Algorithms and Datastructures

Kristoffer Dalby

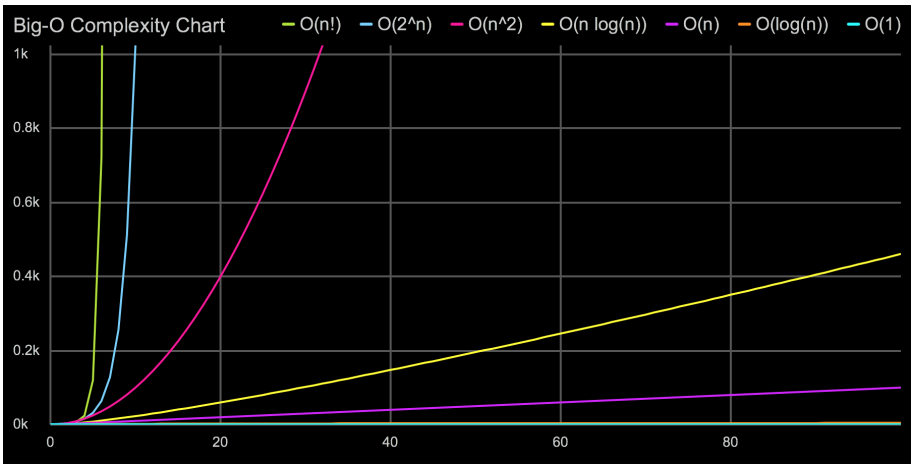
1 Introduction

This is me trying to learn algdat. Credz to hakloev @ github for some of the content in this collection, pretty much or at least the stuff in norwegian and the pseudo code.

2 Notation for asymptotic growth

Name	Notation	Performance
small-oh	$o(n)$	$< n$
big-oh	$O(n)$	$\leq n$
theta	$\Theta(n)$	$= n$
small omega	$\omega(n)$	$> n$
big omega	$\Omega(n)$	$\geq n$

3 Big-O Complexity Chart



3.1 Complexity classes

1. Constant: 1
2. Polylogarithm: $(log_b n)^k$
3. Polynominal: n^k
4. Exponential: $2n$
5. Factorial: $n!$
6. Worse: n^n

3.2 Pseudopolynominal

Gitt en algoritme som tar tallet n som input og har kjøretid $O(n)$ – hvilken kompleksitetsklasse er dette? n betegner her ikke størrelsen på input, men er selv en del av inputen. Det vil si at størrelsen til n = antall bits som kreves $= lg(n)$.

$$n = 2^{lg(n)} = 2w \Rightarrow \text{NB: Exponential time complexity!}$$

4 Time Complexity

Algorithm	Best	Average	Worst
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$
Quicksort	$O(nlog(n))$	$O(nlog(n))$	$O(n^2)$
Counting sort	$O(n+k)$	-	-
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$
Topological sort (DFS)	$O(V+E)$	-	-
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$
BFS	$O(\|E\|+\|V\|)$	-	-
DFS	$O(\|E\|+\|V\|)$	-	-
Bellman-Ford	$O(\|V\|\|E\|)$	-	-
Dijkstra Heap	$O((\|V\|+\|E\|)log\|V\|)$	-	-
Dijkstra Array	$O(\|V\|^2)$	-	-
DAG-shortest (TP)	$O(V+E)$	-	-
Floyd-Warshall	$O(V^3)$ -	-	-
Binarysearch	$O(log(n))$	-	$O(log(n))$
Ford-Fulkerson	$O(Ef)$	$f = maxflowofG$	
Edmons-Karp	$O(VE^2)$	-	-
Huffman	$O(nlog(n))$	-	-
Kruskals	$O(Elog(v))$	-	-
Prims	$O(E+Vlog(v))$	-	-

5 Master method

Masterteoremet er en form for "kokebok"-metode for å løse rekurenser. Man kan som regel løse de fleste rekurrenser av typen:

$T(n) = aT(n/b) + f(n)$ hvor $a \geq 1$ og $b > 1$. $f(n)$ må også være asymptotisk positiv.

