **vertices** (or nodes) V, and
- A set of **edges** E, representing connections between pairs of vertices.

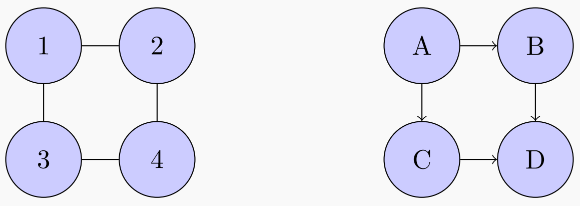
Graphs can be classified into:

- **Undirected Graphs:** Edges have no direction, represented as unordered pairs $\{u, v\}$.
- **Directed Graphs (Digraphs):** Edges have a direction, represented as ordered pairs (u, v).

Graph Theory — **Terms**

- **Adjacent (Neighbors):** Two vertices u and v are adjacent if there is an edge between them.
- **Degree:**
  - **Undirected Graph:** The degree of a vertex v is the number of edges incident to it.
  - **Directed Graph:**
    - **In-Degree:** Number of incoming edges to v.
    - **Out-Degree:** Number of outgoing edges from v.
- **Path:** A sequence of vertices where each adjacent pair is connected by an edge.
  - **Simple Path:** A path with no repeated vertices.
  - **Cycle:** A path that starts and ends at the same vertex.
- **Connected Graph:** A graph where there is a path between any two vertices.
  - **Strongly Connected:** In a directed graph, every vertex is reachable from every other vertex.
  - **Weakly Connected:** A directed graph becomes connected if the direction of edges is ignored.
- **Subgraph:** A graph formed from a subset of vertices and edges of the original graph.
- **Complete Graph:** A graph in which every pair of vertices is connected by an edge.
- **Weighted Graph:** A graph where each edge has an associated weight or cost.
- **Multigraph:** A graph that may have multiple edges (parallel edges) between the same pair of vertices.
- **Self-Loop:** An edge that connects a vertex to itself.
- **Bipartite Graph:** A graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set.
- **Tree:** A graph is a tree if it is connected and has no cycles (sufficient condition), a tree has $n - 1$ edges for n vertices (necessary condition).
- **Reachability:** A vertex u is called *reachable* from v, if there exists a walk or path with endpoints u and v.

Directed and Undirected Graphs — **Example 5.1**



5.1 Paths, Walks, Cycles

Paths, Walks, and Cycles in Graphs — **Definition 5.2**

- **Walk:** A sequence of vertices and edges in a graph, where each edge connects consecutive vertices in the sequence. Vertices and edges may repeat.
- **Path:** A walk with no repeated vertices. In a directed graph, the edges must respect the direction.
- **Cycle:** A path that starts and ends at the same vertex. For a simple cycle, all vertices (except the start/end vertex) and edges are distinct.
- **Eulerian Walk:** A walk that traverses every edge of a graph exactly once.
- **Closed Eulerian Walk (Eulerian Cycle):** An Eulerian walk that starts and ends at the same vertex.
- **Hamiltonian Path:** A path that visits each vertex of a graph exactly once. Edges may or may not repeat.
- **Hamiltonian Cycle:** A Hamiltonian path that starts and ends at the same vertex.

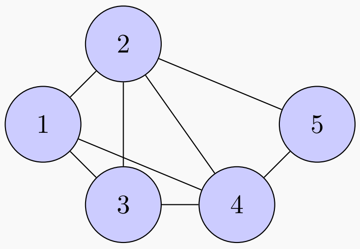
Eulerian Graphs — **Properties**

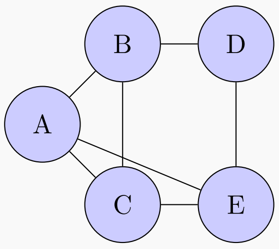
- **Undirected Graph:**
  - A graph has an Eulerian walk if it has exactly two vertices of odd degree (necessary condition).
  - A graph has a closed Eulerian walk if all vertices have even degree (necessary condition).
- **Directed Graph:**
  - A graph has an Eulerian walk if at most one vertex has *in-degree* one greater than its *out-degree*, and at most one vertex has *out-degree* one greater than its *in-degree*. All other vertices must have equal in-degree and out-degree.
  - A graph has a closed Eulerian walk if every vertex has equal in-degree and out-degree.

