NOI.PH Training: Backtracking 1

Complete Search

Carl Joshua Quines

Contents

1 Brute force

We'll draw a distinction between the terms **brute force** and **complete search**. Brute force describes a naive way to solve a problem. It's kind of hard to give a good definition of this, so here are some examples.

Example 1.1. Consider **Anion Find:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/anion-find. The brute force method of solving this would be, for each cation with charge x, to scan the whole grid looking for its partner.

Example 1.2. Consider **Aswang Immortal - Trivia!:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/aswang-immortal-trivia. The brute force method of solving this would be, for each question that is wrong, marking which questions are wrong, and summing up the number of points each question is worth.

Example 1.3. Consider **Evening Gown:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/evening-gown. The brute force method of solving this would be to enumerate all possible paths and finding the one with the least energy.

All of these are brute force approaches, but not all of these are complete search approaches. Complete search refers to the technique of **solving a problem by considering all possibilities**. Of the brute force methods described above, the only complete search method is the one described for Evening Gown.

Try the following exercise before continuing.[1]

Exercise 1.4. Consider the five problems from this year's finals round 1. Which of these problems can be solved with complete search?

1.1 Subtask 1

Why should we care about learning how to brute force? In IOI-style contests, it is almost always true that **one of the first subtasks of a problem can be solved with brute force**. It sometimes happens that you can't make any progress on a problem at all, but you can always try to solve the brute force subtasks. And any points are better than zero points.

It may only count for a few points, but **a few points can mean the difference between first and second place,** as the author will attest. These few points, multiplied over several problems, can also sum up to a lot of points. Programming the brute force quickly and submitting it can lead to getting these crucial points.

A brute force solution is also helpful for **finding and checking solutions for harder subtasks**. Answers may reveal patterns in the data that can lead to a complete solution. And if you're getting a wrong answer verdict, one extremely helpful technique is comparing your solution to the brute force solution for small cases, and figuring out where they differ. This

[1] I strongly encourage you to think about all the exercises in the article. Make sure you at least see the answer before continuing reading. You don't need to send your answers to me. Feel free to discuss the exercises in the article on the NOI.PH Training Discord server.

technique was extremely helpful when I solved Sorting the Planets, for example, since that's the kind of problem where it's easy to come up with wrong heuristics.

For ICPC-style contests, it's sometimes the case that **a problem is meant to be solved through brute force**. These aren't as common now, and when they do appear they're only a small proportion of the problemset. However, since brute force solutions are easy to come up with and easy to debug, it's a helpful skill for teams to be able to quickly write brute force solutions.

Why is it important to learn complete search in particular? Based on a small sample of NOI.PH problems, complete search is the most common kind of brute force needed to solve a problem. And **complete search is the hardest kind of brute force to write**; it's easy to write simulators and to process queries naively, but "enumerate all trees" is not straightforward to write.

One final note about brute force solutions. It's important to estimate the complexity of our solution before we begin programming it, in order to see if it will really pass the bounds. We'll discuss more of this throughout the article.

1.2 Strategy notes

I quote Kevin when he says "Not getting the easy and free points is much more painful than not getting that clever insight."

Always write the brute force for a problem. This is something I regret not doing in my competition days. If you anticipate spending more than thirty minutes solving a problem (which will *almost always* be the case in an IOI-style contest), write a brute force for it and get the easy subtasks.

177	Martin Dinev	Macedonia	30	11	15	100	0	19	175	29.17%
178	Urban Duh	Slovenia	100	17	15	6	0	36	174	29.00%
179	CJ Quines	Philippines	100	5	49	0	0	19	173	28.83%
179	Long Huen Chan	Hong Kong	97	5	15	2	18	36	173	28.83%
179	Reijer van Harten	Netherlands	100	5	15	53	0	0	173	28.83%

IOI 2018. The bronze cutoff was 187 points.

Think about it. For many problems, you should be able to write the brute force within ten minutes.[2] You get *guaranteed* returns, and you can use the brute force to help debug! It's a very low time cost, a very low risk thing to do, and yet the reward is big.

If you remember only one thing from this week, let it be this lesson. It's so important I will say it again. **Always write the brute force.**

[2] If you can't, practice until you *can* write it in ten minutes.

