CityU’s 3 Solutions to Sum Problem

* My GitHub repository (for viewing my code):
* [Approach 1: EnumerateAll](#ny2k12rjym4n)
* [Approach 2: Fold-half Enumerate All](#qr27dtqncxcj)
* [Approach 3: Sum bucket](#cq9f88chryq2)
* [Test cases used & corresponding answers](#g5kr5bztuu)

Approach 1: Enumerate All combinations

**Complexity: O(2^N)**

**Explanation:**

Generate all the possible combinations (2^N) and find the solution that sums up to the target number with the fewest elements.

**Code:**

#include<iostream>

#include<algorithm>

#define MAX\_N 30

using namespace std;

//global variables

int N;

int T;

int arr[MAX\_N];

int sol\_cnt = MAX\_N+1; //how many elements are used in the optimal solution

bool sol\_used[MAX\_N] = { false };

int main() {

while (cin >> N) {

//// PART 0: initialize ////

sol\_cnt = MAX\_N+1;

for (int i = 0; i < N; i++) {

sol\_used[i] = false;

}

//// PART 1: input ////

for (int i = 0; i < N; i++) {

cin >> arr[i];

}

cin >> T;

//// PART 2: enumerate all combinations ////

//cout << (1 << N) << endl;

for (int i = 0; i < (1 << N); i++) {

int temp\_sum = 0;

int temp\_cnt = 0;

bool temp\_used[MAX\_N] = { false };

for (int j = 0; j < N; j++) {

if (i >> j & 1) { //use arr[j]

temp\_cnt++;

temp\_used[j] = true;

temp\_sum += arr[j];

}

}

if (temp\_sum == T) {

if (temp\_cnt < sol\_cnt) {

sol\_cnt = temp\_cnt;

for (int k = 0; k < N; k++) {

sol\_used[k] = temp\_used[k];

}

}

}

}

//// PART 3: Output ////

for (int i = 0; i < N; i++) {

if (sol\_used[i]) {

cout << i << " ";

}

}

if (sol\_cnt == MAX\_N + 1) {

cout << "impossible";

}

cout << endl;

}

}

**Performance:**

The performance is acceptable when the dataset is small, however, when the size reaches around 30, it will take a long time to compute.

Approach 2: Fold-half Enumerate All

**Complexity: O(N \* 2^(N/2))**

**Explanation:**

1. Divide the array into two subarrays with (N/2) size
2. Generate two sets (SetA, SetB) of combinations using the two subarrays’ elements
3. For every combination in SetA, find if (target\_number - this combination’s sum) exists in SetB.
4. If it exists, then compare the combination of solA(from SetA) and solB(from SetB) with the current solution.

**Demonstration:**

1. array=[1,4,2,8,5,7]; target = 12 -> arrA=[1,4,2]; arrB=[8,5,7]
2. SetA={

//value: elements

0:{}

1:{1}

2:{2}

3:{1,2}

4:{4}

5:{1,4}

6:{2,4}

7:{1,2,4}

}

SetB={

0:{}

5:{5}

7:{7}

8:{8}

12:{5,7}

13:{5,8}

15:{7,8}

20:{5,7,8}

}

1. For each element in SetA, find (target-value) in SetB

|  |  |  |
| --- | --- | --- |
| solA | solB | Total element count |
| 0:{} | 12:{5,7} | 2 |
| 1:{1} | Non existance |  |
| 2:{2} | Non existance |  |
| 3:{1,2} | Non existance |  |
| 4:{4} | 8:{8} | 2 |
| 5:{1,4} | 7:{7} | 3 |
| 6:{2,4} | Non existance |  |
| 7:{1,2,4} | 5:{5} | 4 |

1. The overall best solution will be {5,7} and {4,8}

**Code:**

#include<iostream>

#include<algorithm>

#include<map>

#include<vector>

#define MAX\_N 30

using namespace std;

