# Continuous Knapsack Problem

The “simplest” knapsack problem – there is an inventory of unlimited supplies, program must pick the best combination for the knapsack. Items can be cut and the worth will be reduced proportionally.

#include <stdio.h>

#include <stdlib.h>

struct item { double w, v; const char \*name; } items[] = {

{ 3.8, 36, "beef" },

{ 5.4, 43, "pork" },

{ 3.6, 90, "ham" },

{ 2.4, 45, "greaves" },

{ 4.0, 30, "flitch" },

{ 2.5, 56, "brawn" },

{ 3.7, 67, "welt" },

{ 3.0, 95, "salami" },

{ 5.9, 98, "sausage" },

};

int item\_cmp(const void \*aa, const void \*bb)

{

const struct item \*a = aa, \*b = bb;

double ua = a->v / a->w, ub = b->v / b->w;

return ua < ub ? -1 : ua > ub;

}

int main()

{

struct item \*it;

double space = 15;

qsort(items, 9, sizeof(struct item), item\_cmp);

for (it = items + 9; it---items && space > 0; space -= it->w)

if (space >= it->w)

printf("take all %s\n", it->name);

else

printf("take %gkg of %g kg of %s\n",

space, it->w, it->name);

return 0;

}

# Unbounded Knapsack Problem

Another knapsack, but it involves two constraints – e.g. weight + volume. Supplies are unlimited.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct {

double val, wgt, vol;

const char \* name;

} items[] = { // value in hundreds, volume in thousandths

{30, .3, 25, "panacea"},

{18, .2, 15, "ichor"},

{25, 2., 2, "gold"},

{0,0,0,0}

};

/\* silly setup for silly task \*/

int best\_cnt[16] = {0}, cnt[16] = {0};

double best\_v = 0;

void grab\_em(int idx, double cap\_v, double cap\_w, double v)

{

double val;

int t = cap\_w / items[idx].wgt;

cnt[idx] = cap\_v / items[idx].vol;

if (cnt[idx] > t) cnt[idx] = t;

while (cnt[idx] >= 0) {

val = v + cnt[idx] \* items[idx].val;

if (!items[idx + 1].name) {

if (val > best\_v) {

best\_v = val;

memcpy(best\_cnt, cnt, sizeof(int) \* (1 + idx));

}

return;

}

grab\_em(idx + 1, cap\_v - cnt[idx] \* items[idx].vol,

cap\_w - cnt[idx] \* items[idx].wgt, val);

cnt[idx]--;

}

}

int main(void)

{

int i;

grab\_em(0, 250, 25, 0);

printf("value: %g hundreds\n", best\_v);

for (i = 0; items[i].name; i++)

printf("%d %s\n", best\_cnt[i], items[i].name);

return 0;

}

# Bounded Knapsack Problem

Similar to unbounded, but involves a limited supply of items.

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdexcept>

#include <memory>

#include <sys/time.h>

using std::cout;

using std::endl;

class StopTimer

{

public:

StopTimer(): begin\_(getUsec()) {}

unsigned long long getTime() const { return getUsec() - begin\_; }

private:

static unsigned long long getUsec()

{//...you might want to use something else under Windows

timeval tv;

const int res = ::gettimeofday(&tv, 0);

if(res)

return 0;

return tv.tv\_usec + 1000000 \* tv.tv\_sec;

}

unsigned long long begin\_;

};

struct KnapsackTask

{

struct Item

{

std::string name;

unsigned w, v, qty;

Item(): w(), v(), qty() {}

Item(const std::string& iname, unsigned iw, unsigned iv, unsigned iqty):

name(iname), w(iw), v(iv), qty(iqty)

{}

};

typedef std::vector<Item> Items;

struct Solution

{

unsigned v, w;

unsigned long long iterations, usec;

std::vector<unsigned> n;

Solution(): v(), w(), iterations(), usec() {}

};

//...

