

**课 程 实 验 报 告**

**课程名称： 算法设计与分析**

**专业班级： 计算机科学与技术（校际交流）1601班**

**学 号： U201610136**

**姓 名： 朱晓光**

**指导教师：**

**报告日期： 2018年11月14日**

**计算机科学与技术学院**

**目录**

[一、实验情况总览 3](#_Toc530065427)

[二、解题报告 5](#_Toc530065428)

[2.1 最近点对解题报告 5](#_Toc530065429)

[2.2 大整数相乘 12](#_Toc530065430)

[2.3 最优二分查找树 23](#_Toc530065431)

[2.4 Floyd-Warshall算法 27](#_Toc530065432)

[三、心得体会 31](#_Toc530065433)

# 一、实验情况总览

实验完成情况如表1.1所示。

表1.1 实验完成情况表

|  |  |
| --- | --- |
| **实验题目** | **完成情况** |
| 最近点对 | 完成 |
| 大整数相乘 | 完成 |
| 最优二分查找树（15.5-1） | 完成 |
| Floyd-Warshall算法 | 完成 |

实验测试结果如下所示。

1. 最近点对

测试样例：

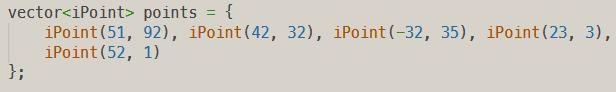


图1.1 最近点对实验测试样例

其中类型iPoint的两个参数分别为该点的横、纵坐标。

算法输出：



图1.2 最近点对算法输出截图

2. 大整数相乘

测试样例：

1） a = 9223372036854775807  
b = 1234567891111  
c = a \* b = 11386878964471969137416693151577

2） a = c  
c = a \* a \* a = 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

3） c = 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033  
d = c + c = 2952868512670402039011831905072118082638114412799042428209387314244123578334594106375861874066