2 Iterative complete search

Consider the following questions:

- Is there a subset of a given list that sums to x?
- How many paths in an $m \times n$ grid don't pass through k given marked squares?
- Which pairs of five digit numbers collectively use all the digits from 0 to 9, and have a quotient that is an integer?

Taking a complete search perspective, we can rewrite these questions like so:

- Enumerate all subsets of a given list. Consider the ones that sum to x. Print whether such a subset exists.
- Enumerate all the paths in an $m \times n$ grid. Consider only the ones that pass through k given squares. Count them.
- Enumerate all pairs of five digit numbers. Consider only the ones that collective use all the digits from 0 to 9, and have a quotient that is an integer. Print all of them.

This is a perspective we will call **filtering**.[3] The complete search solution is **enumerating all possibilities, and only taking the ones that work**.

Example 2.1. Consider this problem: *Find b distinct positive integers, each below k, such that their sum is n, if it exists.* There are multiple ways to rewrite it from a filtering perspective:

- Enumerate all sets of b distinct positive integers, each below k. Consider only the ones whose sum is n.
- Enumerate all sets of positive integers whose sum is n. Consider only the ones that have b distinct positive integers, each below k.
- Enumerate all sets of b positive integers whose sum is n. Consider only the ones that have distinct positive integers, each below k.

It turns out that some of these ways are easier to code, and some of these ways are faster. So it doesn't mean that we have no decisions to make when doing a complete search: sometimes we have to be smart and choose the best way. We'll talk more about this later on.

Exercise 2.2. Rewrite the following problem from a filtering perspective: *How many integers less than n have exactly two distinct digits in its decimal representation?* Can you come up with two genuinely different ways to rewrite this? Which one would run faster?

2.1 "Enumerate all"

The hard part in the filtering perspective isn't the *only taking the possibilities that work* part. Often, this is done through a simple `if`. **The hard part in filtering is enumerating all possibilities** in the first place, which will be the focus of this article. Here are some of the basic "enumerate all" problems:

- Enumerate all subarrays of a given array.

[3] This is not a formal term. I'm making this up.

- Enumerate all subgrids of a given grid.
- Enumerate all subsets of a given set.
- Enumerate all permutations of a given set.
- Enumerate all tuples $(a_1, a_2, \ldots, a_n)$ of positive integers, each less than k.
- Enumerate all subsets of a given set with a certain size.
- Enumerate all paths in a given grid or graph.
- Enumerate all distinct rooted trees with n vertices.

As an example of the first problem in the above list, consider printing all subarrays of a given array. We can do this through something like:

```
// Input is in array a, of size n.
for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
    // Then a[i], a[i+1], ..., a[j] is a subarray.
    for (int k = i; k <= j; k++) {
      cout << a[k] << ' ';
    }
    cout << '\n';
  }
}
```

This goes through all $\mathcal{O}(n^2)$ subarrays, printing each in $\mathcal{O}(n)$, so the whole program is $\mathcal{O}(n^3)$. This is an example of what we call **iterative complete search**. The word *iterative* refers to the fact that we're executing a loop several times until a condition is met. We *iterate* through all possibilities, going through them one-by-one.

Now that we have iterated the code, we can do the filtering part of only taking the possibilities that work. For example, the following code counts the number of subarrays of a given array that have sum at most m:

```
// Input is in array a, of size n.
int count = 0;
for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
    int sum = 0;
    for (int k = i; k <= j; k++) sum += a[k];
    if (sum <= m) count++;
  }
}
cout << count << '\n';
```

Again, the complexity here is $\mathcal{O}(n^3)$.

Exercise 2.3. Modify the above code to solve the following problem: *For each subgrid of a given grid, find its sum. Then print these sums in sorted order.* If the grid has size $m \times n$, your code should run in $\mathcal{O}(m^3 n^3)$.

Here's another example which prints all subsets of $\{0, 1, \ldots, n-1\}$.

```cpp
// Input is the integer n.
for (int i = 0; i < (1 << n); i++) {
  // Then the bits of i that are 1 represent a subset of {0, 1, .., n-1}.
  for (int j = 0; j < n; j++) {
    if (i & (1 << j)) { // If the jth bit is on,
      cout << j << ' ';
    }
  }
  cout << '\n';
}
```

Exercise 2.4. Convince yourself this works. (If you don't understand, ask!)