Case 1: $f(n) = O(n^{log_b a - \varepsilon})$, for en $\varepsilon > 0$
$\Rightarrow$T(n) = $\Theta(n^{log_b a})$
Case 2: $f(n) = \Theta(n^{log_b a})$
$\Rightarrow$T(n) = $\Theta(n^{log_b a} log(n))$
Case 3: $f(n) = \Omega(n^{log_b a + \varepsilon})$, for en $\varepsilon > 0$ og $af(n/b) \leq cf(n)$, hvor $c < 1$
$\Rightarrow$T(n) = $\Theta(f(n))$

6 Datastructures

6.1 Array

6.1.1 Time complexity

	Linked list	Array	Dynamic Array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
ins/del at beginning	$\Theta(1)$	-	$\Theta(n)$
ins/del at end	$\Theta(n)$	-	$\Theta(1)$
ins/del at middle	Search + $\Theta(1)$	-	$\Theta(n)$
wasted space	$\Theta(n)$	0	$\Theta(n)$

6.2 Heap

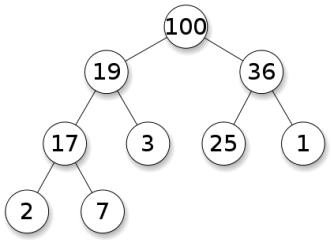
A heap is a specialised tree-based data structure that satisfy the heap property. There is no rules for how the siblings of a node is ordered in a heap. A heap is often implemented as an Array. It is important that the key values from the heaps starts on the 2nd place in the array (index 1 in a 0-index array) to make it easier to calculate the siblings and parents of a node.

6.2.1 Heap property

If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. A heap has to have a full set of siblings for every node except the last one.

Min-heap:
The node with the smallest key value is always on the top of the heap. Every node beneath this node has a bigger key value and the nodes beneath the next node has even bigger. There can not be a node beneath another node with a smaller key value.

Max-heap:
This heap is the same as min-heap just reversed, with the biggest on top.



Figur 1: A max-heep with integers between 1 and 100

6.2.2 Time complexity

Operation	Time complexity
find-min/max	$\Theta(1)$
delete-min/max	$\Theta(\log n)$
insert	$\Theta(\log n)$
decrease-key	$\Theta(\log n)$
merge	$\Theta(n)$

Note: that min/max depends on min/max heap

6.3 Stack

Stack is a last in first out abstract data type. It is often implemented by using an array or a linked list. A stack uses operations as peek, push and pop.

6.4 Queue

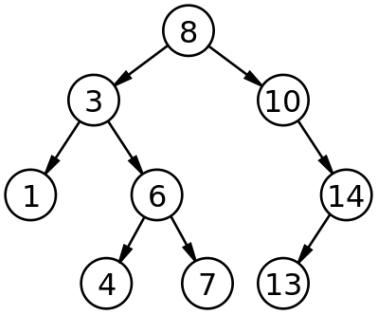
Queue is a first in first out abstract data type. You can implement a queue by using an array (preferably a dynamic one) or a linked list. A queue uses operations as enqueue to put something in the queue, and dequeue to take it out.

6.5 Binary Search tree

Binary search tree is an ordered binary tree.

6.5.1 Properties

The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
The left and right subtree each must also be a binary search tree.
There must be no duplicate nodes.



Figur 2: A binary search tree of size 9 and depth 3

6.5.2 Time complexity

Operation	Average	Worst
Space	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

6.6 Hash table

Hash table or hash map is a data structure used to implement an associative array, a structure that can map keys to values. It works by taking a key and create an index by running it through a hashing algorithm. The index then can be used to find the correct value. The place where the value is stored is often called buckets or slots.

6.6.1 Collisions

The hash structure is very ideal for storing data very efficiently, but has some limitations.
The biggest problem is collisions, there is a possibility that the same hash is generated from two different keys, and therefor two keys can point at the same value. This is why it is important to have a good hashing algorithm.

6.6.2 Perfect hashing

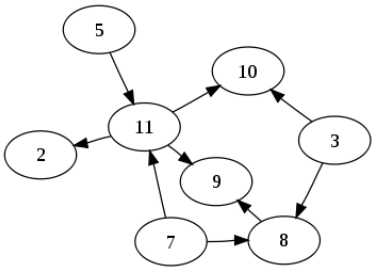
Perfect hashing is not in the curriculum. The concept is built on that before building the hash table you know all the keys, and based on that knowledge you choose the perfect hashing algorithm. This will allowing you to get receive time complexity at linear time.