Hamiltonian Graphs — **Properties**

- Unlike Eulerian walks, there is no simple necessary or sufficient condition for the existence of Hamiltonian paths or cycles.
- A graph with n vertices is Hamiltonian if every vertex has a degree of at least $\lceil n/2 \rceil$ (Dirac's Theorem, sufficient condition).

Eulerian and Hamiltonian — **Example 5.2**





Key Differences Between Eulerian and Hamiltonian Concepts — **Remarks**

- Eulerian paths are concerned with traversing every **edge** exactly once, while Hamiltonian paths are about visiting every **vertex** exactly once.
- Eulerian properties depend on the degree of vertices, whereas Hamiltonian properties depend on overall vertex connectivity.

5.2 Connected Components

Connected Component — Definition 5.3

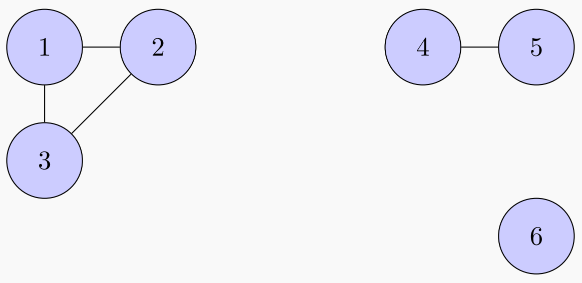
A **connected component** of a graph $G = (V, E)$ is a maximal subset of vertices $C \subseteq V$ such that:

- For every pair of vertices $u, v \in C$, there exists a path connecting u and v.
- Adding any vertex $w \notin C$ to C would violate the connectedness condition.

Key Points About Connected Components — Remarks

- **Undirected Graphs:** A connected component is a subgraph where any two vertices are connected by a path, and which is connected to no additional vertices in the graph.
- **Directed Graphs:** There are two types of connected components:
  - **Strongly Connected Component:** A maximal subset of vertices where every vertex is reachable from every other vertex (considering edge direction).
  - **Weakly Connected Component:** A maximal subset of vertices where connectivity is considered by ignoring edge directions.

Connected Components in a Graph — Example 5.3



Understanding the Example — Remarks

- In the given undirected graph:
  - **Component 1:** $\{1, 2, 3\}$ (fully connected subgraph).
  - **Component 2:** $\{4, 5\}$ (connected by a single edge).
  - **Component 3:** $\{6\}$ (an isolated vertex).
- These subsets are disjoint and collectively partition the graph's vertex set V.

5.3 Topological Sorting / Ordering

Topological Ordering **Definition 5.4**

A **topological ordering** of a directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of its vertices such that for every directed edge $(u, v) \in E$, vertex u comes before vertex v in the ordering.

Topological Ordering **Properties**

- A graph has a topological ordering if and only if it is a DAG.
- The ordering is not unique if the graph contains multiple valid sequences of vertices.
- Common algorithms to compute topological ordering:
  - **DFS-based Approach:** Perform a depth-first search and record vertices in reverse postorder.
  - **Kahn's Algorithm:** Iteratively remove vertices with no incoming edges while maintaining order.

5.4 Graph search

5.4.1 DFS

Depth-First Search (DFS) — Definition 5.5

Depth-First Search is an algorithm for traversing or searching a graph by exploring as far as possible along each branch before backtracking.

Algorithm 13 Depth-First Search (Recursive)

```
procedure DFS($v$)
    Mark $v$ as visited
    for each neighbor $u$ of $v$ do
        if $u$ is not visited then
            DFS($u$)
```

Depth-First Search — Properties

- Can be implemented recursively or iteratively (using a stack).
- Time complexity: $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.
- Used for:
  - Detecting cycles in directed and undirected graphs.
  - Finding connected components in undirected graphs.
  - Computing topological ordering in a DAG.

5.4.2 BFS

Breadth-First Search (BFS) — Definition 5.6

Breadth-First Search is an algorithm for traversing or searching a graph by exploring all neighbors of a vertex before moving to the next level of neighbors.

