

Chapter 5

Complete search

Complete search is a general method that can be used to solve almost any algorithm problem. The idea is to generate all possible solutions to the problem using brute force, and then select the best solution or count the number of solutions, depending on the problem.

Complete search is a good technique if there is enough time to go through all the solutions, because the search is usually easy to implement and it always gives the correct answer. If complete search is too slow, other techniques, such as greedy algorithms or dynamic programming, may be needed.

Generating subsets

We first consider the problem of generating all subsets of a set of n elements. For example, the subsets of $\{0, 1, 2\}$ are \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ and $\{0, 1, 2\}$. There are two common methods to generate subsets: we can either perform a recursive search or exploit the bit representation of integers.

Method 1

An elegant way to go through all subsets of a set is to use recursion. The following function `search` generates the subsets of the set $\{0, 1, \dots, n - 1\}$. The function maintains a vector `subset` that will contain the elements of each subset. The search begins when the function is called with parameter 0.

```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

When the function `search` is called with parameter k , it decides whether to include the element k in the subset or not, and in both cases, then calls itself with parameter $k + 1$. However, if $k = n$, the function notices that all elements have been processed and a subset has been generated.

The following tree illustrates the function calls when $n = 3$. We can always choose either the left branch (k is not included in the subset) or the right branch (k is included in the subset).



Method 2

Another way to generate subsets is based on the bit representation of integers. Each subset of a set of n elements can be represented as a sequence of n bits, which corresponds to an integer between $0 \dots 2^n - 1$. The ones in the bit sequence indicate which elements are included in the subset.

The usual convention is that the last bit corresponds to element 0, the second last bit corresponds to element 1, and so on. For example, the bit representation of 25 is 11001, which corresponds to the subset $\{0, 3, 4\}$.

The following code goes through the subsets of a set of n elements

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}

```

The following code shows how we can find the elements of a subset that corresponds to a bit sequence. When processing each subset, the code builds a vector that contains the elements in the subset.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b & (1<<i)) subset.push_back(i);
    }
}

```

Generating permutations

Next we consider the problem of generating all permutations of a set of n elements. For example, the permutations of $\{0, 1, 2\}$ are $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ and $(2, 1, 0)$. Again, there are two approaches: we can either use recursion or go through the permutations iteratively.

Method 1

Like subsets, permutations can be generated using recursion. The following function `search` goes through the permutations of the set $\{0, 1, \dots, n - 1\}$. The function builds a vector permutation that contains the permutation, and the search begins when the function is called without parameters.

```
void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

Each function call adds a new element to permutation. The array `chosen` indicates which elements are already included in the permutation. If the size of permutation equals the size of the set, a permutation has been generated.

Method 2

Another method for generating permutations is to begin with the permutation $\{0, 1, \dots, n - 1\}$ and repeatedly use a function that constructs the next permutation in increasing order. The C++ standard library contains the function `next_permutation` that can be used for this:

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));
```

Backtracking

A **backtracking** algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the problem of calculating the number of ways n queens can be placed on an $n \times n$ chessboard so that no two queens attack each other. For example, when $n = 4$, there are two possible solutions:



The problem can be solved using backtracking by placing queens to the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed on the board.

For example, when $n = 4$, some partial solutions generated by the backtracking algorithm are as follows:



At the bottom level, the three first configurations are illegal, because the queens attack each other. However, the fourth configuration is valid and it can be extended to a complete solution by placing two more queens to the board. There is only one way to place the two remaining queens.

The algorithm can be implemented as follows:

```

void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}

```

The search begins by calling `search(0)`. The size of the board is $n \times n$, and the code calculates the number of solutions to count.

The code assumes that the rows and columns of the board are numbered from 0 to $n - 1$. When the function `search` is called with parameter y , it places a queen on row y and then calls itself with parameter $y + 1$. Then, if $y = n$, a solution has been found and the variable `count` is increased by one.

The array `column` keeps track of columns that contain a queen, and the arrays `diag1` and `diag2` keep track of diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, the columns and diagonals of the 4×4 board are numbered as follows:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

Let $q(n)$ denote the number of ways to place n queens on an $n \times n$ chessboard. The above backtracking algorithm tells us that, for example, $q(8) = 92$. When n increases, the search quickly becomes slow, because the number of solutions increases exponentially. For example, calculating $q(16) = 14772512$ using the above algorithm already takes about a minute on a modern computer¹.

Pruning the search

We can often optimize backtracking by pruning the search tree. The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.

¹There is no known way to efficiently calculate larger values of $q(n)$. The current record is $q(27) = 234907967154122528$, calculated in 2016 [55].

Let us consider the problem of calculating the number of paths in an $n \times n$ grid from the upper-left corner to the lower-right corner such that the path visits each square exactly once. For example, in a 7×7 grid, there are 111712 such paths. One of the paths is as follows:



We focus on the 7×7 case, because its level of difficulty is appropriate to our needs. We begin with a straightforward backtracking algorithm, and then optimize it step by step using observations of how the search can be pruned. After each optimization, we measure the running time of the algorithm and the number of recursive calls, so that we clearly see the effect of each optimization on the efficiency of the search.

Basic algorithm

The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

- running time: 483 seconds
- number of recursive calls: 76 billion

Optimization 1

In any solution, we first move one step down or right. There are always two paths that are symmetric about the diagonal of the grid after the first step. For example, the following paths are symmetric:



Hence, we can decide that we always first move one step down (or right), and finally multiply the number of solutions by two.

- running time: 244 seconds
- number of recursive calls: 38 billion

Optimization 2

If the path reaches the lower-right square before it has visited all other squares of the grid, it is clear that it will not be possible to complete the solution. An example of this is the following path:



Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- running time: 119 seconds
- number of recursive calls: 20 billion

Optimization 3

If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares. For example, in the following situation, the path can turn either left or right:



In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Optimization 4

The idea of Optimization 3 can be generalized: if the path cannot continue forward but can turn either left or right, the grid splits into two parts that both contain unvisited squares. For example, consider the following path:

The idea is to divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S_A . Correspondingly, the second search creates a list S_B from B . After this, it suffices to check if it is possible to choose one element from S_A and another element from S_B such that their sum is x . This is possible exactly when there is a way to form the sum x using the numbers of the original list.

For example, suppose that the list is $[2, 4, 5, 9]$ and $x = 15$. First, we divide the list into $A = [2, 4]$ and $B = [5, 9]$. After this, we create lists $S_A = [0, 2, 4, 6]$ and $S_B = [0, 5, 9, 14]$. In this case, the sum $x = 15$ is possible to form, because S_A contains the sum 6, S_B contains the sum 9, and $6 + 9 = 15$. This corresponds to the solution $[2, 4, 9]$.

We can implement the algorithm so that its time complexity is $O(2^{n/2})$. First, we generate *sorted* lists S_A and S_B , which can be done in $O(2^{n/2})$ time using a merge-like technique. After this, since the lists are sorted, we can check in $O(2^{n/2})$ time if the sum x can be created from S_A and S_B .

