CS331 | Ch 1.1 – Complexity Theory Basics

Complexity Theory

Space Complexity : how much memory an algorithm needs

Time Complexity : how much time an algorithm needs

Time Complexity : how much time an algorithm needs

- how to measure absolute time?
- a new Computer will be faster of course than an old computer
- Super computers vs Smart phones

Solution : we consider number of steps. It is generic, machine independent etc....

Ex.

12	2	6	8	1

For analyzing algorithms we have to consider the number of items or the "input size"

Sorting 10 items → 100 ms

Sorting 1000 items → 1000ms

Important we want to make a good guess how the algorithm running time depends on the number of items (input size)

~ this is the order of growth : how the algorithm will scale and behave with the input size.

CS331 | Ch1.1 - Complexity Theory Basics

For example: we want to make sure that we end with a sorting algorithm where the algorithm is approximately linear in terms of the input.

Sorting	100 items	→	100 ms	
Sorting	1000 items	→	1000 ms	Good
Sorting	1000 items	→	10000 ms	Bad

Why is it bad? Because it does not scale well

We want to make sure that the sorting in our application is not going to freeze the application itself

~ we like deterministic algorithms where the running times are approximately linear !!!

1.2

CS 331 | Ch 1.2 - Complexity theory illustration

Ex.

First algorithm	Second algorithm
Sorting 10 items : 1ms	Sorting 10 items : 1ms
Sorting 20 items : 2ms	Sorting 20 items : 4 ms
Sorting 100 items : 10 ms	Sorting 100 items : 100 ms
"Linear Time Complexity" $O(N)$	"quadratic time complexity" $O(N^2)$

Usually we are interested in large input sizes!!! ~ asymptotic analysis

1.3

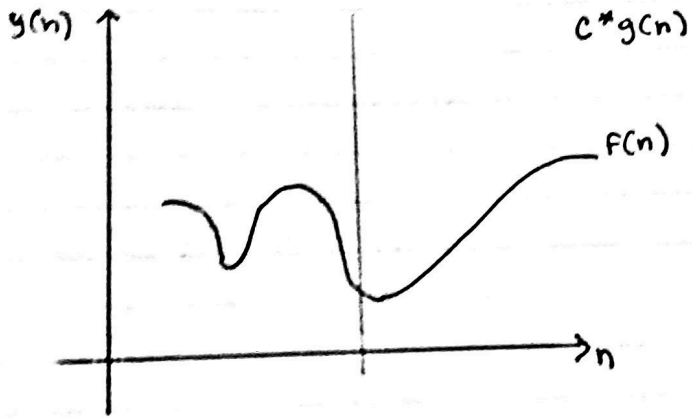
CS331 | Ch1.3 - Complexity Notations - Big O

The big ordo notation (O)

- Laundau Notation
- It describes the limiting behavior of a function, when the argument tends towards a particular value or infinity
- Is used to classify algorithms by how they respond (in their processing time or working space requirements) to changes in input size

The $F(n) = O(g(n))$ expression means, that there is some $c > 0$ value and some $n_0 > 0$ threshold value -> Such that for $n > n_0$ the $|f(n)| \leq c * |g(n)|$

Graph



What does it mean? Kind of an upper bound for the F(n) function

1.3

CS 331 | Ch1.3 - Complexity Notions - Big O

For example: bubble sort has $O(N^2)$ running time complexity

it is also true that bubble sort is in $O(N^3)$ or $O(N^4)$ or $O(N^5)$

// it is like $x < 10$ then it is also true that $x < 100$!!!

CS331 | Ch1.4 - Big Omega

The Big Omega notation (Ω)

- It describes the limiting behaviour of a function, when the argument tends towards a particular value or infinity

- It has two distinct definitions

- According to number theory: it means that an $f(n)$ function is not dominated by $g(n)$ function asymptotically

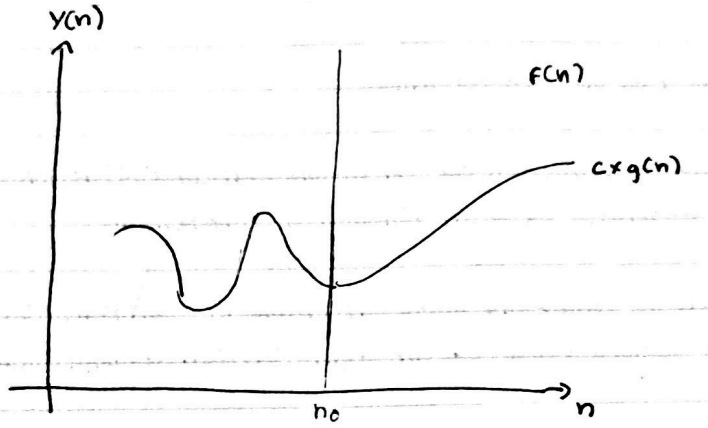
- According to complexity theory: it means that an $f(n)$ function is is bounded below by an $g(n)$ function asymptotically

The $f(n) = \Omega(g(n))$ expression means, that there is some $c > 0$ value and some $n_0 > 0$ threshold value -> such that for $n > n_0$ the $|f(n)| \geq c * |g(n)|$

1.4

CS331 | ch1.4 - Big Omega

The $f(n) = \Omega(g(n))$ expression means, that there is some $c > 0$ value and some $n_0 > 0$ threshold value -> such that for $n > n_0$ the $|f(n)| \geq c * |g(n)|$



What does it mean? kind of a lower bound f(n) function

For example: we have $\Omega(N^2)$ running time complexity

It is also true that is for $\Omega(N)$, $\Omega(\log N)$

// it is like $x > 10$ and $x > 1$ both are true

CS 331 | Ch1.9 - Big Theta

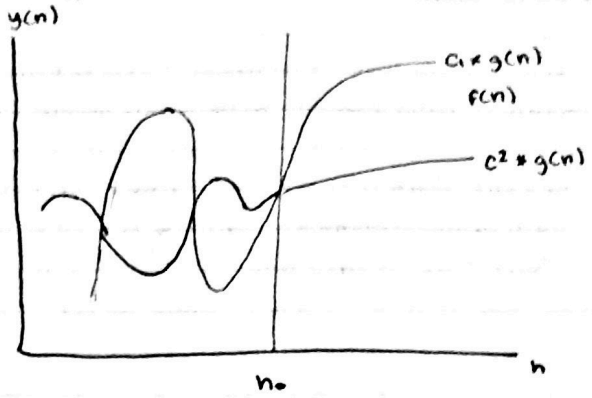
- Big Theta Notation (Θ)

  - It describes the limiting behavior of a function, when the argument tends towards a particular value or ∞

  - An $F(n)$ function is bounded by both above and below by a $g(n)$ function: So both $F(n) = O(g(n))$ and $F(n) = \Omega(g(n))$ are true!!!

The $F(n) = \Theta(g(n))$ expression means, that there is some $c_1, c_2 > 0$ values and some $n_0 > 0$ threshold value
→ Such that for $n > n_0$ the $c_1 * |g(n)| \geq |F(n)| \geq c_2 * |g(n)|$

CS 331 | Ch 1.5 - Big Theta



CS331 | Ch 1.6 - Complexity Example

Examples

$$f(n) = 3n^2 - 100n + 6$$

1.) Lets prove, that $f(n) = O(n^2)$

Definition for $O \rightarrow |f(n)| \leq C * |g(n)|$

If n is large, we can get rid of the absolute values

We have to make sure that $3n^2 \geq 100n$

Proof: So if $n > n_0 = 34 \rightarrow$ we can get rid of the absolute values $\rightarrow |f(n)| = f(n)$

$$3n^2 - 100n + 6 \leq c \cdot n^2$$

IF: $C = 3$ and $n_0 = 34$ It is going to be ok because

$$3n^2 - 100n + 6 \leq 3n^2$$

Important: If $f(n) = O(n^k)$ then $f(n) = O(n^{k+1})$ where $i > 1$ because big O is an upperbound for a given $f(n)$ function

For example: Bubble Sort has $O(n^2)$ running time

It is also true that bubble sort is in $O(N^3)$ $O(N^4)$ and $O(N^5)$....

CS331 / Ch1.6 -

$$F(n) = 3n^2 - 100n + 6$$

3.) Lets prove, that $F(n) = \Omega(n^2)$

Definition : $|f(n)| \geq c * |g(n)|$

So if $n > n_0 = 34$ -> we can get rid of the absolute values -> $|F(n)| = F(n)$

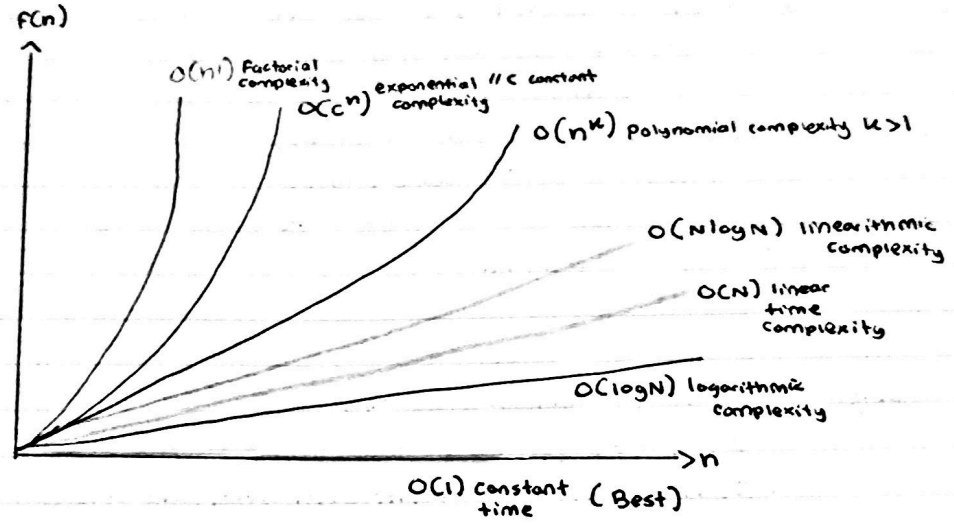
$F(n) \geq c * n^2$ -> $3n^2 - 100n + 6 > c * n^2 = 1 * n^2$

So $c = 1$ and $n_0 = 34$ then it is going to be the lower bound which is omega

And because $F(n) = O(n^2)$ and $F(n) = \Omega(n^2)$ -> $F(n) = \Theta(n^2)$!!!

CS331 | Ch1.7 - Algorithm Running Times

Algorithms Running Times



Constant time Complexity O(1) Swapping two numbers or deciding a number is odd or even

```
Function SwapNums (num1, num2):
    temp = num1
    num1 = num2
    num2 = temp
end
```

Logarithmic time complexity O(logN)

- finding an arbitrary item in a Sorted Array
- check whether there is a cycle in a graph when solving Krustal-Algorithm (with disjoint sets)

1.7

CS331 | Ch1.7 – Algorithm Running Times

Linear time complexity $O(N)$

Ex. Finding the maximum value in an array of numbers

```
Function FindMax(a[])
    max = a[0]
    for i=0 ; i < a.length ; i++
        if ( a[i] > max)
            max = a[i]

    return max
end
```

Linearithmic time complexity $O(N * \log N)$

- mergeSort, quicksort, heapsort
- Finding closest pair of points with divide and conquer method.