//

struct solution {

int cnt=9999;

vector<int>number\_used;

};

//global variables

int N;

int T;

int arr[MAX\_N];

map<int, solution>Sol\_setA; //key: sum; value: combination

map<int, solution>Sol\_setB;

map<int, int>num\_to\_idx;

vector<int>best\_sol;

int main() {

while (cin >> N) {

//// PART 0: initialize ////

Sol\_setA.clear();

Sol\_setB.clear();

num\_to\_idx.clear();

best\_sol.clear();

//// PART 1: input ////

for (int i = 0; i < N; i++) {

cin >> arr[i];

num\_to\_idx[arr[i]] = i;

}

cin >> T;

//// PART 2: enumerate all combinations of set A (first half of the arr) ////

//cout << (1 << N) << endl;

Sol\_setA[0].cnt = 0;

for (int i = 0; i < (1 << (N/2)); i++) {

int temp\_sum = 0;

int temp\_cnt = 0;

vector<int>temp\_used;

temp\_used.clear();

for (int j = 0; j < (N/2); j++) {

if (i >> j & 1) { //use arr[j]

temp\_cnt++;

temp\_used.push\_back(arr[j]);

temp\_sum += arr[j];

}

}

if (Sol\_setA.find(temp\_sum) == Sol\_setA.end() || Sol\_setA[temp\_sum].cnt > temp\_cnt) {

Sol\_setA[temp\_sum].cnt = temp\_cnt;

Sol\_setA[temp\_sum].number\_used.clear();

for (int i = 0; i < temp\_used.size(); i++) {

Sol\_setA[temp\_sum].number\_used.push\_back(temp\_used[i]);

}

}

}

map<int, solution>::iterator ite;

/\*

for (ite = Sol\_setA.begin(); ite != Sol\_setA.end(); ite++) {

cout << "Value: " << ite->first << endl;

cout << "Numbers: ";

for (int i = 0; i < ite->second.cnt; i++) {

cout << ite->second.number\_used[i] << " ";

}

cout << endl;

}

\*/

//// PART 3: enumerate all combinations of set B (later half) and match with Sol\_setA////

Sol\_setB[0].cnt = 0;

int b\_0id = N / 2;

for (int i = 0; i < (1 << (N-b\_0id)); i++) {

int temp\_sum = 0;

int temp\_cnt = 0;

vector<int>temp\_used;

temp\_used.clear();

for (int j = 0; j < (N - b\_0id); j++) {

if (i >> j & 1) { //use arr[j]

temp\_cnt++;

temp\_used.push\_back(arr[b\_0id+j]);

temp\_sum += arr[b\_0id+j];

}

}

if (Sol\_setB.find(temp\_sum) == Sol\_setB.end() || Sol\_setB[temp\_sum].cnt > temp\_cnt) {

Sol\_setB[temp\_sum].cnt = temp\_cnt;

Sol\_setB[temp\_sum].number\_used.clear();

for (int i = 0; i < temp\_used.size(); i++) {

Sol\_setB[temp\_sum].number\_used.push\_back(temp\_used[i]);

}

}

}

/\*

//map<int, solution>::iterator ite;

for (ite = Sol\_setB.begin(); ite != Sol\_setB.end(); ite++) {

cout << "Value: " << ite->first << endl;

cout << "Numbers: ";

for (int i = 0; i < ite->second.cnt; i++) {

cout << ite->second.number\_used[i] << " ";

}

cout << endl;

}

\*/

//// PART 4: Matching Sol\_setA and Sol\_setB ////

map<int, solution>::iterator bite;

for (ite = Sol\_setA.begin(); ite != Sol\_setA.end(); ite++) {

int A\_value = ite->first;

int A\_cnt = ite->second.cnt;

bite = Sol\_setB.find(T - A\_value);

if (bite!= Sol\_setB.end()) {

int B\_cnt = bite->second.cnt;

if (best\_sol.size() == 0||best\_sol.size()>A\_cnt+B\_cnt) {

best\_sol.clear();

for (int i = 0; i < ite->second.number\_used.size(); i++) {

best\_sol.push\_back(ite->second.number\_used[i]);

