INF221

Sammendrag av det viktige.

Series

Arithmetic series

An arithmetic series is the sum of a sequence in which each term is computed from the previous one by adding a constant.

$$S_n = \frac{1}{2}n(2a + (n-1)d)$$

where a is the starting value, d the constant, and n the number of terms.
If the last value, l, is known, the sum can be expressed by:

$$S_n = \frac{1}{2}n(a + l)$$

Geometric series

A geometric series is the sum of terms that have a constant ratio between successive terms.

$$S_n = \frac{a(1 - r^n)}{1 - r}$$

where a is the starting value, r the ratio, and n the number of terms.

Logarithms

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \frac{\log_d a}{\log_d b}$
5. $b^{\log_d a} = a^{\log_d b}$

Sorting

For any valid input:

1. The algorithm terminates
2. The output contains the same keys as the input
3. The output is correctly sorted

A recursive function has two parts:

1. Base case(s) so simple that they can be solved directly
2. Recursive cases(s) that apply recursion to solve subproblems and combine its results

Insertion sort

	INSERTION-SORT(A)	Cost	Executions	
1	`for j = 1 to A.length`	c_1	n	
2	`    key = A[j]`	c_2	n	
3	`    i = j - 1`	c_3	n	
4	`    while i > 0 and A[i] > key`	c_4	$\sum_{j=1}^{n} t_j$	worst case
5	`        A[i + 1] = A[i]`	c_5	$\sum_{j=1}^{n} t_j$	
6	`        i = i - 1`	c_6	$\sum_{j=1}^{n} t_j$	
7	`    A[i + 1] = key`	c_7	n	

$$
\begin{aligned}
T(n) &= \sum \texttt{cost} * \texttt{executions} \\
&= \underbrace{(c_1 + c_2 + c_3 + c_7)}_{\text{B, outer}} n + \underbrace{(c_4 + c_5 + c_6)}_{\text{A, inner}} \sum_{j=1}^{n} t_j + C \\
&= A \sum_{j=1}^{n} t_j + Bn + C
\end{aligned}
$$

where A is a coefficient encapsulating the cost of the inner loop, B the outer loop, and C a fixed cost outside the loops or cost that occur only once regardless of input.

Best case

When the data is already sorted. The `while`-loop is never entered. Running time is linear in n.

$$
\begin{aligned}
T(n) &= (c_1 + c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6) \sum_{j=1}^{n} t_j + C \\
&= (c_1 + c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)0 + C \\
&= Bn + C \\
T(n) &= \Omega(n)
\end{aligned}
$$

Worst case

When the data is reversely sorted. The `while`-loop is fully run for every j. Running time is quadratic in n.

$$
\begin{aligned}
T(n) &= (c_1 + c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\sum_{j=1}^{n} t_j + C \\
&= (c_1 + c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\frac{n(n+1)}{2} + C \\
&= A\frac{n^2}{2} + (A + B)n + C \\
T(n) &= O(n^2)
\end{aligned}
$$

Average case

Assume the `while`-loop is halfway run for every j ($t_j = j/2$). Running time is quadratic in n.

$$
\begin{aligned}
T(n) &= (c_1 + c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\sum_{j=1}^{n} \frac{t_j}{2} + C \\
&= (c_1 + c_2 + c_3 + c_7)n + \frac{(c_4 + c_5 + c_6)}{2}\frac{n(n+1)}{2} + C \\
&= \frac{A}{4}n^2 + (\frac{A}{4} + B)n + C \\
T(n) &= \Theta(n^2)
\end{aligned}
$$

Bubble sort

	BUBBLE-SORT(A)	Cost	Executions	
1	`for i = 1 to A.length - 1`	c_1	n	
2	`    for j = 1 to A.length - i`	c_2	$\sum_{j=1}^{n} t_j$	
3	`        if A[j] > A[j + 1]`	c_3	$\sum_{j=1}^{n} t_j$	
4	`            swap A[j] and A[j + 1]`	c_4	$\sum_{j=1}^{n} t_j$	worst case

Best case

In the best case scenario, the `if`-scope is never run. Running time is quadratic in n.

$$
\begin{aligned}
T(n) &= c_1 n + (c_2 + c_3)\sum_{j=1}^{n} t_j + c_4 0 + C \\
&= c_1 n + (c_2 + c_3)\frac{n(n+1)}{2} + C \\
&= \frac{A}{2}n^2 + (\frac{A}{2} + B)n + C \\
T(n) &= \Omega(n^2)
\end{aligned}
$$

Worst case

Assume the `if`-scope is always run. Running time is quadratic in n.

