Algoritmer och datastrukturer

Laborationsuppgift, 1.5hp

DA346G HT13

Jon Wahlström

19901213

[b12jonwa@student.his.se](mailto:b12jonwa@student.his.se)

Sebastian Zander

19870318

[a12sebza@student.his.se](mailto:a12sebza@student.his.se)

Institutionen för Information och Kommunikation

Högskolan i Skövde

Innehållsförteckning

[Inledning 3](#_Toc376437979)

[1. Modifierad Bucketsort 4](#_Toc376437980)

[1.1 Intuitiv beskrivning 4](#_Toc376437981)

[1.2 Kod 5](#_Toc376437982)

[1.2.1 Pseudokod 5](#_Toc376437983)

[1.2.2 C++ 6](#_Toc376437984)

[1.3 Tidskomplexitetsanalys 7](#_Toc376437985)

[1.4 Analys 8](#_Toc376437986)

[2. Sociala nätverk 9](#_Toc376437987)

[2.1 Intuitiv beskrivning 9](#_Toc376437988)

[2.2 Kod 10](#_Toc376437989)

[2.2.1 Pseudokod 10](#_Toc376437990)

[2.2.2 C++ 11](#_Toc376437991)

[2.3 Tidskomplexitetsanalys 14](#_Toc376437992)

[2.4 Analys 14](#_Toc376437993)

[3. Komprimering genom Huffman-kodning 15](#_Toc376437994)

[3.1 Intuitiv beskrivning 15](#_Toc376437995)

[3.2 Kod 16](#_Toc376437996)

[3.2.1 Pseudokod 16](#_Toc376437997)

[3.2.2 C++ 17](#_Toc376437998)

[3.3 Tidskomplexitetsanalys 21](#_Toc376437999)

[3.4 Analys 21](#_Toc376438000)

[4. Tidskomplexitet 22](#_Toc376438001)

[4.1 Intuitiv beskrivning 22](#_Toc376438002)

[4.2 Kod 22](#_Toc376438003)

[4.2.1 Pseudokod 22](#_Toc376438004)

[4.2.2 C++ 23](#_Toc376438005)

[4.3 Tidskomplexitetsanalys 27](#_Toc376438006)

[4.4 Analys 27](#_Toc376438007)

[5. Handelsresandeproblemet 28](#_Toc376438008)

[5.1 Intuitiv beskrivning 28](#_Toc376438009)

[5.2 Kod 29](#_Toc376438010)

[5.2.1 Pseudokod 29](#_Toc376438011)

[5.2.2 C++ 30](#_Toc376438012)

[5.3 Tidskomplexitetsanalys 35](#_Toc376438013)

[5.4 Analys 35](#_Toc376438014)

# Inledning

Laborationens syfte är att få en bredare bekantskap med olika typer av algoritmer och datastrukturer. Bekantskapen ges genom att lösa fem olika typer av problem, implementera dem samt analysera en tidskomplexitet för dem. De fem uppgifter som skall genomföras är:

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

Detta är en praktisk övning för den teori kursen tagit upp där fokus snarare ligger på problemlösningen än implementationen.

# Modifierad Bucketsort

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

## Intuitiv beskrivning

Denna algoritm bygger på att partitionera en given mängd element i hinkar. Utifrån det placeras sedan värdet från elementet storleksmässigt i hinkarna med minsta värdet först.



Figur 1: Given array som skall sorteras

Antalet hinkar som skapas beror på det största elementet i den array som skall sorteras. Om det största elementet är *k* i denna array kommer det att skapas *k+1* antal hinkar.



Figur 2: Antalet hinkar som skapas utefter största elementet i array

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



Figur 3: Placering av element i hinkarna, sorterade 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;

}

## Tidskomplexitetsanalys

Genom att analysera algoritmen steg för steg har följande tidskomplexitet framtagits,   
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 hinkar.

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

I det specifika fallet där m och n får följande värden:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| m | 25 | 36 | 49 | 64 | 81 | 100 |
| n | 5 | 6 | 7 | 8 | 9 | 10 |

I det fallet gäller att tidskomplexiteten är O(n²) eftersom att m alltid är n² i tabellen ovan. Eftersom att m + n i så fall är n² + n och n² är det mest signifikanta gäller att O(n²).