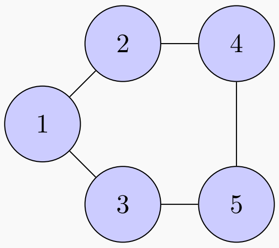
Algorithm 14 Breadth-First Search (Iterative)

```
procedure BFS($start$)
    Initialize queue $Q$ and mark $start$ as visited
    Enqueue $start$ into $Q$
    while $Q$ is not empty do
        $v \leftarrow$ Dequeue($Q$)
        for each neighbor $u$ of $v$ do
            if $u$ is not visited then
                Mark $u$ as visited
                Enqueue $u$ into $Q$
```

Breadth-First Search — Properties

- Implements a queue-based approach for level-order traversal.
- Time complexity: $\mathcal{O}(|V| + |E|)$.
- Used for:
  - Finding shortest paths in unweighted graphs.
  - Checking bipartiteness.

DFS and BFS Traversal **Example 5.4**



DFS Traversal: Starting at 1, a possible traversal is $1 \to 2 \to 4 \to 5 \to 3$.
BFS Traversal: Starting at 1, a possible traversal is $1 \to 2 \to 3 \to 4 \to 5$.

5.5 Shortest path

5.5.1 Dijkstra's Algorithm

Dijkstra's Algorithm — **Definition 5.7**

Dijkstra's Algorithm is a graph search algorithm that finds the shortest paths from a source vertex to all other vertices in a graph with non-negative edge weights.

Algorithm 15 Dijkstra's Algorithm

```
procedure DIJKSTRA($G = (V, E), s$)                                   ▷ $s$ is the source vertex
    Initialize distances: $d[v] \leftarrow \infty\ \forall v \in V, d[s] \leftarrow 0$
    Initialize priority queue $Q$ with all vertices, priority set to $\infty$
    while $Q$ is not empty do
        $u \leftarrow$ Extract-Min($Q$)
        for each neighbor $v$ of $u$ do
            if $d[u] + w(u, v) < d[v]$ then          ▷ If weight through current vertex is lower, update
                $d[v] \leftarrow d[u] + w(u, v)$
                Update $Q$ with new distance of $v$
    Return distances $d$
```

Characteristics of Dijkstra's Algorithm — **Properties**

- **Time Complexity:**
  - $\mathcal{O}\left(|V|^2\right)$ for a simple implementation.
  - $\mathcal{O}\left((|V| + |E|) \log |V|\right)$ using a priority queue.
- Only works with non-negative edge weights.
- Greedy algorithm that processes vertices in increasing order of distance.
- Common applications include navigation systems and network routing.

Dijkstra's Algorithm — **Usage**

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph (non-negative weights).

1. **Initialize:**
   - Set the distance to the source vertex as 0 and to all other vertices as infinity.
   - Mark all vertices as unvisited.
2. **Start from Source:**
   - Select the unvisited vertex with the smallest tentative distance.
3. **Update Distances:**
   - For each unvisited neighbor of the current vertex:
     - Calculate the tentative distance through the current vertex.
     - If the calculated distance is smaller than the current distance, update it.
4. **Mark as Visited:**
   - Mark the current vertex as visited. Once visited, it will not be revisited.
5. **Repeat:**
   - Repeat steps 2-4 until all vertices are visited or the shortest path to all vertices is determined.
6. **End:**
   - The algorithm completes when all vertices are visited or when the shortest paths to all reachable vertices are found.

5.5.2 Bellman-Ford Algorithm

Bellman-Ford Algorithm — **Definition 5.8**

The **Bellman-Ford Algorithm** computes shortest paths from a source vertex to all other vertices, allowing for graphs with negative edge weights.