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4) \sum_{j=1}^{n} t_j + C \\ &= c_1 n + (c_2 + c_3 + c_4) \frac{n(n+1)}{2} + C \\ &= \frac{A}{2} n^2 + (\frac{A}{2} + B) n + C \\ T(n) &= O(n^2) \end{aligned}$$

Average case

In the average case scenario, the `if`-scope is assumed to run every j ($t_j = j/2$). Running time is quadratic in n.

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4) \sum_{j=1}^{n} \frac{t_j}{2} + C \\ &= c_1 n + \frac{(c_2 + c_3 + c_4)}{2} \frac{n(n+1)}{2} + C \\ &= \frac{A}{4} n^2 + (\frac{A}{4} + B) n + C \\ T(n) &= \Theta(n^2) \end{aligned}$$

Selection sort

	SELECTION-SORT(A)	Cost	Executions	
1	`for i = 1 to A.length - 1`	c_1	n	
2	`    idx = i`	c_2	n	
3	`    for j = i + 1 to A.length`	c_3	$\sum_{j=1}^{n} t_j$	
4	`        if A[j] < A[idx]`	c_4	$\sum_{j=1}^{n} t_j$	
5	`            idx = j`	c_5	$\sum_{j=1}^{n} t_j$	worst case
6	`    if idx != i`	c_6	n	
7	`        swap A[i] and A[idx]`	c_7	n	

Best case

In the best case scenario, the `if`-scopes are never run. Running time is quadratic in n.

$$\begin{aligned} T(n) &= (c_1 + c_2 + c_6) n + (c_3 + c_4) \sum_{j=1}^{n} t_j + (c_5 + c_7) 0 + C \\ &= (c_1 + c_2 + c_6) n + (c_3 + c_4) \frac{n(n+1)}{2} + C \\ &= \frac{A}{2} n^2 + (\frac{A}{2} + B) n + C \\ T(n) &= \Omega(n^2) \end{aligned}$$

Worst case

Assume the `if`-scopes are always run. Running time is quadratic in n.

$$
\begin{aligned}
T(n) &= (c_1 + c_2 + c_6 + c_7)n + (c_3 + c_4 + c_5)\sum_{j=1}^{n} t_j + C \\
&= (c_1 + c_2 + c_6 + c_7)n + (c_3 + c_4 + c_5)\frac{n(n+1)}{2} + C \\
&= \frac{A}{2}n^2 + (\frac{A}{2} + B)n + C \\
T(n) &= O(n^2)
\end{aligned}
$$

Average case

In the average case scenario, the `if`-scopes are assumed to run every j ($t_j = j/2$). Running time is quadratic in n.

$$
\begin{aligned}
T(n) &= (c_1 + c_2 + c_6 + c_7)n + (c_3 + c_4 + c_5)\sum_{j=1}^{n} \frac{t_j}{2} + C \\
&= (c_1 + c_2 + c_6 + c_7)n + \frac{(c_3 + c_4 + c_5)}{2}\frac{n(n+1)}{2} + C \\
&= \frac{A}{4}n^2 + (\frac{A}{4} + B)n + C \\
T(n) &= \Theta(n^2)
\end{aligned}
$$

Heap sort

```
    HEAPIFY(A, n, i)
 1  max = i
 2  leftchild = 2i + 1
 3  rightchild = 2i + 2
 4  if leftchild ≤ n and A[i] < A[leftchild]
 5      max = leftchild
 6  else
 7      max = i
 8  if rightchild ≤ n and A[max] > A[rightchild]
 9      max = rightchild
10  if max ≠ i
11      swap A[i] and A[max]
12      HEAPIFY(A, n, max)
```

```
    HEAPSORT(A)
1  for i = A.length/2 downto 1
2      HEAPIFY(A, A.length, i)
3  for i = A.length downto 2
4      swap A[1] and A[i]
5      A.heapsize = A.heapsize - 1
6      HEAPIFY(A, i, 0)
```

Best case

When elements are already sorted.
$\Omega(n \log(n))$

Worst case

When the smallest or largest element is always chosen.
$O(n \log(n))$

Average case

$\Theta(n \log(n))$

Tree sort

```
    NODE(value)
1   Node:
2       value
3       left:  Node
4       right:  Node
```

```
    TRAVERSAL(Node, list)
1   if Node
2       TRAVERSAL(Node.left, list)
3       list.append(Node.value)
4       TRAVERSAL(Node.right, list)
```

```
    INSERT(Node, value) → Node
1   if not Node
2       Node = NODE(value)
3   else if value < Node.value
4       Node.left = INSERT(Node.left, value)
5   else
6       Node.right = INSERT(Node.right, value)
7   return Node
```

```
    BINARY-TREE-SORT(array) → list
1   root = null
2   for value in array
3       root = INSERT(root, value)
4   list = []
5   TRAVERSAL(root, list)
6   return list
```