Exercise 2.5. Consider **Yet Another Packing Problem:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/yet-another-packing-problem. If we solve this by enumerating all possible sets, which subtasks will it solve? Modify the above code to do this.

2.2 `next_permutation()`

The C++ standard library contains the helpful function `next_permutation()`. This allows to easily **iterate through all permutations of a given array**. For example, this code prints out all permutations of the set $\{1, 2, 3, 4, 5\}$:

```cpp
vector<int> a = {1, 2, 3, 4, 5};

do {
  for (int i : a) cout << i << ' ';
  cout << '\n';
} while (next_permutation(a.begin(), a.end()));
```

What `next_permutation()` does is that it takes a given array, and swaps elements around until it gets the next *lexicographically bigger* permutation of the array. It returns `false` if no such permutation exists, and then transforms it to the first permutation.

Example 2.6. If we run `next_permutation()` on the array `{1, 1, 2, 2}`, we don't get `{1, 1, 2, 2}` again, but we get `{1, 2, 1, 2}`. More generally, `next_permutation()` only cares about *distinct* permutations.

Exercise 2.7. Solve **The Next Number:** https://code.google.com/codejam/contest/dashboard?c=186264#s=p1. Careful with leading zeroes!

2.3 Nested loops

We now deal with the fifth problem on our list of "enumerate all" problems, which is to enumerate all tuples $(a_1, a_2, \ldots, a_n)$ of positive integers, each less than k. Consider the problem of printing all such tuples with no two consecutive elements. For small n, this isn't too hard to solve using nested loops:

```
for (int a = 1; a <= k; a++) {
  for (int b = 1; b <= k; b++) {
    for (int c = 1; c <= k; c++) {
      for (int d = 1; d <= k; d++) {
        if (a != b && b != c && c != d) {
          cout << a << ' ' << b << ' ' << c << ' ' << d << '\n';
        }
      }
    }
  }
}
```

Now I'd like to introduce the concept of pruning, which we'll talk about in more detail later. What we're doing here is that we're generating all k^4 possible tuples, and only in the end are we taking the possibilities that work. In fact, our code will be faster if we filtered out the possibilities earlier:

```
for (int a = 1; a <= k; a++) {
  for (int b = 1; b <= k; b++) {
    if (b != a) {
      for (int c = 1; c <= k; c++) {
        if (c != b) {
          for (int d = 1; d <= k; d++) {
            if (d != c) {
              cout << a << ' ' << b << ' ' << c << ' ' << d << '\n';
            }
          }
        }
      }
    }
  }
}
```

In this example, we're removing possibilities that don't work as early as we can. Rather than enumerating all k^4 tuples (a, b, c, d), we already discard all tuples with $a = b$ without looking at the rest of the tuple. This is called **pruning**, where we remove partial solutions that we know won't complete into a full solution.

> **Exercise 2.8.** Solve **Simple Equations:** UVa 11565. Try to write code that prunes as early as possible.

2.4 But which possibilities?

As our last example for iterative complete search, consider **Lotto:** UVa 441. There are several ways to rephrase this problem from a filtering perspective:

- Enumerate all possible subsets, and for each one that has 6 elements, sort it. Then sort the whole list. This one considers at most 2^{12} subsets, but it's inefficient. It's doable, but it's tedious to code. We can simply generate the ones that have 6 elements directly, without considering each possible subset.
- Enumerate all possible tuples of 6 elements, and consider only the ones that have distinct elements and are sorted, then sort them. This one considers at most 12^6 different tuples, and thus around $6 \cdot 12^6$ operations, or around 17 million. And then we need to sort them. Again, it's doable, but tedious.
- Enumerate all possible tuples of 6 distinct elements, and consider only the ones that are sorted, then sort them. This is similar to the approach we did in the previous section, where we can filter out ones that don't have distinct elements earlier on. Doable, but again, I'd rather not do this.
- It turns out that we can enumerate all possible tuples directly, in such a way that we go over them in a sorted manner. Since the input is already sorted, we can do something like:

```
// Input in array s, of size n.
for (int a = 0; a < n - 5; a++) {
  for (int b = a + 1; b < n - 4; b++) {
    for (int c = b + 1; c < n - 3; c++) {
      for (int d = c + 1; d < n - 2; d++) {
        for (int e = d + 1; e < n - 1; e++) {
          for (int f = e + 1; f < n; f++) {
            cout << s[a] << ' ' << s[b] << ' ' << s[c] << ' ' << s[d] << ' ' <<
            ↪  s[e] << ' ' << s[f] << '\n';
          }
        }
      }
    }
  }
}
```

Exercise 2.9. Convince yourself that the above program works and runs within the time limit.