4） e = c - d = -1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

算法输出：

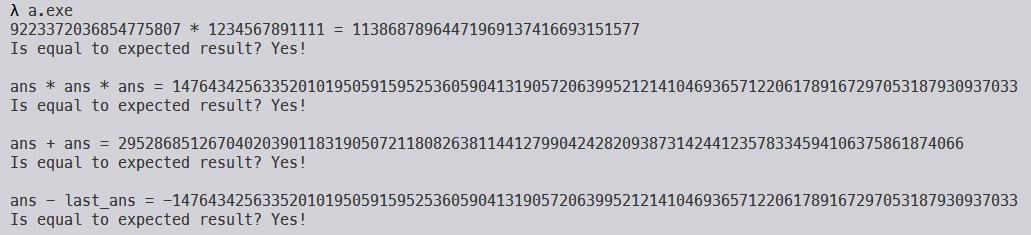


图1.3 大整数相乘算法输出截图

3. 最优二分查找树

测试样例：

测试样例直接选用《算法导论》题目Ch15.5-1中所指定的数据。

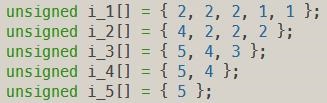


图1.4 最优二分查找树测试样例

算法输出：

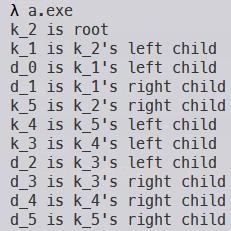


图1.5 最优二分查找树算法输出截图

4. Floyd-Warshall算法

测试样例：

该实验测试样例以邻接矩阵的形式给出。

1） 测试样例1如图1.6所示。

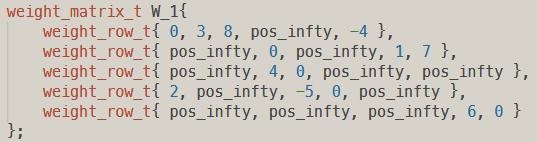


图1.6 Floyd-Warshall算法测试样例1

2） 测试样例2如图1.7所示。

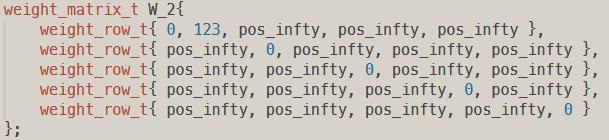


图1.7 Floyd-Warshall算法测试样例2

算法输出：

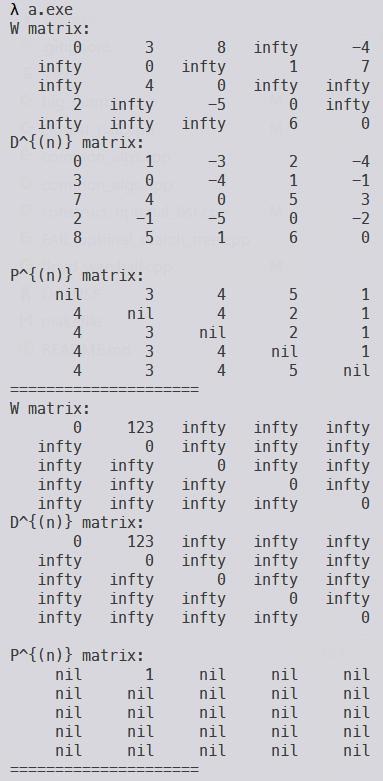


图1.8 Floyd-Warshall算法输出截图

从上至下依次为对测试样例1、测试样例2的输出结果。

# 二、解题报告

## 2.1 最近点对解题报告

2.1.1 题目描述

实现最近点对算法。

2.1.2 算法设计

算法通过分治思想求解最近点对问题，将问题集分为左右两部分，并对左右分别递归调用，通过合并步骤找出横跨左右两子集的解。算法关键部分在于通过证明发现在合并步骤中，对某一个点，只需考虑其附近的点即可，总而降低了合并步骤的复杂度。算法如图2.1所示。

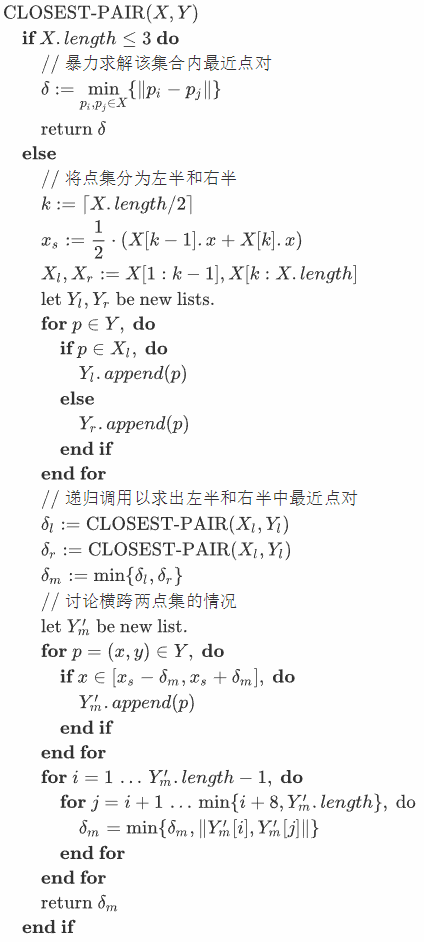


图2.1 最近点对算法

初始时，将问题点集*P*分别按点的*x*坐标和*y*坐标排序形成有序数列*X*, *Y*后调用CLOSEST-PAIR(*X*, *Y*)。

2.1.3 源程序及注释

#include <iostream>

using std::cout; using std::cin; using std::endl;

#include <string>

using std::string; using std::to\_string;

#include <vector>

using std::vector;

#include <array>

using std::array;

#include <cmath>

using std::pow; using std::sqrt;

#include <algorithm>

using std::sort;

template <class DATA\_T>

class Point {

const DATA\_T x;

const DATA\_T y;

public:

Point(DATA\_T x, DATA\_T y) : x(x), y(y) { return; }

virtual ~Point(void) { return; }

virtual Point &operator =(const Point &other) {

\*(DATA\_T \*)&this->x = other.x;

\*(DATA\_T \*)&this->y = other.y;

return \*this;

}

virtual DATA\_T getX(void) const { return this->x; }

virtual DATA\_T getY(void) const { return this->y; }

virtual string str(void) const {

return (

string("(") + to\_string(this->x) + string(", ")

+ to\_string(this->y) + string(")")

);

}

static bool leqX(

const Point<DATA\_T> &lhs,

const Point<DATA\_T> &rhs

) {

return lhs.x <= rhs.x;

}

static bool leqY(

const Point<DATA\_T> &lhs,

const Point<DATA\_T> &rhs

) {

return lhs.y <= rhs.y;

}

static double getL2Distance(

const Point<DATA\_T> &lhs,

const Point<DATA\_T> &rhs

) {

return sqrt(

pow(lhs.getX() - rhs.getX(), 2)

+ pow(lhs.getY() - rhs.getY(), 2)

);

}

};

typedef Point<int> iPoint;

class ClosestPair {

vector<iPoint> points = {

iPoint(51, 92), iPoint(42, 32), iPoint(-32, 35), iPoint(23, 3),

iPoint(52, 1)

};

public:

static bool isIn(

iPoint \*key,

vector<iPoint \*> pool

) {

for (auto &point : pool) {

if (key == point) { return true; }

}

return false;

}

static bool isInVertBand(

iPoint \*ppoint,

double mid\_x,

double delta

) {

int x\_coord = ppoint->getX();

return (

(x\_coord >= mid\_x - delta)