Algorithm 16 Bellman-Ford Algorithm

```
procedure BELLMAN-FORD($G = (V, E), s$)                    ▷ $s$ is the source vertex
    Initialize distances: $d[v] \leftarrow \infty\ \forall v \in V, d[s] \leftarrow 0$
    for $i \leftarrow 1$ to $|V| - 1$ do                    ▷ Relax all edges $|V| - 1$ times
        for each edge $(u, v, w(u, v)) \in E$ do
            if $d[u] + w(u, v) < d[v]$ then
                $d[v] \leftarrow d[u] + w(u, v)$
    for each edge $(u, v, w(u, v)) \in E$ do                ▷ Check for negative-weight cycles
        if $d[u] + w(u, v) < d[v]$ then
            Report Negative Cycle
    return distances $d$
```

Characteristics of Bellman-Ford Algorithm — **Properties**

- **Time Complexity:** $\mathcal{O}(|V| \cdot |E|)$.
- Can handle graphs with negative edge weights but not graphs with negative weight cycles.
- Used for:
  - Detecting negative weight cycles.
  - Computing shortest paths in graphs where Dijkstra's algorithm is not applicable.

Bellman-Ford Algorithm — **Usage**

The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph (handles negative weights).

1. **Initialize:**
   - Set the distance to the source vertex as 0 and to all other vertices as infinity.
2. **Relax Edges:**
   - Repeat for $V - 1$ iterations (where V is the number of vertices):
     - For each edge, update the distance to its destination vertex if the distance through the edge is smaller than the current distance.
3. **Check for Negative Cycles:**
   - Check all edges to see if a shorter path can still be found. If so, the graph contains a negative-weight cycle.
4. **End:**
   - If no negative-weight cycle is found, the algorithm outputs the shortest paths.

Comparison of Dijkstra and Bellman-Ford — **Properties**

Feature	**Dijkstra's Algorithm**	**Bellman-Ford Algorithm**
Handles Negative Weights	No	Yes
Detects Negative Cycles	No	Yes
Time Complexity	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$	$\mathcal{O}(\|V\| \cdot \|E\|)$
Algorithm Type	Greedy	Dynamic Programming

5.6 MST

5.6.1 Prim's algorithm

Definition of Prim's Algorithm — **Definition 5.9**

Prim's Algorithm is a greedy algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted graph. It starts with an arbitrary node and iteratively adds the smallest edge connecting a vertex in the MST to a vertex outside the MST until all vertices are included.

Characteristics and Performance of Prim's Algorithm — **Properties**

- **Graph Type:** Works on undirected, weighted graphs.
- **Approach:** Greedy, vertex-centric.
- **Time Complexity:**
  - With an adjacency matrix: $\Theta\left(V^2\right)$.
  - With a priority queue and adjacency list: $\Theta\left((|V| + |E|)\log(|V|)\right)$.
- **Space Complexity:** Depends on the graph representation, typically $\Theta\left(E + V\right)$.
- **Limitations:** Less efficient than Kruskal's for sparse graphs using adjacency matrices.

Algorithm 17 Prim's Algorithm

```
procedure PRIM($G = (V, E)$, $start$)
    Initialize a priority queue $Q$.
    Initialize $key[v] \leftarrow \infty$ for all $v \in V$, except $key[start] \leftarrow 0$.
    Initialize an empty MST $T$.
    Add all vertices to $Q$ with their key values.
    while $Q$ is not empty do
        $u \leftarrow$ ExtractMin($Q$).
        Add $u$ to $T$.
        for each $(u, v)$ in $E$ do
            if $v$ is in $Q$ and weight$(u, v) < key[v]$ then
                $key[v] \leftarrow$ weight$(u, v)$.
                Update $Q$ with $key[v]$.
    return $T$.
```

Prim's Algorithm — **Usage**

Prim's algorithm is used to find the Minimum Spanning Tree (MST) of a graph. It starts with a single node and grows the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside it.

1. **Initialize:**
   - Pick any starting vertex as part of the MST.
   - Mark the vertex as visited and add it to the MST.
   - Initialize a priority queue to store edges by their weight.
2. **Explore Edges:**
   - Add all edges from the visited vertex to the priority queue.
3. **Pick the Smallest Edge:**
   - Choose the smallest-weight edge from the priority queue that connects a visited vertex to an unvisited vertex.
4. **Add the New Vertex:**
   - Mark the vertex connected by the chosen edge as visited.
   - Add the edge and the vertex to the MST.
5. **Repeat:**
   - Repeat steps 2-4 until all vertices are part of the MST or no more edges are available.
6. **End:**
   - The MST is complete when all vertices are visited, and the selected edges form a connected acyclic graph.

