KTH Computer Science
and Communication

Exercise sets in

Algorithms, data structures and complexity

Fall 2015

Övning 1: Algorithm analysis	2
Övning 2: Data structures and graphs	6
Övning 3: Divide and conquer; and dynamic programming	10
Övning 4: Dynamic programming	15
Övning 5: Graph algorithms and lower bounds	21
Övning 6: Algorithm construction	25
Övning 7: Probabilistic algorithms, reductions	29
Övning 8: Undecidability	33
Övning 9: NP completeness proofs	36
Övning 10: NP complete problems	41
Övning 11: Approximation algorithms	44
Övning 12: Complexity classes and mixed questions	47

Created and collected by Viggo Kann viggo@csc.kth.se
Translated to English by Marcus Dicander 2015.

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 1

Algorithm analysis

Ordo When n grows, which function grows faster? Does $f(n) \in \Theta(g(n))$, $f(n) \in O(g(n))$ or $f(n) \in \Omega(g(n))$.

	$f(n)$	$g(n)$
a)	$100n + \log n$	$n + (\log n)^2$
b)	$\log n$	$\log n^2$
c)	$\frac{n^2}{\log n}$	$n(\log n)^2$
d)	$(\log n)^{\log n}$	$\frac{n}{\log n}$
e)	$\sqrt{n}$	$(\log n)^5$
f)	$n2^n$	3^n
g)	$2^{\sqrt{\log n}}$	$\sqrt{n}$

Division For this task, analyze the division algorithm called long division. This is what long division looks like when 721 is divide by 64. (If you are used to having the divisor on the other side, feel free to swap sides.)

```
   0 1 1
 ---------
64 | 7 2 1
   -0
   ---
     7 2
   -6 4
   ----
     8 1
   -6 4
   ----
     1 7
```

First the algorithm checks how many times 64 divides 7, the most significant digit of 721. Since $64 > 7$ that makes 0 times. We have $0 \cdot 64 = 0$ so we subtract 0 from 7 giving 7. We multiply 7 with 10 (the base), move the next digit (2) in 721 down, and continue by dividing 72 with 64 to get the next digit in the quota. We continue in this way, subtracting the number of times 64 divides every two digit number, move down the next number and end when the integer part of the quota has been calculated. The 17 at the bottom is the remainder.

We can formulate this in the following algorithm where $q = b/a$ where a, b are n bit numbers ($a = a_{n-1}a_{n-2}\cdots a_0$, $b = b_{n-1}b_{n-2}\cdots b_0$) in the base B. Let $x \ll y$ be x left shifted y steps.

To simplify the correctness proof, we have added preconditions, postconditions and invariants (see notes for Lecture 6).

$Div(a, b, n) =$
PRE: $a > 0, b \geq 0$, a and b are stored with n bits.
POST: $qa + r = b, 0 \leq r < a$
$r \leftarrow 0$
for $i \leftarrow n - 1$ **to** 0 **do**
INV: $(q_{n-1}...q_{i+1}) \cdot a + r = (b_{n-1}...b_{i+1}), 0 \leq r < a$
$r \leftarrow (r \ll 1) + b_i$ /* Switch to next digit */
$q' \leftarrow 0$
$a' \leftarrow 0$
while $a' + a \leq r$ **do** /* Find the maximum q' so that $q'a \leq r$ */
INV: $a' = q'a \leq r$
$a' \leftarrow a' + a$
$q' \leftarrow q' + 1$
$q_i \leftarrow q'$
$r \leftarrow r - a'$
return $\langle q, r \rangle$ /* quota and remainder */

What is the time complexity? Should bit cost or unit cost be used?

Euclid's algorithm Analyze Euclid's algorithm for finding the Greatest Common Divisor (GCD) between two integers. Do it once for unit cost, once for bit cost and analyze the difference. Euclid's algorithm reads, assuming that $a \geq b$:

$gcd(a, b)=$
if $b|a$ **then**
$gcd \leftarrow b$
else
$gcd \leftarrow gcd(b, a \bmod b)$

Powers with repeated squaring The following algorithm calculates powers of two when the exponent itself is a power of two.

Input: $m = 2^n$
Output: 2^m
$power(m) =$
$pow \leftarrow 2$
for $i \leftarrow 1$ **to** $\log m$ **do**
$pow \leftarrow pow \cdot pow$
return pow

Analyze the complexity both with unit cost and also with bit cost. Explain which cost model is the most reasonable.

Solutions

Solution to Ordo

a) We calculate the limit for the quota between the functions.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{100n+\log n}{n+(\log n)^2}=\lim_{n\to\infty}\frac{100+(\log n)/n}{1+(\log n)^2/n}=100.$$

Since the limit is constant, we conclude that $f(n)\in\Theta(g(n))$.

b) We note that $\log n^2=2\log n$, so $\log n\in\Theta(\log n^2)$.

c)

$$\frac{g(n)}{f(n)}=\frac{n(\log n)^2}{n^2/\log n}=\frac{(\log n)^3}{n}\to 0 \text{ when } n\to\infty.$$

Therefore $g(n)\in O(f(n))$ and $f(n)\in\Omega(g(n))$.

d) Substitute $m=\log n$ and then compare $f_2(m)=m^m$ and $g_2(m)=2^m/m$:

$$\frac{g_2(m)}{f_2(m)}=\frac{2^m}{m\cdot m^m}=\frac{1}{m}\left(\frac{2}{m}\right)^m\to 0$$

then $n\to\infty$. Therefore $f(n)\in\Omega(g(n))$ and $g(n)\in O(f(n))$.

e) Polynomial functions always grow faster than polylogarithmic functions. Therefore $f(n)\in\Omega(g(n))$ and $g(n)\in O(f(n))$.

For a more formal proof, use the lemma which states that for every constant $c>0$, $a>1$ and all strictly monotonically growing functions $h(n)$ is $(h(n))^c\in O(a^{h(n)})$. If we choose $h(n)=\log n$, $c=5$ and $a=\sqrt{2}$ we get $(\log n)^5\in O(\sqrt{2}^{\log n})=O(\sqrt{n})$ since $\left(2^{1/2}\right)^{\log n}=\left(2^{\log n}\right)^{1/2}=n^{1/2}$.

f) $\frac{f(n)}{g(n)}=\frac{n2^n}{3^n}=n\left(\frac{2}{3}\right)^n$.

An exponential function such as $(2/3)^n$ always defeats a polynomial function (such as n), but let us prove that with l'Hôpital's rule. Rewrite the expression: $n\left(\frac{2}{3}\right)^n=n/\left(\frac{2}{3}\right)^{-n}$. Add variable names for nominator and denominator, $r(n)=n$ and $s(n)=\left(\frac{2}{3}\right)^{-n}$, and derive.

$$r'(n)=1,\qquad s'(n)=\left(\frac{2}{3}\right)^{-2n}\left(\frac{2}{3}\right)^n\ln\left(\frac{3}{2}\right)=\frac{\ln(3/2)}{\left(\frac{2}{3}\right)^n}$$

We can use l'Hôpital since $r(n)\to\infty$, $s(n)\to\infty$ and $s'(n)\neq 0$.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{r(n)}{s(n)}=\lim_{n\to\infty}\frac{r'(n)}{s'(n)}=\frac{\left(\frac{2}{3}\right)^n}{\ln\left(\frac{3}{2}\right)}\to 0$$

and we have proved that $f(n)\in O(g(n))$.

g) $\sqrt{n}\in\Omega(2^{\sqrt{\log n}})$. Note that $\sqrt{n}=2^{\frac{1}{2}\log n}$ and that $\lim_{n\to\infty}2^{\sqrt{\log n}-\frac{1}{2}\log n}=0$.

□

Solution to Division

Since the input always consists of two numbers but the numbers' size may vary, and since the time only depends on the size of the numbers, bit cost is more reasonable. The for loop runs n turns. In every turn we have a constant number of assignments, comparisons, additions and subtractions as well as a while loop which runts at most B turns (since B is the base and the number $B \cdot a \geq r$). Since the base can be regarded as constant, every turn in the for loop contains a constant number of operations. Since all arithmetic is performed with n bit numbers, every comparison, addition and subtraction takes the time $O(n)$. Finally, we get $n \cdot c \cdot O(n) = O(n^2)$. □

Solution to Euclid's algorithm

Since the algorithm is recursive it is not obvious that it always terminates, so we begin by proving this. In termination proofs we often use a power variableåhich has a lower bound (for example 0) and at every call decreases with a whole number step. In this algorithm we can use a as a power variable. Since a is always at least as large as b and the algorithm terminates as soon as $a = b$ or $b = 1$ we have a lower bound for a and we can see that a decreases in every call. The number of calls are apparently dependent on the size of the parameters so let us see how the largest parameter a decreases. Let a_i be a's value in call number i in the algorithm.

Lemma 1 $a_{i+2} \leq a_i/2$.

Proof. We know that $a_{i+2} = b_{i+1}$ and $b_{i+1} = a_i \bmod b_i$, which is why $a_{i+2} = a_i \bmod b_i$. Now assume that $a_{i+2} > a_i/2$. This means that $b_i \geq a_i/2$, which gives a contradiction since $a_i = a_{i+2} + cb_i > a_i/2 + a_i/2 = a_i$. □

Use the lemma to prove that $\lceil \log a_{i+2} \rceil \leq \lceil \log \frac{a_i}{2} \rceil = \lceil \log a_i - \log 2 \rceil = \lceil \log a_i \rceil - 1$. This means that in every second function call the size of the parameter decreases with at least one bit. Therefore the number of calls can be at most $2\lceil \log a \rceil$.

In every recursive call we perform one modulo operation and with unit cost that operation is done in constant time. Our time complexity is therefore $2\lceil \log a \rceil = 2n \in O(n)$.

When we analyze according to bit cost we have to be more careful. A division between two n bit integers takes (as we have seen) $O(n^2)$, and the modulo operation therefore takes $O(n^2)$. So if we make $O(n)$ calls and every call takes $O(n^2)$ the total bit complexity becomes $O(n^3)$. The most appropriate cost model is determined by this questions: Are the numbers stored in variables of fixed length or dynamic length? The answers are unit cost and bit cost respectively. □

Solution to Powers with repeated squaring

The loop runs $\log m = n$ turns. Every turn takes constant time with unit cost. The complexity is therefore $O(n)$.

If we use bit complexity we need to examine the cost of each multiplication in the for loop. We assume that the cost of multiplying an l bit number with itself is $O(l^2)$ (which is an exaggeration. There are faster ways). During turn i the variable $pow = pow_i$ held the value 2^{2^i} so every multiplication costs $O((\log pow_i)^2) = O((\log 2^{2^i})^2) = O(2^{2i})$. If we sum over every turn we get

$$\sum_{i=1}^{\log m} c2^{2i} = c \sum_{i=1}^{\log m} 4^i = 4c \sum_{i=0}^{\log m - 1} 4^i = 4c\frac{4^{\log m} - 1}{4 - 1} \in O(4^{\log m}) = O(4^n).$$

This algorithm has linear complexity when measured with unit cost but exponential complexity when measured with bit cost! If the numbers are saved in fixed length variables the result with quickly overflow. In practice we need variables of dynamic size and bit cost is therefore most suitable. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 2

Data structures and graphs

During this exercise there is also a **hand-in of written solutions for the theory exercises for Lab 1** and oral presentations of the theory exercises. The theory is presented individually, unlike the lab which is done in pairs.

Merging sorted lists Describe an algorithm that combines k sorted lists in time $O(n \log k)$ where n is the total number of elements.

Data structure with maximum and minimum Construct a data structure with the following operations and complexity:

- `insert(x)` (insert an element) takes time $O(\log n)$,
- `deletemin()` (remove the smallest element) takes time $O(\log n)$,
- `deletemax()` (remove the largest element) takes time $O(\log n)$,
- `findmin()` (returns the smallest element) takes time $O(1)$,
- `findmax()` (returns the largest element) takes time $O(1)$.

n is the number of elements currently stored. You may assume that comparisons between two elements cost $O(1)$ time.

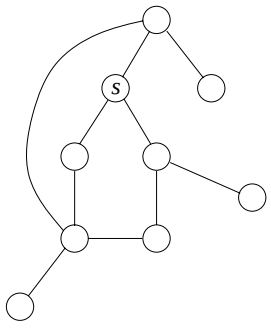
Bloom filter with removal A normal Bloom filter only has two operations: Insert and IsIn. How can we modify the data structure so that it also allows the operation Remove which is not allowed to take longer time than Insert?

Suffix rule lookup in the spell checker Stava Stava has about 1000 suffix rules of the form

-orna ← *-a, -an, -or*

When a word such as *dockorna* is to be checked, Stava will go through all the rules one by one after the first syllable, namely *-ϵ, -a, -na, -rna, -orna, -korna, -ckorna*. How should the suffix rules be stored for fast retrieval?

Graph search Do a DFS and BFS search through the following graph. Start in the vertex s and number the vertices in the order they are visited.



Topological sorting of DAG A Directed Acyclical Graph (DAG) can be topologically sorted. This means that each vertex is assigned a number so that all edges go from a vertex with a lower number to a vertex with a higher number.

Modify the Depth First Search so that it constructs a topological sorting of the DAG in linear time.

Clique A *Clique* in an undirected graph is a set of vertices such that every pair of vertices in the set has an edge between them. An important problem in computer science is finding the largest clique in a graph.

Another famous problem is finding the largest *independent set* in a graph. An independent set is a set of vertices where no pair of vertices have an edge between themselves.

These problems are so alike that if you have an algorithm for finding the largest clique in a graph, that algorithm can be used inside a very simple algorithm for finding the largest independent set. Show this!

Incidence matrix product At the lecture we showed how a directed graph $G = \langle V, E \rangle$ represented as an incidence matrix B of size $|V| \times |E|$. Describe what the elements of the matrix BB^T represent (where B^T is B transposed).

Bipartiteness Describe and analyze an algorithm which determines whether a graph is bipartite or not. The time complexity should be linear in the number of edges and vertices in the graph.

Solutions and hints

Hint to Merging sorted lists
Method 1: Use a heap with an element from each list and place the smallest element at the root. Every heap operation takes $O(\log k)$ and there will be n insert operations and n remove operations.
Method 2: Merge the lists pairwise. Now there are $k/2$ sorted lists. Merge these lists pairwise and so on. After $\log k$ merging rounds there is only one list left. Every merging round takes $O(n)$ because there are n elements in total to be merged in each merging round.