Best case

$\Omega(n \log(n))$

Worst case

Unbalanced: $O(n^2)$
Balanced: $O(n \log(n))$

Average case

$\Theta(n \log(n))$

Merge sort

	MERGE-SORT(A, p, r)	Cost	Executions
1	`if p < r`	c_1	n
2	`    q = floor((p + r) / 2)`	c_2	n
3	`    MERGE-SORT(A, p, q)`	c_3	$n/2$
4	`    MERGE-SORT(A, q + 1, r)`	c_4	$n/2$
5	`    MERGE(A, p, q, r)`	c_5	n

	MERGE(A, p, q, r)	Cost	Executions
1	`n_1 = q - p + 1`	c_1	1
2	`n_2 = r - q`	c_2	1
3	`create arrays L[1..n_1+1] and R[1..n_2+1]`	c_3	1
4	`for i = 1 to n_1`	c_4	$n/2$
5	`    L[i] = A[p + i - 1]`	c_5	$n/2$
6	`for j = 1 to n_2`	c_6	$n/2$
7	`    R[j] = A[q + j]`	c_7	$n/2$
8	`L[n_1 + 1] = R[n_2 + 1] =` ∞	c_8	1
9	`i = j = 1`	c_9	1
10	`for k = p to r`	c_{10}	$n/2$
11	`    if L[i]` $\leq$ `R[j]`	c_{11}	$n/2$
12	`        A[k] = L[i]`	c_{12}	$n/2$
13	`        i = i + 1`	c_{13}	$n/2$
14	`    else`	c_{14}	$n/2$
15	`        A[k] = R[j]`	c_{15}	$n/2$
16	`        j = j + 1`	c_{16}	$n/2$

Best case	Worst case	Average case
$\Omega(n \log(n))$	$O(n \log(n))$	$\Theta(n \log(n))$

Quicksort

	QUICK-SORT(A, p, r)	Cost	Executions
1	`if p < r`	c_1	n
2	`    q = PARTITION(A, p, r)`	c_2	n
3	`    QUICK-SORT(A, p, q - 1)`	c_3	$n/2$
4	`    QUICK-SORT(A, q + 1, r)`	c_4	$n/2$

	PARTITION(A, p, q)	Cost	Executions
1	`x = A[p]`	c_1	1
2	`i = p`	c_2	1
3	`for j = p + 1 to q`	c_3	$n/2$
4	`    if A[j]` $\leq$ `x`	c_4	$n/2$
5	`        i = i + 1`	c_5	$n/2$
6	`    swap A[i] and A[j]`	c_6	$n/2$
7	`swap A[p] and A[i]`	c_7	1

Best case	Worst case	Average case
$\Omega(n \log(n))$	$O(n^2)$	$\Theta(n \log(n))$

Techniques

Divide and conquer

Recursive. Splits the vector into subsets and sort each subset individually, and then combine the results.

Dynamic programming

Brute force through all possible solutions, and store solutions of sub-problems that are repeating, to prevent re-computation.

Greedy algorithms

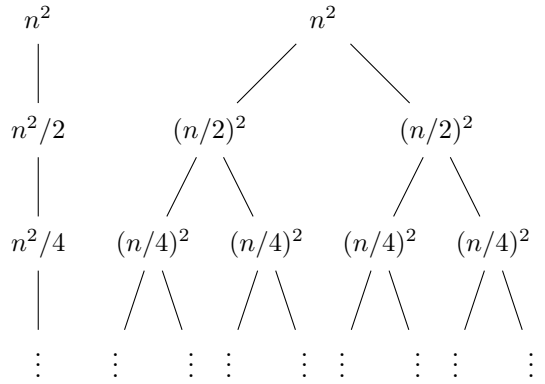
Always make the choice that seems the best at each time step. These algorithms selects locally-optimal choice.

Recursion-tree

A recursion tree is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.
The tree to the right is from the recurrence

$T(n) = 2T(n/2) + n^2$

With a height of $\log n$.
This a geometric series, and in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.



Complexity

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function $T(n)$ - time versus the input size n. We want to define time taken by an algorithm without depending on the implementation details. But you agree that $T(n)$ does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm asymptotically. We will measure time $T(n)$ as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, additon of two bits might take different time, say $c1$ and $c2$, thus the additon of two n-bit integers takes $T(n) = c1 * n$ and $T(n) = c2 * n$ respectively. This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

Asymptotic notations

> The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function T(n) using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement
>
> $T(n) = O(n2)$
>
> says that an algorithm has a quadratic time complexity.