&& (x\_coord <= mid\_x + delta)

);

}

static double getClosestPair(

vector<iPoint \*> x\_sorted,

vector<iPoint \*> y\_sorted

);

public:

ClosestPair(void) { return; }

ClosestPair(vector<iPoint> points) : points(points) { return; }

~ClosestPair(void) { return; }

void solve(void) {

// get pre-sort x-array and y-array

// HACK: use address to uniquely identify Point

vector<iPoint \*> x\_sorted;

sort(this->points.begin(), this->points.end(), iPoint::leqX);

for (auto &point : this->points) {

x\_sorted.push\_back(&point);

}

vector<iPoint \*> y\_sorted;

sort(this->points.begin(), this->points.end(), iPoint::leqY);

for (auto &point : this->points) {

y\_sorted.push\_back(&point);

}

cout << "Distance between closest pair: "

<< ClosestPair::getClosestPair(x\_sorted, y\_sorted) << endl;

return;

}

};

double ClosestPair::getClosestPair(

vector<iPoint \*> x\_sorted,

vector<iPoint \*> y\_sorted

) {

// simplest case return

unsigned range = x\_sorted.size();

if (range <= 3) { // NOTE: promised larger than 1

double dist\_0\_1 = iPoint::getL2Distance(\*x\_sorted[0], \*x\_sorted[1]);

if (range == 2) {

return dist\_0\_1;

}

double dist\_0\_2 = iPoint::getL2Distance(\*x\_sorted[0], \*x\_sorted[2]),

dist\_1\_2 = iPoint::getL2Distance(\*x\_sorted[1], \*x\_sorted[2]);

double min\_dist = (dist\_0\_1 < dist\_0\_2) ? dist\_0\_1 : dist\_0\_2;

min\_dist = (min\_dist < dist\_1\_2) ? min\_dist : dist\_1\_2;

return min\_dist;

}

// split problem set

// make pl and pr, i.e., xl and xr

unsigned split\_idx = (range - 1) / 2 + 1; // NOTE: split\_idx goes to right

double split\_x\_coord = static\_cast<double>(

x\_sorted[split\_idx - 1]->getX()

+ x\_sorted[split\_idx]->getX()

) / 2.0;

vector<iPoint \*> xl, xr;

for (unsigned idx = 0; idx != split\_idx; ++idx) {

xl.push\_back(x\_sorted[idx]);

}

for (unsigned idx = split\_idx; idx != range; ++idx) {

xr.push\_back(x\_sorted[idx]);

}

// make yl and yr, keep order

vector<iPoint \*> yl, yr;

for (auto &ppoint : y\_sorted) {

if (ClosestPair::isIn(ppoint, xl)) {

yl.push\_back(ppoint);

} else {

yr.push\_back(ppoint);

}

}

// recursive call

double delta\_l = ClosestPair::getClosestPair(xl, yl);

double delta\_r = ClosestPair::getClosestPair(xr, yr);

double min\_delta = (delta\_l < delta\_r) ? delta\_l : delta\_r;

// merge step

// focus on points in vertical band

vector<iPoint \*> y\_in\_range;

for (auto &ppoint : y\_sorted) {

if (ClosestPair::isInVertBand(ppoint, split\_x\_coord, min\_delta)) {

y\_in\_range.push\_back(ppoint);

}

}

unsigned y\_in\_range\_len = y\_in\_range.size();

unsigned y\_in\_range\_one\_less\_len = y\_in\_range\_len - 1;

// find minimum with delta\_mid considered

for (unsigned offset = 0; offset != y\_in\_range\_one\_less\_len; ++offset) {

for (unsigned other\_offset = offset + 1,

\_\_virtrange = other\_offset + 8,

other\_range = (

(\_\_virtrange < y\_in\_range\_len) ?

(\_\_virtrange) : y\_in\_range\_len

); other\_offset != other\_range; ++other\_offset) {

double delta\_pair = iPoint::getL2Distance(

\*y\_in\_range[offset],

\*y\_in\_range[other\_offset]

);

if (delta\_pair < min\_delta) {

min\_delta = delta\_pair;

}

}

}

return min\_delta;

}

/\*\*\*\*\*\*\* Main \*\*\*\*\*\*\*/

int main(void) {

vector<iPoint> points = {

iPoint(51, 92), iPoint(42, 32), iPoint(-32, 35), iPoint(23, 3),

iPoint(52, 1)

};

ClosestPair(points).solve();

return 0;

}

2.1.4 本题小结

本题主要难点在于如何有效利用算法的预排序性质降低每次递归调用的复杂度，以及证明对横跨左右子集的点对只需考虑在有限范围内的子集即可。本题充分体现了分治算法的思想，通过巧妙地设置分解和合并步骤，算法能够以更低的复杂度解决问题。

