Algoritmer och datastrukturer

Laborationsuppgift, 1.5hp

HT13

Jon Wahlström, b12jonwa

Sebastian Zander, a12sebza

Institutionen för Information och Kommunikation

Högskolan i Skövde

Innehållsförteckning

[Inledning 4](#_Toc374963640)

[1. Problem – Modifierad Bucketsort 5](#_Toc374963641)

[1.1 Intuitiv beskrivning 5](#_Toc374963642)

[1.2 Kod 6](#_Toc374963643)

[1.2.1 Pseudokod 6](#_Toc374963644)

[1.2.2 C++ 7](#_Toc374963645)

[1.3 Tidskomplexitet 8](#_Toc374963646)

[1.4 Analys 9](#_Toc374963647)

[2. Problem – Sociala nätverk 10](#_Toc374963648)

[2.1 Intuitiv beskrivning 10](#_Toc374963649)

[2.2 Kod 11](#_Toc374963650)

[2.2.1 Pseudokod 11](#_Toc374963651)

[2.2.2 C++ 12](#_Toc374963652)

[2.3 Tidskomplexitet 15](#_Toc374963653)

[2.4 Analys 15](#_Toc374963654)

[3. Problem – Komprimering genom Huffman-kodning 16](#_Toc374963655)

[3.1 Intuitiv beskrivning 16](#_Toc374963656)

[3.2 Kod 17](#_Toc374963657)

[3.2.1 Pseudokod 17](#_Toc374963658)

[3.2.2 C++ 18](#_Toc374963659)

[3.3 Tidskomplexitet 22](#_Toc374963660)

[3.4 Analys 22](#_Toc374963661)

[4. Problem – Tidskomplexitet 22](#_Toc374963662)

[4.1 Intuitiv beskrivning 23](#_Toc374963663)

[4.2 Kod 23](#_Toc374963664)

[4.2.1 Pseudokod 23](#_Toc374963665)

[4.2.2 C++ 24](#_Toc374963666)

[4.3 Tidskomplexitet 28](#_Toc374963667)

[4.4 Analys 28](#_Toc374963668)

[5. Problem – Handelsresandeproblemet 29](#_Toc374963669)

[5.1 Intuitiv beskrivning 29](#_Toc374963670)

[5.2 Kod 29](#_Toc374963671)

[5.3 Tidskomplexitet 29](#_Toc374963672)

[5.4 Analys 29](#_Toc374963673)

# Inledning

Laborationens utgör en grund för bekantskapen med olika typer av algoritmer och datastrukturer. Nedan följer en intuitiv beskrivning, exempelkod samt en tidskomplexitetsberäkning och analys för fem olika typer av problem.

* Modifierad Bucketsort
* Sociala nätverk
* Komprimering via huffman-kodning
* Tidskomplexitet
* Handelsresandeproblemet

Fokus för laborationen som helhet ligger snarare på problemlösningsnivå än implementationsnivå.

# Problem – Modifierad Bucketsort

Problemet som skall lösas är att sortera en given mängd av element med en modifierad Bucket sort.

## Intuitiv beskrivning

Sorteringsalgoritmen Bucket sort bygger på att partitionera en given mängd element i Buckets. Utifrån det placeras sedan värdet från elementet storleksmässigt i Buckets med minsta värdet först.



Figur 1: Given array att sortera

Låt säga att det högsta elementet i array är *k*. Antalet totala Buckets som skapas är då *k+1*.



Figur : Antalet buckets utefter högsta värdet i array

Efter att Buckets har skapats initieras dessa först med värdet noll för att sedan iterativt tilldelas ett eller flera elements värden. Dessa mappas direkt mot dess Bucket vilket ger en sorterad följd som nu kan returneras.



Figur : Placerar värden i buckets i storleksordning

## Kod

### 1.2.1 Pseudokod

function bucketSort(List list)

List buckets

int k => 0

foreach element in list

if element > k

k => element

end

end

bucket.setSize(k)

foreach bucket in buckets

bucket => 0

end

foreach element in list

bucket[element] => bucket[element] + 1

end

for i : 0 to bucket.size()

for 1 to bucket

print (i)

end

end

end

### 1.2.2 C++

#### Main.cpp

#include <iostream>

#include <vector>

#include <random>

#include <algorithm>

#include <iomanip>

#include <chrono>

template <class itrType>

void bucketSort(itrType begin, itrType end)