## 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 hinkar för att sortera två element. Ett sätt att lösa detta skulle vara att skapa en hink för varje unikt värde som finns i listan och mappa den hinken till det värdet.

# 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.



Figur 4: Ett exempel på en fiendegraf.

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.



Figur 5: Trädet som byggs upp från grafen i Figur 4 ovan.

## Kod

### 2.2.1 Pseudokod

def Node

List:Node enemies

end

function ListFriends(Node node)

List current\_level

List next\_level

List friends

Set 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;

}

## Tidskomplexitetsanalys

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(e \* n log n), med en del förkortningar som beskrivet nedan.

def Node

List:Node enemies

end

function ListFriends(Node node)

List current\_level

List next\_level

List friends

Set 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 log 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, d.v.s. antalet utåtvertriser. *n\*e* kommer då bli det totala antalet vertriser i grafen (*v*). Dessa tre loopar kan därmed 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*,* alltså kommer m i genomsnitt vara n/2. Tidskomplexiteten beror då på antalet noder i grafen, *n*, samt antalet vertriser i grafen (rättare sagt hur många fiender genomsnittsnoden har, *e*).

## 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 samt att nya element läggs till i slutet. Den huvudsakliga poängen är att vi har gjort sökningen med bredden först.

# 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 nod med högst vikt läggs först.



Figur 6: Exempel på noder kopplade till frekvensen av karaktärer i den givna stängen

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 att listan fortfarande bevarar en sorterad ordning utefter vikt. Detta illustreras i två steg nedan.

 

Figur 7: Ett frekvenssorterat träd byggs upp steg för steg.

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.



Figur 8: Det slutliga frekvenssorterade trädet.

## 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();

}

## Tidskomplexitetsanalys

Komplexitetet för kodningen analyseras rad för rad enligt följande:

function encode(string)

Table charCount

PriorityQueue treeQ

foreach char in string n

charCount[char]++ log m / 2

end = n log m / 2

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

end = n (log m / 2) + m (log m / 2) +

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

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

Tidskomplexiteten blir då O(n log m + m log m).

Utskrivning av Huffman-trädet är O(m). Detta eftersom att antalet noder i ett (perfekt) binärt träd är 2l – 1 där antalet löv-noder är l. Eftersom att antalet löv-noder i Huffmanträdet är detsamma som antalet unika karaktärer i den kodade stängen beror utskrivningen på m. Ett Huffmanträd är dock inte ett perfekt binärt träd (i de flesta fall), men det är balanserat, så antalet noder är aldrig fler än 2m - 1.

## Analys

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

# 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 . är det minsta heltal som är större än .

## Intuitiv beskrivning

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.

## 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;

}

## Tidskomplexitetsanalys

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

Eftersom att T(*n*) bara räknas ut en gång för varje unikt värde av *n* kommer rekursionen bara ske *n* gånger. T(*n – 1*) respektive T() i uttrycket kommer att räknas ut sammanlagt *n* gånger under en rekursion. Detta eftersom att för några av T(*n - 1*) kommerT() redan ha beräknat. I de fall T(*n*) redan har räknats ut kommer komplexiteten för att slå upp värdet i tabellen vara *log n*, förutsatt att tabellen är implementerat som ett binärt sökträd.

Tidskomplexiteten för denna algoritm kan i stora Oh-notation alltså skrivas som O(n log n).

## Analys

Den första lösningen räknar ut vissa värden för T(*n*) flera gånger, 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 sparar beräkningar.

För att exemplifiera det kan T(5) användas. I det fallet kommer T(n – 1) att räknas ut i ordningen:

T(4), T(3), T(2).

T() kommer att räknas ut i följande ordning:

T(3), T(2).

Om algoritmen löses dynamiskt kommer beräkningarna T(3) och T(2) bara att räknas ut en gång istället för två gånger, alltså sparas två beräkningar. T(n – 1)-delen kommer i så fall att använda resultatet av T(3) och T(2) från tidigare beräkning från T()-beräkning.

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, 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.