## 2.2 大整数相乘

2.2.1 题目描述

利用分治法设计一个计算两个n位的大整数相乘的算法，要求计算时间低于O(n2)。

大整数（big integer）：位数很多的整数，普通的计算机不能直接处理，如：9834975972130802345791023498570345

对大整数的算术运算，显然常规程序语言是无法直接表示的。编程实现大整数的加、减、乘运算，需考虑操作数为0、负数、任意位等各种情况。

2.2.2 算法设计

可以将大数位数对齐后，分高位和低位进行分治，通过合并步骤获得结果中间位数的结果。算法如图2.2所示。

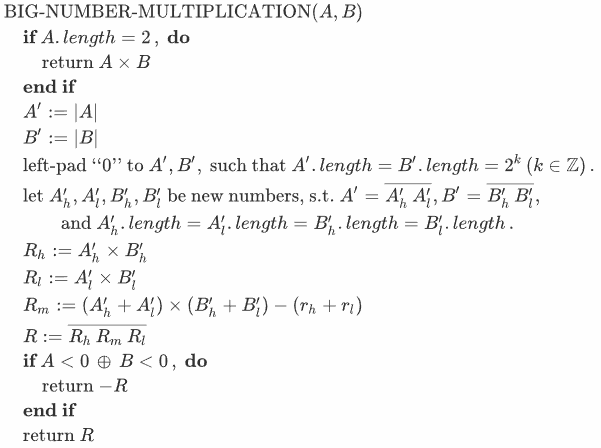


图2.2 大整数相乘算法

初始时，对两大数*A*, *B*调用BIG-NUMBER-MULTIPLICATION(*A*, *B*)。

2.2.3 源代码及注释

#include <iostream>

#include <stdexcept>

#include <string>

#include <list>

class BigNumber {

std::string number;

private:

static void \_makeLengthEven(BigNumber &lhs, BigNumber &rhs, bool round = false) {

if (!lhs.isPositive() || !rhs.isPositive()) {

throw std::runtime\_error(

std::string("BigNumber::\_makeLengthEven : ")

+ std::string("Operands must all be positive")

);

}

int len\_diff = lhs.getLengthDiffAgainst(rhs);

if (len\_diff < 0) {

lhs.number = std::string(-len\_diff, '0') + lhs.number;

} else if (len\_diff > 0) {

rhs.number = std::string(len\_diff, '0') + rhs.number;

}

if (round) {

unsigned target\_len = 1;

while (target\_len < lhs.number.size()) { target\_len \*= 2; }

lhs.number = std::string(target\_len - lhs.number.size(), '0') + lhs.number;

rhs.number = std::string(target\_len - rhs.number.size(), '0') + rhs.number;

}

return;

}

bool isLessThanAbs(const BigNumber &o) const {

if (!(this->isPositive() && o.isPositive())) {

throw std::runtime\_error(

std::string("Operands must all be positive")

);

}

BigNumber lhs(\*this), rhs(o);

BigNumber::\_makeLengthEven(lhs, rhs);

for (unsigned idx = 0, range = lhs.number.size(); idx != range; ++idx) {

if (lhs.number[idx] < rhs.number[idx]) {

return true;

}

if (lhs.number[idx] > rhs.number[idx]) {

return false;

}

}

return false;

}

public:

BigNumber addPositiveValue(const BigNumber &o) const {

if (!(this->isPositive() && o.isPositive())) {

throw std::runtime\_error(

std::string("Operands must be positive numbers")

);

}

// complement length, make even

BigNumber lhs(\*this), rhs(o); // copies

BigNumber::\_makeLengthEven(lhs, rhs);

// std::cout << lhs.str() << " + " << rhs.str() << std::endl;

std::string result;

int carry = 0;

for (int idx = lhs.number.size() - 1; idx != -1; --idx) {

int temp = (

(lhs.number[idx] - '0')

+ (rhs.number[idx] - '0')

+ carry

);

if (temp > 9) {

carry = 1;

temp -= 10;

} else {

carry = 0;

}

result = std::to\_string(temp) + result;

}

if (carry) {

result = std::string("1") + result;

}

BigNumber retbn(result);

return retbn;

}

BigNumber subtractPositiveNoLargerValue(const BigNumber &o) const {

if ((\*this) < o) {

throw std::runtime\_error(

std::string("lhs smaller than rhs: ")

+ this->number + std::string(" and ") + o.number

);

}

BigNumber lhs(\*this), rhs(o);

BigNumber::\_makeLengthEven(lhs, rhs);

int borrow = 0;

std::string result;

for (int idx = lhs.number.size() - 1; idx != -1; --idx) {

int temp = (

(lhs.number[idx] - '0')

- (rhs.number[idx] - '0')

- borrow

);

if (temp < 0) {

borrow = 1;

temp += 10;