As mentioned earlier, **just because we're solving the problem using complete search doesn't mean we don't have any decisions to make**! Often, we need to take the correct perspective in order to solve a problem with complete search. The problems listed at the end of this section all involve complete search, but that doesn't mean you don't have to think about the best way to do it!

3 Recursive backtracking

Our previous solution for enumerating all tuples $(a_1, a_2, \ldots, a_n)$ of positive integers, each less than k, quickly runs into a problem with large n.

Example 3.1. Consider **Exchange Gift:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/exchange-gift. If we enumerate all possible boxes and simulate all queries manually, that will take $\mathcal{O}(3^N BN)$, which is enough to solve the second subtask.

We can enumerate all possible boxes with N nested loops. The problem is, N can be anything from 1 to 12, so we'd have to write a nested loops for each possible N. Even if we combined everything to one huge nested loop, we'd still have to write a nested loop with 12 levels!

In this section we introduce the concept of recursive backtracking. This will solve a lot more "enumerate all" problems than our previous techniques did.

3.1 The generating perspective

Consider these questions:

- Find all sets of b distinct positive integers, each below k, such that their sum is n.
- Is there a subset of a given list that sums to x?
- How many paths in an $m \times n$ grid don't pass through k given marked squares?

Taking a different perspective, we can rewrite these questions like so:

- Start with an empty set. Add in each possible integer less than k, such that it's not currently in the set and the sum doesn't go over n. If we do get a sum of n, print that set.
- Start with an empty set. Add in elements of the list one by one, so that the sum doesn't go over x. If we do get a sum of x, print that such a subset exists.
- Start with the first square in the path. Try each possible next square we can go to. If we do get a path that works, add one to our count.

Instead of enumerating all possibilities and taking the ones that we want, what we called *filtering*, we're now **building up the candidates that work** step-by-step. We'll call this approach **generating**.[4]

In the previous section, what we were doing was called iterative complete search. We iterated over all possibilities and chose the ones that we wanted. On the other hand, the above approaches don't iterate over all possibilities in order, it builds up the candidate as it goes. The end result is the same, but philosophically, they're quite different.

We've already been doing some generating rather than filtering. Recall our problem of printing tuples with distinct elements, where we did *pruning*: we were discarding candidates that we know won't complete into a full solution. Pruning candidates as soon as we know they won't work is one of the main ideas of generating.

[4] Again, this is a term I'm making up. There's also no formal distinction between filtering and generating. I'm introducing the terms in the hopes that it makes the explanation clearer.

3.2 Goodbye, nested loops!

The other main idea is that generating is much easier to do with *recursion*. This is going to look like magic the first time you see it. Consider the previous problem of enumerating all tuples of n positive integers, each less than k, such that no two consecutive elements are the same. Here's the clean way to solve this problem:

```
int a[N]; // Stores the current tuple.

void enumerate_tuples(int n, int i) { // Sets the ith entry of a.
  if (i == n) { // If we've set everything,
    for (int j = 0; j < n; j++) {
      cout << a[j] << ' ';
    }
    cout << '\n';
  } else {
    for (int j = 1; j < k; j++) {
      if (a[i-1] != j) { // For each possible j,
        a[i] = j; // set a[i] = j.
        enumerate_tuples(n, i + 1); // Then set the (i+1)st entry.
      }
    }
  }
}
```

The initial call is `enumerate_tuples(n, 0)`. This is much better than enumerating all the tuples using for loops!

Exercise 3.2. Take a few moments to try to understand for yourself how the above code works. It might help to run a specific case: try taking $n = k = 3$.

What `enumerate_tuples()` does should *feel* more like generating. For each candidate, it tries all of the possible next candidates that work. Then it moves to the next candidate. This feels very *different* than enumerating all possibilities and taking only the ones that work.

The way it's programmed is also very different. Instead of *iterating*, we're now doing *recursion*: we're making a function that calls itself.

Exercise 3.3. Consider the problem Exchange Gift, mentioned earlier. Modify the above code to print out all possible boxes of length N. Your code should print out 3^N boxes.