KnapsackTask(): maxWeight\_(), totalWeight\_() {}

void add(const Item& item)

{

const unsigned totalItemWeight = item.w \* item.qty;

if(const bool invalidItem = !totalItemWeight)

throw std::logic\_error("Invalid item: " + item.name);

totalWeight\_ += totalItemWeight;

items\_.push\_back(item);

}

const Items& getItems() const { return items\_; }

void setMaxWeight(unsigned maxWeight) { maxWeight\_ = maxWeight; }

unsigned getMaxWeight() const { return std::min(totalWeight\_, maxWeight\_); }

private:

unsigned maxWeight\_, totalWeight\_;

Items items\_;

};

class BoundedKnapsackRecursiveSolver

{

public:

typedef KnapsackTask Task;

typedef Task::Item Item;

typedef Task::Items Items;

typedef Task::Solution Solution;

void solve(const Task& task)

{

Impl(task, solution\_).solve();

}

const Solution& getSolution() const { return solution\_; }

private:

class Impl

{

struct Candidate

{

unsigned v, n;

bool visited;

Candidate(): v(), n(), visited(false) {}

};

typedef std::vector<Candidate> Cache;

public:

Impl(const Task& task, Solution& solution):

items\_(task.getItems()),

maxWeight\_(task.getMaxWeight()),

maxColumnIndex\_(task.getItems().size() - 1),

solution\_(solution),

cache\_(task.getMaxWeight() \* task.getItems().size()),

iterations\_(0)

{}

void solve()

{

if(const bool nothingToSolve = !maxWeight\_ || items\_.empty())

return;

StopTimer timer;

Candidate candidate;

solve(candidate, maxWeight\_, items\_.size() - 1);

convertToSolution(candidate);

solution\_.usec = timer.getTime();

}

private:

void solve(Candidate& current, unsigned reminderWeight, const unsigned itemIndex)

{

++iterations\_;

const Item& item(items\_[itemIndex]);

if(const bool firstColumn = !itemIndex)

{

const unsigned maxQty = std::min(item.qty, reminderWeight/item.w);

current.v = item.v \* maxQty;

current.n = maxQty;

current.visited = true;

}

else

{

const unsigned nextItemIndex = itemIndex - 1;

{

Candidate& nextItem = cachedItem(reminderWeight, nextItemIndex);

if(!nextItem.visited)

solve(nextItem, reminderWeight, nextItemIndex);

current.visited = true;

current.v = nextItem.v;

current.n = 0;

}

if(reminderWeight >= item.w)

{

for (unsigned numberOfItems = 1; numberOfItems <= item.qty; ++numberOfItems)

{

reminderWeight -= item.w;

Candidate& nextItem = cachedItem(reminderWeight, nextItemIndex);

if(!nextItem.visited)

solve(nextItem, reminderWeight, nextItemIndex);

const unsigned checkValue = nextItem.v + numberOfItems \* item.v;

if ( checkValue > current.v)

{

current.v = checkValue;

current.n = numberOfItems;

}

if(!(reminderWeight >= item.w))

break;

}

}

}

}

void convertToSolution(const Candidate& candidate)

{

solution\_.iterations = iterations\_;

solution\_.v = candidate.v;

solution\_.n.resize(items\_.size());

const Candidate\* iter = &candidate;

unsigned weight = maxWeight\_, itemIndex = items\_.size() - 1;

while(true)

{

const unsigned currentWeight = iter->n \* items\_[itemIndex].w;

solution\_.n[itemIndex] = iter->n;

weight -= currentWeight;

if(!itemIndex--)

break;

iter = &cachedItem(weight, itemIndex);

}

solution\_.w = maxWeight\_ - weight;

}

Candidate& cachedItem(unsigned weight, unsigned itemIndex)

{

return cache\_[weight \* maxColumnIndex\_ + itemIndex];

}

const Items& items\_;

const unsigned maxWeight\_;

const unsigned maxColumnIndex\_;

Solution& solution\_;

Cache cache\_;

unsigned long long iterations\_;

};

Solution solution\_;

};

void populateDataset(KnapsackTask& task)