# 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. Detta kallas en Hamiltonväg.

## 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.



Figur 9: Noder i grafen

Grafen bildas av noderna ovan genom att koppla en kant mellan varje nod där vikten är det kartesiska avståndet mellan noderna. Låt säga att utgångsnoden är nod 0 i ovanstående graf. Algoritmen kommer söka igenom alla andra noder och räkna ut att den närmaste noden är nod 1.



Figur 10: Vägen från startnod till dess närmsta granne

Detta fortsätter och algoritmen kommer att hitta den närmaste grannen till nod 1, vilket är nod 3. Sedan kommer den fortsätta till nod 4, nod 5, nod 2 till nod 6 för att till sist återgå till startnoden.



Figur 11: Slutgiltig Hamiltonväg genom grafen.

Det går tydligt att se att detta inte är den kortaste Hamiltonvägen i grafen. Algoritmen kan inte hitta en kortare väg med utgångspunkt från nod 0. Däremot är det som sagt möjligt att testa algoritmen med samtliga noder som startnod för att eventuellt hitta en kortare väg. Dock finns det ingen garanti att hitta den kortaste Hamiltonvägen trots detta.

## Kod

### Pseudokod

function NearestNeighbour(Graph graph)

var total\_weight = #INF

Path ret

foreach node in graph

Set unvisited = graph.get\_nodes

var current\_total\_travelling\_cost = 0

var lowest\_total\_travelling\_cost = #INF

Path path

var start\_node = node;

var current\_node = start\_node;

Edge next\_edge(0, 0)

unvisited.remove(current\_node)

while unvisited is not empty ->

var lowest\_travelling\_cost = #INF

Edge current\_edge(current\_node, 0)

Set neighbours = graph.get\_neighbours(current\_node)

foreach neighbour of current\_node in unvisited

current\_edge.end\_node = neighbour;

if travelling\_cost from current\_node to neighbour < lowest\_travelling\_cost

next\_edge = current\_edge

lowest\_travelling\_cost = cost to neighbour

end

end

current\_total\_travelling\_cost += lowest\_travelling\_cost

current\_node = next\_edge.end\_node

path.push(next\_edge)

unvisited.remove(current\_node)

end

Edge join\_path(path.last.end\_node, path.first.start\_node

path.push(join\_path)

current\_total\_travelling\_cost += join\_path.travelling\_cost

if current\_total\_travelling\_cost < lowest\_total\_travelling\_cost

lowest\_total\_travelling\_cost = current\_total\_travelling\_cost

ret = path

end

end

return ret

end

### C++

#### <Edge.h>

#pragma once

#include <algorithm>

namespace tsp

{

class Edge

{

public:

Edge(int start\_node, int end\_node)

:m\_start\_node(start\_node)

,m\_end\_node(end\_node)

{}

void set\_start\_node(int start) { m\_start\_node = start; }

void set\_end\_node(int end) { m\_end\_node = end; }

int get\_start\_node() const { return m\_start\_node; }

int get\_end\_node() const { return m\_end\_node; }

void balance()

{

int old\_start\_node = m\_start\_node;

m\_start\_node = std::min(old\_start\_node, m\_end\_node);

m\_end\_node = std::max(old\_start\_node, m\_end\_node);

}

inline friend bool operator<(const Edge& lhs, const Edge& rhs)

{

return (lhs.m\_start\_node < rhs.m\_start\_node) || (!(rhs.m\_start\_node < lhs.m\_start\_node)) && lhs.m\_end\_node < rhs.m\_end\_node;

}

private:

int m\_start\_node;

int m\_end\_node;

};

}

#### <Graph.h>

#pragma once

#include <set>

#include <map>

#include <exception>

#include <utility>

#include "Region.h"

#include "Edge.h"

namespace tsp

{

struct Path

{

public:

void push(Edge edge) { m\_path.push\_back(edge); }

std::vector<Edge> get\_path() const { return m\_path; }

private:

std::vector<Edge> m\_path;

};

template <class T>

class Graph

{

public:

Graph() {}

Graph(tsp::Region<T>& region);

void set(Edge edge, T weight);

T get\_weight(Edge edge) const;

std::set<int> get\_neighbours(int) const;

std::set<int> get\_nodes() const;

std::map<Edge, T> get\_graph() const;

private:

std::map<Edge, T> m\_map;