}

for (int i = 0; i < bite->second.number\_used.size(); i++) {

best\_sol.push\_back(bite->second.number\_used[i]);

}

}

}

}

if (best\_sol.size() == 0) {

cout << "impossible";

}

else {

for (int i = 0; i < best\_sol.size(); i++) {

//cout << best\_sol[i] << " ";

cout << num\_to\_idx[best\_sol[i]] << " ";

}

}

cout << endl;

}

}

**Performance:**

In comparison with Approach 1, this approach can easily handle array with size 30 since its complexity is significantly lower than that of Approach 1.

**Approach 3: Bucket Sum**

**Complexity**: depends on the nature of the elements in the array. If (after sorting the array), arr[i+1] >>(far greater than) arr[i], this approach might outperform Approach 2. However, its worst-case complexity is same as Approach 1 (O(2^N)). Please check out the ‘Performance” section below for more details.

**Motivation:**

Is it possible to know whether an element is must-needed to the solution (assume that the solution exists)?

Consider the scenario below:

arr=[1 2 6 3 17 82 23 234]; target = 90

I can say that 82 is a must-needed element (if the solution exists), cause if we don’t use 82,

The elements left available are [1,2,6,3,17,23], which only sum up to 52, making it impossible to reach 90.

By the same token, we can expand this scenario to say that for target numbers that lies in between [82,234), 82 is a must-needed element (if the solution exists)

**Explanation:**

1. Preprocess stage

First sort the array, then arrange the elements into buckets by my bucket policy (an element is put into a new bucket if and only if the sum of all elements smaller than it is still smaller than this element)

1. Generate all possible combinations of elements within the bucket
2. Given the target number, find the target\_bucket in which AT LEAST ONE of the element within this bucket must be used. Then break the original problem into subproblem solving (target\_number - the element’s value) and dfs the possible solutions.
3. Check if a solution exists and update the best solution.

**Demonstration:**

1. arr=[1 2 6 3 17 82 23 234]; target = 100

Sort -> arr=[1,2,3,6,17,23,82,234]

Bucketing

|  |  |  |
| --- | --- | --- |
| key | elements | Handle range |
| 1 | 1 | [1,2) |
| 16 | 2,3,6 | [2,17) |
| 81 | 17,23 | [17,82) |
| 233 | 82 | [82,234) |
| 368 | 234 | [234,368] |

1. Generate all combinations within each bucket

|  |  |  |
| --- | --- | --- |
| Bucket key | value | elements |
| 1 | 0 | {} |
| 1 | {1} |
| 16 | 0 | {} |
| 2 | {2} |
| 3 | {3} |
| 5 | {2,3} |
| 6 | {6} |
| 8 | {2,6} |
| 9 | {3,6} |
| 11 | {2,3,6} |
| 81 | 0 | {} |
| 17 | {17} |
| 23 | {23} |
| 40 | {17,23} |
| 233 | 0 | {} |
| 82 | {82} |
| 368 | 0 | {} |
| 234 | {234} |

\*Note that compare with Approach 1 which will have 2^8(256) combinations; Approach 2 will have 2\* 2^4 (32) combinations, this approach only uses 18 combinations

1. Target number =100

-> lies within [82,234) therefore bucket[82] must be used ->

Subproblem => target number become (100-82) =18

-> lies within [17, 82) therefore bucket[16] must be used ->

There are 3 possible usage of this bucket’s combination (excluding 0)

1. Use 17 -> subproblem become (18-17=1)

-> lies within [1,2), bucket[1] must be used

->use element 1 -> subproblem become 0

-> {82,17,1} is a solution

1. Use 23 -> 23>18, cannot use this element, no solution here
2. Use 40 -> 40>18, cannot use this element, no solution here