Polynomial time complexity $O(n^k)$ where $k = 2$ "quadratic"

- bubble sort, insertion sort
- Finding closest pair of points with brute force approach

Exponential time complexity $O(c^N)$ where c is a constant

- towers of Hanoi problem ~ $c = 2$ here in this case
- Calculating Fibonacci numbers with recursive manner ~ $c = 2$ here too
- Travelling salesman problem with dynamic programming implementation

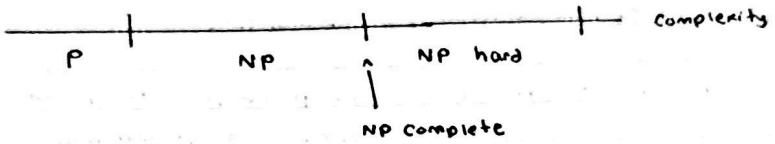
Factorial time complexity $O(N!)$

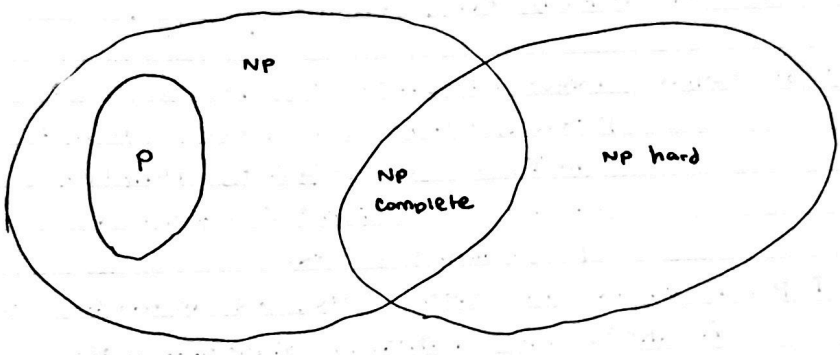
- Solving the traveling salesman problem with brute force search

// we generate all the possible routes which is $N!$ and try to find the one with the minimum overall distance.

CS331 | Ch 1.8 - Complexity Classes

Complexity Classes





P Complexity Class

- P stands for polynomial
- precise defmition : all the definition problems that can be solved with deterministic Turing-machines using polynomial amount of Computational time.

- these problems are efficiently Solvable but not all of these problems have a practical Solutions

- For example : bubble sort is in P but usually we use merge sort for sorting problems.

• NP Complexity Class

- NP stands for non-deterministic polynomial
- If we have a certain solution for the problem: we can verify this solution in polynomial time.

- P is in NP: it is true for the problems in P that we can verify the solution in polynomial time
  // we can even solve it in polynomial time
- One of the most important problems: P = NP ?

- For example: Integer factorization. Very hard to solve the problem, but if we have the solutions X and Y, we can verify it by multiplying the two numbers X * Y which is a polynomial procedure.
- TSP: if we have the solution route, we can make sure it satisfies the conditions in polynomial time.

CS 331 | ch1.8 - Complexity Classes

• Np - Complete Complexity Class

- The hardest problems in Np
- We can transform an Np-Complete problem in polynomial time // Karp Reduction"

- Important in computer science : If we manage to find a polynomial algorithm for an Np-complete problem, it means P = Np

- This is why it is an important Complexity class !!!
- If we know the solution : it is easy to verify in polynomial time
- That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.

- For example graph coloring, Hamiltonian path problems.

CS331 | Ch1.8 - Complexity classes
NP - Hard Complexity class

- problems that are at least as hard as the problems in NP class
- we can transform an NP-hard problem into an NP-complete problem in polinomial time // Karp reduction"

- For example : Hamiting problem

  - This is the problem which asks // given a program and the input, will it run forever?" Yes/No,
    So it's a decision problem.

CS331 | Ch1.9 - Analysis of Algorithms (Loops)

Constant Time Complexity O(1)

Swapping two numbers or deciding whether a number is even or odd

```
function swap ( num1, num2)
    temp = num1;
    num1 = num2;
    num2 = temp;
```

Linear running time : O(N)

When we have a for loop + inside the loop we make O(1) running time operations

```
For i=0 to N
    do O(1) operation
```

For example : we sum up items, or we check whether an item is equal to 10
// Searching in 1 dimensional array.

Quadratic Running time $O(N^2)$

When we have a for loop + we have a nested for loop + inside the loop we make O(1) running time operation

```
For i=0 to N
    For j=0 to N
        do O(1) operation
```

For example : we sum up items, or we check whether an item is equal to 10 // Searching in 2 dimensional array

```
For i=0 to N
    for j=0 to N
        do O(1) operation
```

Hece we make fewer operations but still quadratic!!!

CS331 | Ch L9 - Analysis of Algorithms (Loops)

Polynomial running time : $O(N^K)$

Basically the number of nested for loops is a good indicator for the running time complexity!!!

1.10

CS331 / Ch1.10 - Case Study O(N) - Linear Search

Linear Search

- We have an array of unsorted items / integers
- We want to find a given item
- If we know the index of the item -> we can get it in O(1) constant time complexity
- But usually we do not know the index, so we have to iterate through the array + make checks
- Input size : number of items in the array

Ex. Best Case Scenario

Find(12)

[12] 2 6 8 1 -5 22 68 7 0

Start from left most item

This is the best-case scenario for linear search

~ the first item is the one we are looking for

O(1) running time

Ex. Worst case Scenario

Find(10)

12 2 6 8 1 -5 22 68 7 0 ⊗

This is the worst-case scenario

~ we have considered every item in the array without any result O(N) running time.

Ex. Average Case Scenario

12 2 6 8 1 -5 22 68 7 0

The items we are looking for is uniformly distributed from the first index to the last index

~ on average we have to make N/2 steps to find the item we are looking for

O(N) running time.

1.10

CS331 | Ch 1.10 - Case Study O(N) - Linear Search

Ex.

```
public class App {
        public static void main( String[] args){

        int[] nums = { 1, 4, 5, 6, 10, -4, 67, 100};

       System.out.println( nums[2]); // O(1) Time complexity we know the index

       for ( int i=0; i < nums.length; i++)
       {
            if ( nums[i] == 0)
            {
                System.out.println("The index of the item looking for:" + i);
            }
    }
        // O(N) Linear Time Complexity
    }
```

1.11

CS231 | Ch 1.11 - Case Study O(logN) Binary Search

Binary Search

- Again, we want to find an item in an array
- This time the array is sorted !!!
- If we know the index of the item -> we can get it with O(1) constant time complexity.
- If we don't know the index -> we can start at the middle, on every iteration we can discard half the items.

Ex.

find(22)

22 < 56

↓

12 22 23 34 56 67 76 89 95 97

↓

12 22 23 34

↑

22 < 23

12 [22]

What is the running time of this algorithm?

How many times can we divide N by 2 until we have 1?

$$1 = \frac{N}{2^x}$$

1.11

CS331 | Ch1.11 - Case study O(logN) Binary Search

$$1 = \frac{N}{2^x}$$

$$2^x = N$$

$$\log 2^x = \log N$$

$$x = \log N$$

So we have come to the conclusion that binary search has O(logN) logarithmic running time

What does it mean

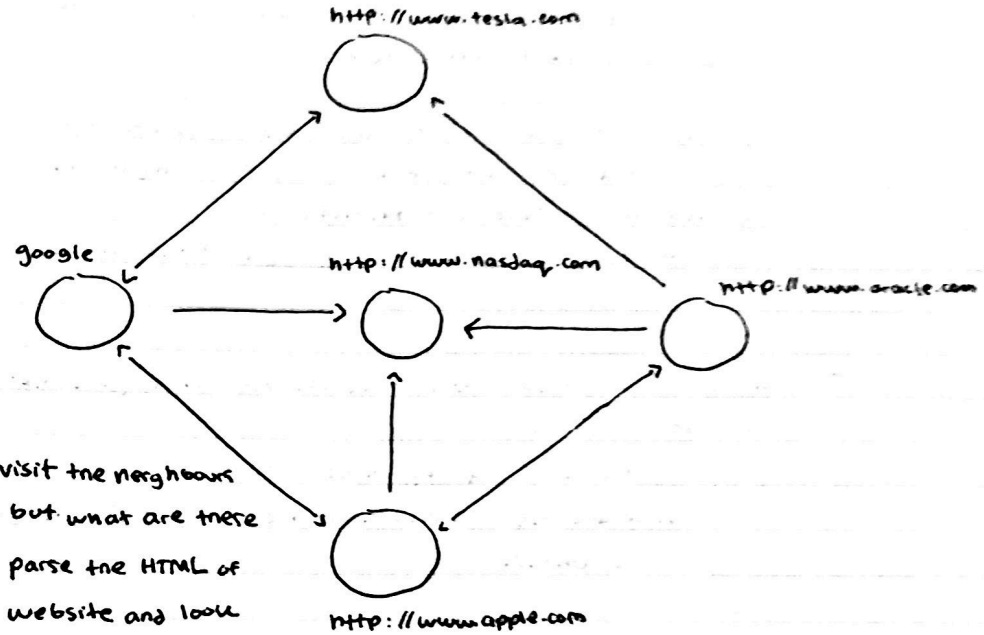
If we have 4x larger array -> algorithm has 2x running time

if we have 9x larger array -> algorithm has 3x running time.

2.1

CS 331 | Ch 2.1 - BFS - Webcrawler (core of search engines)

- Breath-First Search (core of Search Engine?)
- Webcrawler



We visit the neighbours
Ok but what are these
We parse the HTML of
the website and look
For other URL's

- It can be a network out of websites

[2.1]

CS831 | Ch 2.1 - BFS - WebCrawler (core of search engines)

- Basically the whole internet can be represented by a directed graph / network
  - with vertexes → these are the domains / URL's / websites
  - edges → these are the connections

- - - - - - - - - - - -

with breath first search we are able to traverse the web →
this is called a web-crawler that can hop from URL to url
and can observe features of the network
For example : the topology ~ degree of distribution

This kind of web crawler that traverse the web can acquire important parameters of the web

- what is the frequency visited websites
- what are the websites that are important in the network as a whole.

-

Barburabasi model → complex network theory.

3.1

CS 331 | Ch 3.1 - Depth-First Search

• Depth-First Search O(V+E)

- Depth-First Search is a widely used graph traversal algorithm besides breadth-first search

- It was investigated as a strategy for solving mazes by Tremaux in the 19th century.

- It explores as far as possible along each branch before backtracking
  // BFS is a layer by layer algorithm

- Memory complexity: a bit better than that of BFS!!!

Pseudocode

```
dfs(vertex)
    vertex set visited true
    Print vertex
    for v in vertex neighbours
        if v is not visited
            dsfs(v)
```

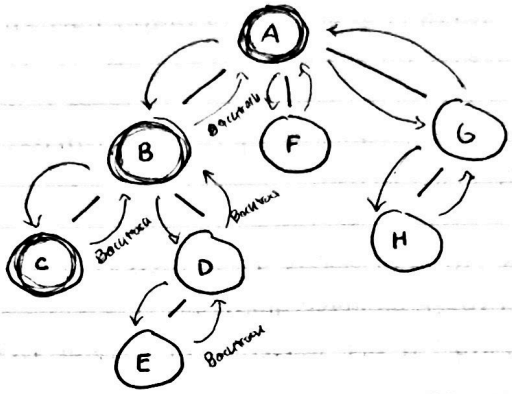
"Recursion"

```
dfs(vertex)
    Stack stack
    vertex set visited true
    Stack.push(vertex)

    While stack not empty
        actual = Stack.pop()

        For v in actual neighbours
            if v is not visited
                v set visited true
                Stack.push(v)
```

"Iterative Using Stack"

3.1

CS331 | Ch3.1 - Depth First Search



Recursively check every unmarked vertex

Applications

- Topological ordering
- Kosaraju algorithm for finding strongly connected components in a graph which can be proved to be very important in recommendation systems (Youtube)

- Detecting Cycles (checking whether a graph is a DAG or not
- Generating mazes or finding way out of a maze.