6.6.3 Universal hashing

Universal hashing is built on choosing a hashing algorithm at random for every key. This will in theory give you a very small collision rate, and in practice none.

6.6.4 Time complexity

Operation	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$



Figur 3: An example of a directed acyclic graph

6.7 Directed acyclic graph

6.8 Adjacency matrix

6.9 Representation of graphs

7 Searching algorithms

7.1 Breadth-first search

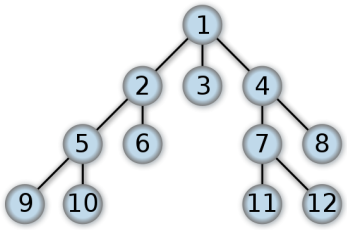
BFS is a strategic algorithm for searching a graph. It has only two operations which is *visit and inspect a node* and *gain access to visit the nodes that is the neighbor of the currently visited node* It can be used to find shortest path. BFS uses a queue to keep track of what nodes to visit.

```
function BFS(G, v)                                   ▷ v er startnode
    lag en kø Q
    legg v inn i Q
    while Q.notEmpty() do
        v = Q.dequeue()
        for each edge e adjacent to v do
            if e not marked then
                mark w
                Q.enqueue(e)
            end if
        end for
    end while
end function
```

7.2 Depth-first search

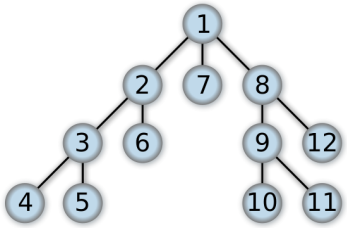
DFS is like BFS an algorithm to search a graph. Instead of a queue, DFS uses a stack and it can be implemented recursively. A directed graph can contain forward edge, backward edge, cross edge and tree edge. A undirected graph can contain backward and cross edge

```
function DFS(G, v)                                   ▷ v er startnode
    initaliser en tom stack, S
    for each vertex u in G do
```



Figur 4: A graph with example of the bfs visit order

set $visited[u] \rightarrow false$
end for
$S.push(v)$
while $S.notEmpty()$ **do**
$u = S.pop()$
for all w adjacent to u **do**
if not $visited[w]$ **then**
$visited[w] \rightarrow true$
$S.push(w)$
end if
end for
end while
end function
maybe talk about all the edgetypes?



Figur 5: A graph with example of the dfs visit order

7.3 Bellman-Ford algorithm

Bellman-Ford is an single-source shortest path algorithm. It is slower than Dijkstra, because it can detect negative cycles by running an additional check. If it detect a negative cycle it returns a false or an error. It is built upon the principle of relaxation, in which an approximation to the correct distance is gradually replaced by a more accurate values until eventually reaching the optimal solution. Bellman-ford is not a greedy algorithm like Dijkstra, instead it checks every outgoing edge form a node.4

function BELLMANN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)

```
    for i = 1 to |G.V| − 1 do
        for each edge (u, v) ∈ G.E do
            RELAX(u, v, w)
        end for
    end for
    for each edge (u, v) ∈ G.E do
        if v.d > u.d + w(u, v) then
            return false
        end if
    end for
    return true
end function
```

7.4 DAG-shortest path

The idea of DAG-shortest path is to use Topological sort on the graph to find the topological sorting of the graph represented in linear ordering. Once we have a linear presentation we can processes every vertex one by one and update the distance using the value of the current vertex.

```
function DAG-SHORTEST-PATH(G, w, s)
    TOPOLOGICAL-SORT(G)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex u, taken in topologically sorted order do
        for each vertex v ∈ G.Adj[u] do
            RELAX(u, v, w)
        end for
    end for
end function
```