Solution to Data structure with maximum and minimum
You can either use a min-heap and a max-heap with the same values or use a balanced search tree with two extra variables, one for the largest value in the tree and one for the smallest.

Both solutions are easy to implement. For the binary tree cases you can trivially write `findmin` and `findmax`. For the three other operations you use the usual binary search tree operations and copy the smallest and largest values to the min and max variables. □

Solution to Bloom filter with removal
Let the Bloom filter consist of numbers (8 bit or 16 bit integers) instead of bits. 0 means false and every positive number means true. Instead of setting a bit to 1 the affected index is increased with 1. Remove works like insert but decreases every affected integer with 1. When an integer reaches 0 there are no elements that hash to that position.

The size of the numbers must be adapted to the number of elements expected to be in the Bloom filter. We don't want integer overflow in our counter. One solution is to add an extra bit for overflow and when an overflowed counter reaches zero the whole bloom filter must be recalculated.

Another option is to never decrease a counter that has reached the ceiling. This is probably safe since there must have been many elements that have been hashed to that position and the probability that they should all be removed is low.

□

Solution to Suffix rule lookup in Stava
We perform lookups on the left hand side of the rules. We call these rules for the entrance suffixes. All rules with an empty entrance suffix are stored in their own list. All other rules are stored in an array sorted by their entrance suffix read backwards. We use lazy hashing on a letter, i.e. we have an alphabetically sorted array giving the index for the first entrance suffix ending on that letter.

When searching, we first check the list of rules with an empty entrance suffix. Then we look up the last letter (a in *dockorna*) with lazy hashing and then go through the rules that only have the only letter as entrance suffix. After that we do a binary search for the second to last letter (n) among the entrance suffixes that end with a and go through the rules that have *na*, the rules that have *rna* as entrance suffix and so on.

This way we get fewer and fewer entrance suffixes to perform the binary search on.

□

Solution to Graph search – Your own exercise!

Solution to Topological sorting of DAG
If you look at the order the DFS passes the vertices when returning (the order in which the vertices are considered done"in the search procedure) you see that it is a reversed topological ordering. If you want to sort the vertices topologically you only have to add a single statement `Push(u)` at the end of the procedure `DFSVISIT(u)`. When the DFS search is done you can pop the vertices one

by one from the stack and they will be returned in a topological order. □

Solution to Clique
Hint: Study the complement graph G' (a graph with the same vertices where the vertices are adjacent iff they are not adjacent in G) □

Solution to Incidence matrix product
The diagonal element (i, i) indicates how many edges have their end point in i. The non diagonal element (i, j) indicates the negated number of edges between the vertices i and j. □

Solution to Bipartiteness
Use DFS but color the vertices alternatingly black and white. If an edge between two vertices of the same color is discovered the graph is not bipartite. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 3

Divide and conquer; and dynamic programming

Maximum and minimum with divide and conquer In the vector $v[1..n]$ there are n numbers. Construct a divide and conquer algorithm which finds the largest and smallest element in v. The algorithm may use at most $\lceil 3n/2 \rceil - 2$ comparisons between elements in v. The number of numbers in v is not guaranteed to be a power of two.

Matrix multiplication Strassen's algorithm can multiply two $n \times n$ matrices in time $O(n^{2.808})$ by dividing the matrices into 2×2 square matrices. The reason why Strassen's is faster than $O(n^3)$ is that it performs seven multiplications instead of eight to create the product matrix.

Another idea is to do the division into 3×3 block matrices instead. A couple of years ago. Viggo tried to find the minimum number of multiplications needed to multiply two 3×3 matrices. He almost managed to get it down to 22 multiplications. Suppose he had succeeded, what time complexity would this yield for two $n \times n$ matrices?

Majority with divide and conquer Input for this exercise is an array A with n elements. Construct and analyze an algorithm which determines whether an element in the array is in majority (it is repeated more than $n/2$ times), and returns it. The algorithm should be a divide and conquer algorithm and the time complexity should be $O(n \log n)$. The only allowed comparison operation on elements in A is $=$. The elements have no order relation.

Mobile Construct an algorithm which balances a mobile (the kinetic sculpture kind of mobile. Not the cell phone) Think of the mobile as a binary tree. The leaves are balls, hanging at the bottom of the mobile. Inner nodes represent thin straight sticks and the children are hanging in threads attached to the ends of each stick. The input is a binary tree with weights in the nodes. The weight in each leaf is the mass of the corresponding ball. The weight of each inner node is the length of the corresponding stick. The algorithm should return the same binary tree but on every inner node the number should be how many centimeters from the left endpoint you have to attach the thread connecting this stick to the one above it to ensure that everything is balanced.

Number sequences Suppose you have been given the recursion for a number sequence. Write the pseudo code for a dynamic programming algorithm to calculate S_n if

a)

$$S_n = \begin{cases} 1 & \text{if } n = 0, \\ 2(S_{n-1}) + 1 & \text{otherwise.} \end{cases}$$

b)

$$S_n = \begin{cases} 4 & \text{if } n = 1, \\ 5 & \text{if } n = 2, \\ \max\left(S_{n-1}, S_{n-2}, S_{n-1} - S_{n-2} + 7\right) & \text{otherwise.} \end{cases}$$

c)

$$S_n = \begin{cases} 4 & \text{if } n = 1, \\ 5 & \text{if } n = 2, \\ \max(S_{n-2}, S_{n-1} - S_{n-2} + 7) & \text{otherwise.} \end{cases}$$

Write down the first 7 numbers. Which values do you need to save during the calculation?

Generalized number sequences a) 2D recursion without input

Given this recursion, construct an algorithm which calculates $M[i, j]$ with dynamic programming. Provide an argument for why the order of calculation is correct.

$$M[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ M[i-1, j-1] + i & \text{otherwise.} \end{cases}$$

b) 2D recursion with input

Given two strings a and b of length n and m, let a_i be the letter in position i in string a and correspondingly b_j is the letter in position j in b, find an evaluation order and write an algorithm which solves the following recursion with dynamic programming.

$$M[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ M[i-1, j-1] + 1 & \text{if } a_i = b_j, \\ 0 & \text{otherwise.} \end{cases}$$

Solutions and hints

Solution to Maximum and minimum with decomposition
When you only have two numbers you only need one comparison to find the largest and the smallest number.

```
MinMax(v,i,j)=
  if i=j then return (v[i],v[i])
  else if i+1=j then
     if v[i]<v[j] then return (v[i],v[j])
     else return (v[j],v[i])
  else
     m:=Floor((j-i)/2)
     if Odd(m) then m:=m+1;
     (min1,max1):=MinMax(v,i,i+m-1);
     (min2,max2):=MinMax(v,i+m,j);
     min:=(if min1<min2 then min1 else min2);
     max:=(if max1>max2 then max1 else max2);
     return (min,max);
```

The tree of calculations will have $\lceil n/2 \rceil$ leaves and $\lceil n/2 \rceil - 1$ inner nodes. If n is even then all $n/2$ leaves are two element sequences and perform one comparison. If n is odd one leaf (the rightmost) is a single element which requires no comparison. In the leaves we perform $\lfloor n/2 \rfloor$ comparisons. In every inner node we perform two comparisons, adding up to $2\lceil n/2 \rceil - 2$. In total we get $\lfloor n/2 \rfloor + 2\lceil n/2 \rceil - 2 = \lceil 3n/2 \rceil - 2$ comparisons.

It can be proved that this can not be done with fewer comparisons. □

Solution to Matrix multiplication
The recursion equation is $T(n) = 22 \cdot T(n/3) + O(n^2)$. Use the Master Theorem to get the solution $T(n) = O(n^{\log_3 22}) = O(n^{2.814})$. □

Solution to Majority with divide and conquer
If there is a majority element it has to be in majority in at least one half of the array.
Recursive approach: Check for majority recursively in the left and right half and count how many times the half array majority elements occur in the whole array. If one element is in total majority it is returned.

```
Majority(A[1..n]) =
    if n = 1 then return A[1]
    mid← ⌈(n + 1)/2⌉
    majInLeft← Majority(A[1..mid−1])
    majInRight← Majority(A[mid..n])
    if majInleft=majInRight then return majInLeft
    noOfMajInLeft← 0
    noOfMajInRight← 0
    for i ← 1  to n do
        if A[i] =majInLeft then noOfMajInLeft←noOfMajInLeft+1
        else if A[i] =majInRight then noOfMajInRight←noOfMajInRight+1
    if noOfMajInLeft≥ m then return majInLeft
    if noOfMajInRight≥ m then return majInRight
    else return NULL
```

Time complexity: We have two recursive calls with the half array followed by the after work $O(n)$. Use the Master Theorem to get the time complexity $O(n \log n)$. □

Solution to Mobile
Consider a stick of length l in the mobile. Assume that the weight v is hanging from the left end and the weight w from the right. Let x be the distance from the sticks left end to the thread attaching the stick to the stick right above it. In order for the momentums to cancel each other out, we need according to the laws of mechanics $xv = (l-x)w$, in other words $x = lw/(v+w)$. We can now calculate x for every stick recursively from top to bottom in the tree. Let the recursive function return the weight of the mobile hanging in the current stick. The algorithm runs in linear time.

```
Balance(p)=
        if p.left =NIL then return p.num
        left ← Balance(p.left)
        right ← Balance(p.right)
        p.x ← p.num · right/(left + right)
        return left + right
```

□

Solution to Number sequences
a) 1, 3, 7, 15, 31, 63, 127

To calculate a number you only need the previous number. The order of evaluation is from the base case towards higher numbers.

```
NumberSequenceA(n) =
    v = 1
    for i = 1 to n do
      v = 2*v+1
    return v
```

b) 4, 5, 8, 10, 10, 10, 10

To calculate a number we need to know the two previous numbers. The order of evaluation is from the base case and onwards to higher numbers.

```
NumberSequenceB(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v1, v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

c) 4, 5, 8, 10, 9, 10, 9

To calculate a number we need to know the two previous numbers. The order of evaluation is from the base case and onwards upwards.

```
NumberSequenceC(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

□

Solution to Generalized number sequences

a) 2D recursion with no input.
Since we do not know what we are looking for, we let the algorithm return all of M. This is not what one would typically do if the algorithm was to be used to answer a specific question. Regardless of what the question would be, the first thing we need to do is to calculate all of M. When the whole matrix is filled we can answer questions about it such as what is the largest value or how many times does the largest value occur.

We have to start with a base case, but if we imagine a matrix M which is to be filled with correct values, the whole first row (where $i = 0$) and the whole left column (where $j = 0$) will be zero and this should not be calculated recursively. A typical order of evaluation for a program would be row by row from the top or column by column left to right. We chose row by row and put the values on the first row. After this we loop over all other rows and put the values column by column. The order of evaluation works because the information we need to calculate a new value is either on a line we have already calculated or earlier on the same line.

We can now calculate all of M. Here is our algorithm:

for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i, 0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    $M[i, j] \leftarrow M[i-1, j-1] + i$
return M

The time complexity will be dominated by the nested for loop and therefore $\Theta(nm)$.

b) 2D recursion with input
We start with the base cases. All the elements on the first row (top) and the first column (left) fulfil the condition for base case and can therefore be set to 0. We chose to start with the first row. After this, we calculate the values on all following rows from top to bottom, column by column. According to the recursion, the value is 0 if $a_i \neq b_j$, and otherwise calculated with the help of previously calculated values. We can always look at the previous row $(i-1)$, since we calculate one row at a time. On that row we can look at which elements we want, since the whole row is filled. In particular we may look at the element in column $j-1$. The order of evaluation works. The algorithm looks like this:

for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i, 0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    if $a_i = b_j$ **then**
      $M[i, j] \leftarrow M[i-1, j-1] + 1$
    else $M[i, j] \leftarrow 0$
return M

The time is, like in the previous example $\Theta(nm)$. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 4

Dynamic programming

This exercise also features a **hand-in of written solutions to the theory exercises for lab 2** and oral presentation of the theory exercises.

Swamp trek Tina is walking through a swamp represented by an $n \times n$ square grid from the left edge to the right. In every step she can go straight right, diagonally up right or diagonally down right. Different squares in the swamp have different costs. Ending up on square (i, j) costs $A[i, j]$, which is a positive whole number.

Describe an algorithm which finds the minimum cost for passing through the swamp.

a) Assume that Tina may start anywhere on the leftmost column.

b) Assume that Tina always starts on square $(n/2, 1)$.

c) Assume that Tina also has to end on square $(n/2, n)$.

d) Print the least costly route for Tina in c).

Combine coins and bills Given a (constant) set of coins and bills with values $v_1, \ldots, v_k$ dollars (All numbers in the input are positive whole numbers.) Formulate a recursion for how many ways you can select coins and bills to reach the sum n.

Longest common substring The strings `ALGORITHM` and `PHANTASMAGORIA` have the common substring `GORI`. The *longest common substring* between these strings has length 4. The substring has to be connected in both strings.

Construct an efficient algorithm which given two strings $a_1 a_2 \ldots a_m$ and $b_1 b_2 \ldots b_n$ calculates and returns the length of the longest common substring. The algorithm should be based on dynamic programming and the time complexity should be $O(nm)$.

Here are two fun but more complicated examples which we will probably not have time for in the exercise session.

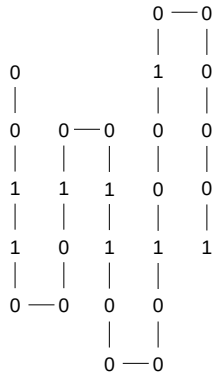
Protein folding A protein is a long chain of amino acids. The protein chain is not a straight line. It is folded in a complicated way which minimizes the potential energy. We want to calculate how a protein will fold. In this exercise we will study a simple model of protein folding where the amino acids are either *hydrophobic* or *hydrophilic*. Hydrophobic amino acids tend to clump together.

The protein may be seen as a binary string where the ones are the hydrophobic amino acids and the zeroes are the hydrophilic amino acids. The string (the protein) is to be folded in to a square grating. The goal is to get the hydrophobic amino acids to lump together. We have an optimization problem where the goal is to maximize the number of pairs of ones that are next to each other in the grating without being next to each other in the string.

You are to construct an algorithm which with the help of dynamic programming constructs an optimal *accordion folding* of a given protein string of length n. An accordion folding is a folding where the strings first goes straight down, then straight up, then straight down and

so on. In such a folding the horizontal pairs in the string always follow each other in the string so it is only vertical pairs (and not foldings) that add to the target function.