CS 331 | Ch 4.1 - Dijkstra Algorithm Basics

Greedy

Dijkstra Algorithm $O(V \times \log V + E)$

- It was constructed by Computer Scientist Edsger Dijkstra in 1956
- Dijkstra can handle positive edge weights !!! Bellman-Ford algorithm can have negative weights as well
- Several variants: It can find the shortest path from A to B, but it is able to construct a shorter shortest path tree as well -> defines the shortest paths from a source to all the other nodes.

- This is asymptotically the fastest known Single-Source Shortest path algorithm for arbitrary directed graphs with unbounded non-negative weights.

Time Complexity

- Dijkstra's algorithm time complexity: $O(V * \log V + E)$
- Dijkstra's algorithm is a greedy one: it tries to find the global optimum with the help of local minimum -> it turns out to be good !!!

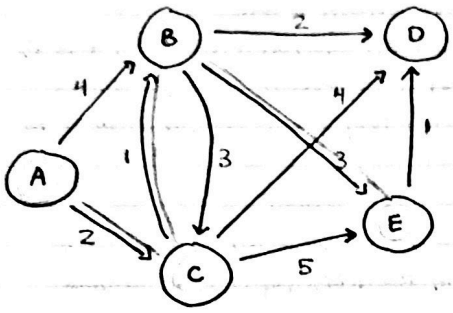
- It is greedy -> on every iteration we want to find the minimum distance to the next vertex possible -> appropriate data structures: heaps (binary or Fibonacci) or in general a Priority Queue.

4.2

CS 331 / Ch4.2 - Dijkstra Algorithm Example

Greedy

Dijkstra Shortest path From
A → E Implementation



Source = A
Target = E

Initial condition
A : 0
B : ∞
C : ∞
D : ∞
E : ∞

Unvisited nodes
{A, B, C, D, E}

Greedy A=0 path
1st visits Neighbor Nodes
A : 0
B : 4
C : 2
D : ∞
E : ∞

Unvisited Nodes
{A, B, C, D, E}

Greedy C=2 path
(visited) A : 0
B : 3
(visited) C : 2
D : 6
E : 7

Greedy B=3 path
(visited) A : 0
(visited) B : 3
(visited) C : 2
D : 5
E : 6

Unvisited Nodes
{A, B, C, D, E}

Greedy D=5 path Ignored No connection to E
(visited) A : 0
(visited) B : 3
(visited) C : 2
D : 5
E : 6

Unvisited Nodes
{A, B, C, D, E}

No change

Greedy E=6
(visited) A : 0
(visited) B : 3
(visited) C 2
(visited) D : 5
E : 6

Backtracking
A → C → B → E
with a cost of
6

4.3

CS331 | Ch4.3 - Dijkstra Pseudocode

Greedy

- Dijkstra Algorithm: Pseudocode

```
Function DijkstraAlgorithm(Graph, Source, Target)
        distance[source] = 0
        Create vertex queue Q

        For v in Graph
            distance[v] = ∞
            predecessor[v] = undefined
            add v to Q
        reached[]
        while Q not empty
            U = vertex in Q with min distance
            remove U from Q
            if U = target
                reached = U
                break
            For each neighbor V of U
                tempDist = distance[u] + distanceBetween(U,V)
                if tempDist < distance[V]
                    distance[V] = tempDistance
                    predecessor[V] = U

return reached[] // contains the shortest distances from the source to other nodes
```

4.4

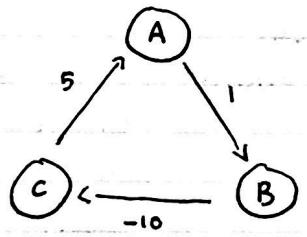
CS 331 - | Ch 4.4 - Bellman-Ford algorithm

- Invented in 1958 by Bellman and Ford independently
- Slower than Dijkstra's but more robust: it can handle negative edge weights too
- Dijkstras algorithm chooses the edge greedy, with the lowest cost
  Bellman-Ford relaxes all the edges at the same time for V-1 iterations

- Running time is $O(V * E)$ — V: number of vertices, E: number of edges

- Does V-1 iteration +1 to detect cycles: if costs decrease in the V-th iteration, than there is a negative cycle, because all the paths are traversed up to the V-1 iteration!!!

Negative Cycle:



what is the problem
If we would like to find a path with the minimum cost we have to go A → B → C → A to decreare the overall cost.

And a next cycle: decrease the cost again
And again ....

4.4

Cs 331 | Ch 4.4 - Bellman-Ford Algorithm

- **Bellman - Ford :** pseudocode

```
Function BellmanFordAlgorithm ( vertices , edges , source)

        distance [ source] = 0
        For v in Graph
              distance [ v] = ∞
              predecessor [v] = undefined  // Previous Node to the Shortest path

      for i = 1 .... num_vertices - 1
         For each edge ( u, v) with weight w in edges  // For all edges, if the distance to
              tempDist = distance [u] + w              // the destination can be shorter
              if tempDist < distance[v]                // by taking the edge, the edge
                    distance [v] = tempDist            // is updated to the
                    predeccessor [v] = u               // new lower value
    For each edge (u,v) with weight w in edges         // V-1 times -> we make
                                                          relaxation
       if distance[u] + w < distance [v]
             error: „ Negative Cycle detected"  // Since the longest possible path
                                                   without a cycle can have V-1
                                                   edges, the edges must be
                                                   scanned V-1 times to ensure
                                                   the shortest path has
                                                   been found for all nodes


          // A Final scan of all the edges is performed
             and if any distance is updated -> means
             there is a negative cycle.
```

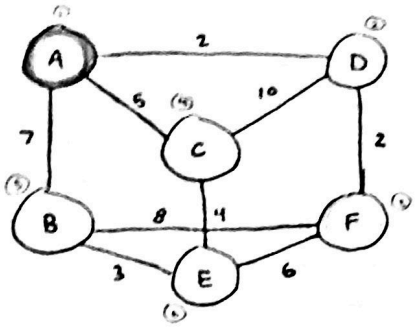
Applications:

- Cycle detection can prove very important
- Negative cycles as well -> we have to run the Bellman Ford algorithm that can handle negative edge weights by default.
- on the Forex market it can detect arbitrage situations.

CS331 | Ch 4 5 - Dijkstra Algorithm [Adjacency Matrix]

- Adjacency Matrix

Undirected graph



Matrix Edge weights

	A	B	C	D	E	F
0 A	0	7	5	2	0	0
1 B	7	0	0	0	3	0
2 C	5	0	0	10	4	0
3 D	2	0	10	0	0	2
4 E	0	3	4	0	0	6
5 F	0	8	0	2	6	0

on every iteration we consider the possible routes were able to take.

The Starting vertex is node A + initialize all the other distances to be infinity We track: the minimum distance + where did we come here (predecessor)

V	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
A		7	5	[2]	∞	∞
D		7	5		∞	[4]
F		7	[5]		10	
C		[7]			9	
B					[9]	

③

Node F connects to B, E, D

(we have already visited D)

We can get to B: min(7, 8+4)

= 7

We can get to E: min(∞, 4+6) = 10

④ Calculate the minimum value in the last row; it is 5 so node C

min(10, 5+4) = 9 we have found a shorter path

we have to calculate: min(∞, 7) for node B

②

Math.min(10+2; 5) = 5

Math.min(∞, 4) = 4 change Column F

⑤ Calculate the minimum: it is node B -> so we consider node B

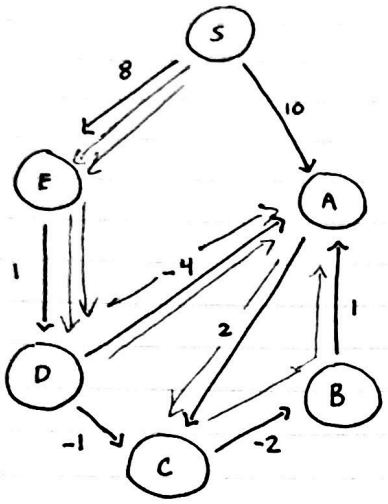
min(9, 7+3) = 9

Conclusion: red values represent what are the shortest path values from A to the given node If we want the path itself: we have to „backtrack", have to store predecessors

4.9

CS381 | Ch 4.9 - Bellman-Ford Implementation Proof I

Ex. 6 vertices = 5 iterations



0 ∞ ∞ ∞ ∞ ∞ — 0th Iteration
S A B C D E

10 12 9
0 10 ∞ ∞ ∞ 8 — 1st Iteration
S A B C D E
S→A C→B A→C E→D S→E | Relaxing all the edges

5 8 Improves the Path
0 ~~6~~ 10 ~~12~~ 9 8 — 2nd Iteration
S A B C D E | Relaxes all the edges
D→A D→C

Improves the path
5 7
0 5 ~~6~~ ~~8~~ 9 8 — 3rd Iteration
S A B C D E
C→B A→C

0 5 5 7 9 8 — 4th Iteration
S A B C D E

0 5 5 7 9 8 — 5th Iteration
S A B C D E

Shortest path

S E D A or SEDACBA

Shortest path

5.1

CS 331 | Ch5.1 - Union-Find data structure (disjoint sets)

Disjoint Sets

- Data structure to keep track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets
- Three main operations: union and find and makeSet.
- In Kruskal algorithm it will be useful: with disjoint sets we can decide in approximately O(1) time whether two vertexes are in the same set or not.

makeSet

```
function makeSet(x)
    x.parent = x
```

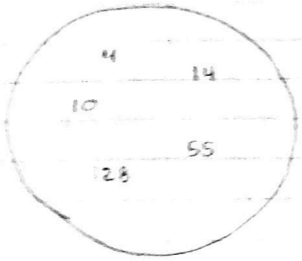
So basically we create set sol to all items / nodes.

find

```
Function find(x)
    if x.parent == x
        return x
    else
        return find(x.parent)
```

- Several items can belong to the same set → we usually represent the set with one of its items "representative of the set"
  When we search for an item with find()
  then the operation is going to return with the representative.