7.5 Dijkstra's algorithm

Dijkstra's shortest path algorithm is faster than Bellman-Ford because it do expect that the graph has no negative cycle. Because of this, it will not work on graphs with negative cycle and it will not finish.
Dijkstra can be implemented with an ordinary array as priority queue., but will den not achieve that great time complexity. If it is implemented with a min-heap or a Fibonacci-heap then you can receive great time complexity.

```
function DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u] do
            RELAX(u, v, w)
        end for
    end while
```

end function

7.6 Floyd-Warshall algorithm

Floyd-Warshall is an algorithm for finding all to all shortest path. It is an example of dynamic programming. By default it only finds the shortest way by value, not by weight. It is possible to write the algorithm with path reconstruction.

function FLOYD-WARSHALL(W)
    $D^{(0)} = W$
    for $k = 1$ to n **do**
        let $D^{(k)}$ be a n x n matrix
        for $i = 1$ to n **do**
            for $j = 1$ to n **do**
                $D_{ij}^{(k)} = min(\mathrm{D}_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$
            end for
        end for
    end for
    return $D^{(n)}$
end function

7.7 Binary search

In computer science, a binary search or half-interval search algorithm finds the position of a specified input value (the search key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned.

8 Flow networks

8.1 Ford-Fulkerson method

Ford-Fulkerson is an algorithm/method which computes the maximum flow in a flow network.
The idea behind the algorithm is as follows: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. The algorithm uses DFS to find the augmenting path

8.1.1 Augmented path

An augmenting path is simply a path through the graph using only edges with positive capacity from the source to the sink.

8.2 Edmonds-Karp

It is exactly the same as Ford-Fulkerson, but instead of using DFS it uses BFS.

9 Sorting algorithms

9.1 Insertion Sort

Insertion sort is a simple sorting algorithm which sorts items one at a time. It is not efficient, but has advantages as simple implementation, efficient for small data sets and stable.

for $j = 0$ to $A.length$ **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$ **do**
        $A[i + 1] = A[i]$
        $i = i - 1$
    end while
    $A[i] + 1 = key$
end for

9.2 Heapsort

Heapsort is a comparison-based sorting algorithm to create a sorted array. The algorithm works in two parts. The first part is to build a heap out of the data.
The second part is taking out the biggest element of the heap and putting it into an array as long as there is still items in the heap. The heap is reconstructed every time. The order of the sorted array as a result is based on if the heap was a max heap or a min heap.

BUILDMAXHEAP(A)
for $i = A.length$ downto 2 **do**
    exchange $A[1]$ with $A[i]$
    $A.heapSize = A.heapSize - 1$
    MAX-HEAPIFY($A, 1$)
end for

9.3 Quicksort

Quicksort is one of the most used sorting algorithm and is on the average case very fast. It is a divide and conquer sorting algorithm which working in the following way:

Step 1: Pick an element, called a pivot element from the list to be sorted.
Step 2: Reorder the list so that all elements with value less than the pivot come before the pivot and all bigger come after. After this, the pivot is on its final position. This is called the partition operation.
Step 3: Recursively apply the above steps to the sublist now created on the sides of the pivot element.

Quicksort is often depended on choosing a good pivot element to have good time complexity. This is often solved by choosing one at random.

```
function QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A.q + 1, r)
    end if
end function

function PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] ≤ x then
            i = i + 1
            exchange A[i] with A[j]
        end if
    end for
    exchange A[i + 1] with A[j]
    return i + 1
end function
```

9.4 Counting sort

Counting sort is a sorting algorithm that can sort only whole integers and is therefor a integer sorting algorithm. It counts the amount of time a integer or key is in an array and then fill in the amount of times the integer appears in the index of the integers value in the counting array. After the counting array is done, it will adjust the counting array so that the value of an index represent the index of the final index in the final sorted array. Counting sort is often used as a subroutine in algorithms such as Radix sort.

```
function COUNTINGSORT(A, B, k)
    let C[0 . . . k] be a new array
    for i = 0 to k do
        C[i] = 0
    end for
    for j = 1 to A.length do
        C[A[j]] = C[A[j]] + 1
    end for
    for i = 0 to k do
        C[i] = C[i] + C[i − 1]
    end for
    for j = A.length downto 1 do
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
    end for
end function
```

9.5 Radix sort

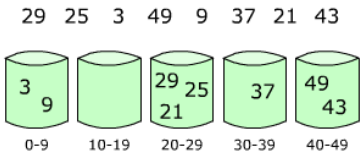
Radix sort is a sorting algorithm that sorts data based on a part-value at a position in the value. It sorts first by the part-value at place -1 and from there it goes to -2 and so on. It uses counting sort for the sorting operation.