{

int k = \*std::max\_element(begin, end);

std::vector<int> buckets(k + 1, 0);

for(itrType itr = begin; itr != end; ++itr)

{

++buckets[\*itr];

}

int val = 0;

for(auto bucket : buckets)

{

for (int i = 0; i < bucket; ++i)

{

\*begin = val;

++begin;

}

++val;

}

}

int main()

{

std::mt19937 engine;

std::uniform\_int\_distribution<int> distro(0, 127);

std::vector<int> list(5000000);

for (auto iter = list.begin(); iter != list.end(); ++iter)

{

\*iter = distro(engine);

}

auto now = std::chrono::system\_clock::now();

bucketSort(list.begin(),list.end());

auto then = std::chrono::system\_clock::now();

auto diff1 = std::chrono::duration\_cast<std::chrono::microseconds>(then - now).count();

int i = 0;

for(auto itr : list)

{

++i;

if (i % 5 == 0)

std::cout << std::endl;

std::cout << std::setw(5) << itr;

}

std::cout << std::endl;

std::cout << "Bucket sort took: " << diff1 << " microsecs." << std::endl;

return 0;

}

## Tidskomplexitet

Genom att analysera algoritmen steg för steg har vi kommit fram till följande tidskomplexitet. T(n) = 3m + 7 + n. Detta kan skrivas i stora Oh notation som O(m + n), där m är antalet element som skall sorteras och n är antalet buckets. I värsta fall är m = n vilket gör att komplexiteten blir O(m²).

function bucketSort(List list)

List buckets 0

int k => 0 1

foreach element in list m

if element > k 1

k => element 1

end

end

bucket.setSize(k) 1

foreach bucket in buckets n

bucket => 0 1

end

foreach element in list m

bucket[element] => bucket[element] + 1 1

end

for i : 0 to bucket.size() n

for 1 to bucket i \* antal element i varje bucket

print (i) 1

end

end

end # n \* den genomsnittliga   
 element i varje bucket = m

UPPGIFT 1.4??

T(5) = 3\*25 + 7 + 5

O(5) = 25 + 5

## Analys

Problemet med denna algoritm är att den inte tar hänsyn till hur indatat är distribuerat. Exempelvis om endast två element skall sorteras där det ena har värdet 1 och det andra har värdet fem miljoner. I det fallet kommer det skapas fem miljoner buckets för att sortera två element. Ett sätt att lösa detta skulle vara att skapa en bucket för varje unikt värde som finns i listan och mappa den bucketen till det värdet.

# Problem – Sociala nätverk

Problemet som skall lösas är att utifrån ett nätverk av fiender hitta vänner till en person. Utifrån filosofin ”min fiendes fiender är min vän”.

## Intuitiv beskrivning

Utifrån en given startnod byggs ett träd upp av noderna i grafen genom att söka igenom grafen med bredden först. Vartannat lager i trädet innehåller noder som är fiender till startnoden och vartannat lager är vänner till startnoden.

Som ett exempel skall vänner till a hittas i nedanstående graf av fiender.



Det trädet som byggs från ovanstående graf illustreras nedan. Notera att cyklar tas bort och bara den kortaste vägen mellan två noder ges. Lagret precis under roten består av ovänner till a, därefter består vartannat kommande lager av ovänner och alla andra lager av vänner.



## Kod

### 2.2.1 Pseudokod

def Node

List:Node enemies

end

function ListFriends(Node node)

List current\_level

List next\_level

List friends

List visited;

children.Add(node)

visited.Add(node)

bool enemyIsFriend = false

while not children.IsEmpty()

foreach child in current\_level

foreach enemy in child.enemies

if enemy not in visited

if enemyIsFriend

friends.Add(enemy)

end

else

visited.Add(enemy)

next\_level.Add(enemy)

end

end

end

enemyIsFriend = not enemyIsFriend

current\_level = next\_level

next\_level.clear()

end

return friends;

end

### 2.2.2 C++

#### Relations.h

#pragma once

#include <vector>

#include <memory>

#include <set>

#include <string>

namespace relations

{

class node : public std::enable\_shared\_from\_this<node>

{

private:

struct detail

{

struct this\_is\_private {};

};

public:

typedef std::shared\_ptr<node> ptr\_t;

std::vector<node::ptr\_t> list\_friends();

void add\_enemy(node::ptr\_t);

static node::ptr\_t make(std::string);

node(std::string, detail::this\_is\_private&);

const std::string& name();

private:

std::string name\_;

std::set<node::ptr\_t> enemies\_;