Find



Find(4) = 4
Find(10) = 4
Find(55) = 4

Union

```
Function union(x, y)
    xRoot = find(x)
    yRoot = find(y)

    xRoot.parent = yRoot
```

The union operation is merge two disjoint sets togetner by connection them according to the representatives

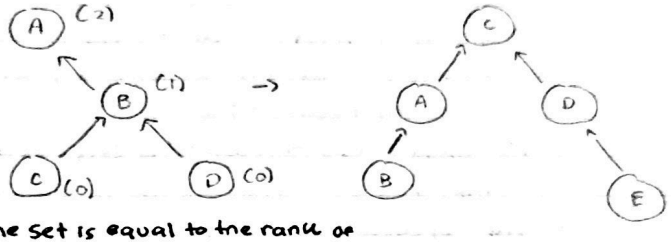
Problem: this tree like structure can become unbalanced

1.) union by rank -> always attach the smaller tree to the root of the larger
The tree will become more balance

2) path compression -> Flattening the structure of tree

CS331 | Ch5.1 - Union-Find

rank basically the depth of the tree



The rank of the set is equal to the rank of the representative // ~ the root node

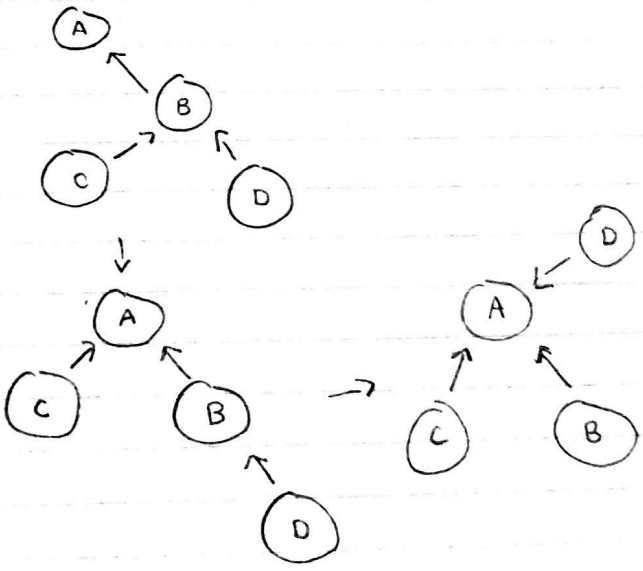
We attach the smaller tree to the larger one -> it means

pathCompression

```
Function find(x)
    if x.parent != x
        x.parent = find(x.parent)
    return x.parent
```

Find(C)

Find(D)



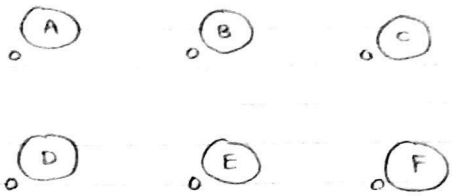
Why is it good? The next time we want to Find(c) or Find(D) it will take O(1) time because they are the direct neighbour of the representative!!! ~ the algorithm will be faster because "path compression"

CS331 | ch5.1

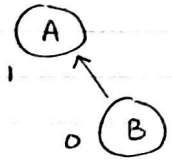
Applications

- It is used mostly in kruskal-algorithm implementation
- we have to check whether adding a given edge to the MST would form a cycle or not
- For checking this → Union-Find data structure is extremely helpful
- We can check whether a cycle is present → in asymtotically O(1) constant time complexity !!!

CS331 | Ch5.2 – Union-Find data structure

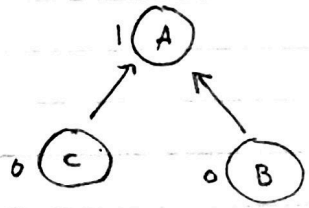


merge(A,B) we make the set with lower rank to be the child of the set with higher rank
~ it keeps the depth of the tree as low as possible !!!
+ we have to operate with the representatives always

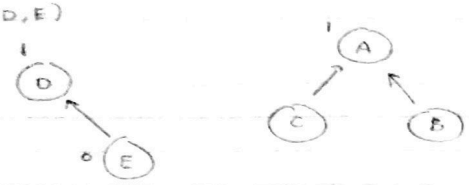


we increment the rank only if the rank parameters were the same before the merge operation

merge(B,C) First we have to find the representatives in both sets : and merge them together !!!

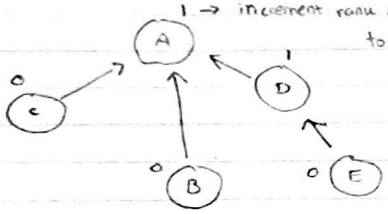


merge (D, E)

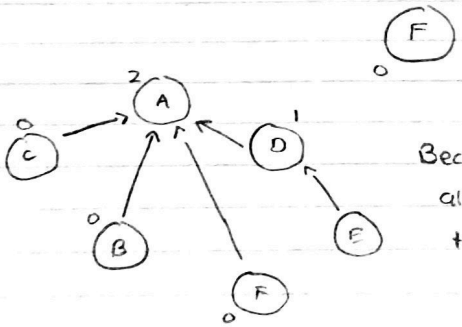


merge (E, C)

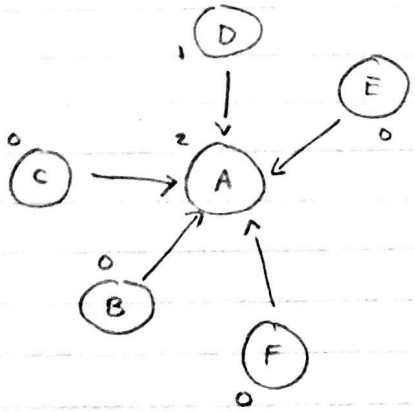
1 → increment rank parameter to 2



merge (A, F)



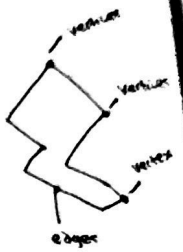
Because of the path compression -> all the nodes will connect to the representatives directly. Finding the representative takes O(1) for every node!!!



CS 331 | Ch 5.3 - Kruskal Algorithm

Spanning Trees

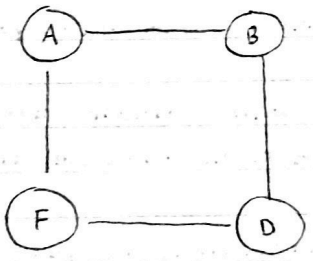
- A Spanning tree of an undirected graph G graph is a subgraph that includes all the vertices of G
- In general, a tree may have several Spanning trees
- We can assign a weight to each edge.



- A minimum Spanning tree is then a Spanning tree with weight less than or equal to the weight of every other Spanning tree.

- Has lots of applications: in big data analysis, clustering algorithms, finding minimum cost for telecommunications company laying cable to a new neighbourhood.
- Standard algorithms: Prim's - Jarnik, Kruskal - greedy algorithms.

A graph may have several Spanning tree!!!



Usually we are looking for the minimum Spanning tree: the spanning tree where the sum of edge weights is the lowest possible.

CS331 | Ch 5.3 - Kruskal Algorithm

Kruskal - Algorithm

Steps:

- We sort the edges according to their edge weights
- It can be done in $O(N * \log N)$ with mergesort or quicksort
- Union find data structure: „ disjoint set"

We start adding edges to the MST and we want to make sure there will be no cycles in the spanning tree. It can be done in $O(\log V)$ with the help of union find data structure.

// we could use heap instead.

Continued

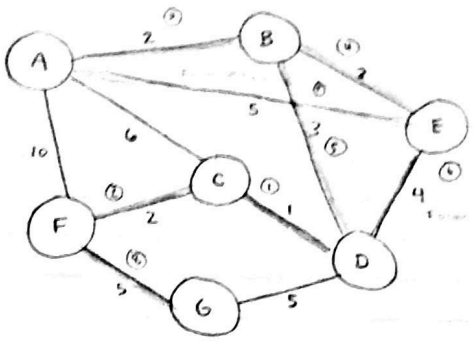
- Worst case running time: $O(E * \log E)$, so we can use it for huge graphs too.
- If the edges are sorted: the algorithm will be quasi-linear

- If we multiply the weights with a constant or add a constant to the edge weights: the result will be the same.

// in physics, an invariant is a property of the system that remains unchanged under some transformation

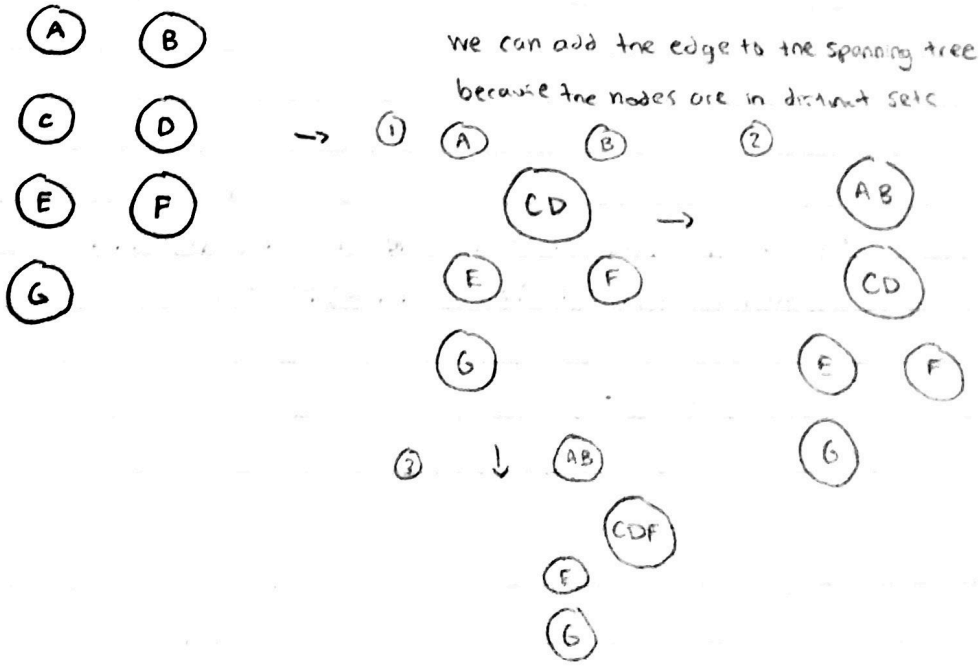
In Kruskal algorithm, Spanning trees are invariant under the transformation of these weights (addition, multiplication)

CS 331 | Ch 5.3 - Kruskal Algorithm
Ex. Kruskal Graph



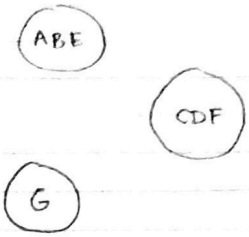
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 6, 10
On every iteration we have to make sure whether by adding the new edge → will there be a cycle or not.

Disjoint sets: at the beginning we have as many sets as the number of vertices. When adding an edge, we merge two sets together ... the algorithm stops when there is only a single set remains

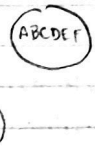


CS 331 | Ch 5.3 - Kruskal Algorithm

4



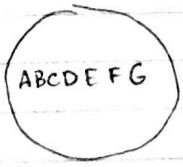
5



6 There in same set can create a cycle so do not add to the set

7

8



We have a single set so it is the end of the algorithm

Minimal cost is : 2+3+3+1+2+5 = 16

5.7

cs3911 ch 5.7 - Lazy Prim's Algorithm

Greedy

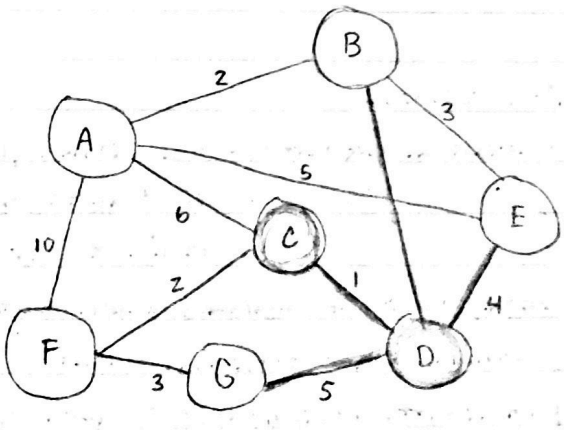
- Prim-Jarnik Algorithm
  - In Krusual Implementation we build the spanning tree seperately adding the smallest edge to the spanning tree if there it no cycle
  - In prims algorithm we build the spanning tree from a given vertex, adding the smallest edge to the MST
    - Krushal -> Edge Based Prims -> Vertex Based
- There are two implementations: lazy and eager
- Lazy Implementation: add the new neighbours edges to the heap without deleting its content
- Eager Implementation: we keep updating the heap if the distance from a vertex to the MST has changed

Average Running Time: $O(E \times \log E)$ but we need additional memory space $O(E)$

Worst Case: $O(E \log V)$

We consider edges to vertices we have not visited yet

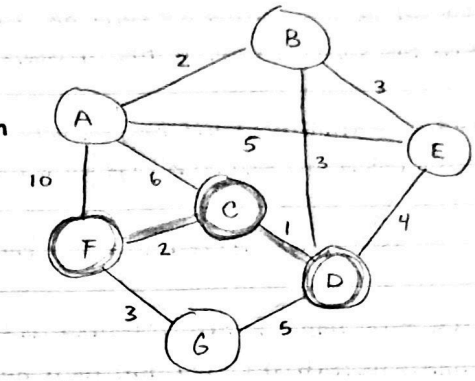
We pick the one with the lowest cost



Heap Content: DG-5, DC-1, DB-3, DE-4

5.7

CS331 / Ch5.7 - Lazy Prim's Algorithm



We pick the one with the lowest cost

Visited = { D, C, F }

Heap Content : DG-5, DB-3, DE-4, CA-6, CF-2

↓ add F's neighbour vertex

Heap Content : DG-5, DB-3, DE-4, CA-6, FA-10, FA-G-3

Visited = { D, C, F, B }

Heap Content : DG-5, DE-4, CA-6, FA-10, FG-3, BA-2, BE-3

Visited = { D, C, F, B, A }

Heap Content : DG-5, DE-4, CA-6, FA-10, FG-3, BE-3, AE-5

Visited { D, C, F, B, A, E, G }

Prims vs Kruskal

- Kruskal can have better performance if the edges can be sorted Perfoms better in (Sparse graph)

we have visited all the nodes, so we are done!

the minimum cost : 14

- Prims algorithm is better significally faster in the limit when your got a really dense graphs with many more edges than vertices.
- prims is better if the number of edges to vertices is high (dense graphs)
- Kruskal can have better performance if the edges can be sorted in linear time O(n) or the edges are already sorted.