function RADIXSORT(A, d)
  for $i = 1$ to d **do**
    Bruk *Stable Sort* til å sortere array A
  end for
end function

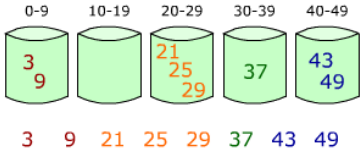
9.6 Bucket sort

Bucket sort that works by partitioning an array into a number of buckets. After the initial sorting, the buckets are sorted individually, by applying bucket sort again or by using another sorting algorithm.

function BUCKETSORT(A)
  $n = A.length$
  la $B[0 \ldots n-1]$ være et nytt array
  for $i = 0$ to $n - 1$ **do**
    Gjør $B[i]$ til en tom liste
  end for
  for $i = 1$ to n **do**
    Sett inn $A[i]$ i lista $B[\lfloor nA[i] \rfloor]$
  end for
  for $i = 0$ to $n - 1$ **do**
    Sorter lista $B[i]$ med INSERTIONSORT(B[i])
  end for
  Konkatiner listene $B[0], B[1], \ldots, B[n-1]$ i rekkefølge
end function



Figur 6: Some unsorted buckets



Figur 7: Some sorted buckets

9.7 Topological sort

Topological sort is a linear ordering of a directed acyclic graphs vertices. One way to topological sort is to use an algorithm based on DFS. Every DAG has at least one topological sort.

function TOPOLOGICALSORT(G)
    DFS(G) to compute finish times $v.f$ for each vertex v.
    as each vertex is finished, insert it onto the end of the list
    return the list of verticies
end function

9.8 Bubble sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

for $i = 0$ to $A.length - 1$ **do**
    for $j = A.length$ downto $i + 1$ **do**
        if $A[j] < A[j - 1]$ **then**
            exchange $A[j]$ with $A[j - 1]$
        end if
    end for
end for

9.9 Merge sort

Merge sort is a divide and conquer sorting algorithm that splits a list in two and then calls itself recursively. When every list is a single element list, it will start the merge part where it merges the list back together in sorted order.

function MERGE-SORT(A, p, r)
    if $p < r$ **then**
        $q = \lfloor (p + r)/2 \rfloor$
        MERGE-SORT(A, p, q)
        MERGE-SORT($A, q + 1, r$)
        MERGE(A, p, q, r)
    end if
end function

function MERGE(A, p, q, r)
    $n_1 = q - pr + 1$
    $n_2 = r - q$
    let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1$ be new arrays
    for $i = 1$ to n_1 **do**
        $L[i] = A[p + i - 1]$
    end for
    for $j = 1$ to n_2 **do**
        $R[j] = A[q + j]$
    end for
    $L[n_1] + 1] = \infty$

```
    r[n_2] + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            if A[k] = R[j] then
                j = j + 1
            end if
        end if
    end for
end function
```

9.10 Huffman

10 Dynamic Programming

Overlapping part problem and Optimal substructure. If a locally optimal problem is a global solution, then it is a optimally global problem, then you can use greedy algorithm.
DP = memoising, recursion and guessing.
Dynamic programming is pretty much shortest path with DAG.

10.1 Five easy steps to DP

1. Define subproblems
2. Guess (part of solution)
3. Relate subproblem solutions
4. Recurse and memoize
5. or
6. Build DP table bottom-up
7. Solve original problem

10.2 Memoising

For memoising to work, the problem need to be acyclic, else it will run infinite.