For any monotonic functions $f(n)$ and $g(n)$ from the positive integers to the positive integers, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that

$f(n) \leq c \times g(n),$ for all $n \leq n_0$

Intuitively, this means that function $f(n)$ does not grow faster than $g(n)$, or that function $g(n)$ is an **upper bound** for $f(n)$, for all sufficiently large $n \to \infty$.

Master method

The master method is a formula for solving recurrence relations of the form:

$T(n) = aT(n/b) + f(n)$

where n is the input size, a the number of sub-problems (constant), n/b the size of each sub-problem (they are all equal size, constant) and $f(n)$ the cost done outside the recursion.

$T(n)$ has the following asymptotic bounds:

1. $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

where $\epsilon > 0$ is a constant.

Which can be interpreted as

1. If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level *i.e.*, $n^{\log_b a}$.
2. If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$. Thus, the time complexity will be $f(n)$ times the total number of levels *i.e.*, $n^{\log_b a} \times \log n$.
3. If the cost of solving the sub-problems at each level decreases by a certain factor, the value of f(n) will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of $f(n)$.

Examples

a) $T(n) = 2T(n/4) + 1$

We see that $a = 2$ and $b = 4$, and that $\log_4 2 = 1/2$.
From the equation we have that

$f(n) = O(1) = O(n^{\log_4 2 - \epsilon}) = O(n^{1/2 - 1/2}) \qquad \Rightarrow \qquad \epsilon = 1/2$

Therefore, the asymptotic bounds follows case 1.

$T(n) = \Theta(n^{\log_4 2}) = \Theta(n^{1/2}) = \Theta(\sqrt{n})$

b) $T(n) = 2T(n/4) + \sqrt{n}$

We see that $a = 2$ and $b = 4$, and that $\log_4 2 = 1/2$.
From the equation we have that

$f(n) = O(\sqrt{n}) = O(n^{1/2}) = O(n^{1/2})$

Therefore, the asymptotic bounds follows case 2.

$T(n) = \Theta(n^{\log_4 2} \times \log n) = \Theta(n^{1/2} \times \log n) = \Theta(\sqrt{n} \times \log n)$

c) $T(n) = 2T(n/4) + n$

We see that $a = 2$ and $b = 4$, and that $\log_4 2 = 1/2$.
From the equation we have that

$$f(n) = O(n) = O(n^{\log_4 2+\epsilon}) \qquad \Rightarrow \qquad \epsilon = 1/2$$

Therefore, the asymptotic bounds follows case 3.

$$T(n) = \Theta(f(n)) = \Theta(n)$$

d) $T(n) = 2T(n/4) + n^2$

We see that $a = 2$ and $b = 4$, and that $\log_4 2 = 1/2$.
From the equation we have that

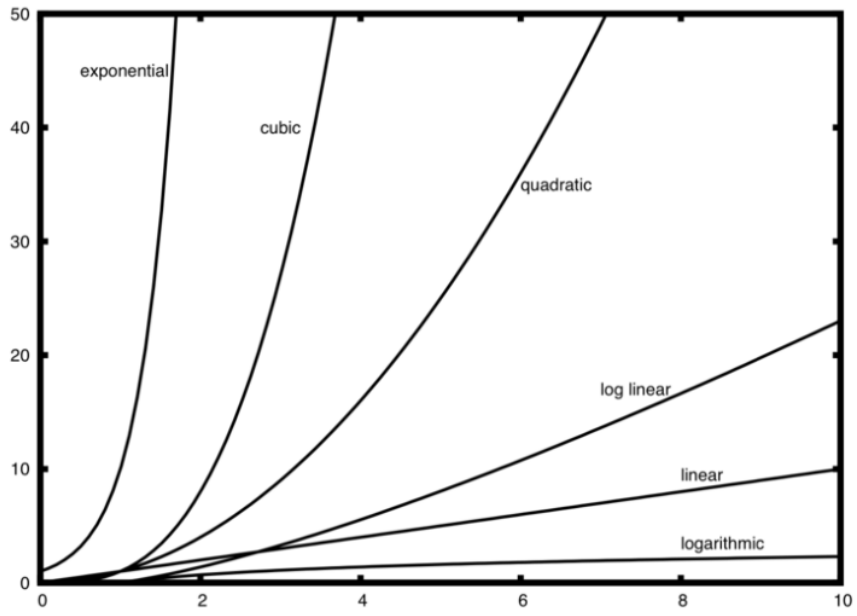
$$f(n) = O(n^2) = O(n^{\log_4 2+\epsilon}) \qquad \Rightarrow \qquad \epsilon = 3/2$$

Therefore, the asymptotic bounds follows case 3.

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

Growth

The growth of an algorithm gives a simple characterization of the algorithm's efficiency. In addition, it is a good metric for comparing relative performance of alternative algorithms.
Usually, an algorithm that is asymptotically more efficient is the better choice for all but very small inputs.