CS 331 | ch10.1 - Recursion Intro

Recursion

- A method / procedure where the solution to a problem depends on solutions to smaller instances of the same problem.
- So we break the tasks into smaller subtasks
- The approach can be applied to many types of problems and recursion is one of the central ideas of computer science.
- We have to define a base case in order to avoid infinite loops.
- We can solve problems with recursion or with iteration.

Ex: We want to add the first N numbers:

Usually we use a simple for / while loops but we can solve it with the help of recursive method calls.

```
public int iterationSum(int N) {
    int result = 0;
    for(int i=1 ; i<N ; ++i)
        result = result + i;
    return result;
}

// iteration
```

```
public int recursionSum(int N) {
    if( N == 1) return 1;
    return N + recursionSum(N-1);
}

// recursion
```

CS 331 | Ch10.1 - Recursion Intro

Head vs Tail Recursion

- If the recursive call occurs at the end of a method -> it is called tail recursion.

- Tail recursion is similar to a loop.
- The method executes all the statements before jumping into the next recursive call

- If the recursive call occurs at the beginning of a method, it is called a head recursion.
- This method saves the state before jumping into the next recursive call

Ex:

```
public void tail(int N) {
    if (N==1) return;
    System.out.println(N);
    tail(N-1);
}
// Tail Recursion
```

```
public void head(int N) {
    if (N==1) return;
    head(N-1);
    System.out.println(N);
}
// Head Recursion
```

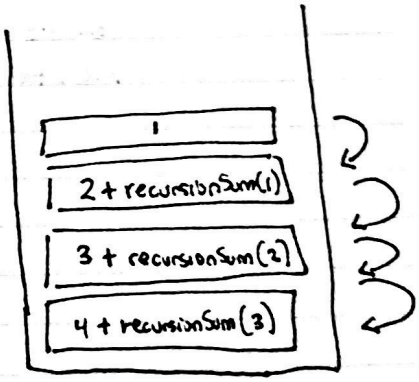
CS 381 | ch 10.1 - Recursion Intro

Stack with Recursion

- We have to trace during recursion who called the given method and what arguments are to be handed over.
- and we have to trace the pending calls !!!
- We just need a single stack data structure: the operating system does everything for us.
- These important information are to be pushed to the stack
- Values are to be popped off from the stack.

Ex: We want to add the first N numbers:
- Usually we use a simple for / while loop but we can solve it with the help of recursive method calls.

```
public int recursionSum(int N) {
    if( N ==1) return 1;
    return N + recursionSum(N-1);
}

// Recursion
```



Stack

10.1

CS331 | ch10.1 - Recursion Intro

When we used recursionSum(int N) method:

```
recursionSum(4)
  recursionSum(3)
    recursionSum(2)
      recursionSum(1)
      return 1
    return 2+1
  return 3+2+1
return 4+3+2+1
```

So these method calls and values are stored on the stack

Comparing recursive implementation against iterative implementation

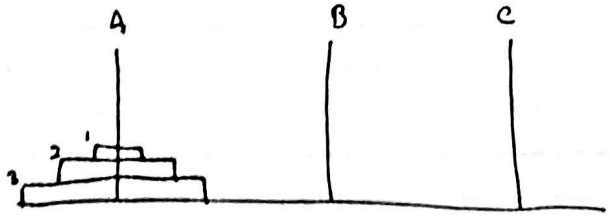
-> recursion is at least twice slower because first we unfold recursive calls (pushing them on the stack) until we reach the base case and then we traverse the stack and retrieve all recursive calls.

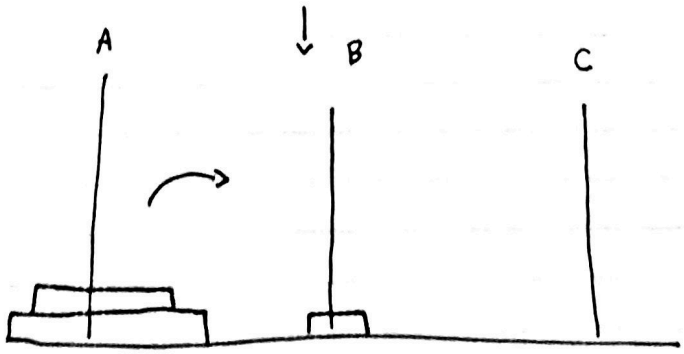
CS 3311 Ch10.6 - Towers of Hanoi intro

Towers of Hanoi

- It consists of three rods and number of disus of different sizes which can slide onto any rod.
- The puzzle starts with the disus in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape
- The minimum number of moves required to solve a Tower of Hanoi problem is $2^n - 1$ // $O(2^n)$ exponential time complexity.

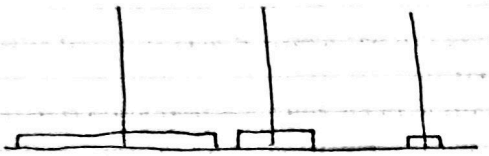
- We have some rules
  - only one disk can be moved at a time.
  - each move consists of taking the upper disk from one of the stacks and placing it on top of another stack -> a disk can only be moved if it is the uppermost disk on a stack.
  - no disk may be placed on top of a smaller disk.



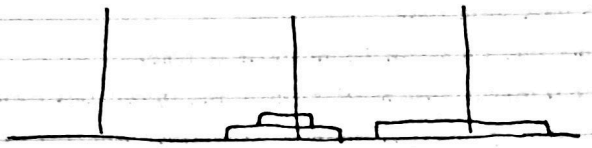


10.6

CS 331 | Ch 10.6 - Towers of Hanoi intro



There will be always a situation like this: we have managed to shift n-1 plates to the auxiliary rod -> we just have to put the largest to the last rod.



CS331 | Ch11.1 - Selection algorithms

Selection Algorithms

- Selection algorithm is an algorithm for finding the kth smallest / largest number in a list/array such a number is called the kth order statistic

- For example: finding maximum, minimum or median.
- The aim is to acheive O(N) linear time complexity for this particular operation!!! ~ not that easy.

- Methods: quickselect, median of medians method...

Sorting

- Intuition: let's sort the array in which we want to find the given item.
- After sorting -> we can access the item with the help of the index.
- For example: if we sort an array in descending order, the array[0] yeilds the maximum item.

- Inefficient approach: if we want to find just a single item (maximum, minimum or median)

- Efficient approach: if we want to find just a single several items at the same time

- Why? O(nlog(n)) vs O(N)

- Intuition: selection can be reduced to sorting and vice versa.

CS331 | Ch11.1 - Sorting/selection algorithms

Data Structures

- We can use a data structure in order to find items
- Sublinear time can be reached: $O(\log(n))$
- For example: Construct a balanced binary search tree or a heap
- Problem: it has some memory complexity, we have to construct the data structure
- So overall not the best solution

Online Selection

- We keep downloading data and we want to find items at runtime
- Problem: we do not know all values in advance
- We will not be able to construct an algorithm that finds the best solution: we can have a good guess ... a value that probably the one we are looking for.

„Secretary Problem"

11.2

CS3B1 | Ch11.2 - Quickselect intro - Hoare algorithm

Quickselect

- It is a selection algorithm to find the Kth smallest / largest item in an unordered array.

- Hoare constructed the algorithm -> „Hoare-algorithm"
- It has a very good average case running time: O(n)
- Worst case scenario: $O(n^2)$
- In-place algorithm
- Concept is similar to quicksort
  - Choose a pivot element at random
  - Partition the array
  - Instead of recursing into both sides, we just take one side
  - O(NlogN) -> O(N)

Hoare algorithm

1) Partition

- The partition method is just for partitioning the array according to the pivot
  - Choose a pivot value at random: we generate a random number in the range [FirstIndex, lastIndex]
- rearranges the list in a way that all elements less than pivot are on left side of pivot and others on right. It then returns index of the pivot element.

Ex:

7	-2	5	8	1	6

Generate a pivot item at random!!!

Time Complexity

Best case performance	O(N)
Worst case performance	$O(N^2)$
Average	O(N)

CS 331 | Ch11.2 -

7	-2	5	8	1	6

We are done, we return the index of the pivot! of course
In the course of the algorithm, we may have
to make several partitions procedures

Important: We just need one „half" of the array
Left Side: If we want to find the
small items
For example: third smallest value

Right Subarray: we want the large items
For example: Second largest value.

2) Select
-After the partitioning -> we are looking for the Kth smallest item
for example. So we keep the left subarray in the partition phase.
After partition there are 3 Cases
1.) K == pivot
- It means we have found the Kth smallest item we are after,
because this is how partitioning works: there are exactly
K-1 items that are smaller than the pivot //
in this case pivot == K !!!
2.) K < pivot
The Kth smallest item is on the left side of the pivot,
thats why we can

11.3

CS 331 | Ch11.3 - Quickselect Simulation

Hoare Algorithm

1	-2	5	8	7	6

quickselect.select(2) So we are looking for the second largest item

Pseudocode

```
Select ( indexFirst , indexLast, k)                    //k' = k-1
    pivot = partition (indexFirst, indexLast)          //    = (2)-1
    if( pivot > k')                                    //  k' = 1
        return select (indexFirst, pivot-1, k)
    else if ( pivot < k')
        return select(pivot +1, indexLast, k')
    return nums[k]
```

8	7	6	5	1	-2

[11.3]

CS381 | ch11.3 - Quickselect Simulation

Hoare algorithm

Pseudocode

```
partition(indexFirst, indexSLast)
    pivot = random(indexFirst, indexLast)
    Swap(indexLast, pivot)

For(int i= indexFirst; i < indexLast; i++)
    if(nums[i] > nums[indexLast])
        Swap(i, indexFirst)
        indexFirst++;

Swap(indexFirst, indexLast);
return indexFirst
```

CS 331 | Ch11.5 - Advanced Selection, median of medians, introselect

Median of medians Select:

It is basically the same as quickselect, the only difference is how we get the pivot value.