10.3 Rod cutting

10.4 Matrix-chain multiplication

11 Greedy algorithm

11.1 Huffman

12 Minimal spanning trees

12.1 Growing a minimal spanning tree

12.2 Kruskals algorithm

Kruskals algoritme sorterer alle kanter etter kostnaden og velger den billigste tilgjengelige kanten , og legger til denne med noder i treet. Dette kun hvis den ikke allerede er brukt og vil danne en sykel. Fortsetter til det ikke finnes kanter som kan legges til i treet.

function KRUSKAL(G, w)
  $A = \emptyset$
  for each vertex $v \in G.V$ **do**
    MAKE-SET(v)
  end for
  sort edges of $G.E$ into nondecreasing order by weight w
  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight **do**
    if FIND-SET(u) $\neq$ FIND-SET(v) **then**
      $A = A \cup \{(u, v)\}$
      UNION(u,v)
    end if
  end for
  return A

12.3 Prims algorithm

Prims algoritme velger en tilfeldig node, og legger alle kantene inn i en prioritetskø etter vekt. Velger den billigste kanten og legger den til i treet. Det vil si at den legger til den billigste kanten som er mulig å legge til fra treet den bygger.

function PRIM(G, w, r)
  for each $u \in G.V$ **do**
    $u.key = \infty$
    $u.\pi = NIL$
  end for
  $r.key = 0$
  $Q = G.V$
  while $Q \neq \emptyset$ **do**
    u = EXTRACT-MIN(Q)

```
        for each v ∈ G.Adj[u] do
            if v ∈ Q and w(u, v) < v.key then
                v.π = u
                v.key = w(u, v)
            end if
        end for
    end while
end function
```

13 NP

13.1 Reduction

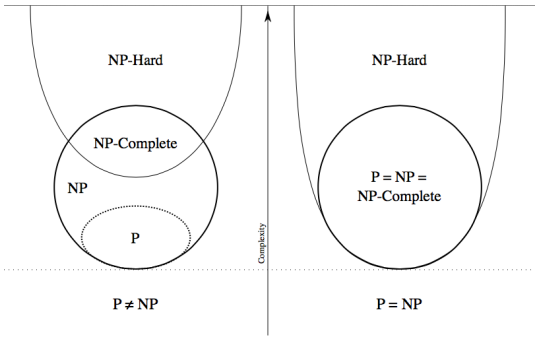
If you have A and B, and A is a known hard problem, and you want to show that B is a hard problem, then you can reduce A to B to show that B is a hard problem. If a problem is in both NP and NP-Hard then it is NPC (NP Complete)

14 NP-Komplette problemer

NP ⇒ Nondeterministic Polynomial
NPC ⇒ NP-Complete
P ⇒ Polynomial

Sammenhengen mellom alle NP-problemer:
$P \subseteq NP$
$NPC \subseteq NP$

Hvis et problem $A \in NPC$, så vil også $A \in NP$. Dette er fordi $NP = P + NPC$. Hvis noe ikke er i P eller NPC, er det heller ikke i NP, fordi $P \cap NPC = \emptyset$.



Figur 8: Venndiagramet, med med $P \neq NP$ og $P = NP$

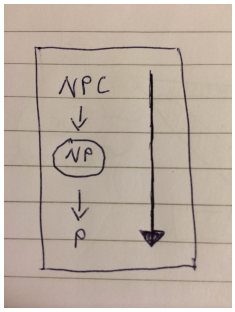
Du vet at problem A er i NP og problem B er i NPC. Du vil vise at A også er i NPC. Da reduserer du fra B til A.

Du står overfor de tre problemene A, B og C. Alle tre befinner seg i mengden NP. Du vet at A er i mengden P og at B er i mengden NPC. Anta at du skal bruke polynomiske reduksjoner mellom disse problemene til å vise ...:

1. ... at C er i P må $C \leq A$ (C reduseres til A)
2. ... at C er i NPC må $B \leq C$ (B reduseres til C)
3. ... hvis B kan reduseres til A er $P = NP$ (NB: ikke løst enda)
4. Alle disse reduksjonene skjer i polynomisk tid

To ikke helt legitime, men greie huskeregler:

reduser fra litt til mer // fra litt ... $\leq$ ... til mer
reduser nedover: NPC $\rightarrow$ (NP) $\rightarrow$ P (forsiktig med denne)



Figur 9: En tvilsom huskeregel, NB!

Liste/strukturen til NPC-problemer redusert fra CIRCUT-SAT:

- CIRCUT-SAT
- SAT
- 3-CNF-SAT
  - SUBSET-SUM
  - CLIQUE
    - VERTEX-COVER
    - HAM-CYCLE
    - TSP (Traveling Salesman Problem)