5.6.2 Kruskal's algorithm

Kruskal's Algorithm — **Definition 5.10**

Kruskal's Algorithm is a greedy algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted graph. It sorts all the edges by weight and adds them to the MST in order of increasing weight, provided they do not form a cycle with the edges already included.

Characteristics and Performance — **Properties**

- **Graph Type:** Works on undirected, weighted graphs.
- **Approach:** Greedy, edge-centric.
- **Time Complexity:** $\mathcal{O}(|E|\log(|E|))$ (for sort), $\mathcal{O}(|V|\log(|V|))$ (for union find data structure).
  Time complexity $\mathcal{O}(|E|\log(|E|) + |V|\log(|V|))$
- **Space Complexity:** Depends on the graph representation, typically $\Theta(E + V)$.
- **Limitations:** Requires sorting of edges, which can dominate runtime.

Algorithm 18 Kruskal's Algorithm

```
procedure KRUSKAL($G = (V, E)$)
    Sort all edges $E$ in non-decreasing order of weight.
    Initialize an empty MST $T$.
    Initialize a disjoint-set data structure for $V$.
    for each edge $(u, v)$ in $E$ (in sorted order) do
        if $u$ and $v$ belong to different components in the disjoint set then
            Add $(u, v)$ to $T$.
            Union the sets containing $u$ and $v$.
    return $T$.
```

Kruskal's Algorithm — **Usage**

Kruskal's algorithm finds the Minimum Spanning Tree (MST) by sorting edges and adding them to the MST, provided they don't form a cycle.

1. **Sort Edges:**
   - Sort all edges in ascending order by their weights.
2. **Initialize Disjoint Sets (union find):**
   - Assign each vertex to its own disjoint set to track connected components.
3. **Iterate Over Edges:**
   - For each edge in the sorted list:
     - Check if the edge connects vertices from different sets.
     - If it does, add the edge to the MST and merge the sets.
4. **Stop When Done:**
   - Stop when the MST contains $n - 1$ edges (for a graph with n vertices).
5. **End:**
   - The MST is complete, and all selected edges form a connected acyclic graph.

Union-Find

Union-Find Data Structure - Step-by-Step Execution

Usage

The Union-Find data structure efficiently supports two primary operations for disjoint sets:

- **Union:** Merge two sets into one.
- **Find:** Identify the representative (or root) of the set containing a given element.

Steps for Using the Union-Find Data Structure:

1. **Initialization:**
   - Create an array *parent*, where $parent[i] = i$, indicating that each element is its own parent (a singleton set).
   - Optionally, maintain a *rank* array to track the rank (or size) of each set for optimization.
2. **Find Operation:**
   - To find the representative (or root) of a set containing element x:
     (a) Follow the *parent* array recursively until $parent[x] = x$.
     (b) Apply **path compression** by updating $parent[x]$ directly to the root to flatten the tree structure:
     $$parent[x] = \text{Find}(parent[x])$$
3. **Union Operation:**
   - To merge the sets containing x and y:
     (a) Find the roots of x and y using the Find operation.
     (b) Compare their ranks:
       – Attach the smaller tree under the larger tree to keep the structure shallow.
       – If ranks are equal, arbitrarily choose one as the root and increment its rank.
4. **Optimization Techniques:**
   - **Path Compression:** Flattens the tree during Find operations, reducing the time complexity.
   - **Union by Rank/Size:** Ensures smaller trees are always attached under larger trees, maintaining a logarithmic depth.
5. **End:**
   - After performing Find and Union operations, the Union-Find structure can determine connectivity between elements or group them into distinct sets efficiently.

Performance

Properties

- MAKE(V): Initialize data structure $\mathcal{O}(n)$
- SAME(u, v): Check if two components belong to the same set $\mathcal{O}(1)$ or $\mathcal{O}(n)$, depending on if the representant is stored in an array or not
- UNION(u, v): Combine two sets, $\mathcal{O}(\log(n))$, in Kruskal we call this $\mathcal{O}(n)$ times, so total number (amortised) is $\mathcal{O}(n \log(n))$

5.6.3 Boruvka's algorithm

Definition of Borůvka's Algorithm **Definition 5.11**

Borůvka's Algorithm is a greedy algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted graph. It repeatedly selects the smallest edge from each connected component and adds it to the MST, merging the components until only one component remains.