- quickselect: we generate a random index
- median of medians: we calculate the approximate median

O(N) running time guaranteed

O(logN) worst case memory complexity

CS 381 | Ch 11.5 - Advanced Selection, median of medians, introselect

Introselect:

It is a hybrid algorithm: combining two algorithms in order to take advantage of the best features

- quickselect is in place algorithm, this is the advantage.
- median of medians select: always fast O(N)

Let's combine them: introselect starts with quickselect in order to obtain good average performance, and then falls back to median of medians if progress is slow.

CS331 | ch11.6 - online Selection

- Online algorithm related problem
- We want to find the kth smallest / largest item of a stream
- Partition based algorithms can not be used: we do not know the data in advance.

- The problem is to select (under these constraints) a specific element of the input sequence of data with largest probability.

Secretary Problem

- very important problem of optimal stopping theory
- Also known as „best choice problem"

▷ Problem: we want to hire the best secretary of of N applicants. Applicants are interviewed one by one + after rejecting, the applicant can not be recalled. We can rank the applicant among all applicants interviewed so far, but we are unaware of the quality of yet unseen applicants.

▷ What is the optimal strategy?

▷ We want to maximize the probability of selecting the best applicant.

11.6

CS 331 | Ch11.6 - Online selection Secretary Problem

▷ If we can make the decision at the end : we just have to make a maximum finding

▷ It can be done in O(N) ... no problem

▷ But we have to make the decision immediately

Solution

always reject the first $\frac{n}{e}$ applicants and then

we have to stop at the one who is better than all the previous ones.

e: natural log ~ 2.718...

- it is very popular problem because it has a well defined Solution.
- the probability of choosing the best applicants is $\frac{1}{e}$
- So 37% chance we find the optimal one.

CS331 | Ch12.1 - Backtracking intro

Backtracking

- It is a form of Recursion!!!
- General algorithm for finding all solutions to some computational problems → "Constraint Satisfaction" problems.
- We incrementally build candidates to the solutions.
- If partial candidate A cannot be completed to a valid solution: we abandon A as a solution.
- For example: eight-queens problem or Sodoku.
- Backtracking is often much faster than brute force enumeration of all complete candidates, because it can eleminate a large number of candidates with a single test.
- Backtracking is an important tool for solving Constraint Satisfaction problems → Combinatorial optimization problems.

The method

- The partial candidates are represented as the nodes of a tree structure.
- „Potential Search trees"
- Each partial candidate is the parent of the candidates that differ from it by a single extension step.
- The leaves of the tree are the partial candidates that cannot be extended any further.
- The backtracking algorithm traverses this search tree recursively, from the root down (like DFS)

CS331 | Ch12.1 - Backtracking Intro

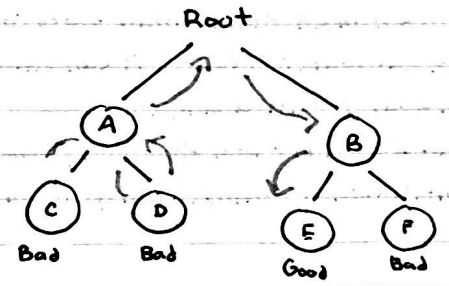
The Method Continued

▷ This is why backtracking is sometimes called depth-first search!!!

1.) For every node the algorithm checks whether the given node can be completed to a valid solution.

2.) If it can not → the whole subtree is skipped!!!

3.) Recursively enumerates all subtree of the node.



- we have several options: we can choose A or B at the beginning
- after every choice → we have a new set of options
- if we make good choices → we end up with a good state
- if not we have to backtrack!!!

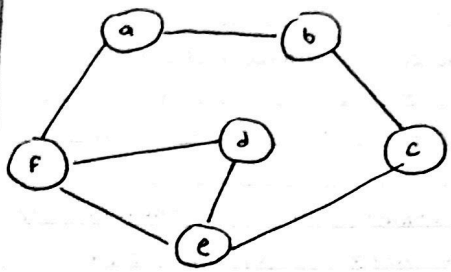
CS 331 | Ch12.8 - Hamiltonian Cycle Intro

Hamiltonian Cycle

$G(V, E)$

V: vertices in the graph

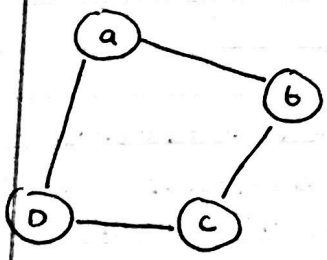
E: edges in the graph



	a	b	c	d	e	f
a	0	1	0	0	0	1
b	1	0	1	0	0	0
c	0	1	0	0	1	0
d	0	0	0	0	1	1
e	0	0	1	1	0	1
f	1	0	0	1	1	0

AC(i,j) = { 1 - if there is a connection between i and j ;
0 - if no connection }

Hamiltonian Path



A <u>hamiltonian path</u> in an undirected graph is a path that visits every node exactly once!!!

Hamiltonian cycle : the first node and the last node of the path are the same vertexes.

StartingPoint == EndPoint

A valid hamiltonian path is : { a b c d a }

There may be several hamiltonian path in a given graph!!!

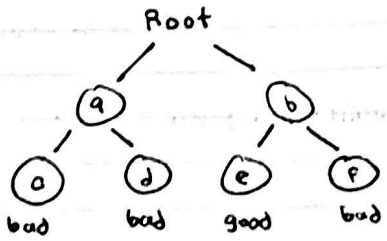
CS331 / Ch12.5 - Hamiltonian Cycle Intro

Hamiltonian problem

- Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem.
- This is an NP-Complete problem !!!
- Dirac-principle : a simple graph with N vertices is Hamiltonian if every vertex has degree N/2 or greater (degree is the number of edges of a vertex)
- Important fact : finding a Hamiltonian path is NP-complete, but we can decide whether such path exists in linear time complexity with topological ordering.

Solutions:

- Constructing a tree



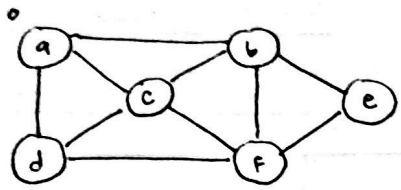
This tree is an abstract model of the possible sequences of choices we could make. Here we do a depth-first search on the tree.

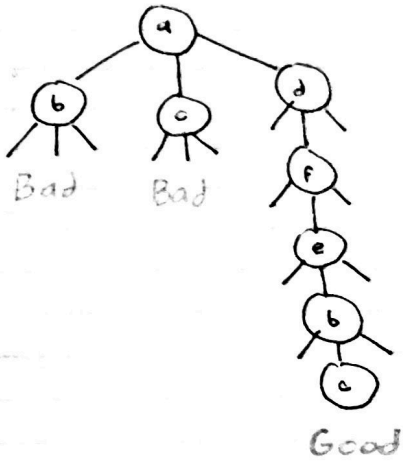
problem: hard to construct a tree if N is big

- Backtracking :
- we use recursion to solve the problem
- Create an empty path array and add vertex 0 to it as the starting vertex
- Add other vertices, starting from the vertex 1
- Before adding a vertex, check whether it is adjacent to the previously added vertex + make sure it is not already added
- If we find such a vertex → we add the vertex as part of the solution
- If we do not find a vertex → we return false.

CS 331 / Ch 12.6 - Hamiltonian Cycle illustration

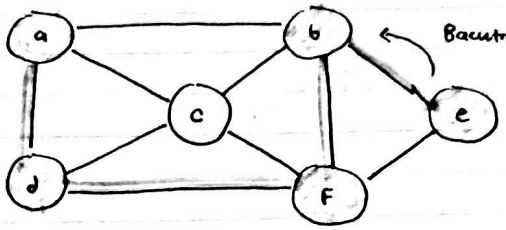
State-Space Tree



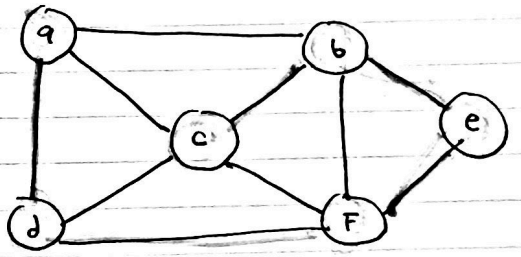


12.6

CS331 | Ch12.6 - Hamiltonian Cycle illustration



Not a Hamiltonian Cycle



we have found a Hamiltonian-Cycle in graph

$\{a, d, f, e, b, c, d\}$

12.8

CS331 | Ch12.8 - Coloring Problem Intro

Backtracking Coloring Problem

- NP-Complete problem !!! ~ exponential running time
- Problem: Coloring the vertices of a graph such that no two adjacent vertices share the same color.
- This is called vertex coloring
- Reached popularity with the general public in the form of the popular number puzzle Sodoku.
- The smallest number of colors needed to color a graph G is called its chromatic number.
- There may be more than one solution: for example we can color a graph with 4 vertices in 12 ways with 3 colors!!!

Applications

- Bipartite graphs
  - Determine if a graph can be colored with 2 colors is equivalent to determining whether or not the graph is bipartite, and thus computable in linear time using breadth first search
  - Bipartite graph: a graph whose vertices can be divided into two disjoint sets U and V (U and V are independent sets) such that every edge connects a vertex in U to one in V.

12.8

CS 381 | Ch12.8 - Coloring Problem Intro

Applications

- Making Schedules for university students problem.
- Radio frequencies assignment problem.
- Register allocation
  - In compiler optimization -> register allocation is the process of assigning a large number of target program variables onto a smaller number of CPU registers.
- Map Coloring

Solutions

- Greedy approach -> finds the solution but not the most optimal one. - It may uses more colors the necessary !!!
- Powell-Welsh algorithm -> relies on sorting the nodes according to the degrees + we start assigning colors to the nodes with the most neighbours !!!

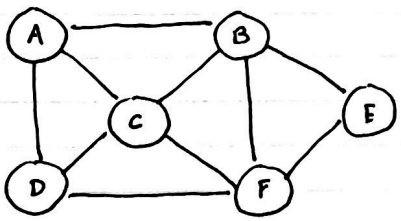
▷ Backtracking

CS331 | Ch12.8 - Coloring Problem Intro

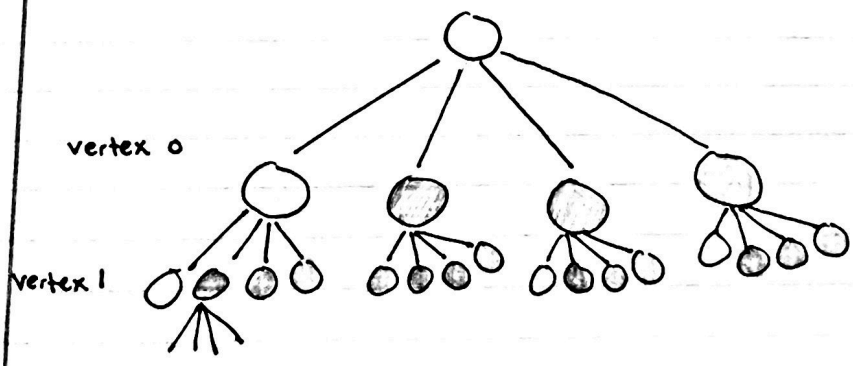
Backtracking Coloring

- We assign colors one by one to different vertices starting from the first vertex (optional)
- Before assigning a color -> we check for safety by considering already assigned colors to the adjacent vertices.
- If we find a color assignment which is feasible -> we mark the color assignment as part of a solution.
- If we do not find a color due to clashes -> we backtrack !!!

Ex.



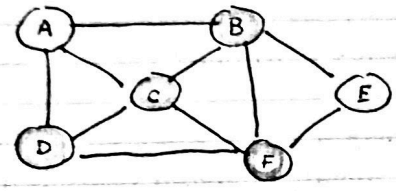
we start at a random vertex.