In the following illustration the string 00110001001100001001000001 is accordion folded so that the target function becomes 4.



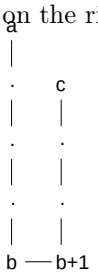
Definition of the problem Accordion Folding of Protein:

> Input: A binary string with n characters ('0' or '1').
>
> Problem: Find the accordion folding of the input string that gives the largest value for the goal function. In other words the largest numbers of pairs of ones that are next to each other in the grating but not in the string.

Construct and analyze the time complexity for an algorithm which solves the accordion folding of a protein chain with dynamic programming

We strongly suggest that you use this algorithm to calculate the number of pairs of ones in a turn (between to adjacent parts of the string) in an accordion folding that are adjacent in the grating but not in the string. Assume that the protein is stored in an array `p[1..n]`. The parameters `a` and `b` give index in the array for the first part's end points. The parameter `c` give the index for the second part's endpoint. See illustration below on the right.

```
profit(a,b,c) =
   shortest←min(b-a,c-(b+1));
   s←0;
   for i←1 to shortest do
      if p[b-i]=1 and p[b+1+i]=1 then
         s←s+1;
   return s;
```



Note: The protein folding problem is an important algorithmic problem which is studied in the course *Algorithmic Bioinformatics*. The course is resting as of 2015 but contact Lars Arvestad if you want to take it as an independent reading course.

Analysis tool for context free grammar A *context free grammar* is among other things used to describe the syntax of a programming language. A context free grammar in *Chomsky normal form* is described by

- a set of termination symbols T (written in lowercase),
- a set of non termination symbols N (written with uppercase),
- the start symbol S (one of the non terminal symbols in N),
- a set of rules on the form $A \to BC$ or $A \to a$, where $A, B, C \in N$ and $a \in T$.

If $A \in N$ then $\mathcal{L}(A)$ is defined through

$$\mathcal{L}(A) = \{bc : b \in \mathcal{L}(B) \text{ and } c \in \mathcal{L}(C) \text{ where } A \to BC\} \cup \{a : A \to a\}.$$

The language generated by the grammar is defined as $\mathcal{L}(S)$, which is all the strings of termination symbols that can be created by applying the rules starting from the start symbol S.

Example: Consider the grammar with $T = \{a, b\}$, $N = \{S, A, B, R\}$, the start symbol S and the rules $S \to AR$, $S \to AB$, $A \to a$, $B \to b$, $R \to SB$. We can see that the string $aabb$ is in the language through this chain of rule applications:

$$S \to AR \to aR \to aSB \to aSb \to aABb \to aaBb \to aabb.$$

In this case we can prove that the language consists of all strings with k number of a:s followed by k b:s where k is a positive integer.

Your task is to *construct* and *analyze* an efficient algorithm which determines whether or not a string is part of the language generated by a grammar. Input is a context free grammar on Chomsky normal form and a set of termination symbols. Output is true or false depending on whether the string could be generated by the grammar or not. Express the time complexity of your algorithm in the number of rules in the grammar and the length n of the string.
You can learn more about grammars in the language *Automata and languages*.

Solutions

Solution to Swamp trek
a) We let $W[i, j]$ be the minimum work required to reach square (i, j). Tina may start anywhere in the leftmost column. The base cases is to end up on a first square (from someplace outside of the swamp. That only costs $A[i, 0]$ for every i. We will express the cost of arriving at square i, j from square $i, j-1$, which is the cost $A[i, j]$ plus the cost of arriving at square $i, j-1$. We want to find the *least costly* way of arriving at every square i, j, so in general there are three places Tina may have appeared from, all in the column $j - 1$, and we must now compare the cost of arriving from each and one of those and chose the one with the lowest cost. It seems reasonable to calculate the cost column by column. Tina can not reach a square from the top edge so that value is set to infinity to prevent our comparison from favoring that direction. We deal with the bottom edge in the same way. Our recursion is therefore: $W[i, j] = \min(W[i-1, j-1], W[i, j-1], W[i+1, j-1]) + A[i, j]$. We calculate all values in the new matrix W.

The base cases:
for $j \leftarrow 1$ **to** n
  $W[1, j] = A[1, j]$

Fill in the matrix:
for $j \leftarrow 2$ **to** n
  for $i \leftarrow 1$ **to** n
    $fromabove = W[i-1, j-1]$ **if** $i > 1$, **else** ∞
    $fromsame = W[i, j-1]$
    $frombelow = W[i+1, j-1]$ **if** $i < n$, **else** ∞
    $W[i, j] = \min(fromabove, fromsame, frombelow) + A[i, j]$

Tina was allowed to leave the swamp from any square to the right. Therefore we can go through the last column in our matrix and look for the smallest value there.

Find the answer:
$opt \leftarrow \infty$
for $i \leftarrow 1$ **to** n
  if $W[i, n] < opt$ **then**
    $opt \leftarrow W[i, n]$
return *opt*

If we put all these three steps together we get a dynamic programming algorithm which calculates the least costly way of getting from the left side of the swamp to the right side.

b) If Tina always starts in square $(n/2, 1)$ we have a slightly different base case. Since Tina never can move straight up or straight down all other squares in the leftmost column are unreachable, i.e. the cost of getting there is infinite. The recursion is the same as above.

c) If Tina always ends in the square $(n/2, n)$, we will only be interested in the result in square $(n/2, n)$. This number will also be what we return. The recursion works as before.

d) For the construction of an optimal path we can either save the optimal source previous position in every position, in other words let W contain both cost and a previous pointer or just take our end square, subtract its value from the A matrix and find a possible square that we could have arrived from and repeat the process until we reach the starting square. If there is more than one path to a square that was equally costly then no matter. We just pick one of the least costly paths through the swamp and there may be many that are equally costly.

Since we started with the value in the end square and figured out from where we could have come to get that value and since we repeat this for every square in the path we will reach the starting square (otherwise we can not have any value in the ending square). In every square we have lowest cost of arrival from the starting square, through one of the least costly squares and if we subtract the cost of this square from the cost of arrival, what remains must be the cost of arriving at the previous square. We will always reach the starting square without passing through some unreachable square since $W[i, j] - A[i, j] \neq \infty$ unless $W[i, j] = \infty$. Since we are looking at the path backwards we can save the path in a stack to print them in the right order. □

Solution to Combine coins and bills

Let $N[b, j]$ be the number of ways to create the sum b using coins and bills of values $v_1, \ldots, v_j$ dollars.

N can be defined recursively this way:

$$N[b,j] = \begin{cases} 1 & \text{if } b = 0 \text{ or } (j = 1) \wedge (b \mod v_1 = 0), \\ 0 & \text{if } j = 1 \text{ and } b \mod v_1 \neq 0, \\ \sum_{i=0}^{\lfloor b/v_j \rfloor} N[b - i \cdot v_j, j-1] & \text{if } 0 < b \leq n \end{cases}$$

Another way to express the recursion is that a combination of the coin and bill values $v_1, \ldots, v_j$ either contains the value v_j or just the values $v_1, \ldots, v_{j-1}$:

$$N[b,j] = \begin{cases} 1 & \text{if } b = 0, \\ 0 & \text{if } j = 0 \text{ and } b > 0, \\ N[b, j-1] & \text{if } 0 < b < v_j \text{ and } j > 0 \\ N[b - v_j, j] + N[b, j-1] & \text{if } v_j < b \leq n \text{ and } j > 0 \end{cases}$$

□

Solution to longest common substring
For every pair of characters from each string, let $M[i, j]$ be the number of matching letters to the left of (and including) a_i which corresponds to as many letters to the left of and including b_j. The length of the longest common substring will be the largest number in the matrix M.

M can be defined recursively as:

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ M[i-1, j-1] + 1 & \text{if } a_i = b_j, \\ 0 & \text{otherwise.} \end{cases}$$

Here we can see that the recursion we have arrived at is the same as the one we were looking at last exercise. Then we came up with the following algorithm:

for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i, 0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    if $a_i = b_j$ **then**
      $M[i, j] \leftarrow M[i-1, j-1] + 1$
    else $M[i, j] \leftarrow 0$
return M

To answer the question in this assignment we need to find the largest number in M. We could first calculate M and go through the matrix to compare values, but it is better to compare the values during the calculation of the matrix. That way we do not have to return the whole matrix. The following algorithm calculates all of M and returns the largest value in M.

$max \leftarrow 0$
for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i, 0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    if $a_i = b_j$ **then**
      $M[i, j] \leftarrow M[i-1, j-1] + 1$
      if $M[i, j] > max$ **then** $max \leftarrow M[i, j]$
    else $M[i, j] \leftarrow 0$
return max

The running time is still dominated by the nested for loop and is therefore $\Theta(nm)$. □

Solution to Protein folding
Let $q_{a,b}$ be the maximum value of the goal function we can get so that a folding of the part $p[a..n]$ of the protein, where the first substring in the fold has endpoints a and b. We can express $q_{a,b}$ recursively in the following way:

$$q_{a,b} = \max_{b+1<c\leq n} (\texttt{profit}(a, b, c) + q_{b+1,c}).$$

The base cases are: $q_{a,n} = 0$ for $1 \leq a < n$. The answer is then found as $\max_{1<b\leq n} q_{1,b}$.
Now we need to calculate $q_{a,b}$ according to these formulae and in the right order:

```
for a←1 to n-1 do q[a,n]←0;
for b←n-1 downto 2 do
```

```
  for a←1 to b-1 do
    t←-1;
    for c←b+2 to n do
      v←profit(a,b,c)+q[b+1,c];
      if v>t then t←v;
    q[a,b]←t;
max←0;
for b←2 to n do
  if q[1,b]>max then max←q[1,b];
return max;
```

We have at most 3 nested `for` loops and one call to `profit` which takes time $O(n)$ our time complexity is $O(n^4)$. □

Solution to Analysis tool for context free grammar

We use dynamic programming in approximately the same way as when we are looking for an optimal matrix chain multiplication order. Here we are instead going to determine in which order and on which substring the rules are to be applied.

Input is a set of rules R and a vector `w[1..n]` which is indexed from 1 to n. Let us construct a matrix `M[1..n,1..n]` where the element `M[i,j]` indicates the non terminal symbols from which we can use grammar rules to derive the substring `w[i..j]`.

Recursive definition of `M[i,j]`:

$$\texttt{M}[i,j] = \begin{cases} \{X : (X \to \texttt{w}[i]) \in R\} & \text{om } i = j \\ \{X : (X \to AB) \in R \wedge \exists k : A \in \texttt{M}[i,k-1] \wedge B \in \texttt{M}[k,j]\} & \text{if } i < j \end{cases}$$

Since every position in the matrix is a set of non terminal symbols we have to choose a suitable data structure for this. Let us represent a set of non terminal symbols as a bit vector indexed with non terminal symbols. 1 means that the symbol is in the set and 0 that it is not in the set. Example: If `M[i,j][`B`]`$= 1$ then the non terminal symbol B is in the set `M[i,j]`, which means that there is a chain of substitution rules from B to the substring `w[i..j]`.

The algorithm that calculates the matrix `M[i,j]` and returns true if the string is part of the language generated by the grammar:

```
for i←1 to n do
  M[i,i]←0; /* all bits are set to zero */
  for every rule X →w[i] do
    M[i,i][X]←1;
for len←2 to n do
  for i←1 to n-len+1 do
    j←i+len-1;
    M[i,j]←0;
    for k←i+1 to j do
      for every rule X → AB do
        if M[i,k-1][A]=1 and M[k,j][B]=1 then
          M[i,j][X]←1;
return M[1,n][S]= 1;
```

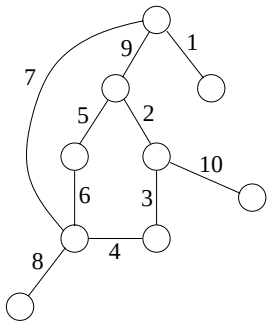
Time: $O(n^3m)$. Memory: $O(n^2m)$ (since m is an upper bound for the number of non terminal symbols). □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 5

Graph algorithms and lower bounds

Spanning trees Show how a minimal spanning tree can be found in the following graph using Prim and Kruskal's algorithms.



Altered flow

a) Describe an efficient algorithm which finds a new maximum flow if the capacity along a particular edge is *increased* one unit.

   The algorithms complexity should be linear: $O(|V| + |E|)$.

b) Describe an efficient algorithm which finds a new maximal flow if the capacity along a particular edge is *decreased* with one unit.

   The algorithms complexity should be linear: $O(|V| + |E|)$.

Euler cycle Take an undirected graph $G = \langle V, E \rangle$ where all vertices have even degree, find an euler cycle, a closed path that passes every edge in E exactly once. The algorithm must have linear time complexity.

Christmas present distribution A father is going to give his n children one Christmas present each. Every child has written a wish list. The father wants to give every child a present from their wish list but he does not want to give the same present to more than one child.

Construct and analyze an efficient algorithm for this problem. You may assume that there are at most m items on each wish list.

Upper bound for searching in sorted array As you know, a binary search in a sorted array with n numbers takes $O(\log n)$. Prove that $\Omega(\log n)$ is a lower bound for the worst case of this problem.

Fix broken Christmas lighting Jonas's Christmas lighting with n serially connected lamps does not illuminate. A single broken lamp is enough to make the system not illuminate. Jonas has bought n new light bulbs that are guaranteed to work. He does not want to replace any whole lamps with new, because that would be a waste of lamps. He naturally wants to make as few lamp tests as possible.

Construct and analyze an efficient algorithm for how Jonas can get the light to work with as few lamp switches as possible. Analyze the worst case complexity for the algorithm in detail (not just Ordo). A lamp switch takes time 1. The faster algorithm the better.

Find a lower bound for the worst case complexity that matches your algorithm.

Solutions and hints

Solution to Spanning trees – your own exercise!