4. The solution is {82,17,1}

**Code:**

#include<iostream>

#include<algorithm>

#include<map>

#include<vector>

#include<set>

#define MAX\_N 30

using namespace std;

//global variables

int N;

int T;

int arr[MAX\_N];

vector<int>key\_arr;

map<int, int>num\_to\_idx;

map<int, vector<int>>bucket\_element;

map<int, map<int, vector<int>>>bucket\_combinations;

vector<int>Solution;

//global functions

int num\_to\_key(int num);

void print\_bucket\_element();

void print\_bucket\_combinations();

void dfs(int t, vector<int>sol, set<int>used\_bucket);

int main() {

while (cin >> N) {

//// PART 0: initialize ////

bucket\_element.clear();

num\_to\_idx.clear();

Solution.clear();

bucket\_combinations.clear();

key\_arr.clear();

//// PART 1: input ////

for (int i = 0; i < N; i++) {

cin >> arr[i];

num\_to\_idx[arr[i]] = i;

}

cin >> T;

//// PART 2: Preprocess input ////

/\*

Range will become:

(first element to next bucket\_element first element -1)

\*/

sort(arr, arr + N);

//cout << "Case:" << endl;

int sum = 0;

vector<int>temp\_bucket;

for (int i = 0; i < N; i++) {

if (arr[i] > sum) {

key\_arr.push\_back(arr[i] - 1);

bucket\_element[arr[i] - 1] = temp\_bucket;

temp\_bucket.clear();

}

sum += arr[i];

temp\_bucket.push\_back(arr[i]);

}

//remember the last bucket!

bucket\_element[sum] = temp\_bucket;

key\_arr.push\_back(sum);

//print\_bucket\_element();

//// PART 3: from bucket\_element to bucket\_combination ////

map<int, vector<int>>::iterator ite; //ite points to bucket\_element

for (ite = bucket\_element.begin(); ite != bucket\_element.end(); ite++) {

map<int, vector<int>>this\_bucket\_comb; //combination for the elements within this bucket

this\_bucket\_comb.clear();

int ele\_cnt = ite->second.size();

for (int i = 0; i < (1 << ele\_cnt); i++) {

int temp\_sum = 0;

int temp\_cnt = 0;

vector<int>temp\_used;

temp\_used.clear();

for (int j = 0; j < ele\_cnt; j++) {

if (i >> j & 1) { //use the j-th element in the bucket

temp\_cnt++;

temp\_used.push\_back(ite->second[j]);

temp\_sum += ite->second[j];

}

}

if (this\_bucket\_comb.find(temp\_sum) == this\_bucket\_comb.end() || this\_bucket\_comb[temp\_sum].size() > temp\_cnt) {

this\_bucket\_comb[temp\_sum].clear();

for (int i = 0; i < temp\_used.size(); i++) {

this\_bucket\_comb[temp\_sum].push\_back(temp\_used[i]);

}

}

}

bucket\_combinations[ite->first] = this\_bucket\_comb;

}

//print\_bucket\_combinations();

//// PART 4: from Target to range ////

int target = T;

vector<int>temp\_v;

temp\_v.clear();

set<int>temp\_s;

temp\_s.clear();

dfs(target, temp\_v, temp\_s);

//// PART 6: Output ////

cout << "Solution: ";

if (Solution.size() > 0) {

for (int i = 0; i < Solution.size(); i++) {

//cout << Solution[i] << " ";

cout<<num\_to\_idx[Solution[i]]<<" ";

}

cout << endl;

}

else {

cout << "impossible" << endl;

}

}

}

int num\_to\_key(int num) {

int idx = lower\_bound(key\_arr.begin(), key\_arr.end(), num) - key\_arr.begin();

if (idx >= key\_arr.size()) {

return -1;

}

return key\_arr[idx];