Characteristics and Performance of Borůvka's Algorithm **Properties**

- **Graph Type:** Works on undirected, weighted graphs.
- **Approach:** Greedy, component-centric.
- **Time Complexity:** $\Theta\left((|V| + |E|)\log(|V|)\right)$.
- **Space Complexity:** Depends on the graph representation, typically $\Theta\left(E + V\right)$.
- **Limitations:** Efficient for parallel implementations but less commonly used in practice compared to Kruskal's and Prim's.

Algorithm 19 Borůvka's Algorithm

```
procedure BORUVKA($G = (V, E)$)
    Initialize a forest $F$ with each vertex as a separate component.
    Initialize an empty MST $T$.
    while the number of components in $F > 1$ do
        Initialize an empty set $minEdges$.
        for each component $C$ in $F$ do
            Find the smallest edge $(u, v)$ such that $u \in C$ and $v \notin C$.
            Add $(u, v)$ to $minEdges$.
        for each edge $(u, v)$ in $minEdges$ do
            if $u$ and $v$ are in different components in $F$ then
                Add $(u, v)$ to $T$.
                Merge the components containing $u$ and $v$ in $F$.
    return $T$.
```

Borůvka's Algorithm **Usage**

Borůvka's algorithm finds the Minimum Spanning Tree (MST) by repeatedly finding the smallest outgoing edge from each connected component.

1. **Initialize:**
   - Assign each vertex to its own component.
2. **Find Smallest Edges:**
   - For each component, find the smallest outgoing edge. After combination, the other vertex will only be evaluated if it has an even lower weight edge outgoing from it.
3. **Merge Components:**
   - Add the selected edges to the MST.
   - Merge the connected components of the graph.
4. **Repeat:**
   - Repeat steps 2-3 until only one component remains (all vertices are connected).
5. **End:**
   - The MST is complete, and all selected edges form a connected acyclic graph.

5.7 All-Pair Shortest Paths

We can also use n-times dijkstra or any other shortest path algorithm, or any of the dedicated ones

5.7.1 Floyd-Warshall Algorithm

Floyd-Warshall Algorithm — **Definition 5.12**

The **Floyd-Warshall Algorithm** is a dynamic programming algorithm used to compute the shortest paths between all pairs of vertices in a weighted graph.

Algorithm 20 FloydWarshall(G)

```
procedure Floyd-Warshall($G = (V, E)$)
    Initialize distance matrix $d[i][j]$: $d[i][j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$
    for each intermediate vertex $k \in V$ do
        for each pair of vertices $i, j \in V$ do
            $d[i][j] \leftarrow \min(d[i][j], d[i][k] + d[k][j])$
    Return $d$
```

Characteristics of Floyd-Warshall Algorithm — **Properties**

- **Time Complexity:** $\mathcal{O}\left(|V|^3\right)$.
- Works for graphs with negative edge weights but no negative weight cycles.
- Computes shortest paths for all pairs in one execution.

Floyd-Warshall Algorithm — **Usage**

The Floyd-Warshall algorithm computes shortest paths between all pairs of vertices in a weighted graph (handles negative weights but no negative cycles).

1. **Initialize:**
   - Create a distance matrix D, where $D[i][j]$ is the weight of the edge from vertex i to vertex j, or infinity if no edge exists. Set $D[i][i] = 0$.
2. **Iterate Over Intermediate Vertices:**
   - For each vertex k (acting as an intermediate vertex):
     - Update $D[i][j]$ for all pairs of vertices i, j using:

$$D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$$

3. **Repeat:**
   - Repeat for all vertices as intermediate vertices ($k = 1, 2, \dots, n$).
4. **End:**
   - The final distance matrix D contains the shortest path distances between all pairs of vertices.