};

template <class T>

Graph<T>::Graph(tsp::Region<T>& region)

{

for(int i = 0; i < region.size() - 1; ++i)

{

for(int j = i + 1; j < region.size(); ++j)

{

set(Edge(i, j), region.distance(region[i], region[j]));

}

}

}

template <class T>

void Graph<T>::set(Edge edge, T weight)

{

edge.balance();

m\_map[edge] = weight;

}

template <class T>

T Graph<T>::get\_weight(Edge edge) const

{

edge.balance();

auto find = m\_map.find(edge);

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

throw std::out\_of\_range("Edge not found");

return find->second;

}

template <class T>

std::set<int> Graph<T>::get\_neighbours(int node) const

{

std::set<int> ret;

for(auto itr : m\_map)

{

if(itr.first.get\_start\_node() == node)

ret.insert(itr.first.get\_end\_node());

if(itr.first.get\_end\_node() == node)

ret.insert(itr.first.get\_start\_node());

}

return ret;

}

template <class T>

std::set<int> Graph<T>::get\_nodes() const

{

std::set<int> ret;

for(auto node : m\_map)

{

ret.insert(node.first.get\_start\_node());

ret.insert(node.first.get\_end\_node());

}

return ret;

}

template <class T>

std::map<Edge, T> Graph<T>::get\_graph() const

{

return m\_map;

}

}

#### <Region.h>

#pragma once

#include <vector>

#include <cmath>

namespace tsp

{

template <class T>

class Region

{

public:

void add\_city(T x, T y);

T distance(std::pair<T, T> c0, std::pair<T, T> c1) const;

int size() { return static\_cast<int>(m\_city\_coords.size()); }

std::pair<T, T> operator[](int index) { return m\_city\_coords[index]; }

private:

std::vector<std::pair<T, T>> m\_city\_coords;

};

template <class T>

void Region<T>::add\_city(T x, T y)

{

m\_city\_coords.push\_back(std::make\_pair(x, y));

}

template <class T>

T Region<T>::distance(std::pair<T, T> c0, std::pair<T, T> c1) const

{

T x = c0.first - c1.first;

T y = c0.second - c1.second;

T ret = std::sqrt(x \* x + y \* y);

return std::sqrt(x \* x + y \* y);

}

};

#### <NearestNeighbour.h>

#pragma once

#include "Graph.h"

#include <vector>

#include <numeric>

namespace tsp

{

namespace algorithm

{

template <class T>

tsp::Path NearestNeighbour(Graph<T> graph)

{

std::set<int> nodes = graph.get\_nodes();

T total\_weight = std::numeric\_limits<T>::max();

tsp::Path ret;

for(auto count : nodes)

{

T current\_total\_weight = 0;

tsp::Path path;

std::set<int> unvisited;

int start\_node = count;

int current\_node = start\_node;

for(auto node : nodes)

{

unvisited.insert(node);

}

tsp::Edge next\_edge(0, 0);

unvisited.erase(current\_node);

while(!unvisited.empty())

{

T min\_weight = std::numeric\_limits<T>::max();

tsp::Edge current\_edge(current\_node, 0);

std::set<int> neighbours = graph.get\_neighbours(current\_node);

for(auto neighbour : neighbours)

{

if(unvisited.count(neighbour) == 0)

continue;

current\_edge.set\_end\_node(neighbour);

if(graph.get\_weight(current\_edge) < min\_weight)

{

next\_edge = current\_edge;

min\_weight = graph.get\_weight(current\_edge);

}

}

current\_total\_weight += min\_weight;

current\_node = next\_edge.get\_end\_node();

path.push(next\_edge);

unvisited.erase(current\_node);

}

path.push(Edge(path.get\_path().back().get\_end\_node(), path.get\_path().front().get\_start\_node()));

current\_total\_weight += graph.get\_weight(path.get\_path().back());

if(current\_total\_weight < total\_weight)

{

total\_weight = current\_total\_weight;

ret = path;

}

}

return ret;

}

}

}

#### <Main.cpp>

#include "Graph.h"

#include "NearestNeighbour.h"

#include <iostream>

int main()