Solution to Altered flow

a) Assume that the edge from the vertex u to v gets its capacity increased with one. Since the time complexity should be linear we can not calculate whole new solution. We need to use the solution to the previous flow problem. Let Φ be this solution (What does Φ consist of?) and just perform a new iteration of Ford Fulkerson's algorithm with the altered graph: In the rest flow graph the capacity in the edge (u, v) is increased with one. Perform a graph search (in time $O(|V| + |E|)$) to see if there now is a path in the residual flow graph where the flow can increase. If there is such a path, the flow has to be of size 1 (since all flows are integers). If there is no flow in the residual flow graph then Φ is still the maximum flow.

b) Assume that the edge from u to v gets its capacity decreased with one. If the previous maximum flow Φ did not utilize the full capacity of (u, v) then the flow is not altered at all. Otherwise we have to update the flow in the following way:

Since a larger flow arrives to u than the flow that leaves it and a smaller flow arrives at v than the flow that leaves it we have to find a way to lead one unit of flow from u to v some other way. Search in the residual flow graph for a path from u to v that can carry a flow of one. This can be done with a graph search in time $O(|V| + |E|)$. If there is such a path we update Φ with that flow.

If there is no such path we have to decrease the flow from s to u and from v to t with one unit. We do this by finding a path from u to s in the residual flow graph along which the flow can be increased with one and a path from t to v in the residual flow graph along which the flow can increase with one. (There has to be such a path since we had a flow from s to t via (u, v).) Update Φ with these two flows.

□

Solution to Euler cycle
The general idea is to start in a vertex v and search through the graph until v has been reached. The search path P will then either have visited all edges there remains, if we remove P one or more connected components $G_1, G_2, \ldots, G_n$. In the latter case we can redo the same sort of search for every component to get and find an Euler cycle for every G_i and then add all these Euler cycles to P to get a total Euler cycle.

Our linear time algorithm uses two players P_1 and P_2 who walk through the graph together. Both players start in v. Player P_1 creates the path P by walking along edges randomly. He marks every visited edge as visited and stops when he returns to v again. P_2 follows along the path of P, but for every new vertex he checks if every edge from u are visited. If they are all visited he continues along P. If there are unvisited edges (there is always an even number) he sends out $P1$ to find a new path that begins and ends in u. P_2 then walks this new path before continuing from u. After repetitions P_2 will finally have walked along every path and created an Euler cycle. P_1 has not walked along the same edge more than once either, so the total runtime is linear.

In the algorithm below we call P_1 $PathFinder$ and $P_2 Straggler$

```
EulerCycle(G) =
    cycle ← {1}
    choose edge (1, t)
    mark (1, t)
    path ← PathFinder(G, 1, t)
    Straggler(path)
    return cycle

PathFinder(G, start, cur) =
    append(cur, path)
    while cur ≠ start do
        choose unmarked edge (cur, v)
        mark (cur, v)
        append(v, path)
        cur ← v
    return path

Straggler(path) =
    while path ≠ ∅ do
        u ← next(path)
        append(u, cycle)
        for all edges (u, v) do
            if unmarked (u, v) then
                mark (u, v)
                p ← PathFinder(G, u, v)
                Straggler(p)
```

□

Solution to Christmas present division

The problem is a bipartite matching problem. The object is to find a matching in a graph where the left vertices are children (n vertices) and the right vertices are items. There is an edge between a child and an item if the item is on the child's wishing list. Bipartite matching can be solved as a matching problem in a graph with $O(nm)$ edges. Since the flow increases with one every in every turn of Ford Fulkerson's algorithm and the matching is at most n the matching is at most n the complexity is $O(n^2 m)$. □

Hint to Lower bound for searching in sorted array

Construct a decision tree for an arbitrary algorithm which searches in a sorted array. Examine how many leaves the tree must have and how high it needs to be.

Solution to Repair Christmas lighting
Number the light bulb positions from 1 to n.

switch all bulbs except the one in place 1 against new ones
put the old bulbs in a sac
$i \leftarrow 1$
while (there are still bulbs in the sac) **do**
    if (the lights are on) **then**
        $i \leftarrow i + 1$
        unscrew the new bulb in position i
    else
        unscrew the old broken bulb in position i
    take a bulb from the sac and put it in position i
if not (the lights are on) **then**
    unscrew the old broken bulb in position i
    put a new bulb in position i

It is easy to see that the algorithm is correct with the following invariant for the for loop. (the invariant is true in the beginning of every turn of the loop): All bulbs in position $< i$ are old and work, the lamp in position i is old, all lamps in position $> i$ are new.

The worst case is that every bulb in the lighting are broken and then the algorithm performs $(n-1)+(n-1)+1 = 2n-1$ lamp switches.

$2n-1$ is a lower bound. Think about it this way: The only way to determine if a particular bulb in the lighting is whole or broken is to ensure that all other bulbs are whole and see if the lights are on. A tough opponent makes sure that the Christmas lighting still does not turn on after $n-1$ new bulbs have been switched by simply making the first bulb broken. All $n-1$ removed original bulbs must then be tested again because the tough opponent has not allowed any information about these bulbs to slip out and that requires another set of $n-1$ lamp switches. Finally the opponent makes sure that the last tested original lamp is broken and then yet another lamp switch is needed to fix it.

The tough opponent just needs two broken bulbs to make the algorithm require $2n-1$ switches!
□

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 6

Algorithm construction

On this exercise session there is also a **hand-in of written solutions for the theory exercises for lab 3** and oral presentation of the theory assignments.

Inside or outside? Let P be a convex polygon with n vertices, described as an array with the vertices $p_1, p_2, \ldots p_n$ in cyclic order. Construct an algorithm which determines whether q is inside P. The algorithm should take time $O(\log n)$ in the worst case.

Sorting small integers Construct an algorithm which sorts n integers in the range $[1..n^3]$ in time $O(n)$ with unit cost.
Hint: Think about bucket sort and radix sort.

Find the missing number On a datafile there are 999 999 999 integers, every integer between 1 and 1 000 000 000 except one. Which number is missing? Construct an algorithm that solves this problem in constant memory and linear time in a calculation model with 32 bit arithmetic (meaning it can calculate with numbers less than $2^{31} = 2147483648$)

Complex multiplication If we multiply two complex numbers $a + bi$ with $c + di$ in the normal way we get four multiplications and two additions of real numbers. Since multiplications are more expensive than additions (and subtractions) it would be beneficial to decrease the number of multiplications. We may be dealing with large numbers here. Find an algorithm which only uses three multiplications (but more additions) to multiply two complex numbers.

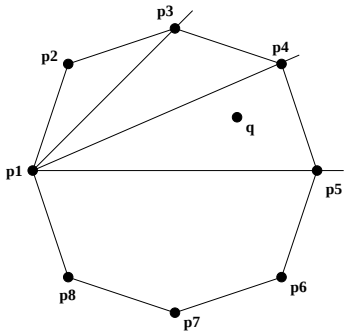
Two dimensional Fourier transformation Suppose you have access to an FFT implementation for the one dimensional case but you need to transform two dimensional data, for example to transform an image represented as a matrix $(a_{ij})_{i,j\in\{0,\ldots,N-1\}}$ to a 2D frequency spectrum. How would you use FFT for this? What is the time complexity?

Definition of the FFT:

$$\hat{a}_{kl} = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_{ij} w^{ik+jl}$$

where w is a suitable root of unity ($w^N = 1$).

Binary tree with mirrored structure Two binary trees are said to have a *mirrored structure* if one is a mirror image of the other. In other words: if we change left for right everywhere in one tree then the trees are structurally equivalent. Construct and analyze the time complexity for an efficient divide and conquer algorithm which determines if two binary trees have mirrored structure.



Figur 1: A convex polygon and the lines used by the algorithm to divide it.

The party problem Input is a list of n persons, an integer k and a list of which persons know each other. You want to invite as many of the persons as possible to your party, but to ensure that all guests enjoy the party you want every invited guest to know at least k of the other guests. Construct and analyze an algorithm which solves this problem in linear time in the size of the input.

Solutions

Solution to Inside or outside?
If the polygon had not been convex we could have counted the number of intersections it has with a line from q to a point outside P, which takes time $O(n)$ but there is not time for that now. Instead we will use an interval halving like search to exclude half of the remaining polygon at a time until only one triangle remains. Then we can easily (with a constant number of comparisons) determine whether q is in P. See illustration 1

$InsideConvex(P, q, l, u) =$
  if $u = l + 1$ **then** /* a triangle */
    choose a point q' outside the triangle p_1–p_l–p_u
    if the line q–q' intersects exactly one line in the triangle **then**
      return inside
    else
      return outside
  else
    $mid \leftarrow \lceil \frac{l+u}{2} \rceil$
    if q is in the same side of the line p_1–p_{mid} as p_{mid+1} **then**
      return $InsideConvex(P, q, mid, u)$
    else
      return $InsideConvex(P, q, l, mid)$

The algorithm is called with $InsideConvex(P, q, 2, n)$.

If we assume that $InsideConvex(P, q, 2, n)$ takes time $T(n)$ then we get the recursive equation

$$T(n) = T\left(\frac{n}{2}\right) + c$$

which has the solution $c \log n$. The time complexity is therefore $T(n) \in O(\log n)$. □

Solution to Sorting small integers
Consider the integers in base n. All numbers will then have at most 3 digits, except n^3 which is written as 1000 and can be handled separately. (Go through all the numbers, remove all occurrences of n^3 and place them last in the sorted sequence.) Now add leading zeros on numbers that are smaller than n^2. Perform radix sort on the numbers. Since there are 3 digit numbers, we get 3 turns of the radix sort, where very turn sorts the numbers after a digit in the interval $[0..n-1]$. That sort of sorting is done in linear time with counting sort (which is stable if implemented like the lecture notes from lecture 16).

Analysis: Three turns are performed and every turn takes time $O(n)$. The special handling of n^3 does not take more time than $O(n)$ either. The total time will therefore be $O(n)$. □

Hint to Find the missing number
Sum all numbers modulo for example 1 000 000 000.

Solution to Complex multiplication
Your own exercise. □

Solution to 2D Fourier transform
Note first that the sum w^{ik} can be moved out of the inner sum. If we now regard the inner sum as a 1D Fourier transform $\hat{b}_{il}$ then $\hat{a}_{kl}$ is a 1D transformation of $\hat{b}_{il}$.

$$\hat{b}_{il} = \frac{1}{N}\sum_{j=0}^{N-1} a_{ij}w^{jl}, \qquad \hat{a}_{kl} = \frac{1}{N}\sum_{i=0}^{N-1} \hat{b}_{il}w^{ik}$$

We have used the fact that the Fourier transform is separable. The above equations tell us that we can Fourier transform 2D data by first transforming every column in the matrix (a_{ij}) and then transform the rows in the resulting matrix (b_{il}) (or the other way around if you so wish) The algorithm is:

2D-FFT$(a_{0,0}, a_{0,1}, \ldots, a_{N-1,N-1}, N) =$
    for $i \leftarrow 0$ **to** $N-1$
        $b_{i,0}, \ldots, b_{i,N-1} \leftarrow FFT(a_{i,0}, a_{i,1}, \ldots, a_{i,N-1}, N)$
    for $i \leftarrow 0$ **to** $N-1$
        $c_{0,i}, \ldots, c_{N-1,i} \leftarrow FFT(b_{0,i}, b_{1,i}, \ldots, b_{N-1,i}, N)$
    return $c_{0,0}, c_{0,1}, \ldots, c_{N-1,N-1}$

FFT is called $2N$ times. Since every call takes $O(N \log N)$ the total time becomes $O(N^2 \log N)$.
□

Solution to Binary tree with mirrored structure
We use divide and conquer to step down both trees at the same time. In order for the trees to be each other's mirror images the left subtree of the first tree has to be the mirror image of the right subtree of the other tree and the right subtree of the first tree must be the mirror image of the left subtree of the other tree. The base case is that at least one of the trees are empty. In that case the trees are mirror images of each other if both trees are empty.

MirroredTrees(T_1, T_2)=
    if T_1 =**NIL or** T_2 =**NIL then**
        return T_1 =**NIL and** T_2 =**NIL**
    return MirroredTrees(T_1**.left,**T_2**.right) and MirroredTrees**(T_1**.right,**T_2**.left)**

Since the function will be called (at most) once for every subtree (root) and the cost of a function call apart from the recursion is constant the complexity will be linear in the size of the tree.

It is easy to see that the algorithm is correct with the help of so called structural induction, induction over the tree structure. We first check that the base case is correct (if one tree is empty the other tree must also be empty for them to be mirror images) and then check that the induction step is correct (if the first tree's left subtree is the mirror image of the other tree's right subtree and the the first tree's right subtree is the mirror image of the other tree's left subtree then the trees must be each other's mirror images). □

Solution to The party problem

Assume that the list of guests who know each other consist of m rows. It is easy to represent input as a graph $G = (V, E)$ where each person is a vertex and every edge (x, y) means that two people x and y know each other. Use an edge list to represent the graph.

The sought solution is the largest (induced) subgraph where every vertex has degree at least k. We can find this subgraph by removing every vertex with degree less than k. When a vertex is removed we also remove all edges to that vertex, which is why other vertices get lower degree and may need to be remove. This is easiest to implement with a variable d_x in every vertex x which keeps track of the degree. It is updated every time an edge to the vertex is removed. Let us also have a queue Q where we put all vertices with degree less than k which are awaiting removal of their edges.

```
foreach x ∈ V do
   if d_x < k then Q.Put(x)
while not Q.Empty() do
    x ← Q.Get()
    foreach (x, v) ∈ x.edge list do
       if d_v ≥ k then
          d_v ← d_v − 1
          if d_v < k then Q.Put(v)
write 'The solution consists of:'
foreach x ∈ V do
   if d_x ≥ k then write x
```

Note first that as soon as d_v gets smaller than k the vertex v is put in the queue and once it has been pushed there d_v will not change again (because of the if statement). Since there are n vertices and every vertex is treated a constant number of times (initialization of Q, push in queue, removal from queue, printing), and since there are m edges and every edge is handled at most twice (it occurs in two edge lists) the total complexity becomes $O(n + m)$ which is linear in the size of the input.

It is easy to see that the algorithm is correct because every vertex printed in the solution still has at least k neighbours left in the graph, and no vertex with less than k neighbours has been removed during the run of the algorithm. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 7

Probabilistic algorithms, reductions

Skip lists The probability p that an element at level i in a skip list is also at level $i+1$ can be varied after need. On page 5 in the skip list article there is an analysis of how the search paths expected length and the expected number of pointers in an element depend on p. It is concluded that the shortest search path is found if $p = 1/e \approx 0.3679$. In spite of this it may be better to choose $p = 1/2$ or $p = 1/4$. Why is that?