} else {

borrow = 0;

}

result = std::string(std::to\_string(temp)) + result;

}

return BigNumber(result);

}

BigNumber &powerByTen(unsigned times) {

this->number += std::string(times, '0');

return \*this;

}

public:

BigNumber() : number("0") { return; }

~BigNumber(void) { return; }

BigNumber(const BigNumber &o) : number(o.number) { return; }

BigNumber(const std::string &s) : number(s) {

if (s.empty()) {

throw std::runtime\_error(std::string("Empty input value"));

}

if ((s[0] != '-') && (s[0] > '9' || s[0] < '0')) {

throw std::runtime\_error(std::string("Wrong input value: ") + s);

}

bool check = true;

for (unsigned idx = 1, range = s.size(); idx != range; ++idx) {

if (s[idx] > '9' || s[idx] < '0') {

check = false;

break;

}

}

if (check) {

this->number = s;

} else {

throw std::runtime\_error(std::string("Wrong input value: ") + s);

}

}

virtual std::string str(void) const {

BigNumber temp(\*this);

temp.shrinkSpace();

return temp.number;

}

virtual bool isPositive(void) const { return this->number[0] != '-'; }

virtual unsigned len(void) const {

if (this->isPositive()) {

return this->number.size();

}

return this->number.size() - 1;

}

virtual void shrinkSpace(void) {

bool is\_neg = (this->number[0] == '-');

if (is\_neg) {

this->number = this->number.substr(1);

}

auto first\_not\_zero = this->number.find\_first\_not\_of('0');

if (first\_not\_zero != std::string::npos) {

this->number = this->number.substr(first\_not\_zero);

}

if (is\_neg) {

this->number = std::string("-") + this->number;

}

}

virtual bool isCompatibleWith(const BigNumber &o) const {

return this->len() == o.len();

}

virtual BigNumber getAbsoluteValue(void) const {

BigNumber retbn;

if (this->isPositive()) {

retbn = BigNumber(\*this);

} else {

retbn = BigNumber(this->number.substr(1));

}

return retbn;

}

virtual int getLengthDiffAgainst(const BigNumber &o) const {

return this->len() - o.len();

}

virtual bool operator==(const BigNumber &o) const {

BigNumber lhs(\*this), rhs(o);

lhs.shrinkSpace();

rhs.shrinkSpace();

return lhs.number == rhs.number;

}

virtual bool operator<(const BigNumber &o) const {

bool lhs\_neg = !this->isPositive(),

rhs\_neg = !o.isPositive();

BigNumber lhs(this->getAbsoluteValue()), rhs(o.getAbsoluteValue());

if (lhs\_neg && rhs\_neg) {

return !lhs.isLessThanAbs(rhs);

}

if (lhs\_neg && !rhs\_neg) {

return true;

}

if (!lhs\_neg && rhs\_neg) {

return false;

}

return lhs.isLessThanAbs(rhs);

}

virtual bool operator>(const BigNumber &o) const {

return ( !((\*this) < o) && !((\*this) == o) );

}

virtual bool operator<=(const BigNumber &o) const {

return ( (\*this) < o || (\*this) == o );

}

virtual bool operator>=(const BigNumber &o) const {

return ( !((\*this) < o) );

}

virtual BigNumber operator-(void) const {

BigNumber ret = \*this;

ret.number = std::string("-") + ret.number;

return ret;

}

virtual BigNumber operator+(const BigNumber &o) const {

bool

is\_lhs\_pos = this->isPositive(),

is\_rhs\_pos = o.isPositive();

if (is\_lhs\_pos && is\_rhs\_pos) { // if both positive

return this->addPositiveValue(o);

}

if (!is\_lhs\_pos && !is\_rhs\_pos) { // if both negative

return -(

this->getAbsoluteValue()

.addPositiveValue(

o.getAbsoluteValue()

)

);

}

// one positive, one negative

bool negate = false;

BigNumber

abs\_lhs = this->getAbsoluteValue(),

abs\_rhs = o.getAbsoluteValue(),

abs\_dist;

// get absolute distance

if (abs\_lhs > abs\_rhs) {

abs\_dist = abs\_lhs.subtractPositiveNoLargerValue(abs\_rhs);

if (!is\_lhs\_pos) {

negate = true;

}

} else {

abs\_dist = abs\_rhs.subtractPositiveNoLargerValue(abs\_lhs);

if (!is\_rhs\_pos) {

negate = true;

}

}

// return truth value

if (negate) {

return -abs\_dist;

}

return abs\_dist;

}

virtual BigNumber operator-(const BigNumber &o) const {

return (\*this) + (-o);

}

virtual BigNumber operator\*(const BigNumber &o) const {

// smallest case return

if (this->len() == 2) {

int rst = (

std::stoi(this->number)

\* std::stoi(o.number)