{

typedef KnapsackTask::Item Item;

task.setMaxWeight( 400 );

task.add(Item("map",9,150,1));

task.add(Item("compass",13,35,1));

task.add(Item("water",153,200,2));

task.add(Item("sandwich",50,60,2));

task.add(Item("glucose",15,60,2));

task.add(Item("tin",68,45,3));

task.add(Item("banana",27,60,3));

task.add(Item("apple",39,40,3));

task.add(Item("cheese",23,30,1));

task.add(Item("beer",52,10,3));

task.add(Item("suntancream",11,70,1));

task.add(Item("camera",32,30,1));

task.add(Item("T-shirt",24,15,2));

task.add(Item("trousers",48,10,2));

task.add(Item("umbrella",73,40,1));

task.add(Item("w-trousers",42,70,1));

task.add(Item("w-overclothes",43,75,1));

task.add(Item("note-case",22,80,1));

task.add(Item("sunglasses",7,20,1));

task.add(Item("towel",18,12,2));

task.add(Item("socks",4,50,1));

task.add(Item("book",30,10,2));

}

int main()

{

KnapsackTask task;

populateDataset(task);

BoundedKnapsackRecursiveSolver solver;

solver.solve(task);

const KnapsackTask::Solution& solution = solver.getSolution();

cout << "Iterations to solve: " << solution.iterations << endl;

cout << "Time to solve: " << solution.usec << " usec" << endl;

cout << "Solution:" << endl;

for (unsigned i = 0; i < solution.n.size(); ++i)

{

if (const bool itemIsNotInKnapsack = !solution.n[i])

continue;

cout << " " << solution.n[i] << ' ' << task.getItems()[i].name << " ( item weight = " << task.getItems()[i].w << " )" << endl;

}

cout << "Weight: " << solution.w << " Value: " << solution.v << endl;

return 0;

}

# Knapsack Problem

Yet another knapsack. However, this time round we can only take 1 of each item, or not take the item at all.

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

typedef struct {

const char \* name;

int weight, value;

} item\_t;

item\_t item[] = {

{"map", 9, 150},

{"compass", 13, 35},

{"water", 153, 200},

{"sandwich", 50, 160},

{"glucose", 15, 60},

{"tin", 68, 45},

{"banana", 27, 60},

{"apple", 39, 40},

{"cheese", 23, 30},

{"beer", 52, 10},

{"suntancream", 11, 70},

{"camera", 32, 30},

{"T-shirt", 24, 15},

{"trousers", 48, 10},

{"umbrella", 73, 40},

{"waterproof trousers", 42, 70},

{"waterproof overclothes", 43, 75},

{"note-case", 22, 80},

{"sunglasses", 7, 20},

{"towel", 18, 12},

{"socks", 4, 50},

{"book", 30, 10}

};

#define n\_items (sizeof(item)/sizeof(item\_t))

typedef struct {

uint32\_t bits; /\* 32 bits, can solve up to 32 items \*/

int value;

} solution;

void optimal(int weight, int idx, solution \*s)

{

solution v1, v2;

if (idx < 0) {

s->bits = s->value = 0;

return;

}

if (weight < item[idx].weight) {

optimal(weight, idx - 1, s);

return;

}

optimal(weight, idx - 1, &v1);

optimal(weight - item[idx].weight, idx - 1, &v2);

v2.value += item[idx].value;

v2.bits |= (1 << idx);

\*s = (v1.value >= v2.value) ? v1 : v2;

}

int main(void)

{

int i = 0, w = 0;

solution s = {0, 0};

optimal(400, n\_items - 1, &s);

for (i = 0; i < n\_items; i++) {

if (s.bits & (1 << i)) {

printf("%s\n", item[i].name);

w += item[i].weight;

}

}

printf("Total value: %d; weight: %d\n", s.value, w);

return 0;

}

# Josephus Problem

An executioner walks along the circle, starting from prisoner 0, removing every k-th prisoner and killing him, until only one survivor is left, who is then freed. Find the number of this survivor.