Verification of matrix multiplication The fastest known method for matrix multiplication of two $n \times n$ matrices is $O(n^{2,376})$. Show that it is possible to probabilistically verify a matrix multiplication in $O(n^2)$. To be more precise, construct a Monte Carlo algorithm that in time $O(kn^2)$ verifies that $AB = C$ and has an error probability of at most 2^{-k}.

Reduction with a negative result To find the median element (the $\lceil n/2 \rceil$th smallest element) of n positive integers requires at least $2n$ comparisons in the worst case. Use this result to determine a lower bound for the number of comparisons required to find the $\lceil n/2 \rceil + 10$th smallest element.

Reduction which gives a positive result A useful way of solving a problem is to find a reduction to a problem which we already know how to solve. Use this method to solve the disconnected graph problem defined like this:

> INPUT: A connected undirected graph $G = (V, E)$ and a positive integer K between 1 and $|V|$.
> PROBLEM: Is it enough to remove K well chosen edges from the graph G to make it disconnected?

Express an optimization problem as a decision problem The slacking problem is defined as an optimization problem in the following way:

Input: The length of the work day T and n tasks. Tark i costs time t_i and effort w_i. All numbers are positive integers.

Solution: A subset of the tasks $D \subseteq [1..n]$ which together fill up a work day. In other words $\sum_{i \in D} t_i \geq T$.

Goal function: The effort required to perform a day's work D, in other words $\sum_{i \in D} w_i$.

Goal: Minimize the goal function. Find the daily work that minimizes the effort.

a) Express the optimization problem as a decision problem. Expand the input with a goal value I and formulate a yes or no question.

b) Make a turing reduction from the optimization problem to the decision problem. In other words show how we can solve the optimization problem with the help of an algorithm to solve the decision problem.

Reductions between decision, optimization and construction problems Assume that the algorithm GraphColouring(G,k) in time $T(n)$ (where n are the number of vertices in G) answers 1 iff the vertices in G can be colored with k colors so that no edge connects two vertices of the same color.

a) Construct an algorithm which given a graph G with n vertices determines the minimum number of colors needed to color G. The time should be $O(\log n \cdot T(n))$.

b) Construct an algorithm which given a graph G with n vertices colors the vertices with the minimum number of colors in time $O(P(n)T(n))$ where $P(n)$ is a polynomial.

Solutions

Solution to Skip lists
We want to be able to calculate RandomLevel quickly, where RandomLevel is the level where the element is to be put. If the randomness can be expressed on the form $p = 2^{-k}$ this can be done efficiently by reading k bits of a random string (a stream of bits) and (for example) check if all k bits are 1. □

Solution to Verification of matrix multiplication
Idea: Randomize a n vector $\mathbf{r} \in \{0,1\}^n$, calculate $\mathbf{a} = C \circ \mathbf{r}$ and $\mathbf{b} = A \circ (B \circ \mathbf{r})$ and verify that $\mathbf{a} = \mathbf{b}$.

If $AB = C$ then $\mathbf{a} = \mathbf{b}$ will always hold. If $AB \neq C$ then $D = AB - C \neq 0$. Then there must be some index i in $\mathbf{r}$ which affects the value of $D\mathbf{r}$ if you switch $\mathbf{r}[i]$ so that it is 1 if it was 0 and 0 if it was 1. Also, if the algorithm randomizes the elements in the vector $\mathbf{r}$ with probability $1/2$ for 0 and $1/2$ for 1 then $AB\mathbf{r} \neq C\mathbf{r}$ will hold with probability $1/2$. If we repeat this k times the probability that the algorithm is fooled all k times will be less than $(1/2)^k$.

The algorithm is:

```
VerifyMult(n, A, B, C, k) =
        for i ← 1  to k do
            r ← randomVector∈ {0,1}^n
            a ← C ∘ r
            b ← A ∘ (B ∘ r)
            if a ≠ b then return false
        return true
```

The time complexity is $O(kn^2)$ since the algorithm runs k turns and for every turn there are three matrix/vector multiplications which take $O(n^2)$. □

Solution to Reduction with a negative result
Reduce the median problem to the new problem by adding so many ones to the input so that the earlier median element now is the $\lceil m/2 \rceil + 10$th smallest element, where m is the number of elements after the ones have been added. That means we have to add 20 ones and $m = n + 20$. The lower bound $2n$ expressed in m is: $2n = 2m - 40$. □

Solution to Reduction to prove a positive result
First we need to understand the problem. Assume that X is a minimum set of edges that make G disconnected if we remove them. (This should be interpreted such that no strict subset of X fulfills

the same.) Then G is divided into two components and all edges in X connect those components (otherwise X would not be minimal). X corresponds to a cut in the graph (a division of the vertex set in two parts). The number of edges in X is the size of the cut. This means that the lowest number of edges that we need to remove to make G disconnected — call this $\lambda(G)$ — is equal to the size of a minimum cut (V_1, V_2) in G.

Minimum cut equals maximum flow. We should therefore try to reduce the disconnected graph problem to maximum flow between two vertices s and t in a graph. If we expand G to a flow graph G' by giving every edge a reverse edge and capacity 1 in both directions. We can then calculate $\lambda(G)$ by calculating the maximum flow from s and t for different s and t. We don't need to vary both of these. If we choose s arbitrarily then it has to belong to one of the sets V_1 and V_2 above. By varying t over all vertices except s we are guaranteed to eventually get some vertex in the other subset. The conclusion is that for an arbitrary vertex s:

$$\lambda(G) = \min_{t \in V - \{s\}} \{\text{MaxFlow}(G', s, t)\}.$$

The exercise was to calculate whether $K \geq \lambda(G)$. We can answer NO if $K < \text{MaxFlow}(G', s, t)$ for all $t \in V - \{s\}$ and YES otherwise.

The max flow algorithm is called $|V| - 1$ times. Every flow calculation can be done in time $O(|V|^3)$ (There is no need to memorize that.) The complexity of our algorithm is therefore $O(|V|^4)$. □

Solution to Express an optimization problem as a decision problem

a) **Input:** Length of the working day T, the goal I and n tasks. Task i takes time t_i and costs the effort w_i.

Question: Is there a subset $D \subseteq [1..n]$ so that $\sum_{i \in D} t_i \geq T$ and $\sum_{i \in D} w_i \leq I$?

b) Assume that Slacking$(T, I, n, \{t_i\}, \{w_i\})$ is an algorithm that solves the decision problem. The optimal total effort has to be an integer between 0 and $\sum_{1 \leq i \leq n} w_i$. Perform a binary search between these extreme values after the lowest value I for which Slacking$(T, I, n, \{t_i\}, \{w_i\})$ answers yes.

□

Solution to Reductions between decision, optimization and construction problems

a) We know that the number of colors are between 1 and n. Use binary search in this interval and call the decision problem GraphColoring to find a k so that GraphColoring$(G, k) = 1$ and GraphColoring$(G, k-1) = 0$. This means that the minimal coloring has k colors. We need $\log n$ halvings of n to get down to 1. The time complexity is therefore $O(\log n \cdot T(n))$.

b) Find the minimum number of colors k with the method above. We want to color the vertices in G with colors numbered from 1 to k. The following algorithm does that:

$CreateColouring(G = (V, E),\ k)$=
$u \leftarrow$ first vertex in V
$C \leftarrow \{u\}; u.colour \leftarrow k$
foreach $v \in V - \{u\}$ **do**
  if $(u, v) \notin E$ **then**
    if GraphColoring$((V, E \cup \{(u, v)\}), k) = 1$ **then** $E \leftarrow E \cup \{(u, v)\}$
    else $C \leftarrow C \cup \{v\}; v.colour \leftarrow k$
if $k > 1$ **then** $CreateColouring((V - C, E), k - 1)$

GraphColoring is called once for every pair of vertices in the graph. The time complexity for the whole algorithm is therefore $O(\log n \cdot T(n) + n^2 \cdot T(n)) = O(n^2 \cdot T(n))$. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 8

Undecidability

One of the two hours this exercise set are spent on Mästarprov 1.

Undecidability 1 The *worm problem* takes as input a set T of tile types and two points p_1 and p_2 in the plane. The question is whether or not we can use the tile types in T to build a worm that coils from p_1 to p_2. The worm may not break the tile pattern anywhere and it must stick in the upper half plane.

Are the following versions of the worm problem decidable or undecidable?

a) Input is expanded with a fourth parameter, a tile type $t \in T$. The first tile (the one we place on p_1) must be of type t.

b) Input is expanded with an integer N and the question is expanded with the condition that the worm must use at most N tiles.

c) Input is expanded with an integer N and the question expanded with the condition that the tile must consist of at least N tiles.

Undecidability 2 Is there a program P such that given y it is decidable whether P stops for input y?

Undecidability 3 Is there an explicit program P such that given y it is undecidable whether P stops for input y?

Undecidability 4 If the program x stops on empty input we let $f(x)$ be the number of steps before the program stops. Otherwise we set $f(x) = 0$. Now define $MR(y)$, the maximum runtime of all programs whose binary coding are less than y.

$$MR(y) = \max_{x \leq y} f(x),$$

Is MR computable?

Undecidability 5 Show that the function MR in the previous exercise grows faster than every recursive function. In particular, show that for every recursive function g there is a y such that $MR(y) > g(y)$.

Undecidability 6 Assume that a Turing Machine M, with input X (written on the tape at the start) and an integer constant K are given. Is the following problem decidable or undecidable?

Does M stop on input X after having used at most K squares on the tape. (a used square may be written to and read from more than once)?

Undecidability 7 A *recursively enumerable set* was defined at the lecture as a language which can be recognized by a function whose yes solutions can be verified in finite time. An alternative definition is a set which is countable (the elements can be numbered with natural numbers) and can be produced by an algorithm. Use the latter definition to prove that every recursive set is recursively enumerable.

Undecidability 8 The *diagonalized halting set* is the set of all programs p that reach the halting state on input p. Show that this set is recursively enumerable.

Solutions

Solution to Undecidability 1

a) Undecidable. We reduce the *worm problem* to *wormProblemWithSpecialStartTile* in the following way.

$$
\begin{aligned}
&\textit{worm problem}(T, p_1, p_2) = \\
&\qquad \textbf{for } t \in T \textbf{ do} \\
&\qquad\quad \textbf{if } \textit{wormProblemWithSpecialStartTile}(T, p_1, p_2, t) \textbf{ then} \\
&\qquad\qquad \textbf{return true} \\
&\qquad \textbf{return false}
\end{aligned}
$$

b) Decidable. We just need to test all imaginable worms of length at most N that start in p_1 and see if they solve the problem. Since this is a finite set (bounded by $(3|T|)^N$) the algorithm is finite and the problem decidable.

c) Undecidable. We reduce the *wormProblem* to *wormProblemWithMinimumLength* in the following way.

$$
\begin{aligned}
&\textit{wormProblem}(T, p_1, p_2) = \\
&\qquad \textbf{return } \textit{wormProblemWithMinimumLength}(T, p_1, p_2, 1)
\end{aligned}
$$

□

Solution to Undecidability 2
Yes, there are plenty of such programs. Take for example the simple program $P(y) = \textbf{return}$. It is easy to see that P stops for all inputs. □

Solution to Undecidability 3
Yes. Let for example P be an interpreter and input y be a program x followed by input y' to that program. This means that $P(y)$ behaves just like x would behave on input y', and it is undecidable whether x stops on a certain input y'. □

Solution to Undecidability 4
No, *MR* can not be calculated. We know that it is undecidable whether a program stops on blank input (the empty string). Therefore, reduce this program to *MR*. Note that $MR(y)$ is the maximum number of steps that every program of size at most y needs before it stops if it is started with blank input.

StopBlank(y) =
    $s \leftarrow MR(y)$
    if $s = 0$ **then return false**
    else
        simulate y on blank input in s steps (or until y stops)
        if y stopped **then return true**
        else return false

If y has not stopped within s steps we know that it will never stop. Since StopBlank is not computable MR can not be computable either. □

Solution to Undecidability 5
Assume the opposite, that there is a recursive function g so that $g(y) \geq MR(y)$ for all y. Then we could change the first row $s \leftarrow MR(y)$ in the solution to the previous exercise to $s \leftarrow g(y)$ and the program would still work. The program would then become computable which we know it can not be. Therefore we have a contradiction and our assumption was false. □

Solution to Undecidability 6
The problem is decidable. Since the Turing machine has to stay within K squares on the tape the number of configurations the Turing machine can be in is finite. If the machine has m states then the number of configurations is $m \cdot K \cdot 3^K$ (the number of states multiplied with all positions for the read/write head multiplied by all possible tape configurations). Simulate the machine in $m \cdot K \cdot 3^K + 1$ steps and check for every step that the machine does not move outside of the K squares on the tape. If the Turing machine stops within this time we answer *yes*. If the Turing machine has not stopped after this time it must have returned to one of its previous configurations, meaning that it is in an infinite loop and we can safely answer *no*. □

Solution to Undecidability 7
Assume that S is a recursive set. This means that there is an algorithm $A(x)$ which determines whether $x \in S$. The elements in S are naturally stored as binary strings. The following program generates the set S:

for $i \leftarrow 0$ **to** ∞ **do**
    if $A(i)$ **then write** i

□

Solution to Undecidability 8

for $i \leftarrow 0$ **to** ∞ **do**
    for $p \leftarrow 0$ **to** i **do**
        simulate the calculation $p(p)$ during i steps
        if $p(p)$ stops within i steps **then write** p

The same p will be written many times. If we want to avoid this we can make sure to only print p if either $p(p)$ stops at exactly i steps or $p = i$. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 9

NP completeness proofs

This time there is also a **hand-in of written solutions for the theory exercises for lab 4** and oral presentation of the assignments.