O-notation

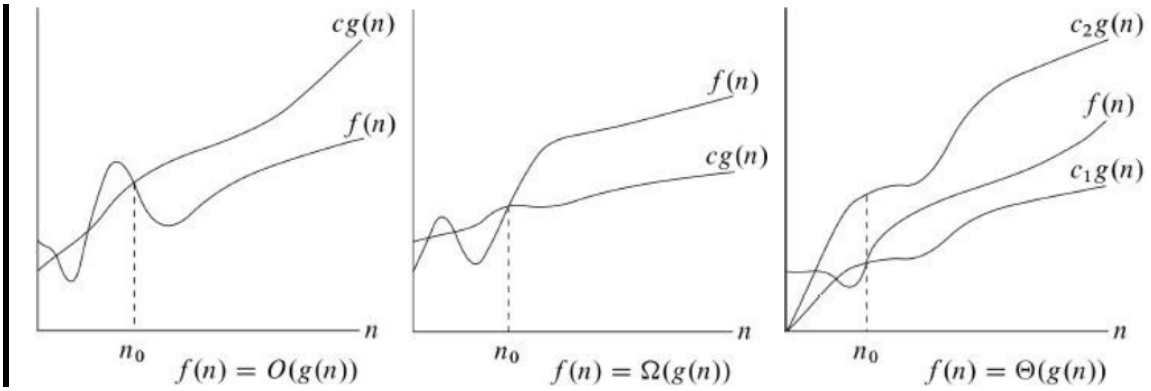
We use O-notation when we have only an asymptotic upper bound.
When applied to a function, the O-notation is an approximation of said function, within a constant factor.
Here, $f(n)$ is the function we are trying to describe in terms of growth rate. $g(n)$ is a simpler function, used for comparison, where c is a constant defining the bounds. n_0 is a threshold beyond which the bounds hold true.

Ω-notation

We use Ω-notation when we have only an asymptotic lower bound.
When applied to a function, the Ω-notation is an approximation of said function, within a constant factor.
Here, $f(n)$ is the function we are trying to describe in terms of growth rate. $g(n)$ is a simpler function, used for comparison, where c is a constant defining the bounds. n_0 is a threshold beyond which the bounds hold true.

Θ-notation

We use Θ-notation for analyzing the average-case complexity of an algorithm, since it represents the upper and the lower bound of the running time of an algorithm.
Here, $f(n)$ is the function we are trying to describe in terms of growth rate. $g(n)$ is a simpler function, used for comparison, where c_1 and c_2 are constants defining the bounds. n_0 is a threshold beyond which the bounds hold true.


cg(n)
f(n)
n
n0
f(n) = O(g(n))
f(n)
cg(n)
n
n0
f(n) = Ω(g(n))
c2g(n)
f(n)
c1g(n)
n
n0
f(n) = Θ(g(n))

Proof

Direct

A direct proof is one of the most familiar forms of proof. We use it to prove statements of the form `if p then q` or `p implies q` which we can write as `p` $\Rightarrow$ `q`. The method of the proof is to takes an original statement `p`, which we assume to be true, and use it to show directly that another statement `q` is true. So a direct proof has the following steps:

- Assume the statement `p` is true.
- Use what we know about `p` and other facts as necessary to deduce that another statement `q` is true, that is show `p` $\Rightarrow$ `q` is true.

Example

Directly prove that if n is an odd integer then n^2 is also an odd integer.

Let `p` be the statement that n is an odd integer and `q` be the statement that n^2 is an odd integer. Assume that n is an odd integer, then by definition $n = 2k + 1$ for some integer k. We will now use this to show that n^2 is also an odd integer.

$$\begin{aligned} n^2 &= (2k+1)^2 && \text{since } n = 2k+1 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

Hence we have shown that n^2 has the form of an odd integer since $2k^2 + 2k$ is an integer. Therefore we have shown that `p` $\Rightarrow$ `q` and so we have completed our proof.

Contradiction

We begin by making a certain assumption. If we can then show that this assumption in all possible cases will lead to two statements contradicting each other, we conclude that the assumption must be wrong. If we want to prove that a certain proposition is true, we typically assume the opposite; if that leads to a contradiction, we can conclude that the original proposition is true.

Example

There is no rational number, x, i.e., no $x = p/q$ with $p, q \in \mathbb{N}$, such that $x^2 = 2$. More colloquially: $\sqrt{2}$ is not a rational number, i.e., it cannot be expressed as a fraction of two natural numbers.