};

}

#### Relations.cpp

#include "Relations.h"

#include <utility>

#include <deque>

namespace relations

{

node::ptr\_t node::make(std::string name)

{

return std::make\_shared<node>(std::move(name), detail::this\_is\_private());

}

void node::add\_enemy(node::ptr\_t n)

{

enemies\_.insert(n);

}

node::node(std::string name, detail::this\_is\_private&)

:name\_(std::move(name))

{}

std::vector<node::ptr\_t> node::list\_friends()

{

std::vector<node::ptr\_t> current\_level, next\_level;

std::vector<node::ptr\_t> friends;

std::set<node::ptr\_t> visited;

current\_level.push\_back(shared\_from\_this());

visited.insert(shared\_from\_this());

bool enemy\_is\_friend = false;

while(!current\_level.empty())

{

for (auto& node : current\_level)

{

for(auto& enemy : node->enemies\_)

{

if (visited.count(enemy) == 0)

{

if(enemy\_is\_friend)

{

friends.push\_back(enemy);

}

visited.insert(enemy);

next\_level.push\_back(enemy);

}

}

}

enemy\_is\_friend = !enemy\_is\_friend;

current\_level = std::move(next\_level);

}

return std::move(friends);

}

const std::string& node::name()

{

return name\_;

}

}

#### Main.cpp

#include "Relations.h"

#include <iostream>

#include <algorithm>

#include <iomanip>

template <class forward\_iterator>

void pretty(forward\_iterator begin, forward\_iterator end,

int width, int elem\_per\_row)

{

while (begin != end)

{

if (std::distance(begin, end) % elem\_per\_row == 0)

std::cout << std::endl;

std::left(std::cout);

std::cout << std::setw(width)<< (\*(begin++))->name();

}

}

int main()

{

auto a = relations::node::make("a");

auto b = relations::node::make("b");

auto c = relations::node::make("c");

auto d = relations::node::make("d");

a->add\_enemy(b);

b->add\_enemy(c);

b->add\_enemy(d);

c->add\_enemy(b);

c->add\_enemy(d);

auto friends = a->list\_friends();

pretty(friends.begin(), friends.end(), 4, 10);

return 0;

}

## Tidskomplexitet

Analys av algoritmen rad för rad ger följande tidskomplexitet. T(n) = 11 + d \* c\* e \* m vilket kan uttryckas i stora Oh-notation som O(n\*n) .

def Node

List:Node enemies

end

function ListFriends(Node node)

List current\_level

List next\_level

List friends

List visited;

current\_level.Add(node) 1

visited.Add(node) 1

bool enemyIsFriend = false 1

while not current\_level.IsEmpty() d

foreach child in current\_level c

foreach enemy in child.enemies e

if enemy not in visited m

if enemyIsFriend 1

friends.Add(enemy) 1

end

visited.Add(enemy) 1

next\_level.Add(enemy) 1

end

end

end

enemyIsFriend = not enemyIsFriend 1

current\_level = next\_level 1

next\_level.clear() 1

end

return friends; 1

end

*d* är djupet för trädet av grafen eftersom att trädet söks igenom nivå för nivå. *c* är antalet noder i den nuvarande nivån. Detta betyder att *d\*c* är *n,* där *n* är antalet noder. *e*  är antalet fiender varje nod har, dvs antalet utåtvertriser. *n\*e* kommer då bli det totala antalet vertriser i grafen (*v*). Dessa tre loopar kan då ses som en loop vilket beror på antalet vertriser i grafen. En extra variabel i funktionen är *m*, vilket är det totala antalet besökta noder, vilket kommer att gå från 1 till *n* allteftersom algoritmen itererar*.*

## Analys

Det kan se ut som att vi har avvikit från uppgiften eftersom vår algoritm inte explicit använder en kö. Specifikt används inte std::queue utan std::vector används istället. Dock används vektorn som om det vore en kö eftersom den iteraras från början till slut och nya element läggs till i slutet. Den huvudsakliga poängen är att vi har gjort sökningen med bredden först.

# Problem – Komprimering genom Huffman-kodning

Problemet som skall lösas är att givet en sträng bygga upp en huffman-kodning för strängen. En huffman-kodning är ett frekvenssorterat binärt träd. Detta kan användas för att komprimera och avkomprimera strängen.