What do the reductions say? A, B, C, D and E are decision problems. Assume that B is NP complete and that there are polynomial Karp reductions between the problems as illustrated below (a reduction from A to B is denoted $A \rightarrow B$):

$$\begin{array}{ccccccc} \text{A} & \rightarrow & \text{B} & \leftrightarrow & \text{C} & \leftarrow & \text{D} \\ & & \downarrow & & & & \\ & & \text{E} & & & & \end{array}$$

What do we know about the complexity for A, C, D and E? Write a cross in the table below for when it is certain and a ring where it is possible but we can't know for certain.

	is in NP	is NP complete	is NP hard
A			
C			
D			
E			

Frequency allocation Within the cell phone carrier industry one of the problems to solve is the *frequency allocation problem* which works like this. There are a number of transmitters. Every transmitter can broadcast on a given set of frequencies. Some towers are so close to each other that they can not broadcast on the same frequency because then they would interfere with each other. (This is not just because of proximity but also because of the presence of mountains, houses or other things that block radio signals.)
We know the transmitters, each transmitter's frequency set and which pairs of transmitters would interfere with each other if they broadcasted on the same frequency. The problem is to determine whether there is a possible choice of frequencies so that no transmitter interferes with another.

a) Express this problem as a graph problem
b) Show that the problem is in NP by:
b1) Suggesting how a solution may look.
b2) Show that if the answer is yes, then the solution can be verified.
b3) Show that the verification takes polynomial time.
c) Show that the problem is NP hard by:
c1) Find a known NP complete problem to reduce.
c2) Describe the reduction between the known problem and the frequency allocation problem.
c3) Show that the reduction is polynomial.
c4) Show that the reduction is correct.

Then you will know that the problem is NP complete!

Hamiltonian path in graph Show that the problem Hamiltonian Path is NP complete. The problem is to determine whether there is a simple path which passes through all vertices in a graph.

Spanning tree with limited degree Show that the following problem is NP complete: Given an undirected graph $G = (V, E)$ and an integer k, determine whether G contains a spanning tree T so that every vertex in the tree has degree at most k.

Polynomial reduction Construct a polynomial reduction from 3CNF SAT to Eq-gf[2], the satisfiability problem for a system of polynomial equations over GF[2] (integers modulo 2).

Is an Euler graph k colorable? Many problems of the type *determine if the graph G has property e* can be simplified if we assume that the graph has another special property. We will now study a special case. We say that a connected graph is an *Euler graph* if every vertex in the graph has even degree. We would now like to determine whether such a graph is k colorable. We have the following problem:

> Input: An Euler graph G and an integer k.
> Output: **YES** if the graph is k colorable. **NO** otherwise.

For $k \leq 2$ there is a polynomial algorithm for determining colorability for general graphs. We can therefore assume that $k \geq 3$. Now determine if there is a polynomial algorithm that solves the problem above or if the problem is NP complete.

Solutions

Solution to What do the reductions say?

	is in NP	is NP complete	is NP hard
A	x	o	o
C	x	x	x
D	x	o	o
E	o	o	x

□

Solution to Frequency allocation

a) Express the frequency allocation problem as a graph problem.

Let the vertices represent transmitters and edges represent transmitters that interfere with one another. Every vertex v_i is marked with a frequency set F_i. The question is if it is possible to assign to every vertex a frequency from its set of frequencies so that no vertices with the same frequency have an edge between them.

b) Show that the graph problem is in NP.
b1) A solution is an assignment of frequencies to each vertex.
b2) Go through each vertex and verify that its frequency is in the set of frequencies. Also go through every edge and verify that the endpoints' frequencies are different.
b3) The verification takes linear time in the size of the graph.

c) Show that the graph problem is NP hard.
c1) We try to reduce the k coloring problem.
c2)

k coloring$(G, k) =$
for every vertex v_i in the graph G
$F_i \leftarrow \{1, \ldots, k\}$
return frequency allocation$(G, \{F_i\})$

c3) The reduction just creates a k set for each of the vertices. It is obviously polynomial.
c4) Now show that there is a k coloring of the graph G iff there is an allowed frequency assignment for G where all vertices have the frequency set $\{1, \ldots, k\}$.

Assume that we have a k coloring of G. Number the colors 1 to k. If a vertex has been given color i we let the corresponding vertex (transmitter) in the frequency allocation problem get the frequency i. This will be an allowed frequency assignment since we have based it on an allowed k coloring.

In the other direction: Assume that we have an allowed frequency assignment. We get a k coloring by coloring a vertex with the color i if the corresponding transmitter has been assigned frequency i.

□

Solution to Hamiltonian path in graph
A Hamiltonian path is a simple open path that contains each vertex in a graph exactly once. The HAMILTONIAN PATH problem is the problem to determine whether a given graph contains a Hamiltonian path.

To show that this problem is NP-complete we first need to show that it actually belongs to the class NP and then find a known NP-complete problem that can be reduced to HAMILTONIAN PATH.

For a given graph G we can solve HAMILTONIAN PATH by nondeterministically choosing edges from G that are to be included in the path. Then we traverse the path and make sure that we visit each vertex exactly once. This obviously can be done in polynomial time, and hence, the problem belongs to NP.

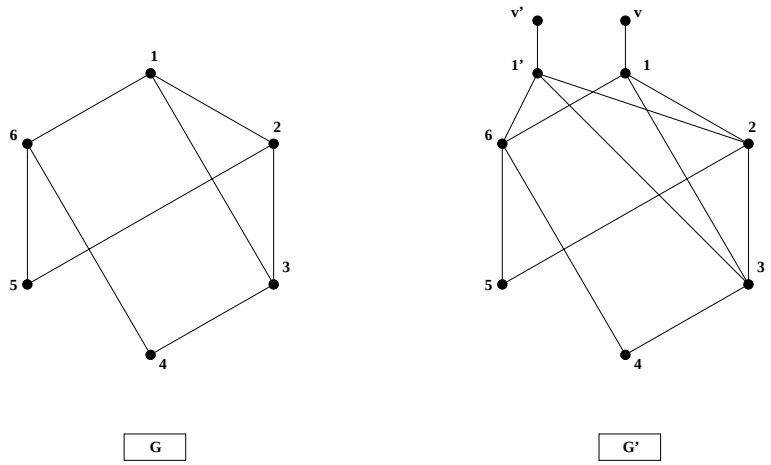
Now we have to find an NP-complete problem that can be reduced to HAMILTONIAN PATH. A closely related problem is the problem to determine whether a graph contains a Hamiltonian cycle, that is, a Hamiltonian path that begin and end in the same vertex. Moreover, we know that HAMILTONIAN CYCLE is NP-complete, so we may try to reduce this problem to HAMILTONIAN PATH.

Given a graph $G = \langle V, E \rangle$ we construct a graph G' such that G contains a Hamiltonian cycle if and only if G' contains a Hamiltonian path. This is done by choosing an arbitrary vertex u in G and adding a copy, u', of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' with u'; see Figure 2 for an example.

Suppose first that G contains a Hamiltonian cycle. Then we get a Hamiltonian path in G' if we start in v, follow the cycle that we got from G back to u' instead of u and finally end in v'. For example, consider the left graph, G, in Figure 2 which contains the Hamiltonian cycle $1, 2, 5, 6, 4, 3, 1$. In G' this corresponds to the path $v, 1, 2, 5, 6, 4, 3, 1', v'$.

Conversely, suppose G' contains a Hamiltonian path. In that case, the path must necessarily have endpoints in v and v'. This path can be transformed to a cycle in G. Namely, if we disregard v and v', the path must have endpoints in u and u' and if we remove u' we get a cycle in G if we close the path back to u instead of u'.

The construction won't work when G is a single edge, so this has to be taken care of as a special case. Hence, we have shown that G contains a Hamiltonian cycle if and only if G' contains a Hamiltonian path, which concludes the proof that HAMILTONIAN PATH is NP-complete. □



Figur 2: A graph G and the Hamiltonian path reduction's constructed graph G'.

Solution to Spanning tree with limited degree
In this solution we use the alternative (original) definition of NP as *nondeterministic polynomial time*. First note that the problem can be solved by the following nondeterministic algorithm:

1. For each edge in E, choose nondeterministically if it is to be included in T.
2. Check that T is a tree and that each vertex has degree less than k.

This means that the problem is in NP. Now we need to reduce a problem known to be NP-complete to our spanning tree problem. In this way we can state that determining whether a graph has a k-spanning tree is at least as hard as every other problem in NP.

Consider the problem Hamiltonian Path that was shown to be NP-complete in the previous exercise. Can we reduce this problem to the spanning tree problem? That is, can we solve Hamiltonian Path if we know how to solve the spanning tree problem? We claim that we can and that G has a Hamiltonian path if and only if it has a spanning tree with vertex degree ≤ 2.

It is easy to see that such a spanning tree is a Hamiltonian path. Since it has degree ≤ 2 it cannot branch and since it is spanning only two vertices can have degree < 2. So the spanning tree is a Hamiltonian path. If, on the other hand, G contains a Hamiltonian path, this path must be a spanning tree since the path visits every node and a path trivially is a tree.

We have reduced Hamiltonian Path to the spanning tree problem and, therefore, our problem is NP-complete. □

Solution to Polynomial reduction
Suppose we have a formula $\varphi \in$ 3Cnf-sat, for example

$$\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_3 \vee x_4)$$

Reducing 3Cnf-sat to Eq-gf[2] means that, for every formula φ, we can create an equation that is solvable if and only if φ is satisfiable. Now, φ is only satisfiable if every clause can be satisfied simultaneously, so let's start with finding an equation for an arbitrary clause $(x \vee y \vee z)$, where x, y and z are considered as literals rather than variables. For a start, the clause is satisfiable with one of the literals set to true, so in that case the equation $x + y + z = 1$ is also solved. But this fails if two literals are true! We can fix this by adding three more terms, where one and only one is 1 if two variables are 1. We get

$$x + y + z + xy + yz + xz = 1$$

Again, this is not complete. if all variables are set to 1, it fails. Of course, the solution is to add a term for this case, xyz. Our result is

$$x + y + z + xy + yz + xz + xyz = 1$$

If one of the literals in the clause happened to be inverted, e.g. we have $\overline{x}$ instead of x, replace x by $(1 + x)$ in the equation. The first clause in the example would be turned into

$$x_1 + x_2 + (1 + x_3) + x_1x_2 + x_1(1 + x_3) + x_2(1 + x_3) + x_1x_2(1 + x_3) = 1$$

We are now ready to create an equation from an arbitrary 3-CNF formula φ. For every clause φ_i in φ, create a corresponding equation Q_i. Our claim is that this system of equations is solvable if and only if φ is satisfiable and we must prove that this is correct.

To begin with, we note that $(1+x)$ is a proper way to handle inverses, $1+1 = 0$ and $1+0 = 1$, so in the following we will only consider literal values and not variables.

1. ($\Leftarrow$) If an equation in Q_i is satisfiable, then there exists an assignment to the variables such that each left hand expression is summed to 1. Hence, at least one of the literals is 1 and the corresponding literal in the boolean formula set true would make its clause satisfied. The equation system Q_i being satisfied means that each equation is satisfied and consequently is each clause in φ satisfied.

2. ($\Rightarrow$) When φ is satisfied, we have an assignment on the variables such that each clause is satisfied. There are three cases for the clauses, (i) one literal is true, (ii) two literals are true and (iii) three literals are true. By construction, the corresponding equations are all satisfied.

□

Solution to Is an Euler graph k colorable?
The problem is NP complete. It is easy to verify that if a solution is a correct coloring then the problem is in NP. To show completeness we reduce the general k coloring problem to our problem. We assume that $k \geq 3$. Assume that we have a graph G. We turn it into an Euler graph G' in the following way. There has to be an even number of vertices with odd degree. For each pair of odd degree vertices we add a new vertex. To this vertex we connect two new edges, one from each of the odd degree vertices. The graph G plus the new vertices becomes the graph G'. It is easy to see that G' is an Euler graph.

We now show that G is k colorable $\Leftrightarrow$ G' is k colorable. Assume that f is a coloring of G, i.e. every vertex x received a color $f(x)$. We then define a k coloring f' of G' in the following way: If x is a vertex in G' which also is in G then $f'(x) = f(x)$. If x is not in G then there are two neighbouring vertices y and z in G. We then set $f(x)$ to some other free color than $f(y)$ and $f(z)$. (It is possible to do that since $k \geq 3$.) The implication in the other problem is trivial. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 10

NP complete problems

Construct a knapsack solution The knapsack problem is a well known NP complete problem (see lecture 21). Input is a set P of items with weight w_i and value v_i, a knapsack size S and a goal K. The question in the knapsack problem is whether we can select items of total value at least K so that their total weight is at most S. All numbers in input are non-negative integers.

Assume that we have an algorithm A that solves the decision problem. Construct an algorithm which with the help of calls to A solves the constructive knapsack problem, i.e. with the same input as A both answers the knapsack problem and, if the answer is yes, returns the items that are to be packed into the knapsack. The algorithm may call A $O(|P|)$ times but other than that it may not take more than polynomial time.

We have to construct a Turing reduction of the constructive knapsack problem to the more common knapsack problem (a decision problem).

Reliability of the Internet Computers and connections between them may break down from time to time. Since we still want to be able to communicate through the rest of the Internet we would like there to be alternative paths between every pair of computer in the net. If there are three different paths through the Internet from computer A to computer B and these paths are disjoint (apart from the end points) then two computers, any two can disappear and A and B can still communicate.

In this problem we think of the Internet as an undirected graph where the vertices correspond to the computers on the Internet and every edge (X, Y) is a *possible direct connection* between the computers X and Y. For every pair of computers (X, Y) we also have a desired reliability $T(X, Y)$ which denotes how many disjoint paths we need between X and Y. Finally there is a budget B that denotes how many direct connections we can afford to construct.

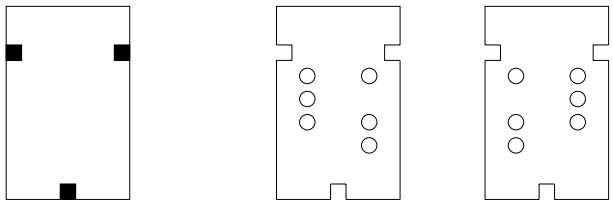
The question is: can we pick a subgraph with B edges so that for every pair of vertices (X, Y) there are at least $T(X, Y)$ disjoint paths in the subgraph between X and Y.

Show that this problem is NP complete!

Håstad's toy store Håstad's toy store sells a puzzle which consists of a box and a number of cards, see figure 3. Cards can be placed in the box with either the front side up or the back side up, since there are groves in the cards that have to match the strips in the box. On every card there are two columns of holes but some of the holes are filled. The goal of the puzzle is to place the cards in the box in such a way that the whole bottom is covered. For every hole position at least one of the cards has to have a filled hole.

Prove that the language TOYS $= \{\langle c_1, c_2, \ldots, c_n\rangle$: ($c_i$ describes cards) $\wedge$ (the puzzle is solvable)$\}$ is NP complete.