);

return BigNumber(std::to\_string(rst));

}

// get final result sign flag

std::string sign\_flag;

if (this->isPositive() ^ o.isPositive()) {

sign\_flag = "-";

} // else omit '+' sign

// make copy

BigNumber lhs(this->getAbsoluteValue()), rhs(o.getAbsoluteValue());

BigNumber::\_makeLengthEven(lhs, rhs, true);

// split

unsigned split\_idx = lhs.number.size() / 2;

BigNumber lhs\_high(lhs.number.substr(0, split\_idx)),

lhs\_low(lhs.number.substr(split\_idx)),

rhs\_high(rhs.number.substr(0, split\_idx)),

rhs\_low(rhs.number.substr(split\_idx));

BigNumber temp\_high(lhs\_high \* rhs\_high),

temp\_low(lhs\_low \* rhs\_low),

temp\_mid(

(

(lhs\_high.addPositiveValue(lhs\_low))

.operator\*(

rhs\_high.addPositiveValue(rhs\_low)

)

).subtractPositiveNoLargerValue(

temp\_high.addPositiveValue(temp\_low)

)

);

BigNumber retbn(

temp\_high.powerByTen(lhs.number.size())

.addPositiveValue(temp\_mid.powerByTen(split\_idx))

.addPositiveValue(temp\_low)

);

retbn.number = sign\_flag + retbn.number;

return retbn;

}

};

int main(void) {

BigNumber

a(std::string("9223372036854775807")),

b(std::string("1234567891111")),

target(std::string("11386878964471969137416693151577"));

BigNumber result = a \* b;

std::cout << a.str() << " \* " << b.str() << " = "

<< result.str() << std::endl;

std::cout << "Is equal to expected result? ";

if (result == target) {

std::cout << "Yes!" << std::endl;

} else {

std::cout << "Nein!" << std::endl;

}

std::cout << std::endl;

a = target;

target = BigNumber(std::string("1476434256335201019505915952536059041319"

"057206399521214104693657122061789167297053187930937033"));

result = a \* a \* a;

std::cout << "ans \* ans \* ans = " << result.str() << std::endl;

std::cout << "Is equal to expected result? ";

if (result == target) {

std::cout << "Yes!" << std::endl;

} else {

std::cout << "Nein!" << std::endl;

}

std::cout << std::endl;

a = target; b = target;

target = BigNumber(std::string("29528685126704020390118319050721180826381"

"14412799042428209387314244123578334594106375861874066"));

result = a + a;

std::cout << "ans + ans = " << result.str() << std::endl;

std::cout << "Is equal to expected result? ";

if (result == target) {

std::cout << "Yes!" << std::endl;

} else {

std::cout << "Nein!" << std::endl;

}

std::cout << std::endl;

b = result;

target = BigNumber(std::string("-1476434256335201019505915952536059041319"

"057206399521214104693657122061789167297053187930937033"));

result = a - b;

std::cout << "ans - last\_ans = " << result.str() << std::endl;

std::cout << "Is equal to expected result? ";

if (result == target) {

std::cout << "Yes!" << std::endl;

} else {

std::cout << "Nein!" << std::endl;

}

return 0;

}

2.2.4 本题小结

本题难点在于推导出正确的合并步骤公式，确保其在最后的结果合成步骤不会出现值不正确或位数错乱的问题。

本实现在数字位数等于2时，即将原用字符串表示的大整数转换为无符号整型直接求解，不必将递归调用进行到一位数字的情况，降低了复杂度。

## 2.3 最优二分查找树

2.3.1 题目描述

（《算法导论》 15.5-1）实现算法CONSTRUCT-OPTIMAL-BST(*root*)，输入为表*root*，输出是最优二叉搜索树的结构。

2.3.2 算法设计

算法只需根据*root*表输出对应的二叉树即可，通过观察输出样例可知为对构造号的二叉树的先序遍历输出。算法如图2.3所示。

初始时，调用CONSTRUCT-OPTIMAL-BST(*root*, 1, *root*.*columns*, NIL,   
“root”)。

2.3.3 源程序及注释

#include <iostream>

#include <stdexcept>

#include <string>

#include <iterator>

#include <vector>

#include <queue>

#include <array>

std::string str\_k\_(unsigned x) {

return std::string("k\_") + std::to\_string(x);

}

std::string str\_d\_(unsigned x) {

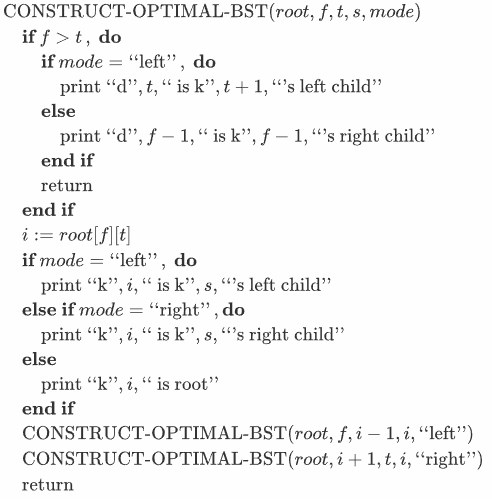


图2.3 最优二分查找树算法

return std::string("d\_") + std::to\_string(x);