## Intuitiv beskrivning

Som ett exempel används här strängen ”aaaabbbc”. Efter att ha räknat mängden karaktärer av samma typ skapas noder av respektive karaktär tillsammans med antalet av den typen (dess vikt) vilket illustreras nedan. Noderna sorteras i ordning där den nog med högst vikt läggs först.



Trädet byggs genom att kontinuerligt söka upp två noder med lägst vikt. Dessa två noder kopplas därefter ihop till ett subträd där rotnoden får den sammanlagda vikten från de bägge noderna. Roten läggs tillbaka i listan på den platsen så att listan fortfarande behåller sorterad ordning utefter vikt. Detta illustreras i två steg nedan.



Varje karaktär kan nu representeras av en binär sträng där en etta signalerar ett steg till vänster och en nolla signalerar ett steg till höger (i trädet). Exempelvis kan karaktären crepresenteras av bitsträngen 10 eftersom att denna nod nås genom att gå till vänster ett steg och sedan till höger ett steg. Hur karaktärerna i detta exempel kan nås illustreras i grafen nedan.



## Kod

### Pseudokod

define Tree

function construct(weight, char)

this.weight = weight

this.char = char

this.left = null

this.right = null

end

function construct(Tree lhs, Tree rhs)

this.weight = lhs.weight + rhs.weight

this.char = 0

this.left = lhs

this.right = rhs

end

function printTree(bitString)

if this.char != 0

print(bitString + " : " + this.char)

end

if this.right != null

bitString.push(0)

this.right.printTree(bitString)

bitString.pop()

end

if this.left != null

bitString.push(1)

this.left.printTree(bitString)

bitString.pop()

end

end

end

function encode(string)

Table charCount

PriorityQueue treeQ

foreach char in string

charCount[char]++

end

foreach char in charCount

treeQ.pushWithPriority(char.value, new Tree(char.value, char.key))

end

while !treeQ.empty

Tree lhs = treeQ.popAndRead()

if treeQ.empty

return lhs

end

Tree rhs = treeQ.popAndRead()

Tree newTree = new Tree(lhs, rhs)

treeQ.pushWithPriority(newTree.weight, newTree)

end

end

### 3.2.2 C++

#### Tree.h

#pragma once

#include <memory>

#include <string>

#include <map>

#include <vector>

#include <iostream>

class Tree

{

public:

Tree(int weight, char c);

Tree(Tree\* t1, Tree\* t2);

int getWeight() const;

void printTree(std::ostream& stream = std::cout) const;

std::map<char, std::vector<bool>> getEncodings();

Tree\* getLeft() const { return m\_Left.get(); };

Tree\* getRight() const { return m\_Right.get(); };

void setLeft(Tree\* left) { m\_Left = std::unique\_ptr<Tree>(left); };

void setRight(Tree\* right) { m\_Right = std::unique\_ptr<Tree>(right); };

char getChar() const { return m\_Char; }

void setChar(char chr) { m\_Char = chr; }

private:

void getEncodings(std::map<char, std::vector<bool>>& val, std::vector<bool> bitString);

void printTree(std::string& bitString, std::ostream& stream) const;

std::unique\_ptr<Tree> m\_Left;

std::unique\_ptr<Tree> m\_Right;

int m\_Weight;

char m\_Char;

}

#### Tree.cpp

#include "Tree.h"

Tree::Tree(int weight, char c)

:m\_Weight(weight)

,m\_Char(c)

{}

Tree::Tree(Tree\* t1, Tree\* t2)

:m\_Weight(!t1 || !t2 ? 0 : t1->m\_Weight + t2->m\_Weight)

,m\_Left(!t1 || !t2 ? nullptr : (t1->m\_Weight > t2->m\_Weight ? t1 : t2))

,m\_Right(!t1 || !t2 ? nullptr : (t1->m\_Weight <= t2->m\_Weight ? t1 : t2))

,m\_Char(0)

{}

int Tree::getWeight() const

{

return m\_Weight;

}

void Tree::printTree(std::ostream& stream) const

{

std::string bitString;

printTree(bitString, stream);

}

void Tree::printTree(std::string& bitString, std::ostream& stream) const

{

if (m\_Char != 0)

{

stream << bitString << ":" << m\_Char << std::endl;

}

if (m\_Right != nullptr)