{

tsp::Region<float> region;

region.add\_city(15, 30);

region.add\_city(43, 18);

region.add\_city(95, 80);

region.add\_city(15, 65);

region.add\_city(0, 60);

region.add\_city(47, 55);

region.add\_city(32, 27);

tsp::Graph<float> graph(region);

auto path = tsp::algorithm::NearestNeighbour(graph);

float total\_weight = 0;

for(auto edge : path.get\_path())

{

total\_weight += graph.get\_weight(edge);

std::cout << edge.get\_start\_node() << " -> " << edge.get\_end\_node() << " (" << graph.get\_weight(edge) << ")" << std::endl;

}

std::cout << "Total weight: " << total\_weight << std::endl;

return 0;

}

## Tidskomplexitetsanalys

Tidskomplexiteten för algoritmen är O(n³). För varje nod hittar algoritmen den närmaste grannen (*n*). Detta görs för varje nod i grafen till en komplett väg har hittats (*n*). Detta i sin tur görs för varje nod i grafen som startnod (*n*).

För varje kontroll att hitta den nuvarande nodens grannar måste algoritmen hålla koll på vilka av grannarna som inte redan ingår i vägen. Detta kan göras linjärt så det påverkar inte tidskomplexiteten.

## Analys

Att kontrollera vilka noder som redan ingår i vägen kan göras på olika sätt där det bästa vore om det kan göras linjärt. En linjär lösning kan göras exempelvis med hjälp av en hash-tabell eller genom att markera noder som besökts.

Den mest naiva *nearest neighbour* har tidskomplexiteten n². Vår lösning får istället n³ eftersom att varje nod i grafen testas som startnod. Detta kommer att öka tiden för uträkningen dramatiskt (dock blir det fortfarande mindre än mer exakta lösningar) men algoritmen kommer hitta den kortaste vägen som går att hitta via *nearest neighbour*. Detta har båda för- och nackdelar, vi ansåg att fördelarna vägde upp nackdelarna i det här fallet. En av de stora nackdelarna är då den algoritmens kortaste möjliga väg hittas redan under den första iterationen av noder som startnod. Detta kommer att leda till en stor mängd onödiga beräkningar, men samtidigt går detta inte att veta om och när detta kommer att ske. Det är helt enkelt övervägning mellan effektivitet och exakthet.

Oavsett hur en sådan girig algoritm implementeras kommer den aldrig med säkerhet hitta den absolut kortaste vägen. Till och med är det så att för varje antal städer finns det en graf där algoritmen ger den sämsta möjliga vägen, vilket är ett stort problem för algoritmen.

Det finns väldigt många sätt att representera en graf som kan användas för den här algoritmen. Vi har valt att använda en key-value representation, med en uppslagningstabell som mappar ett par av noder till en längd. Denna representation är i grunden asymmetriskt, men vi har gjort den symmetrisk genom att alltid sätta den lägsta noden (om noder representeras av ett heltal) som det högra elementet i paret, i C++-koden. Det beror på domänen om grafen bör vara symmetrisk eller inte.

Key-value passar bäst om grafen inte är tät, med andra ord om det finns få kopplingar mellan noder. Detta är inte nödvändigtvis fallet för problemet (det beror på domänen).

En annan representation är genom en grannmatris, där varje element i matrisen representerar vikten mellan noden i elementets kolon och noden i elementets rad. En sådan representation passar bäst om grafen är tät, dessutom är representationen också cache-vänlig. Om grafen inte är tät och det finns väldigt många noder kräver matrisen väldigt mycket onödigt minne (n²). En key-value representation kräver dock generellt sätt mer minne om grafen är extremt tät än en matris-representation, men minnet som behövs beror på antalet kopplingar istället för antalet noder.

En otät graf kan representeras som en matris utan att ta upp onödigt minne om en icke-naiv implementation används.

Representation av grafen påverkar till stor del hur mycket minne algoritmen kommer att använda. Snabbheten påverkas också av representationen, dels genomsökningen och hur element i grafen hittas, men även cachen kan spela en roll vilket kan vara viktigt.

Problemet kan parallelliseras, genom att exempelvis köra algoritmen för varje nod som startnod i separata trådar och därefter jämföra resultatet från varje tråd. Detta ökar snabbheten för sökningen.