Assumption: There is a rational number x such that $x^2 = 2$.
This assumption implies that there are $p, q \in \mathbb{N}$ such that $x = p/q$. We can further assume that p and q are the smallest such number, i.e., that call common factors have been divided out.
We then have

$$x^2 = \left(\frac{p}{q}\right)^2 = \frac{p^2}{q^2} = 2 \Longleftrightarrow p^2 = 2q^2$$

This means that p^2 is even, and therefore that p is even. We can therefore write $p = 2s$ for some $s \in \mathbb{N}$. Then we have

$$p^2 = (2s)^2 = 4s^2 = 2q^2 \Longleftrightarrow 2s^2 = q^2.$$

Thus, also q^2 and therefore q is even and we can write $q = 2t$ for some $t \in \mathbb{N}$.
This means that we have

$$x = \frac{p}{q} = \frac{2s}{2t}$$

Induction

A proof by induction of $P(n)$, a mathematical statement involving a value n, involves these main steps:

- Prove directly that P is correct for the initial value of n (for most examples you will see this is zero or one). This is called the **base case**.
- Assume for some value k that $P(k)$ is correct. This is called the **induction hypothesis**.
- We will now prove directly that $P(k) \Rightarrow P(k+1)$. That means prove directly that $P(k+1)$ is correct by using the fact that $P(k)$ is correct. This is called the **induction step**.

The combination of these steps shows that $P(n)$ is true for all values of n.

Example

Prove by induction that $n^3 + 2n$ is divisible by 3 for every non-negative integer n.

Let P(n) be the mathematical statement $n^3 + 2n$ is divisible by 3.
Base case: When $n = 0$ we have $0^3 + 0 = 0 = 3 \times 0$. So $P(0)$ is correct.
Induction hypothesis: Assume that $P(k)$ is correct for some positive integer k. That means $k^3 + 2k$ is divisible by 3 and hence that $k^3 + 2k = 3m$ for some integer m.
Induction step: We will now show that $P(k+1)$ is correct. So we will take the original formula and replace the n with $k + 1$ to get $(k + 1)^3 + 2(k + 1)$ and we will show that this is divisible by 3. At some stage in the proof we will need to use the fact that $k^3 + 2k = 3m$, so when we re-arrange the formula we will try to get $k^3 + 2k$ as part of it.

$$\begin{aligned}(k+1)^3 + 2(k+1) &= k^3 + 3k^2 + 3k + 1 + 2k + 2 \\ &= 3m + 3(k^2 + k + 1) \\ &= 3(m + k^2 + k + 1)\end{aligned}$$

As $m + k^2 + k + 1$ is an integer we have that $(k + 1)^3 + 2(k + 1)$ is divisible by 3, so $P(k + 1)$ is correct.
Hence by mathematical induction $P(n)$ is correct for all non-negative integers n.

Binary tree

A **binary tree** is a rooted tree in which every node has at most two children.
A **full binary tree** is a special type of binary tree in which every parent node/internal node as either two or no children.
A **complete binary tree** is a binary tree in which all the levels are completely filled (except possibly the lowest, which is filled from the left).
Basic operations take time proportional to height of the tree.

- Complete binary tree with n nodes; worst case $\Theta(\texttt{log}(n))$
- Linear chain of n nodes; worst case $\Theta(n)$

Binary search tree

Left-child, right-sibling representation.
For any two elements at the same height, the left element is less than (or equal to) the right element.

Traversal in-order

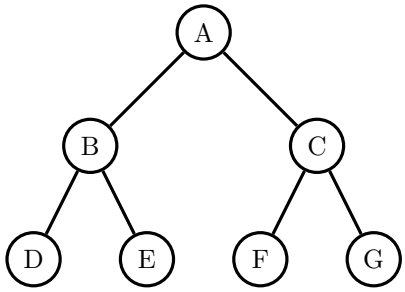
D B E A F C G

Traversal pre-order

A B D E C F G

Traversal post-order

D E B F G C A



Querying

Recursively search each sub-tree, left before right.

```
    TREE-SEARCH(Node, value)
1   if not Node.value or value = Node.value
2       return Node.value
3   if value < Node.value
4       return TREE-SEARCH(Node.left, value)
5   else
6       return TREE-SEARCH(Node.right, value)
```

Insertion

```
    INSERT(Node = root, value) → Node
1   if not Node
2       Node = NODE(value)
3   else if value < Node.value
4       Node.left = INSERT(Node.left, value)
5   else
6       Node.right = INSERT(Node.right, value)
7   return Node
```

Deletion

Kanskje skrive noe her. Forelesning 14.

Transplant

Kanskje skrive noe her. Forelesning 14.