Processor scheduling We have n jobs to be executed and a sufficient amount of processors. Every job takes on unit of time to execute. There are also conditions on the form *job i can not be executed on the same time as j*. We call the problem to determine whether the jobs



Figur 3: Håstad's puzzle. The box with three strips, a card with holes and groves and the same card back side up.

can be scheduled to be executed in three time units SCHEDULE. Show that this problem is NP complete.

Is the problem still NP complete if we demand that the jobs should be executed after k time units where $k > 3$? Is it true for $k = 2$?

Solutions

Solution to Construct knapsack solution

```
Kappsäck(P,S,K)=
  if not A(P,S,K) then return "No solution"
  foreach p in P do
    if A(P-p,S,K) then P:=P-{p}
  return P
```

The algorithm calls A at most $|P|+1$ times. The returned set P is a solution because it fulfils the following two criteria.

- *P contains no superfluous elements* since the for loop runs over all elements of P and in every turn we check if the element p is superfluous. If it is then it is removed from P.
- *P contains a solution* since this is an invariant for the turn. If we reach the for loop this is fulfilled and after every turn of the loop it is fulfilled.

□

Hint to Reliability of the Internet
Reduce the problem *Hamiltonian circuit.*

Solution to Håstad's toy box
Look at the cards in the box to verify that every hole position is covered. Thus the problem is in NP.

What remains is to show that every problem in NP can be reduced to TOYS. We do that by reducing CNF-SAT, known to be NP-complete, to TOYS. More precise, we must show that every CNF-formula

$$\varphi = (l_1 \vee l_2 \vee \ldots \vee l_i) \wedge (l_j \vee \ldots \vee l_k) \wedge \ldots \wedge (\ldots)$$

with n variables and m clauses reduced to a problem γ in TOYS is satisfiable if and only if γ is solvable. Consider the following rules.

1. Variable x_i corresponds to a card c_i.

2. There are two columns with m positions on every card, so there is one row for each clause.
3. If x_i occurs in clause j then the hole in position j in the left column is covered.
4. If $\bar{x}_i$ occurs in clause j then the hole in position j in the right column is covered.
5. There are holes in all other positions.
6. There is an additional card c_{n+1} with holes only in the left column.

If φ is satisfied for an assignment of $x_1, \ldots, x_n$ then the corresponding puzzle γ is solvable, because if $x_i = 1$ then card c_i can be put down "right side up" and thus covering the holes in the left column corresponding to the clauses x_i satisfies. If $\bar{x}_i = 1$ then the card c_i should be placed "upside-down" and in the same way covering holes. Since the formula was satisfied, every clause must be satisfied and therefore the left holes in every row are covered. The extra card c_{n+1} is used for covering the holes in the right column that might occur when all variables in a clause are set to 1, so all right holes are covered too.

Conversely, if the puzzle γ is solved, we can easily find an assignment to $x_1, \ldots, x_n$ satisfying φ from the following rule:

$$x_i = \begin{cases} 1, & \text{if } c_i \text{ is right side up} \\ 0, & \text{otherwise} \end{cases}$$

We also need to check the placing of the extra card; if it is up-side-down we simply invert the assignment of all variables. When all holes are covered we also have, by construction, that all clauses are satisfied. We have shown that CNF-SAT can be reduced to TOYS. □

Solution to Processor Scheduling

First we need to show that the problem is in NP. This is true because we can guess a scheduling of the jobs and then easily verify that it is valid.

Next, to prove that SCHEDULE is NP-complete, we have to reduce an NP-complete problem to it. We choose to reduce the NP-complete problem 3-COLORING to SCHEDULE. This means that, given a graph $G = (V, E)$, we want to create an instance of SCHEDULE such that the graph can be colored with 3 colors if and only if the jobs can be scheduled in 3 time units.

We create a job for each vertex in the graph and if the vertex i is a neighbor to the vertex j we add the condition that job i can not be executed at the same time as job j. Let each color correspond to a time unit. Then we claim that the graph can be colored with 3 colors if and only if the jobs can be executed in 3 time units. First, if the graph can be colored with 3 colors let C_1 be the vertices that is colored with the first color. There is no edge between the vertices in C_1 and hence the corresponding jobs can all be executed in the first time unit. The same is true for colors 2 and 3 which gives that all jobs can be executed in 3 time units. On the other hand, if the jobs can be executed in 3 time units, the vertices that correspond to the jobs in each time unit can be given one color so that the graph can be 3-colored.

In this way we have reduced 3-COLORING to SCHEDULE and, consequently, SCHEDULE is NP-complete.

With the same argument we can reduce k-COLORING to SCHEDULE with k time units so that the latter problem is NP-complete for $k \geq 3$.

If we have 2 time steps, the problem can be solved by performing the reduction in the opposite way. That is, given the jobs and the conditions, we construct a graph which we then try to color with 2 colors. This can be done in linear time with depth first search, since if a vertex v has color 1, all neighbors to v must have color 2. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 11

Approximation algorithms

Approximation of independent set Let INDEPENDENT SET$-B$ be the problem of finding a maximum set of independent vertices in a graph where every vertex has degree $<= B$. Show that this problem is in APX, in other words it can be approximated within some constant in polynomial time.

Probabilistic not-all-equal-satisfiability The NP hard problem MAX NOT-ALL-EQUAL 3-CNF SAT is defined as follows:

> INPUT: A CNF formula consisting of the clauses $c_1, c_2, \ldots, c_m$ where every clause is a disjunction of exactly three literals (variables or negated variables). The variables are called $x_1, x_2, \ldots, x_n$.
> SOLUTION: A variable assignment.
> GOAL FUNCTION: The number of clauses that contain at least one true literal and one false literal.
> PROBLEM: Maximize the goal function.

Since this problem is NP complete we want an algorithm that approximates this within a constant in polynomial time. Construct a probabilistic approximation algorithm for the problem which gives an expected approximation quota of $4/3$. Analyze the complexity of the algorithm and expected approximation quota. You may use a random number generator in your algorithm.

Approximation of linear inequalities MAX SAT LR$^{\geq}$ (maximum satisfiable linear subsystem) is the problem that, given a set of linear inequalities on the form $\geq$ (for example $2x_1 - 8x_3 + 3x_8 \geq 3$), find a variable assignment which satisfies as many inequalities as possible Construct an approximation algorithm that approximates MAX SAT LR$^{\geq}$ within a factor of 2.

Upper bound for approximation of homogeneous bipolar inequalities

MAX HOM BIPOLAR SAT LR$^{\geq}$ (maximum homogeneous bipolar satisfiable linear subsystem) is the same problem as MAX SAT LR$^{\geq}$ but the variables may only contain the values 1 and -1, and all inequalities are homogeneous, in other words they have no constant terms (zero on the right hand side).

Show that MAX HOM BIPOLAR SAT LR$^{\geq}$ can be approximated within a factor of 2 and that MAX HOM BIPOLAR SAT LR$^{>}$ can be approximated within a factor 4.

Upper bound for approximation of binary inequalities MAX BINARY SAT LR$^{\geq}$ (maximum binary satisfiable linear subsystem) is the same problem as MAX SAT LR$^{\geq}$ where the variables may only contain 0 or 1.

Show that MAX BINARY SAT LR$^{\geq} \notin$APX.

Solutions

Solution to Approximation of independent set
Given a graph $G = (V, E)$ with vertex degrees limited by B, construct an independent set of vertices in the following way.

$V' \leftarrow \emptyset$
$W \leftarrow V$
for $v \in W$ **do**
  $V' \leftarrow V' \cup \{v\}$
  $W \leftarrow W - \{w \in W : (w, v) \in E\} - \{v\}$

After the algorithm V' is an independent set of vertices. It is also easy to see that V' is a dominating set. The optimal independent set of vertices V'_{opt} can at most be B times larger than $|V'|$. To see this we need only look at one of the vertices in V' and its neighbours in V. If all its neighbours are in V'_{opt} then there are B of them, and it can not be any worse. Since V' is a dominating set, every vertex in V'_{opt} is either in V' or the neighbour of a vertex in V'. This means that V'_{opt} is at most B times larger than V', in other words the approximation quota is B, which is constant. □

Solution to Probabilistic not-all-equal-satisfiability
The algorithm is simply: Assign to every variable a randomly chosen value. It should now be as probable that the variable is true as false. If we look at an arbitrary clause then there are only two cases of eight (false-false-false and true-true-true) that should not count towards the goal function. The expected value for the number of clauses is

$$
\begin{aligned}
&E[\text{the number of clauses with both false and true literals}] = \\
&= m \cdot Pr[\text{an arbitrary clause which is neither entirely true or entirely false}] = \\
&= m \cdot (1 - Pr[\text{an arbitrary clause which is either entirely true or entirely false}]) = \\
&= m \cdot (1 - 2/8) = 3m/4.
\end{aligned}
$$

Because at most m clauses can be counted the expected approximation quota is

$$\frac{OPT}{APPROX} \geq \frac{m}{3m/4} = \frac{4}{3}.$$

The algorithm takes linear time and requires linearly many random numbers in the number of variables. □

Solution to Approximation of linear inequalities
Assume that the input is given as a set X of rational variables and a set E of linear inequalities over X.

while $E \neq \emptyset$ **do**
  if (there are inequalities in E with a single variable) **then**
    $U \leftarrow \{x \in X : x$ is a single variable in at least one inequality in $E\}$
    Chose an arbitrary $y \in U$.
    $F(y) \leftarrow \{e \in E : e$ only contains the variable $y\}$
    Give y a value that satisfies as many inequalities in $F(y)$ as possible.
    $E \leftarrow E - F(y)$
  else
    Chose arbitrary $y \in X$.
    $y \leftarrow 0$
  Evaluate if the inequalities in E which contain y.
  $X \leftarrow X - \{y\}$

The algorithm always assigns y a value that satisfies at least half of the inequalities in $F(y)$. Therefore it constructs a solution which satisfies at least half of the inequalities in the input. Since at most all inequalities can be satisfied the approximation quota for the algorithm is 2.

The time complexity is polynomial since every variable and term are only used once. □

Solution to Upper bound for approximation of homogeneous bipolar inequalities
We begin with an approximation of Max Hom Bipolar Sat $\mathrm{LR}^{\geq}$. Take an arbitrary bipolar vector $\mathbf{x}$ and look at the number of satisfied inequalities if the variables are set to $\mathbf{x}$ and $-\mathbf{x}$ respectively. If the left hand side of an inequality is positive for $\mathbf{x}$ then it is negative for $-\mathbf{x}$ and vice versa. Therefore one of these two vectors will satisfy at least half of the inequalities, in other words achieve the approximation quota 2.

This trivial algorithm does not work for Max Hom Bipolar Sat $\mathrm{LR}^{>}$, because many relations may be 0 for both vectors. Therefore we start by finding a solution with many non-zero relations. The approximation algorithm for Max Sat $\mathrm{LR}^{\geq}$ above can be modified so that it finds a solution $\mathbf{x}$ for which at least half of the relations are non-zero. The same thing then applies to $-\mathbf{x}$, and one of these two vectors have to satisfy at least $1/4$ of all inequalities. □

Solution to Lower bound for approximation of binary inequalities
Show this by reducing Max Clique to Max Binary Sat $\mathrm{LR}^{\geq}$ with an approximation preserving reduction. Since we know that Max Clique is not in APX we know that Max Binary Sat $\mathrm{LR}^{\geq} \notin$ APX.

Let $G = (V, E)$ be the input for Max Clique. For every vertex $v_i \in V$ construct a variable x_i and the inequality

$$x_i - \sum_{j \in N(v_i)} x_j \geq 1$$

where j is in $N(v_i)$ if $v_j \neq v_i$ and $(v_i, v_j) \notin E$ (i.e. v_j is not a neighbour of v_i). Now we have a system with $|V|$ inequalities. Note that the i:th inequality is satisfied iff $x_i = 1$ and $x_j = 0$ for all $j \in N(v_i)$.

It is easy to verify that if we have a s clique $V' \subseteq V$ then we can get a binary solution that satisfies the s corresponding inequalities by setting $x_i = 1$ if $v_i \in V'$ and $x_i = 0$ otherwise. On the other hand, if we get a binary solution $\mathbf{x}$ that satisfies s inequalities then we can get an s clique by letting V' consist of all vertices v_i which correspond with satisfied inequalities.

This reduction is not just *approximation preserving* but also *cost preserving* because it keeps the goal functions value in its entirety.

Also note that this reduction works for $A\mathbf{x} > \mathbf{0}$ och $A\mathbf{x} = \mathbf{1}$, so we can prove that Max Binary Sat $\mathrm{LR}^{>} \notin$ APX and that Max Binary Sat $\mathrm{LR}^{=} \notin$ APX. □

Algoritmer, datastrukturer och komplexitet, fall 2015

Exercises to övning 12

Complexity classes and mixed questions

Questions on complexity classes

co-NP completeness A discrete technical diagnosis problem can be modelled in the following way: component state, system state and environment state are represented by boolean variables and the system in itself is defined with a boolean formula φ. We know that in all *possible* (working) worlds the system variables will have values such that φ is true.

Assume that the component states are all binary and that a component c is therefore represented by a boolean variable x_c (where true means that the component worlds and false means that it is broken). We know that a component c is broken if the formula we get if we in φ place values for all variables that represent known component, system and environment states and set x_c to true is not satisfiable.

Show that it is co-NP-complete to determine whether or not c is broken.

---

relations between complexity classes PSPACE is the complexity class which consists of all languages for which there is a *deterministic* Turing machine that recognizes the language in polynomial *memory*. EXPTIME consists of all languages for which there is a *deterministic* Turing machine that recognizes the language in exponential time. Show that PSPACE$\subseteq$EXPTIME.

Mixed exercises from old exams

The first exercise is a typical first question on the theory exam. The other tasks are typical questions on the oral examination.

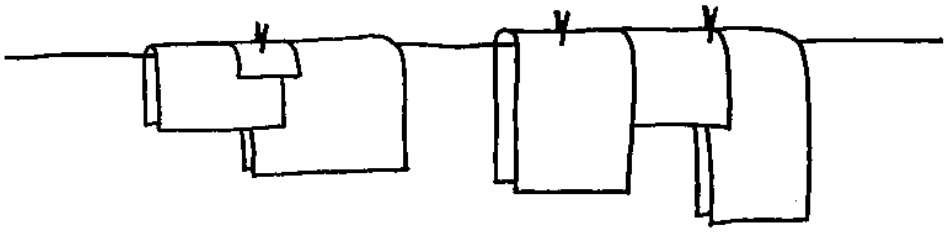
1. Are the following statements true or false? For every question the right answer gives 1 point and a convincingly proved correct answer gives 2 points.

   a) Determining if a number with n digits is a prime number is in the complexity class co-NP.

   b) There is a constant $c > 1$ such that $n^3 \in O(c^{\log n})$.

   c) Binary trees are usually implemented by adding two pointers to every post (`left` and `right`). When we implement ternary trees (where every parent node has three children) we need at least three pointers in every post.