{

bitString.push\_back('0');

m\_Right->printTree(bitString, stream);

bitString.pop\_back();

}

if (m\_Left != nullptr)

{

bitString.push\_back('1');

m\_Left->printTree(bitString, stream);

bitString.pop\_back();

}

}

void Tree::getEncodings(std::map<char, std::vector<bool>>& val, std::vector<bool> bitString)

{

if (m\_Char != 0)

{

val[m\_Char] = bitString;

}

if (m\_Right != nullptr)

{

std::vector<bool> bitStringR(bitString);

bitStringR.push\_back(false);

m\_Right->getEncodings(val, bitStringR);

}

if (m\_Left != nullptr)

{

std::vector<bool> bitStringL(bitString);

bitStringL.push\_back(true);

m\_Left->getEncodings(val, bitStringL);

}

}

std::map<char, std::vector<bool>> Tree::getEncodings()

{

std::map<char, std::vector<bool>> encodings;

std::vector<bool> temp;

getEncodings(encodings, temp);

return encodings;

}

#### Huffman.h

#pragma once

#include "Tree.h"

#include <vector>

#include <string>

class Huffman

{

public:

/\* When creating a huffman, pass in a string of the

text you want to encode. On construction the object

will then encode the string \*/

Huffman(std::string);

/\* Return the huffman tree of the compression \*/

const Tree\* const getTree() const;

private:

void encode(std::string&);

std::unique\_ptr<Tree> m\_Tree;

};

#### Huffman.cpp

#include "Huffman.h"

#include <queue>

#include <map>

Huffman::Huffman(std::string str)

{

encode(str);

}

const Tree\* const Huffman::getTree() const

{

return m\_Tree.get();

}

void Huffman::encode(std::string& str)

{

std::array<int, 256U> charCount;

std::fill(charCount.begin(), charCount.end(), 0);

m\_Tree.reset();

struct priocomp

{

bool operator() (const std::pair<int, Tree\*>& lhs,

const std::pair<int, Tree\*>& rhs) const

{

return (lhs.first > rhs.first );

}

};

std::priority\_queue<std::pair<int, Tree\*>,

std::vector<std::pair<int, Tree\*>>, priocomp> treeQ;

for (auto chr : str)

{

charCount[chr]++;

}

for (auto count = charCount.begin(); count != charCount.end(); ++count)

{

char chr = std::distance(charCount.begin(), count);

if (\*count != 0)

treeQ.emplace(\*count, new Tree(\*count, chr));

}

while (!treeQ.empty())

{

Tree\* lhs = treeQ.top().second;

treeQ.pop();

if (treeQ.empty())

{

m\_Tree = std::unique\_ptr<Tree>(lhs);

break;

}

Tree\* rhs = treeQ.top().second;

treeQ.pop();

Tree\* newTree = new Tree(lhs, rhs);

treeQ.emplace(newTree->getWeight(), newTree);

}

}

#### Main.cpp

int main()

{

Huffman huff("aaaabbbc");

auto tree = huff.getTree();

tree->printTree();

return 0;

}

## Tidskomplexitet

Komplexitetet för kodningen analyseras rad för rad enligt följande:

function encode(string)

Array charCount[256]

PriorityQueue treeQ

foreach char in string n

charCount[char]++ 1

end = n

foreach char in charCount m

treeQ.pushWithPriority(char.value, new Tree(char.value, char.key))

log m /2

end = m log m /2

while !treeQ.empty m

Tree lhs = treeQ.popAndRead() log m / 2

if treeQ.empty 1

return lhs 1

end

Tree rhs = treeQ.popAndRead() log m / 2

Tree newTree = new Tree(lhs, rhs) 1

treeQ.pushWithPriority(newTree.weight, newTree) log m / 2

end = m \* 2 \* log m / 2 + 3

end = n \* log m / 2 + m log m / 2 +

m \* 2 \* log m / 2 + 3

*n* är storleken på strängen som ska kodas, *m* är antalet unika karaktärer i strängen.

Tidskomplexiteten blir då O(n log n).

Utskrivningen av koderna i trädet är linjärt med antalet noder i huffman-trädet. Dvs. O(n) = n.

## Analys

Det som spelar mest roll för tidskomplexiteten är prioritetskön. Om strängen från början hade varit sorterad så hade komplexiteten varit O(n), men då kräver såklart sorteringen som minst O(n log n) om inte speciella förhållanden föreligger av något skäl. Det man då kan tänka på är att en effektiv prioriteringskö spelar större roll för hur lång tid algoritmen tar än något annat.