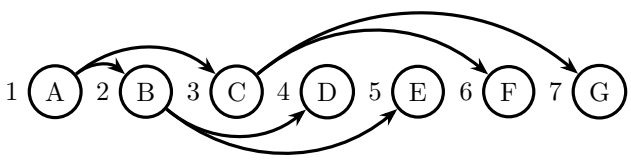
Rotation

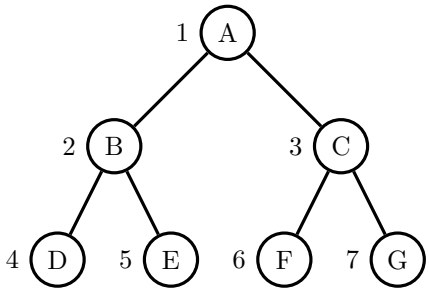
Kanskje skrive noe her. Forelesning 16, nederst.

Heap

A heap is a complete binary tree, which can be represented as an array.
The minimum and maximum numbers of nodes in a heap of height h is

$\texttt{max} = 2^{h+1} - 1$ $\texttt{min} = 2^h$ $h = \texttt{log(\# nodes)}$





Max-heap

```
     MAX-HEAPIFY(A, i)
 1   l = LEFT(i)
 2   r = RIGHT(i)
 3   if l ≤ A.heapsize and A[l] > A[i]
 4       largest = l
 5   else largest = i
 6   if r ≤ A.heapsize and A[r] > A[largest]
 7       largest = r
 8   if largest ≠ i
 9       swap A[i] and A[largest]
10       MAX-HEAPIFY(A, largest)
```

Where each node satisfies the max-heap property of `parent` $\geq$ `child` (values). Thus, the largest element is always stored at the root.

Min-heap

Where each node satisfies the min-heap property of `parent` $\leq$ `child` (values). Thus, the smallest element is always stored at the root.

Stacks and queues

Stack

LIFO: last in, first out. Push and pop ONLY at the top.

Queue

FIFO: first in, first out. Push to the top, pop from the bottom.

Linked list

Flexible with respect to insertion and deletion anywhere – only neighboring pointers needed to be modified. For singly-linked list, only prior and new pointer needs updating (as well as inserted pointer). For doubly-linked list, both prior and next pointers are updated (as well as inserted pointers).
Search requires traversing all elements until correct key is found. The best-case scenario, the search operation can take $O(1)$ time if the desired element is the first node in the list. In the worst-case scenario, if the desired element is not in the list or is the last node, the search operation will have to traverse all the elements, resulting in a time complexity of $O(n)$, where n is the number of nodes in the list.
A circular list is a special case, where the last element points to the first and vice versa.

ADVANTAGES

- *Dynamic Data Structure*

  Linked List being a dynamic data structure can shrink and grow at the runtime by de-allocating or allocating memory, so there is no need for an initial size in linked list.

- *Implementation*

  Some very helpful data structures like queues and stacks can be easily implemented using a Linked List.

- *Insertion and Deletion Operation*

  In a Linked List, insertion and deletion operations are quite easy, as there is no need to shift every element after insertion or deletion.

DISADVANTAGES

- *Memory Usage*

  The memory required by a linked list is more than the memory required by an array, as there is also a pointer field along with the data field in the linked list.

- *Random Access*

  To access node at index x in a linked list, we have to traverse through all the nodes before it. But in the case of an array, we can directly access an element at index x, using arr[x].

Graph

Let `G = (V, E)` be a graph where `V` is the set of vertices and `E` the set of edges. `G` can be directed or undirected.

`|V|` Number of vertices (nodes)

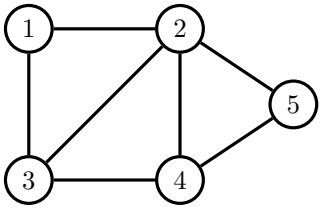
`|E|` Number of edges

Adjacency lists

Array adjacency of `|V|` lists; one per vertex.
List of vertex `u` contains all vertices `v` with `(u, v)` $\in$ `E`.

- List of all vertices adjacent to `u`
- List of all edges "leaving" `u`

`G.Adj[u]` is adjacency list for vertex `u` of graph `G`.



1 → 2 3
2 → 1 3 4 5
3 → 1 2 4
4 → 2 3 5
5 → 2 4

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

For directed graphs, the above representation changes. Here, it is assumed each connection is bidirectional (undirected).

Breadth-first search

```
    BFS(graph, start)
 1  for u in graph.V
 2      u.prev = none
 3      u.dist = inf
 4  start.dist = 0
 5  Q = Dequeue()
 6  Enqueue(Q, start)
 7  while Q
 8      u = Dequeue(Q)
 9      for v in graph.Adj[u]
10          if v.dist == inf
11              v.dist = u.dist + 1
12              v.prev = u
13              Enqueue(Q, v)
```

INPUT

- Graph `graph = (V, E)`, directed or undirected.
- Source vertex `start` $\in$ `V`.