2. (algorithm construction, grade C) On a clothesline, Viggo has hung n towels of various sizes. Viggo wants the towels to give the viewer a harmonic impression. Therefore he was very careful when placing them and has measured exactly where the towels are hanging along the line, so that nobody should want to move them. To ensure that the wind does not interfere with this harmonic order Viggo wants to secure the towels with clothespins, but he does not want to use more clothespins than necessary. It is sufficient that every towel is fixed with one clothespin and two or more towels hanging over each other can share the same clothespin.



Describe an algorithm (in words or with pseudo code) which as fast as possible calculates where on the line Viggo should place the clothespins so that he needs as few as possible. Justify your solutions correctness and analyze with unit cost.

The input is n pairs of numbers (v_1, h_1), (v_2, h_2), ..., (v_n, h_n) where every pair denotes where the left and the right side of each towel is. All measurements denote the distance from the beginning of the clothes line to a towel edge. The pairs are not sorted in any way. Output is a number of coordinates indicating where the clothespins should be pinched (you may assume that a clothespin has width 0).

For maximum points your algorithm must run in time $O(n \log n)$.

3. (Complexity, grade C) In a large organization such as KTH there are many groups of persons, for instance teachers at Nada; teachers at F; and students in the course Algorithms, data structures and complexity; members of the Teknolog choir; etc. etc. Now the headmaster wants to create a group of informers who quickly can get information to all individuals at KTH. He wants every group to be represented in the informer group (at least one member of each group should be in the informer group), but he also wants the informer group to be as small as possible.

   This is an example of a general problem where you given a set of groups want to find a minimum set of members from all groups.

   a) Formulate the problem mathematically as a set problem and describe it as a decision problem.

   b) Show that the problem is NP complete.

4. (algorithm construction, grade A; complexity, grade C; difficult assignment!) The folding rule problem can be described in the following way. A folding rule consists of a chain of wooden pieces held together with hinges at the ends. It is possible (but not necessary) to fold the folding rule at every hinge. The wooden pieces may have different lengths. Then it is no longer certain how to fold the folding rule so that it becomes as short as possible (and fits the carpenters pocket). The problem is: given a folding rule, fold it so that it becomes as short as possible.

   a) Formulate the problem mathematically as a decision problem.

   b) Show that the folding rule problem is NP complete. A good problem to reduce is the set partition problem.

   c) Assume that the wooden pieces have integer lengths and the longest piece is $17n$ where h is the number of wooden pieces that the folding rule consist of. Determine the complexity for this problem.

5. (complexity, grade C) MAX 2$\wedge$SAT is an optimization problem defined in this way:

   INPUT: A positive integer K between 1 and n and n clauses where every clause are one or two literals combined with the operator $\wedge$. For example: $x_1 \wedge \overline{x}_3$, $x_2 \wedge x_3$.
   PROBLEM: Is there a variable assignment that satisfies at least K clauses?

Show that this decision problem version of MAX 2∧SAT is NP complete. You can use the fact that the problem MAX 2SAT – a similar problem with the operator ∨ instead of ∧ – is NP complete.

Solutions to exercises on complexity classes

Solution to co-NP completeness
To prove that the problem is in co-NP we will show that we in polynomial time can verify a no-solution, in other words that c is not broken. We know that c is not broken iff there is a variable assignment which satisfies the formula. If we guess variable values we only need to verify that the formula with this assignment is true. This can be done in polynomial time.

To show that the problem is co-NP hard we reduce co-SAT, the problem to determine whether a given boolean formula ϕ is *not* satisfiable. Because SAT is NP complete per definition co-SAT is co-NP complete.

Given ϕ, construct a system which contains the extra component c (represented by x_c) and as defined by the formula $\varphi = \phi \vee \neg x_c$.

Now note that if x_c is set to true then $\varphi = \phi$ so the problem of determining whether c is broken is exactly the same as the problem of determining whether ϕ is not satisfiable. The reduction is therefore correct. □

Solution to Complexity class relation
If a Turing machine uses polynomial memory there is a constant k such that the number of used squares on the tape are $O(n^k)$ where n is the size of the input. If the alphabet consists of three characters (0,1,blank) then the number of different configurations on the tape is limited by $3^{O(n^k)}$, the number of positions for the read/write head is$O(n^k)$ and the number of states in the Turing machine is finite i.e. $O(1)$. The total number of configurations for a Turing machine which uses polynomial memory is therefore which is exponential in n. Since the Turing machine can not return to the same configuration more than once (then it would run in an infinite loop) this is also an upper bound for the time. Every problem that can be solved with polynomial memory can therefore be solved in exponential time. □

Solutions to mixed questions from exams

1. a) *True*. The problem is in co-NP if the complement problem is in NP. The complement problem is in this case to determine whether a number with n digits can be factorized in at least two factors (greater than 1). This problem is in NP because a solution (a factorization of the number) can be verified in polynomial time (by multiplying the factors and checking that the product is the given number).

   b) *True*. If we assume that $\log n$ is the logarithm in base 2 then we know that $c^{\log n} = 2^{\log c^{\log n}} = 2^{\log n \log c} = n^{\log c}$. If we chose $c \geq 8$ then $\log c \geq 3$ and $n^3 \in O(n^{\log c}) = O(c^{\log n})$.

   c) *False*. For general tree we need two pointers on each post (`firstson` and `next`).

2. A greedy algorithm solves the problem in time $O(n \log n)$.

   1. Sort all pairs based on their left edge and their right edge. Collect the results in two lists, *Lsort* and *Rsort*. During the sorting, keep track of where each post in the first list ends up in the other one. With heap sort this takes time $O(n \log n)$.
   2. As long as there are pairs in the list *Rsort*:
      2.1 Take the first remaining pair, say (l, r). Since the pairs have been sorted after their right edges (l, r) is the towel with the leftmost right edge.

2.2 Pinch a clothespin farthest to the right on this towel, exactly to the left of r.

2.3 Remove all pairs with a left edge to the left of r, in other words remove all towels pinched by the last clothespin. This is done by removing pairs from the beginning of *Lsort* which is sorted in rising left edge order. For every pair that is removed from *Lsort* the same pair should be removed from *Rsort*.

Finally all pairs have been removed and the algorithm terminates. In step 2 each pair is only treated once. The time for step 2 is therefore $O(n)$. The whole algorithm therefore takes time $O(n \log n)$.

Correctness: Since only pairs with a clothespin in them get removed all pairs will be pinched when the algorithm has run. No we will show that the number of clothespin is minimal. The towle which the leftmost right edge (the one that is first in *Rsort*) must have a clothespin pinching it. If we attach the towel in the point p all towels with a left edge less than p will be stuck, since there is no towel tiwh its right edge to the left of p. To pinch as many towels as possible we must choose a p that is as large as possible which means as close to the towels right edge as possible. The same line of reasoning is applied to the other towels.

3. a) Let k be a positive integer, let S be the set of persons and $C = \{C_1, \ldots, C_m\}$ be the m groups. The problem is finding a subset $S' \subseteq S$ with at most k elements so that $S' \cap C_i \neq \emptyset$ for $1 \leq i \leq m$. This problem is known as the *hitting set*.

   b) The problem is in NP since we can guess which k elements are supposed to be in S' and verify that $S' \cap C_i \neq \emptyset$ for $1 \leq i \leq m$ in polynomial time.

   The problem is NP hard because it is a generalization of the vertex cover problem which is known to be NP complete. Given a graph $G = (V, E)$, let $S = V$ and $C = E$. A vertex cover of size k is exactly a subset $S' \subseteq S$ of size k which contains at least one element from each C_i.

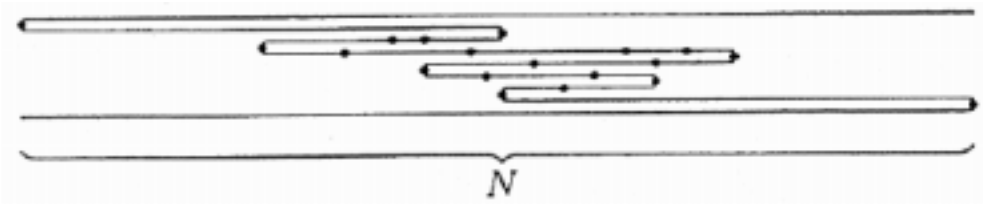
4. a) Input is n positive numbers $l_1, \ldots, l_n$, which in turn give the lengths of the n wooden pieces included in the folding rule, and a positive number K indicating the length of the carpenters pocket. The question is if we can choose directions $r_1, \ldots, r_n$, where $r_i \in \{-1, 1\}$, for the n wooden pieces so that if we fold the folding rule according to the directions its length will be at most K, in other words $\max_{j \in [0..n]} \left\{ \sum_{i=1}^{j} r_i l_i \right\} - \min_{j \in [0..n]} \left\{ \sum_{i=1}^{j} r_i l_i \right\} \leq K$.

   b) The folding rule problem is in NP because we can guess the values of $r_1, \ldots, r_n$ and verify that the length of the folding rule is at most K in polynomial time.

   We show that it is NP hard by reducing the set partition problem which is known to be NP complete. The input for the partitioning problem is n positive integers $t_1 \ldots, t_n$ and the question is if the numbers can be divided into two groups so that their sums are equal.

   Given $t_1 \ldots, t_n$, construct input for the folding rule problem in the following way: $l_1 = N$, $l_2 = N/2$, $l_{i+2} = t_i$ for $1 \leq i \leq n$, $l_{n+3} = N/2$, $l_{n+4} = N$ and $K = N$, where $N = \sum_{i=1}^{n} t_i$.

   Now show that there is an even partitioning iff there is a folding of the folding rule so that the length is at most N.



If we have a partitioning we can fild the folding rule so that wooden pieces that correspond to numbers in the first subset ar folded to the left (direction -1), pieces that correspond to numbers in the second subgroup are folded to the right (direction $+1$), $r_1 = -1$, $r_2 = 1$, $r_{n+3} = 1$, $r_{n+4} = -1$. It is easy to verify that the length of the folding rule is N, see illustration.

Conversely, if we have a folding which ensures that the folding rule fits the pocket of size N we can assume that $r_1 = 1$. If $r_1 = -1$ we first switch signs on all directions, we flip the sliding rule over, which of course still has the same length. Since the length of the first wooden piece is N the next piece must have direction $r_2 = -1$ and go back to a position on the middle of the last wooden piece. To ensure that the length of the sliding rule is at most N the end of the other wooden pieces position must be the same as the second to last piece's beginning. This means that all pieces between them must start and end in the same position, i.e. $\sum_{i=3}^{n+2} r_i l_i = 0$, which can be written as $\sum_{i:3\leq i\leq n+2 \wedge r_i=1} l_i - \sum_{i:3\leq i\leq n+2\wedge r_i=-1} l_i = 0$. Since these wooden pieces lengths exactly correspond to numbers in the partitioning problem we get a solution for this problem by partitioning according to how the wooden pieces are directed.

c) The problem can be solved in polynomial time with dynamic programming by keeping track of all possible subset sums. If the longest piece of wood is $17n$ we know that for all partial sums we must have $-17n^2 \leq \sum_{i=1}^{j} r_i l_i \leq 17n^2$. Create a boolean array $b[0..n, -17n^2..17n^2]$ where all elements are initially false. The algorithm fills the array so that $b[j,k]$ is true iff there are values for r_i such that $\sum_{i=1}^{j} r_i l_i = k$. The recursion equation is $b[j,k] = b[j-1, k+l_j] \vee b[j-1, k-l_j]$. Also create a boolean array $ends[0..n, -17n^2..17n^2, -17n^2..0, 0..17n^2]$ where $ends[j,k,l,r]$ is true if there is some folding of the j first pieces of wood that end in position k have their leftmost point in position l and their rightmost point in position r.

$b[0,0] \leftarrow$ true
for $j \leftarrow 1$ **to** n **do for** $k \leftarrow -17n^2$ **to** $17n^2$ **do**
  if $b[j-1,k]$ **then**
    $b[j, k-l_j] \leftarrow b[j, k+l_j] \leftarrow$ true
    for $l \leftarrow -17n^2$ **to** 0 **do for** $r \leftarrow 0$ **to** $17n^2$ **do**
      if $ends[j-1,k,l,r]$ **then**
        $ends[j, k-l_j, min(l, k-l_j), r] \leftarrow ends[j, k+l_j, l, max(r, k+l_j)] \leftarrow$ true
$minlength \leftarrow 17n^2$
for $k \leftarrow -17n^2$ **to** $17n^2$ **do**
  if $b[n,k]$ **then**
    for $l \leftarrow -17n^2$ **to** 0 **do for** $r \leftarrow 0$ **to** $17n^2$ **do**
      if $ends[j,k,l,r] \wedge (r - l < minlength)$ **then** $minlength \leftarrow r - l$
return $(minlength)$

It is easy to verify that the algorithm returns the smallest length of folding rule achievable. The algorithm takes time $O(n^7)$, which is polynomial (but quite slow!).

5. MAX 2∧SAT is in NP because it is easy to verify that a variable assignment (a solution) satisfies at least K of the clauses We show that it is NP hard by reducing MAX 2-SAT. Every 2-SAT clause with a single literal is moved unaltered to the MAX 2∧SAT-probleminstance. For every clause $l_i \vee l_j$ in the MAX 2-SAT-instance we construct three clauses in the MAX 2∧SAT-probleminstance: $l_i \wedge l_j$, $\overline{l}_i \wedge l_j$ and $l_i \wedge \overline{l}_j$. We see that $l_i \vee l_j$ is true iff one of the tree constructed clauses is true. If $l_i \vee l_j$ is false then all the three constructed clauses are false. THe number of satisfied MAX 2-SAT clauses is equivalent with the number of satisfied MAX 2∧SAT-clauses. Choose the goal K for the constructed problem to the same value as K for the MAX 2-SAT-problem.