# Problem – Tidskomplexitet

Problemet som skall lösas är att jämföra två implementationer av en rekursiv algoritm som skall beräkna tidskomplexiteten T(n). Tidskomplexiteten kan uttryckas som en rekursiv funktion: där basfallet är .

## Intuitiv beskrivning

**Väldigt konstigt skrivet, korrigera detta stycke.**

En rekursiv beräkning av tidskomplexiteten nedan kan lösas på olika sätt. I denna lösning finns både en trivial lösning som för varje rekursion gör en ny beräkning för uttrycket T(n). Det finns även en lösning där metoden dynamisk programmering tillämpas. Detta bygger på att spara ner det senaste beräknade T(n) för att undvika att beräkna uttrycket för samma *n* flera gånger. Istället börjar algoritmen med att se om uttrycket för det nuvarande *n* redan har beräknats. Om så är fallet hämtas värdet och algoritmen går vidare till nästa *n.* Annars beräknas uttrycket och sparas undan för senare eventuella likadana beräkningar.

Det tidskomplexitetsuttryck algoritmen skall beräkna illustreras ovan där är det minsta heltal som är större än .

## Kod

### Pseudokod

#### Uppgift 4.1

function expr(n)

if n = 1

return 1

else

return expr(n - 1) + expr(ceil(n/2)) + n

end

end

#### Uppgift 4.2

Table table

table[1] = 1

function expr(n)

if n not in table

table[n] = expr(n - 1) + expr(ceil(n/2)) + n

end

return table[n]

end

### C++

#### Main.cpp (Uppgft 4.1)

#include <iostream>

#include <cmath>

#include <cassert>

#include <limits>

#include <chrono>

//---- 4.1

template <class T, class U>

typename std::enable\_if<std::numeric\_limits<T>::digits >= std::numeric\_limits<U>::digits, T>::type

ceildiv (T x\_, U y\_)

{

typename std::enable\_if<std::is\_integral<T>::value, T>::type x = x\_;

typename std::enable\_if<std::is\_integral<U>::value, U>::type y = y\_;

assert(x >= 0 && y > 0);

if (x == 0)

return (x + y - 1) / y;

else

return 1 + ((x - 1) / y);

}

template <class T, class U>

typename std::enable\_if<std::numeric\_limits<T>::digits < std::numeric\_limits<U>::digits, U>::type

ceildiv (T x\_, U y\_)

{

typename std::enable\_if<std::is\_integral<T>::value, T>::type x = x\_;

typename std::enable\_if<std::is\_integral<U>::value, U>::type y = y\_;

assert(x >= 0 && y > 0);

if (x == 0)

return (x + y - 1) / y;

else

return 1 + ((x - 1) / y);

}

template <class T>

typename std::enable\_if<std::is\_integral<T>::value, T>::type

complexity (T n)

{

if(n == 1)

return 1;

else

//this is wrong for some values due to floating point imprecision. :(

return complexity(n-1) + complexity(static\_cast<T>(ceildiv(n, 2))) + n;

}

int main()

{

std::cout << "T(10) = " << complexity(short(10)) << std::endl;

std::cout << "T(3) = " << complexity(3) << std::endl;

std::cout << "T(4) = " << complexity(4) << std::endl;

auto now = std::chrono::system\_clock::now();

for (int i = 0; i < 10000; ++i)

{

complexity(i % 100 + 1);

}

auto then = std::chrono::system\_clock::now();

auto diff = std::chrono::duration\_cast<std::chrono::microseconds>(then - now).count();

std::cout << "100000 complexities took: " << diff << std::endl;

return 0;

}

#### Complexity\_table.h (uppgift 4.2)

#pragma once

#include <type\_traits>

#include <map>

#include <cassert>

namespace complexity

{

namespace detail

{

template <class T, class U>

typename std::enable\_if<std::numeric\_limits<T>::digits >= std::numeric\_limits<U>::digits, T>::type

ceildiv (T x\_, U y\_)

{

typename std::enable\_if<std::is\_integral<T>::value, T>::type x = x\_;

typename std::enable\_if<std::is\_integral<U>::value, U>::type y = y\_;

assert(x >= 0 && y > 0);

if (x == 0)

return (x + y - 1) / y;

else

return 1 + ((x - 1) / y);