12.8

CS331 | Ch12.8 - Coloring problem intro

Ex Continued



Assign Different Colors adjacent

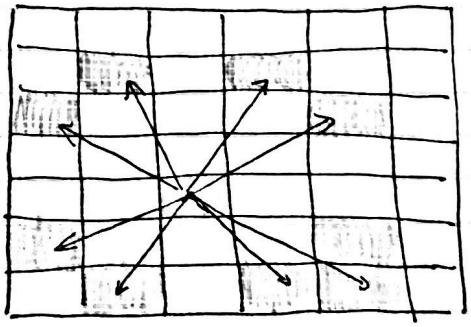
12.10

CS 3311 Ch12.10 – Knights Tour Problem Intro

Knights tour problem

- A sequence of moves of a knight on a chessboard such that the knight visits every square exactly once.
- Closed tour: when the knight end point is the same as the starting point.
- The knights tour problem is an instance of the more general Hamiltonian-path problem.
- Closed knight tour ~ Hamiltonian-cycle problem !!!
- Solutions: Bruteforce approach + backtracking

Ex. Chess board knight



Schwenk theorem

- For an m x n chessboard the closed knight tour problem is always feasible unless:
  - m and n are both odds
  - m = 1, 2, or 4
  - m = 3 and n = 4, 6, or 8

12.10

CS331 | ch12.10 - Knights tour intro

Knight Backtracking

- Start with an empty solution matrix / 2D array
- When adding a new item → we check whether adding the current item violates the problem or not.
- Yes: we backtrack
- No: we add the item to the solution set and go to the next item.
- If we have considered all the items we are ready to go!!!

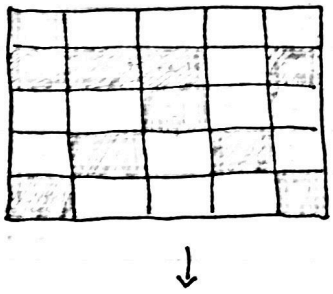
12 12

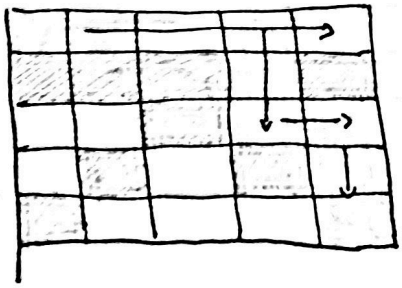
CS331 | Ch12.12 - Maze Problem Intro

Maze Solver

- It is an important problem in robotetics: how to navigate a given robot
- for example: vacum cleaner, roomba
- There may be several obstacles
- So again -> there are constraints // obstacles
- It is like a depth-first search

Ex.



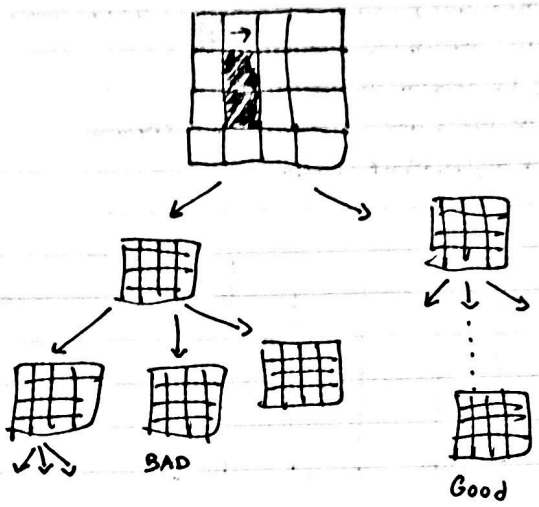


← Deadend: have to backtrack!!!

12.12

CS331 | Ch12.12 - Maze problem intro

Ex. Decision Tree



12.14

CS321 | Ch 12.14 - Soduku Intro

Soduku

A single box

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Rules

- The aim of Soduku is to fill a 9x9 chessboard-like grid with digits
- We have some rules:
- Each column + each row, and each of the nine 3x3 sub-grids that compose the grid (boxes) contains all of the digits from 1 to 9
- Initially we have -> a partially completed grid, which for a well posed puzzle has a unique solution.
- The same integer may can not appear twice in the same row + column or in any of the nine 3x3 subregions / boxes of the 9x9 grid.

12.14

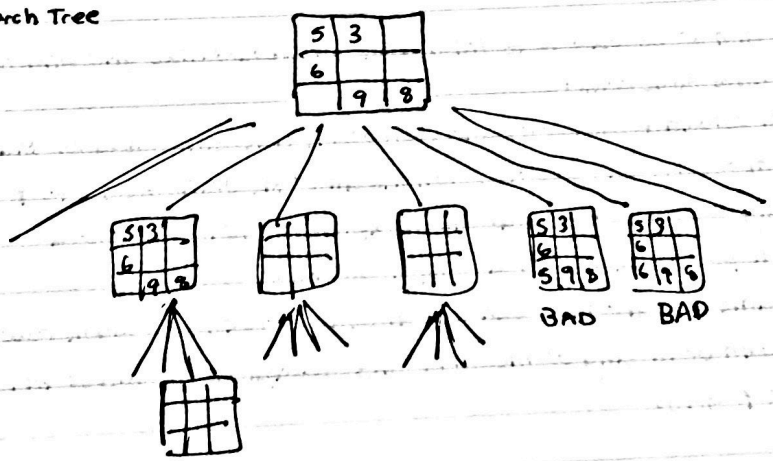
CS 381 | Ch 12.14 - Soduku Intro

Soduku

- The problem itself is NP-complete
- Running time complexity : $O(m^n)$
- m : number of possibilities for a single cell (9)
- n : number of blank fields at the beginning
- Backtracking

- Iterates all the possibilities for the given Soduku.
- If the solutions assigned do not lead to the solution of Soduku, the algorithm discards the solutions and rollbacks to the original solutions and retries the origin.

Search Tree



CS 331 | Ch 12.16 - NP-Complete problems

Backtracking NP

- N-queens problem → exponential running time with backtracking
- Coloring problem → exponential running time
- Sudoku problem → exponential running time

- Usually NP-Complete and NP-hard problems are very slow to solve.
- Solution: we do not want to get the exact solution, just an approximation will be fine!!!

12.16

CS331 Ch12.16 - NP Complete problems

- Meta-heuristics

▷ Usually for NP-hard problems we are looking for an approximate solution instead of the exact one.

▷ Methods: ant-colony optimization, genetic algorithms, simulated annealing.

▷ Not always find the optimal solution (global minimum) But the g-algorithm will be fast.

▷ This is why Artificial Intelligence is important!!!

CS331 Ch13.1 - Dynamic Programming intro

• Dynamic Programming

▷ Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.

▷ It is applicable to problems exhibiting the properties of overlapping subproblems.

▷ The method takes far less time than other methods that don't take advantage of the subproblem overlap.

▷ We need to solve different parts of the problem (subproblems) + combine the solutions of the subproblems to reach an overall solution

▷ We solve each subproblem only once → we reduce the number of computations

▷ Subproblems can be stored („memoization") !!!

CS 331 | Ch13.1 - Dynamic Programming Intro

- Dynamic programming vs. "divide and conquer" method.

▷ Several problems can be solved by combining solutions to non-overlapping sub-problems.

▷ This strategy is called "divide and conquer" method

▷ This is why merge sort / quick sort are not classified as dynamic programming problems.

▷ Overlapping subproblems -> dynamic programming.

▷ Non-overlapping subproblems -> divide and conquer method.

CS331 | Ch13.2 - Fibonacci numbers intro

Fibonacci Sequence: 0 1 1 2 3 5 8 13 21 34 .....

Fibonacci numbers are defined by the recurrance relation

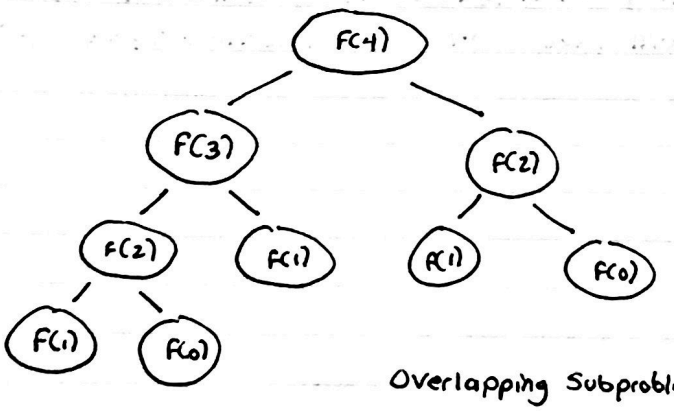
$$F(N) = F(N-1) + F(N-2)$$

$$F(0) = 0 \qquad F(1) = 1$$

with generator functions we can get a close form: „Binet Formula"

What is the problem with the recursive formula? We calculate same problems over and over again

$$F(n) = F(n-1) + F(n-2)$$



Overlapping Subproblems !!!

CS331 | Ch 13.2 - Fibonacci Numbers Intro
Solution Fibonacci numbers

▷ Solution: Use dynamic programming and memoization in order to avoid recalculating a subproblem over and over again

▷ We should use an associative array abstract data type (hash table) to store the solution for the subproblems // O(1) time complexity.

▷ On every F() method call → we insert the calculated value if necessary.

▷ Why is it good? Instead of the exponential time complexity we will have O(N) time complexity + requires O(N) space.

13.4

CS3311 ch13.4 - Knapsack problem intro

Knapsack Problem

- Its is a problem in combinatorial optimization.
- Given a set of items, each with a mass W and a value V, determine the number of each item to include in a collection so that the total weight M is less than or equal to a given limit and the total value is as large as possible.
- The problem often arises in resourse allocation where there are financial constraints.

Applications

- Finding the least wasteful way to cut raw materials.
- Selection of investments and portfolios.
- Selection of assets for asset-backed securization.
- Construction and Scoring of tests in which the test-takers have a choice as to which questions they answer.

2 Important Knapsack problem types.

Divisible problem

- If we can take fractions of the given problems items, then the greedy approach can be used.
- Sort the items according to their values, it can be done in O(N * log(N)) time complexity.
- Start with the item that is the most valueable and take as much as possible.
- Than try with the next item from our sorted list.
- This is linear search has O(N) time complexity.
- Overall complexity : O(N * log(N)) + O(N) = O(N * log(N)) !!!

- So we can have a divisible Knapsack problem qurte fast.

13.4

CS331 | Ch13.4 - Knapsack problem intro

0-1 Knapsack Problem

- In this case we are not able to take fractions : we have to decide whether to take an item or not.

- Greedy algorithm will not provide the optimal result !!!
- Another approach would be to sort by cost per unit weight and include from highest on down until knapsack is full .... not good solution too
- Dynammic programming is the right way !!!

Dynamic Programming

- Solves larger problems by relating it to overlapping subproblems and then solves the subproblems

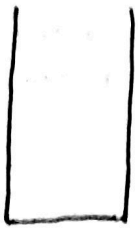
- It works through the exponential set of solutions, but does not examine them all explicitly.

- Stores intermediate results so that they are not recomputed
  " memoization"

- Solution to original problem is easily computed from the solutions to the subproblems.

CS331 | ch13.4 - Knapsack problem intro

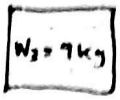
Ex.