Exercise 3.4. Modify the above code to print all permutations of $\{0, 1, \ldots, n-1\}$. Your code should run in $\mathcal{O}\left(n^{n+1}\right)$.

3.3 Eight queens

In the hopes of making things clearer, let's look at another classic example of using recursion to enumerate all possibilities.

Example 3.5 (Eight queens)**.** In an 8×8 board, find all possible ways to place eight queens such that no two queens attack each other.

Here are some filtering approaches to this problem:

- Try all choices of 8 out of 64 squares. You could program this with loops nested 8 times. There are $\binom{64}{8}$ possibilities, which is around 4 billion—much too large.
- Each row can only have one queen, so try all 8 possibilities of placing a queen in each of the 8 rows. Again, you can program this with loops nested 8 times, but this is much more doable to program. This is still 8^8 possibilities, or around 17 million; doable, but still big.
- But each column can only have one queen too, so we don't have to try all 8^8 tuples, but only 8! permutations. For example, you could describe a placement of queens like $\{1, 3, 0, 2, 5, 4, 7, 6\}$: the queen in row 0 is placed in column 1, the queen in row 1 is placed in column 3, etc.

  This is 8! possibilities, which is only around 40 thousand. This is much more doable, and you only need to filter out ones where the diagonal condition doesn't hold. But there is still a way that considers much less possibilities than 40 thousand…

and that's by using a generative approach. Here, we *generate* the placement of queens row-by-row:

```
int col[8]; // The queen in row r is placed in column col[r].
// Initially all the queens are unplaced; the board is blank.

// Can we currently place a queen at row r and column c?
bool can_place(int r, int c) {} // Omitted.

void enumerate_positions(int i) { // Set the ith entry of col.
  if (i == 8) { // If we've set everything,
    for (int j = 0; j < n; j++) {
      cout << col[j] << ' ';
    }
    cout << '\n';
  } else {
    for (int j = 0; j < 8; j++) {
      if (can_place(i, j)) { // For each possible j,
        col[i] = j; // set col[i] = j.
        enumerate_positions(i + 1); // Then set the (i+1)st entry.
      }
    }
  }
}
```

Exercise 3.6. What happens when `enumerate_positions()` can't place the queen on the ith row on any column?

Exercise 3.7. Try running the code yourself. No, not running it with a computer, but try executing the algorithm by hand. You can use this page as a chessboard: http://www.hbmeyer.de/backtrack/achtdamen/eight.htm. Run the code until you understand what's happening; because it'll take quite a while before you get a valid placement.

Exercise 3.8. Fill out the `can_place()` function above. Check that your code prints 92 solutions.

3.4 Recursive backtracking

And now we take a step back and take a look at what's happening:

Exercise 3.9. Compare `enumerate_positions()` to `enumerate_tuples()`. How are they similar? What are the parts that are changed?

Both of these functions follow the same general pattern:

```
def backtrack(candidate):
  if candidate is done:
    do something
  else:
    for each possible next_candidate:
      candidate = next_candidate
      backtrack(next_candidate)
```

Then `backtrack(root)` is called, for the "empty" candidate. This pattern is called **recursive backtracking**. The *recursive* part is from how it uses recursion: it's a function that calls itself. The *backtracking* part is how, if a partial solution doesn't have a next candidate, it goes *up* to the previous candidate. Hence it *backtracks*.

Exercise 3.10. If you're familiar with depth-first search, consider the following rooted tree. Each candidate solution has a vertex, with the "empty" candidate as the root vertex. The children of each vertex is each possible next candidate from that vertex. How does the `backtrack()` function search this tree?

What makes this approach generative is how the *each possible next candidate* part is done. In `enumerate_tuples()`, a partial candidate was extended by setting a possible next element in the tuple. In `enumerate_positions()`, a partial candidate was extended by adding a possible queen in the next row.

We only **extend a candidate to solutions that seem possible**, or in other words, we're pruning candidates that we know aren't possible. Figuring out how to program a backtracking solution involves two pieces:

- Figure out how to extend the candidate.
- Figure out how to represent the candidate.

These aren't mutually exclusive: you want to **represent the candidate in a way that makes it easy to extend**. The next example should make it clear what I'm talking about.