}

template <class T, class U>

typename std::enable\_if<std::numeric\_limits<T>::digits < std::numeric\_limits<U>::digits, U>::type

ceildiv (T x\_, U y\_)

{

typename std::enable\_if<std::is\_integral<T>::value, T>::type x = x\_;

typename std::enable\_if<std::is\_integral<U>::value, U>::type y = y\_;

assert(x >= 0 && y > 0);

if (x == 0)

return (x + y - 1) / y;

else

return 1 + ((x - 1) / y);

}

template <class T>

typename std::enable\_if<std::is\_integral<T>::value, T>::type

complexity (T n)

{

if(n == 1)

return 1;

else

return complexity(n-1) + complexity(static\_cast<T>(ceildiv(n, 2))) + n;

}

}

template<class T, class Enable = void>

class complexity\_table; // undefined

template <class T>

class complexity\_table<T, typename std::enable\_if<std::is\_integral<T>::value>::type>

{

public:

T complexity(T);

inline T operator()(T);

private:

std::map<T, T> m\_Table;

};

template <class T>

T complexity\_table<T, typename std::enable\_if<std::is\_integral<T>::value>::type>::operator()(T n)

{

return complexity(n);

}

template <class T>

T complexity\_table<T, typename std::enable\_if<std::is\_integral<T>::value>::type>::complexity(T n)

{

auto find = m\_Table.find(n);

if (find == m\_Table.end())

{

T val = detail::complexity(n);

m\_Table.insert(std::make\_pair(n, val));

return val;

}

else

return find->second;

}

}

#### Main.cpp (uppgift 4.2)

#include <iostream>

#include "complexity\_table.h"

#include <chrono>

int main()

{

complexity::complexity\_table<int> complexity;

std::cout << "T(10) = " << complexity(short(10)) << std::endl;

std::cout << "T(3) = " << complexity(3) << std::endl;

std::cout << "T(4) = " << complexity(4) << std::endl;

auto now = std::chrono::system\_clock::now();

for (int i = 0; i < 10000; ++i)

{

complexity(i % 100 + 1);

}

auto then = std::chrono::system\_clock::now();

auto diff = std::chrono::duration\_cast<std::chrono::microseconds>(then - now).count();

std::cout << "100000 complexities took: " << diff << std::endl;

return 0;

}

## Tidskomplexitet

## Analys

Den första lösningen räknar ut visa värden för T(*n*) flera, specifikt för de *n* som fås av för det specifika förstavärdet på *n*, vilket även gäller rekursivt. Eftersom att T(*n*) i sig är rekursiv är det extra kostsamt att räkna ut dessa flera gånger. Att då spara undan redan uträknade värden i en tabell

Vi har löst problemet top- down, dvs. ett värde för t(*n*) räknas inte ut förrän det behövs. Eftersom att T(*n - 1*) ingår i den rekursiva funktionen vet vi att alla T(*k*) kommer att räknas ut hade det gått lika bra att gör det bottom-up, eftersom att samma T(*n*) kommer att räknas ut i vilket fall. Top-down passar bättre för algoritmer där inte alla utfall behöver räknas ut, eller om det inte går att veta vilka utfall som räknas ut från början.

Ett resonemang och ett exempel som visar varför det blir skillnad. Det

räcker inte med att ni har konstaterat genom mätningar att en av algoritmerna

ar snabbare. Ni Måste ha med ett resonemang som visar att ni har

förstått varför det blir skillnad.

# Problem – Handelsresandeproblemet

Handelsresandeproblemet går ut på att hitta den kortaste vägen genom alla noder i en viktad graf, utan att passera samma nod mer än en gång (en Hamiltonväg m.a.o.).

## Intuitiv beskrivning

Den enklaste algoritmen är nearest neighbour, vilken är en girig algoritm. Algoritmen väljer ut en nod som startpunkt, och går sedan till den granne som är närmast (vars kant har lägst vikt) och gör detta succesivt med den grannen som ny nod, tills alla noder har besökts. Utifrån detta skapas en väg, för att vägen sen ska bli en full cykel kopplas startnoden och slutnoden ihop. Eftersom att olika val av startnod ger olika totallängd för vägen används här alla noder i grafen som startnod. På så sätt ges lika många olika vägar som det finns noder, och den kortaste vägen av dessa väljs ut.

## Kod

## Tidskomplexitet

## Analys