OUTPUT

- For all `v` $\in$ `V`, the distance `v.dist` from `source` to `v`, i.e., smallest number of edges between `source` and `v`.
- For each `v` also `v.prev`, the predecessor of `v` on a shortest path from `start` to `v`, i.e., the vertex `u` such that `(u, v)` is the last edge on a shortest path from `start` to `v`.
- The set of all edges {`(v, prev, v)`|`v` $\neq$ `start`} forms a tree.

Depth-first search

```
    DFS(graph)
1   for u in graph.V
2       u.prev = none
3       u.visited = false
4   t = 0
5   for u in graph.V
6       if not u.visited
7           t = DFS-VISIT(graph, u, t)
```

```
     DFS-VISIT(graph, u, t) → t
 1   t = t + 1
 2   u.discovery = t
 3   for v in graph.Adj[u]
 4       if not v.visited
 5           v.prev = u
 6           t = DFS-VISIT(graph, v, t)
 7   u.visited = true
 8   t = t + 1
 9   u.final = t
10   return t
```

Recursively explore *every* edge in graph. Each vertex is assigned a discovery time `discovery`, and finishing time `final`, including its predecessor, `prev`.

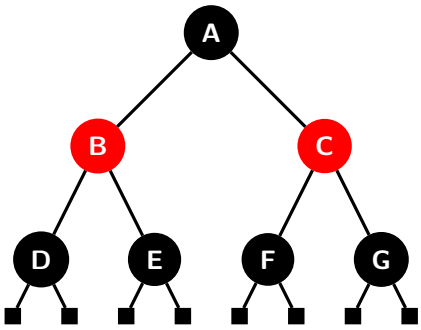
Topological sort

Once we have a finish time for a vertex, we have explored all descendants of said vertex.
Any vertices that have not finished at this time, may be its predecessor.
Start with activity with latest finishing time, then execute activities in order of descending finishing time.

Red-black trees

Keeps tree height bounded by `O(log(n))`. Each node has a color as an additional property (either red or black). The tree remains balanced by maintaining rules about color relationships between nodes.

- The root is black.
- Every leaf (■) is black.
- If a node is red, then both children are black.
- For each node, all paths from the node to descendant leaves contain the same number of black nodes.



Any node with height h has black height $\geq h/2$.
The subtree rooted at an arbitrary node x contains $\geq 2^{bh(x)} - 1$ internal nodes.
A red-black tree with n internal nodes has height $\leq 2\texttt{log}(n+1)$.

Insertion

Ahr, kjedelig. Forelesning 16, nederst.

Hash tables

Like a dictionary of lists. Thus, improve search time, by dividing original list into multiple doubly linked sub-lists, each with a corresponding key. Doubly linked to speed up deletion.
Each entry of a hash table is a reference to head of list of elements with corresponding key (or null if no elements exists for said key).

Hash function

A hash function must consistently map input to a fixed-size output, be fast to compute, and distribute inputs uniformly across the output space to minimize collisions.
But simple uniform hashing impossible to guarantee in practice, since distribution of keys not known in advance.
Thus, use heuristics (experience) to choose hash functions that work well for the domain of keys at hand.

Division method

One method for hashing is:

$h(k) = k \ \texttt{mod} \ m$

Which is fast, but needs some consideration. For instance, $m = 2^p$ for all $p \in \mathbb{N}$ is a bad choice.
A good choice is a prime, not close to a power of 2.

Chaining

When storing data of each key in a linked list.
Load factor in hashing is a measure of how full a hash table is.

$$\texttt{load factor} = \frac{\texttt{\# items}}{\texttt{\# slots (keys)}}$$

The disadvantage of chaining in hash tables is due to there potentially being long chains at certain keys/slots, slowing down search to linear time.

Open addressing

For a key k, start probe sequence at $\texttt{hash}'(k)$. Move to next slot if occupied. Repeat until empty slot is found (wrap at end of table).

$\texttt{hash}(k, i) = (\texttt{hash}'(k) + i) \ \texttt{mod} \ m$

Randomized algorithms

> Hoppet over dette. Forelesning 18.

Backtracking

> Used to solve problems with a large search space, by systematically eliminating possibilities.

```
    BACKTRACK(state)
1   for choice in possibilities
2       new state = make choice
3       result = BACKTRACK(new state)
4       return result if solution
5   return failure
```

Minimax algorithm

> Minimizing the worst-case potential loss (minimizing the maximum risk). In the context of a two-player game, one player is assumed to be the maximizer and the other is the minimizer. The algorithm recursively evaluates all possible outcomes and chooses the move that minimizes the maximum possible loss.

Djikstra algorithm

> Ahr, kjedelig. Nederst forelesning 22.