3.5 Combinations

We *finally* have enough to tackle the next "enumerate all" problem. Given a set with n elements, enumerate all subsets with size k. Suppose we're just trying to find the sum of each subset, and print all such sums in some order.

We can try to use a filtering approach. We can enumerate all 2^n possible subsets, and take only the ones of a certain size. But this is really inefficient: **we're looking at 2^n possibilities when only $\binom{n}{k} = \mathcal{O}(n^k)$ work.**[5] Instead, we'll tackle this using recursive backtracking.

How do we extend a candidate? If our candidate has k elements, we're done. Otherwise, we try all possible next elements that come after the last element we added. To be able to do that, we need to represent both the current number of elements, and the index of the last element.

What else do we need to be part of the representation? Well, we need the sum of each subset, so we keep a running total of the sum. Note that we don't actually need the subset itself as part of our representation. Here's the program:

```
1  int arr[N]; // Set with n elements.
2
3  // Representing the candidate:
4  // s is the current number of elements,
5  // i is the index of the last element, and
6  // sum is the running total of the sum.
7  void backtrack(int s, int i, int sum) {
8    if (s == k) {
9      cout << sum << '\n';
10   } else {
11     for (int j = i + 1; j < n; j++) {
12       backtrack(s + 1, j, sum + arr[j]);
13     }
14   }
15 }
```

Then calling `backtrack(0, -1, 0)` solves the problem.

> **Exercise 3.11.** Convince yourself this works. What's the complexity?

> **Exercise 3.12.** Modify line 11 to read `for (int j = i + 1; j <= n - (k - s); j++)` instead. Does it still work? Why or why not?

Sometimes, we want to make the variables representing a candidate global instead of being passed as a parameter. In particular, it's easier to code a backtracking solution if **arrays are declared as global**. Here's how to modify the above code so that `sum` is global:

```
1  int sum;
2
3  void backtrack(int s, int i) {
```

[5] Doing CS is so cool, I can just say $\binom{n}{k} \approx n^k$.

```
4     if (s == k) {
5       cout << sum << '\n';
6     } else {
7       for (int j = i + 1; j < n; j++) {
8         sum += arr[j];
9         backtrack(s + 1, j);
10        sum -= arr[j]; // This is important!
11      }
12    }
13  }
```

Exercise 3.13. Why is line 10 in the above code important?

Exercise 3.14. *Do not discuss this exercise on the Discord.* Modify the above code to solve **Basic subset enumeration:** https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-subset-enumeration. You want the array storing the current subset to be a global variable. (Alternatively, you can pass the current subset by reference.)

3.6 Paths in a grid

For our last example, we'll tackle the next "enumerate all" problem: enumerating paths in a grid. We'll leave the last "enumerate all" problem, along with a couple more, as problems. The problem we'll solve is **Evening Gown:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/evening-gown.

How do we extend a candidate? We need to know our current position, of course. And we need to know how many steps we've taken. We also need to output the minimum amount of energy, so that should be part of our representation as well.

That gives us our representation: the row, the column, the number of steps, and the amount of energy. Now we just put it all together and backtrack:

```
1   int e[N][N]; // Energy needed to step on each stone.
2   int answer = 999999999;
3
4   int dx[] = {0, 1, 0, -1};
5   int dy[] = {1, 0, -1, 0};
6
7   void backtrack(int r, int c, int steps, int energy) {
8     if (r == m - 1 && c == n - 1) {
9       if (energy < answer) answer = energy;
10    } else {
11      for (int k = 0; k < 4; k++) {
12        int nr = r + dx[k];
13        int nc = c + dy[k];
14        if (0 <= nr && nr < m && 0 <= nc && nc < n && (steps + 1 - e[nr][nc]) %
          ↪  3 != 0) { // Move to (nr, nc):
```

```
            backtrack(nr, nc, steps + 1, energy + e[nr][nc]);
        }
      }
    }
}
```

Then we call `backtrack(0, 0, 0, 0)`, and then `answer` will contain the answer to the problem. Unfortunately, we have a problem:

Exercise 3.15. *Do not discuss this exercise on the Discord.* Construct a grid such that the above code runs in an infinite loop.

To make our solution complete, then, we have to bound the number of steps so that it doesn't run in an infinite loop. It turns out that, for the first subtask, we can just stop trying to backtrack when we reach around 150 steps.