}

void print\_bucket\_element() {

cout << "Bucket & its elements:" << endl;

map<int, vector<int>>::iterator ite;

for (ite = bucket\_element.begin(); ite != bucket\_element.end(); ite++) {

if (ite == bucket\_element.begin())continue;

cout << "Key range: " << ite->second[0] << " " << ite->first << endl;

cout << "Numbers: ";

for (int i = 0; i < ite->second.size(); i++) {

cout << ite->second[i] << " ";

}

cout << endl;

cout << endl;

}

cout << endl;

}

void print\_bucket\_combinations() {

cout << "Bucket & its inner elements' combinations:" << endl;

map<int, map<int, vector<int>>>::iterator cite; //combination ite

for (cite = bucket\_combinations.begin(); cite != bucket\_combinations.end(); cite++) {

if (cite == bucket\_combinations.begin())continue;

cout << "Key: " << cite->first << endl;

cout << "Combinations: " << endl;

map<int, vector<int>>::iterator vite; //value ite

for (vite = (cite->second).begin(); vite != (cite->second).end(); vite++) {

cout << "Value: " << vite->first << " =";

for (int i = 0; i < vite->second.size(); i++) {

cout << vite->second[i] << " ";

}

cout << endl;

}

cout << endl;

}

}

void dfs(int t, vector<int>sol,set<int>used\_bucket) {

if (t == 66) {

cout << "here";

}

//end circumstance

if (t == 0) {

if (Solution.size() == 0 || Solution.size() > sol.size()) {

Solution.clear();

for (int i = 0; i < sol.size(); i++) {

Solution.push\_back(sol[i]);

}

}

return;

}

//find possible combinations in target bucket

int b\_key = num\_to\_key(t);

//if this bucket is already used or no bucket available

if (b\_key == -1 || used\_bucket.find(b\_key) != used\_bucket.end()) {

return;

}

else {

map<int, vector<int>>t\_bucket = bucket\_combinations[b\_key];

map<int, vector<int>>::iterator ite;

for (ite = t\_bucket.begin(); ite != t\_bucket.end(); ite++) {

//try this

if (ite->first <= t) {

vector<int>new\_vec;

new\_vec.clear();

//copy(sol.begin(), sol.end(), new\_vec.begin());

for (int i = 0; i < sol.size(); i++) {

new\_vec.push\_back(sol[i]);

}

new\_vec.insert(new\_vec.end(), ite->second.begin(), ite->second.end());

set<int>new\_used\_b;

new\_used\_b.clear();

new\_used\_b = used\_bucket;

new\_used\_b.insert(b\_key);

dfs(t - ite->first, new\_vec, new\_used\_b);

}

else {

continue;

}

//return to orig state

}

return;

}

**}**

**Performance:**

This approach’s performance highly depends on the nature of the array. One of the best test case will be arr={1,2,4,8,16,32,64 …} since each bucket only has one element and with binary search we may find the target bucket with O(logN) repetitively until we decrease the subproblem to 0. On the other hand, the worst case scenarios will be those array whose elements are relatively close and all elements go into the same bucket, making it a 2^N combinations generation at step 2.

**Test cases used & corresponding answers**

Test cases:

8

1 2 6 3 17 82 23 234

26

8

1 2 6 3 17 82 23 234

40

8

1 2 6 3 17 82 23 234

23

7

1 2 3 4 11 19 41

59

7

1 2 3 4 11 19 42

41

15

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

25

30

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

142

30

1 2 3 4 5 16 17 18 19 20 106 107 108 109 110 646 647 648 649 650 3886 3887 3888 3889 3890 23326 23327 23328 23329 23330

14285

Answer:

3 6

4 6

6

2 3 4 6

impossible

11 12

20 21 22 23 24 26

4 8 16 17 18 19 22 23 24

Note that the last two test cases are used to compare the performance of Approach 2 and Approach 3. Approach 3 performs bad with the sequence one but outperform Approach 2 in the sparse one.