5.7.2 Johnson's Algorithm

Johnson's Algorithm **Definition 5.13**

The **Johnson's Algorithm** computes shortest paths between all pairs of vertices in a sparse graph. It uses the Bellman-Ford algorithm as a preprocessing step to reweight edges and ensures all weights are non-negative.

Characteristics of Johnson's Algorithm **Properties**

- **Steps:**
  1. Add a new vertex s with edges of weight 0 to all vertices.
  2. Run Bellman-Ford from s to detect negative cycles and compute vertex potentials.
  3. Reweight edges to remove negative weights.
  4. Use Dijkstra's algorithm for each vertex to find shortest paths.
- **Time Complexity:** $\mathcal{O}(|V| \cdot (|E| + |V| \log |V|))$.
- Efficient for sparse graphs compared to Floyd-Warshall.

Johnson's Algorithm **Usage**

Johnson's algorithm computes shortest paths between all pairs of vertices in a weighted graph (handles negative weights but no negative cycles).

1. **Add a New Vertex:**
   - Add a new vertex s to the graph and connect it to all vertices with zero-weight edges.
2. **Run Bellman-Ford:**
   - Use the Bellman-Ford algorithm starting from s to compute the shortest distance $h[v]$ from s to each vertex v.
   - If a negative-weight cycle is detected, stop.
3. **Reweight Edges:**
   - For each edge $u \to v$ with weight $w(u, v)$, reweight it as:
   $$w'(u, v) = w(u, v) + h[u] - h[v]$$
   - This ensures all edge weights are non-negative.
4. **Run Dijkstra's Algorithm:**
   - For each vertex v, use Dijkstra's algorithm to compute the shortest paths to all other vertices.
5. **Adjust Back:**
   - Convert the distances back to the original weights using:
   $$d'(u, v) = d'(u, v) - h[u] + h[v]$$
6. **End:**
   - The resulting shortest path distances between all pairs of vertices are valid.

5.7.3 Comparison

Algorithm	**Primary Use**	**Time Complexity**	**Remarks**
Floyd-Warshall	AP-SP	$\mathcal{O}(\|V\|^3)$	Handles negative weights
Johnson's Algorithm	AP-SP (sparse graphs)	$\mathcal{O}(\|V\|(\|E\| + \|V\| \log \|V\|))$	Requires reweighting

Table 5: Comparison of the All-Pair Shortest path (AP-SP) algorithms discussed in the lecture

5.8 Matrix Multiplication

5.8.1 Strassen's Algorithm

Strassen's Algorithm **Definition 5.14**

The **Strassen's Algorithm** is an efficient algorithm for matrix multiplication, reducing the asymptotic complexity compared to the standard method.

Characteristics of Strassen's Algorithm **Properties**

- **Standard Multiplication:** Requires $\mathcal{O}\left(n^3\right)$ time for two $n \times n$ matrices.
- **Strassen's Approach:** Reduces the complexity to $\mathcal{O}\left(n^{\log_2 7}\right)$ (approximately $\mathcal{O}\left(n^{2.81}\right)$).
- **Idea:** Uses divide-and-conquer to reduce the number of scalar multiplications from 8 to 7 in each recursive step.
- Useful for applications involving large matrix multiplications.

Strassen's Algorithm **Usage**

Strassen's algorithm is used for matrix multiplication, reducing the computational complexity from $O(n^3)$ to approximately $O(n^{2.81})$.

1. **Divide Matrices:**
   - Split the input matrices A and B into four submatrices each:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

2. **Compute 7 Products:**
   - Calculate seven intermediate products using combinations of the submatrices:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

3. **Combine Results:**
   - Use the intermediate products to compute the submatrices of the result C:

$$C_{11} = M_1 + M_4 - M_5 + M_7, \quad C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4, \quad C_{22} = M_1 - M_2 + M_3 + M_6$$

4. **Repeat Recursively:**
   - If the submatrices are larger than a certain threshold, repeat the process recursively.
5. **End:**
   - The resulting matrix C is the product of A and B.

6 Authors

Created by Janis Hutz (`https://janishutz.com`).

Some examples were taken from the script, others were AI-Generated (because it's a bit of a pain to draw trees by hand using Tikz)