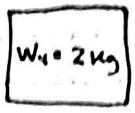
W = 10 kg

x_1 $V_1 = \$10$

x_2 $V_2 = \$13$



x_3 $V_3 = \$19$



x_4 $V_4 = \$4$

We are not able to carry more than 10 kg

Formula

x_i = items we have // we have N items

V_i value of the i-th item

W_i weight of the i-th item

W maximum capacity of knapsack

$$\text{maximize} \sum_{i=1}^{N} V_i \times x_i \quad \text{subject to} \quad \sum_{i=1}^{N} W_i \times x_i \leq W$$

x_i 0 if we do not take the ith item, 1 if we take it.

13.4

CS331 | Ch13.4 - Knapsack problem intro

Knapsack with dynamic programming $O(n * W)$

- We have to define subproblems: we have N items so we have to make N decisions whether to take the item with given index or not.
- Subproblems: the solution considering every possible combination of remaining items and remaining weight.
- S[i][w] the solution to the subproblem corresponding to the first i items and available weight W
- Or in other words....
- S[i][w] = the maximum cost of items that fit inside a knapsack of size (weight) W, choosing from the first i items!!
- We have to decide whether to take the item or not

$$S[i][W] = \text{Math.max}(S[i-1][W] \text{ ; Vi + } S[i-1][W-W_i])$$

do not take i-th item — we take i-th item

But!! We are only considering $S[i-1][W-W_i]$ if it can fit

$W \geq W_i$

If there is not common for it -> the answer is just

$S[i-1][W]$!!!

CS331 | Ch13 6 - Coin Change Problem intro

- Coin Change Problem
  - ▷ Given a set of coins V[] For example {1,2,3}
  - ▷ Given an M amount → the total
  - ▷ How many ways the coins V[] can be combined in order to get the total M?
  - ▷ The order of the coins does not matter?
  - ▷ This is the coin change problem

Ex. Coins V[] → {1,2,3}
Total amount M → 4

The order of coins does not matter!!!
For example {1,3} = {3,1}

Solution to the coin change problem:
{1,1,1,1} {1,1,2} {1,3} {2,2}

We can solve this problem 2 ways with recursion or Dynamic Programming

Recursion

- ▷ The naive approach is to use a simple recursive method / function
- ▷ For every single coin we have two options: include it in our solution or excluded.
- ▷ Problems: time complexity + overlapping subproblems
- ▷ Exponential time complexity: $O(2^N)$ where N is the number of coins.
- ▷ For every coin we have 2 options whether to take it or not.

13.6

05331 | Ch13.6 - Coin Change Problem Intro

Dynamic Programming approach

We have to create a solution matrix:

dpTable [numOfCoins + 1] [totalAmount + 1]

rows columns

We have to define the base cases:

- IF the totalAmount is 0 → there is 1 way to make the change

Because we do not include any coin!!!

- IF numOfCoins is 0 → there is 0 way to change the amount In this case there is no solution!!!

Complexity: O(V*M)

For every coin: make a decision whether to include that coin and excluding that coin

1.) include the coin: reduce the amount by coin value and use the subproblem solution

// totalAmount - V[i]

2.) exclude the coin: solution for the same amount without considering that coin.

13.6

CS331 | ch13.6 - coin change problem

Equations:

If the coin value is smaller than the amount:
it means we can consider that coin !!!

Base cases

$$dpTable[i][j] = \begin{cases} 0 \text{ if } i = 0 \\ 1 \text{ if } j = 0 \\ dpTable[i-1][j] + dpTable[i][j-v[i-1]] \text{ if } v[i] \leq j \\ dpTable[i-1][j] \text{ if } v[i] > j \end{cases}$$

How many ways the first i coins can be combined in order

If the coin value is greater than the amount: it means we can not consider that coin !!!

13.8

CS331 | Ch13-8 - Rod cutting problem intro

Rod Cutting Problem

- Given a rod with certain length L
- Given the prices of different lengths
- How to cut the rod in order to maximize the profit?
- This is the rod cutting problem

Rod length -> L = 5m

Prices for different length:

- 1m -> $2
- 2m -> $5
- 3m -> $7
- 4m -> $3

Solution to the rod cutting problem:

{2,3} so a cut the road to get a 2m piece and a 3m piece

OR

{2,2,1} 2 2m pieces and a single 1m piece, it is going to be the same $12 profit

Total valve for both Solution: $12

Recursion

- The naive approach is to use a simple recursive method / function
- N-1 cuts can be made in the road rod of length N
- There are 2^{N-1} ways to cut the rod
- Problems: time complexity + overlapping subproblems
- Exponential time complexity: $O(2^N)$ where N is the length of the rod in units.

- (For every length we have 2 options whether to cut or not)

CS331 | Ch13.8 - Rod Cutting Problem Intro

Dynamic Programming

We have to create a solution matrix

dpTable[numOfLengths + 1] [originalLength + 1]

rows columns

We have to define the base cases:

- if originalLength 0 -> 0 is the profit
- if we do not consider any length -> 0 is the profit

Complexity : O(numOfLengths * originalLength)

These are the

$$dpTable[i][j] = \begin{cases} 0 \text{ if } j=0 \text{ and } 0 \text{ if } i=0 \\ \max\{ dpTable[i-1][j];\ Prices[i] + dpTable[i][j-1] \text{ if } i \le j \\ \quad dpTable[i-1][j] \text{ if } i > j \end{cases}$$

The total value when total length is j and we have the first i pieces.

CS331 | Ch13.11 - Subset Sum problem

Subset Sum Problem

- One of the most important problems in Complexity theory
- The problem: given an S set of integers, is there a non-empty Subset whose S sum is zero or a given integer?
- For example: given the set $\{5, 2, 1, 3\}$ and $S = 9$ the answer is Yes because the subset $\{5, 3, 1\}$ sums to 9
- The problem is NP-Complete → we have efficient algorithms when the problem is small !!!
- Special case of knapsack problem

Solutions

1.) Naive approach „brute force Search"
   - Generate all the subsets of the given set of integers
   - N is the number of integers in the set S
   - Check whether the sum of all subsets is equal to s or not
   - Time Complexity: exponential // $O(N * 2^N)$

2.) Dynamic Programming: we want to avoid the same problems over and over again ... we create a dynamic programming table and memoize.

13.11

CS 331 | Ch 13.11 – Subset Sum problem

Of course if j can be constructed with the i-1 integers -> there must be a subset with sum i as well (include)

Base cases

dpTable[i][j] =

true if j = 0 and false if i = 0

dpTable[i][j] = dpTable[i-1][j] if dpTable[i-1][j] is true

dpTable[i][j] = dpTable[i-1][j-s[i-1]] else

There is a non-empty subset of the first i integers that sums to j

if j - actual integer can be constructed with the i-1 integers (EXCLUDE)

Ex. {1,2,3} -> 5

{1,2,3,4} -> s = 5 we can still solve this problem !!!

even if we add more integers

we can still get 5 based

on the subset problem

CS331 | Ch14.1 - Bin Packing Problem

Bin Packing

- Objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used.

- In the main: how to fit several things into containers in an efficient way.
- It is an NP-complete problem
- When the number of bins is restricted to 1 and each item is characterized by both a volume and a value, the problem of maximizing the value of items that can fit in the bin is known as the knapsack problem.

Solutions

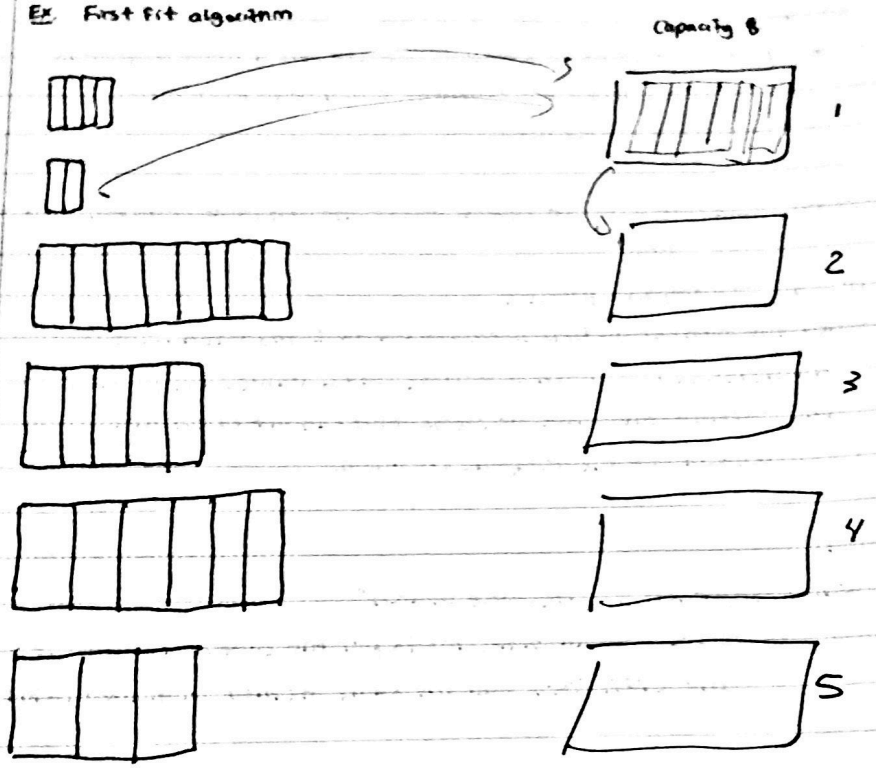
1.) Naive approach „brute-force search"
   - Iterate over all bins, try to put the current item in the bin and (if it fits) call the same method with the next item.

2.) First-fit algorithm
   - Iterate over all the items we want to put into bins → if we are not able to put it into a given bin we try to put it into the next one.
   - Yeilds non-optimal solutions in the main.

3.) First-fit decreasing algorithm
   - Sorting the items in decreasing order may be helpful. After sorting we use first-fit algorithm.
   - Yeilds non-optimal solutions in the main.

CS 331 | Ch14.1 - Bin packing Problem

Ex First fit algorithm



Example in power point for first-fit decreasing

First-fit decreasing algorithm : $O(N \log N)$

Applications:

- we have N groups of people with group sizes $w_1, w_2 \ldots w_n$. We have minibuses with capacity C. What is the optimal number of minibuses with the groups must stay together.
- Virtual machines often have this problem
- Television advertisements: we are given a certain time slot (for example 10 minutes). How do we pack the most commercials into each time slot and maximize our daily profits?

14.3

CS381 | Ch 14.3 - Closest Pair of Points Problem

Closest Pair of Points (Divide and Conquer)

- We are given an array of N points in the 2D plane and the problem is to find the closest pair of points in an array.
- It has several applications: for example air-traffic control
- We may want to monitor planes that come too close together → they might collide.
- Brute-force approach: $O(N^2)$ very slow, we need something faster
- Divide and conquer approach might help to achieve $O(N * \log N)$ time complexity.

Brute Force Search approach

```
public Double bruteForceSearchApproach(List<Point> points) {
        double minDistance = Double.MAX_VALUE;
        for(int i=0; i < points.size(); ++i) {
            for(int j=0; j < points.size(); ++j) {
                if(distance(points.get(i), points.get(j)) < minDistance) {
                    minDistance = distance(points.get(i), points.get(j));
                }
            }
        }
        return minDistance;
}
```

14.3

CS331 | Ch14.3 – Closest pair of points problem

Algorithm

1.) Sort all points according to the X-coordinates
2.) Divide all points into two subsets with the help of a middleIndex
3.) Find the minimum distance recursively in the two subsets // d_1 and d_2
4.) Calculate the minimum of these smallest distances //
$d = \min(d_1, d_2)$
5.) Check the neighbourhood of the middle line (strip) there may be points that are closer to each other than $\min(d_1, d_2)$.
So we get a stripMinimum
6.) Find the smallest distance in the strip
7.) Finally return $\min(d, \text{StripMinimum})$