}

typedef std::vector<std::vector<unsigned> > root\_table\_t;

typedef std::array<unsigned, 3> triplet; // { from, split, to }

unsigned &get\_item(root\_table\_t &tab,

size\_t i,

size\_t j) {

size\_t n = tab.size();

size\_t col\_len = tab[i - 1].size();

return tab[i - 1][n - j];

}

enum \_ConstructMode { LEFT, RIGHT, ROOT };

void \_construct\_tree(root\_table\_t &tab,

unsigned split, unsigned from, unsigned to,

\_ConstructMode mode) {

/\*\*

\* Note:

\* While `mode` is `ROOT`, `split` is ignored

\*/

// simplest case return: leaf node

if (from > to) {

switch (mode) {

case LEFT: {

std::cout << str\_d\_(to) << " is " << str\_k\_(to + 1)

<< "'s left child" << std::endl;

break;

}

case RIGHT: {

std::cout << str\_d\_(from - 1) << " is " << str\_k\_(from - 1)

<< "'s right child" << std::endl;

break;

}

}

return;

}

// normal cases

unsigned new\_split\_num = get\_item(tab, from, to);

switch (mode) {

case LEFT: {

std::cout << str\_k\_(new\_split\_num) << " is " << str\_k\_(split)

<< "'s left child" << std::endl;

split = new\_split\_num;

break;

}

case RIGHT: {

std::cout << str\_k\_(new\_split\_num) << " is " << str\_k\_(split)

<< "'s right child" << std::endl;

split = new\_split\_num;

break;

}

case ROOT: {

std::cout << str\_k\_(new\_split\_num) << " is root" << std::endl;

split = new\_split\_num;

break;

}

throw std::runtime\_error("should not reach this line");

}

\_construct\_tree(tab, split, from, new\_split\_num - 1, LEFT);

\_construct\_tree(tab, split, new\_split\_num + 1, to, RIGHT);

return;

}

void construct\_optimal\_bst(root\_table\_t tab) {

\_construct\_tree(tab, 0, 1, tab.size(), ROOT);

return;

}

int main(void) {

unsigned i\_1[] = { 2, 2, 2, 1, 1 };

unsigned i\_2[] = { 4, 2, 2, 2 };

unsigned i\_3[] = { 5, 4, 3 };

unsigned i\_4[] = { 5, 4 };

unsigned i\_5[] = { 5 };

root\_table\_t tab;

tab.push\_back(std::vector<unsigned>(std::begin(i\_1), std::end(i\_1)));

tab.push\_back(std::vector<unsigned>(std::begin(i\_2), std::end(i\_2)));

tab.push\_back(std::vector<unsigned>(std::begin(i\_3), std::end(i\_3)));

tab.push\_back(std::vector<unsigned>(std::begin(i\_4), std::end(i\_4)));

tab.push\_back(std::vector<unsigned>(std::begin(i\_5), std::end(i\_5)));

construct\_optimal\_bst(tab);

return 0;

}

2.3.4 本题小结

本题为简单的查表先序遍历，注意叶子节点（*d*节点）的输出判断条件即可。

## 2.4 Floyd-Warshall算法

2.4.1 题目描述

补充ALL-PATHS算法，增加前驱矩阵（《算法导论》Chp25.2），使得在求出结点间的最短路径长度矩阵A后，能够推导出每对结点间的最短路径。

2.4.2 算法设计

在《算法导论》中描述的Floyd-Warshall算法中加入对表П的更新即可。算法如图2.4所示。

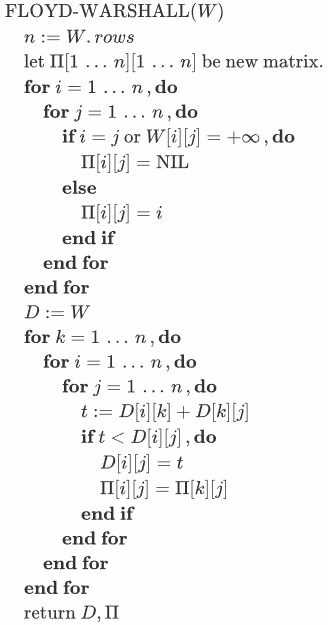


图2.4 Floyd-Warshall算法

初始时，对领接矩阵*W*调用FLOYD-WARSHALL(*W*)即可，返回最短路径权重矩阵*D*及前驱矩阵П。

2.4.3 源代码及注释

#include <iostream>

#include <iomanip>

#include <limits>

#include <iterator>

#include <array>

#include <vector>

const unsigned n = 5;