#include <iostream>

#include <vector>

using namespace std;

typedef unsigned long long bigint;

class josephus

{

public:

bigint findSurvivors( bigint n, bigint k, bigint s = 0 )

{

bigint i = s + 1;

for( bigint x = i; x <= n; x++, i++ )

s = ( s + k ) % i;

return s;

}

void getExecutionList( bigint n, bigint k, bigint s = 1 )

{

cout << endl << endl << "Execution list: " << endl;

prisoners.clear();

for( bigint x = 0; x < n; x++ )

prisoners.push\_back( x );

bigint index = 0;

while( prisoners.size() > s )

{

index += k - 1;

if( index >= prisoners.size() ) index %= prisoners.size();

cout << prisoners[static\_cast<unsigned int>( index )] << ", ";

vector<bigint>::iterator it = prisoners.begin() + static\_cast<unsigned int>( index );

prisoners.erase( it );

}

}

private:

vector<bigint> prisoners;

};

int main( int argc, char\* argv[] )

{

josephus jo;

bigint n, k, s;

while( true )

{

system( "cls" );

cout << "Number of prisoners( 0 to QUIT ): "; cin >> n;

if( !n ) return 0;

cout << "Execution step: "; cin >> k;

cout << "How many survivors: "; cin >> s;

cout << endl << "Survivor";

if( s == 1 )

{

cout << ": " << jo.findSurvivors( n, k );

jo.getExecutionList( n, k );

}

else

{

cout << "s: ";

for( bigint x = 0; x < s; x++ )

cout << jo.findSurvivors( n, k, x ) << ", ";

jo.getExecutionList( n, k, s );

}

cout << endl << endl;

system( "pause" );

}

return 0;

}

# Mergesort

#include <stdio.h>

#include <stdlib.h>

int \_mergeSort(int arr[], int temp[], int left, int right);

int merge(int arr[], int temp[], int left, int mid, int right);

/\* This function sorts the input array and returns the

number of inversions in the array \*/

int mergeSort(int arr[], int array\_size)

{

int \*temp = (int \*)malloc(sizeof(int)\*array\_size);

return \_mergeSort(arr, temp, 0, array\_size - 1);

}

/\* An auxiliary recursive function that sorts the input array and

returns the number of inversions in the array. \*/

int \_mergeSort(int arr[], int temp[], int left, int right)

{

int mid, inv\_count = 0;

if (right > left)

{

/\* Divide the array into two parts and call \_mergeSortAndCountInv()

for each of the parts \*/

mid = (right + left)/2;

/\* Inversion count will be sum of inversions in left-part, right-part

and number of inversions in merging \*/

inv\_count = \_mergeSort(arr, temp, left, mid);

inv\_count += \_mergeSort(arr, temp, mid+1, right);

/\*Merge the two parts\*/

inv\_count += merge(arr, temp, left, mid+1, right);

}

return inv\_count;

}

/\* This funt merges two sorted arrays and returns inversion count in

the arrays.\*/

int merge(int arr[], int temp[], int left, int mid, int right)

{

int i, j, k;

int inv\_count = 0;

i = left; /\* i is index for left subarray\*/

j = mid; /\* i is index for right subarray\*/

k = left; /\* i is index for resultant merged subarray\*/

while ((i <= mid - 1) && (j <= right))

{

if (arr[i] <= arr[j])

{

temp[k++] = arr[i++];

}

else

{

temp[k++] = arr[j++];

/\*this is tricky -- see above explanation/diagram for merge()\*/

inv\_count = inv\_count + (mid - i);

}

}

/\* Copy the remaining elements of left subarray

(if there are any) to temp\*/

while (i <= mid - 1)

temp[k++] = arr[i++];

/\* Copy the remaining elements of right subarray

(if there are any) to temp\*/

while (j <= right)

temp[k++] = arr[j++];

/\*Copy back the merged elements to original array\*/

for (i=left; i <= right; i++)

arr[i] = temp[i];

return inv\_count;

}

/\* Driver progra to test above functions \*/

int main(int argv, char\*\* args)

{

int n;

scanf("%d", &n);

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

{

int d;

scanf("%d", &d);

int arr[d];

for(int j=0;j<d;j++)

{

scanf("%d", &arr[j]);

}

printf("%d\n", mergeSort(arr, d));

}

return 0;

}

# Sorting

Use std qsort (check c++ ref)/sort