typedef int weight\_t;

typedef std::array<weight\_t, n> weight\_row\_t;

typedef std::array<weight\_row\_t, n> weight\_matrix\_t;

typedef std::array<std::array<int, n>, n> precursor\_matrix\_t;

const weight\_t pos\_infty = std::numeric\_limits<weight\_t>::max() / 2; // HACK: `/2` to allow two infty add up

const int nil = -1;

class Solution {

public:

weight\_matrix\_t W;

precursor\_matrix\_t P;

};

Solution floyd\_warshall(weight\_matrix\_t W) {

unsigned n = W.size();

// initialized precursor matrix

precursor\_matrix\_t P;

for (unsigned i = 0; i != n; ++i) {

for (unsigned j = 0; j != n; ++j) {

if (i == j) {

P[i][j] = nil;

} else if (W[i][j] >= pos\_infty) {

P[i][j] = nil;

} else {

P[i][j] = i + 1;

}

}

}

// Floyd-Warshall Algorithm

weight\_matrix\_t D{W};

for (unsigned k = 0; k != n; ++k) {

for (unsigned i = 0; i != n; ++i) {

for (unsigned j = 0; j != n; ++j) {

weight\_t temp;

if (D[i][k] >= pos\_infty || D[k][j] >= pos\_infty) {

temp = pos\_infty;

} else {

temp = D[i][k] + D[k][j];

}

if (temp < pos\_infty && temp < D[i][j]) {

D[i][j] = temp;

P[i][j] = P[k][j];

}

}

} // end for i, j

} // end for k

// end of F-W Alg.

Solution solution;

solution.W = D;

solution.P = P;

return solution;

}

int main(void) {

std::vector<weight\_matrix\_t> testcase;

weight\_matrix\_t W\_1{

weight\_row\_t{ 0, 3, 8, pos\_infty, -4 },

weight\_row\_t{ pos\_infty, 0, pos\_infty, 1, 7 },

weight\_row\_t{ pos\_infty, 4, 0, pos\_infty, pos\_infty },

weight\_row\_t{ 2, pos\_infty, -5, 0, pos\_infty },

weight\_row\_t{ pos\_infty, pos\_infty, pos\_infty, 6, 0 }

};

testcase.push\_back(W\_1);

weight\_matrix\_t W\_2{

weight\_row\_t{ 0, 123, pos\_infty, pos\_infty, pos\_infty },

weight\_row\_t{ pos\_infty, 0, pos\_infty, pos\_infty, pos\_infty },

weight\_row\_t{ pos\_infty, pos\_infty, 0, pos\_infty, pos\_infty },

weight\_row\_t{ pos\_infty, pos\_infty, pos\_infty, 0, pos\_infty },

weight\_row\_t{ pos\_infty, pos\_infty, pos\_infty, pos\_infty, 0 }

};

testcase.push\_back(W\_2);

for (auto &W : testcase) {

std::cout << "W matrix:" << std::endl;

for (auto &row : W) {

for (auto &each : row) {

if (each >= pos\_infty) {

std::cout << std::setw(8) << std::right << "infty";

} else {

std::cout << std::setw(8) << std::right << each;

}

}

std::cout << std::endl;

}

Solution solution = floyd\_warshall(W);

std::cout << "D^{(n)} matrix:" << std::endl;

for (auto &row : solution.W) {

for (auto &each : row) {

if (each < pos\_infty) {

std::cout << std::setw(8) << std::right << each;

} else {

std::cout << std::setw(8) << std::right << "infty";

}

}

std::cout << std::endl;

}

std::cout << std::endl;;

std::cout << "P^{(n)} matrix:" << std::endl;

for (auto &row : solution.P) {

for (auto &each : row) {

if (each == nil) {

std::cout << std::setw(8) << std::right << "nil";

} else {

std::cout << std::setw(8) << std::right << each;

}

}

std::cout << std::endl;

}

std::cout << "=====================" << std::endl;

}

return 0;

}

2.4.4 本题小结

本题按照《算法导论》上所属FLOYD-WARSHALL算法实现，并保持权重矩阵*D*与前驱矩阵П同步更新即可。

# 三、心得体会

刚开始上手实验时，感觉稍微有些难度，主要是对算法思想的掌握不足导致。通过几次对分治法的实现之后，逐渐开始掌握算法背后的思想，之后就可以很快理清思路并着手实现。

实验中发现若是已有现成的伪代码，则代码实现时就比较快。可见伪代码虽然与实际代码有差距，省去了如具体数据结构等实现细节，但仍然具有非常重要的指导意义。