Exercise 3.16. *Do not discuss this exercise on the Discord.* Convince yourself that if a solution to a 6×6 board has more than 150 steps, there's a corresponding solution that has less than 150 steps and less energy. Use this to prove that stopping our backtracking after 150 steps doesn't discard the optimal solution.

Exercise 3.17. Solve the first subtask of **The Room:** https://www.hackerrank.com/contests/noi-ph-2019-finals-1-mirror/challenges/the-room.

Exercise 3.18. Solve the first two subtasks of **Start the Day on Wings:** https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/start-the-day-on-wings.

4 Iterative backtracking

This is an advanced section. I recommend that you not read this on your first reading. First-timers, **feel free to skip this section entirely**; only one of the problems will require this section.

In this section, we will eliminate the recursion involved in recursive backtracking in order to turn it to iterative backtracking. The main reason you want to do this is speed; iterative solutions sometimes reduce the constant by a sizable amount.

4.1 Let's get technical!

I'm going to throw a bunch of new phrases that may not make immediate sense; keep reading and bear with me. Internally, all recursion in a program is done using the **call stack**.

As a simplified explanation, suppose you call a function f on line x. The caller pushes line x, called the **return address** onto the call stack and runs the function f. When f is done, it pops x from the call stack and continues execution from line x, which is why it's called the return address.

This handles recursion quite nicely. However, the call stack *is* slower compared to doing something iteratively. This is because, in addition to the return address, the local variables, the function's arguments, and other metadata about the function is also pushed onto the call stack.

The idea in converting recursion into iteration is to simulate a call stack. But we can get a possible constant speed improvement by only storing the things we need on the stack, often just an integer or two. With backtracking, we often only deal with the case when the recursion is at the end of the function, which makes it easier to convert. In pseudocode, if you have

```
def f(p):
  do something
  f(q)
```

and begin with calling `f(a)`, you can convert this to

```
stack.push(a)
while not stack.empty():
  p = stack.pop()
  do something
  stack.push(q)
```

Observe how the only things we push onto the stack are the arguments for the function!

Exercise 4.1. Convince yourself that these are the same.

Exercise 4.2. Convert our program in subsection 3.2 to use iteration rather than recursion. Now test both of them, possibly on an online judge like https://codeforces.com/problemset/customtest. Do you get a speed improvement?

4.2 `next_permutation()`, revisited

There are several other ways to turn recursion into iteration. For example, using `next_permutation()` is iteration: we can use it to iterate over all possible permutations of an array.

The most interesting part about `next_permutation()` is how it's implemented. Here's the implementation in the standard:

```
template <class BidirIt>
bool next_permutation(BidirIt first, BidirIt last) {
  if (first == last) return false;
  BidirIt i = last;
  if (first == --i) return false;

  while (true) {
    BidirIt i1, i2;
    i1 = i;
    if (*--i < *i1) {
      i2 = last;
      while (!(*i < *--i2));
      std::iter_swap(i, i2);
      std::reverse(i1, last);
      return true;
    }
    if (i == first) {
      std::reverse(first, last);
      return false;
    }
  }
}
```

It's clear that this algorithm runs in linear time. Averaged out over all permutations, it runs in constant time. Instead of explaining how this works, I'm going to invite you to figure out how it works yourself, or read Word Aligned's wonderful explanation.

5 Problems

Please keep in mind the following policies, which will be largely followed for problem sets in the upcoming weeks:

Problem categories: Problems are divided in four categories, each with their own submission methods:

- The **W** problems are warm-ups, intended to be straightforward applications of the topics. They do not need to be submitted.
- The **N** problems are non-coding problems. These include proving problems, explanation problems, debugging problems, among others. All non-coding problems are to be submitted through the template in Google Classroom.
- The **C** problems are coding problems to be submitted in Google Classroom. The file name should be `bt1-CX.cpp` where `X` is the problem number. C++ is strongly recommended since it's currently the only language allowed in the IOI.
- The **S** problems are coding problems to be submitted to their respective online judges. These will be checked using ProgVar.Fun. Note that the only allowed languages are C++ and Java.

Points: Each problem will be worth a certain number of points, labeled next to each problem. You don't need to solve all problems; you just need to accumulate the amount of points outlined in each section. You should also solve any problems marked mandatory. In general, harder problems will be worth more points.

First-timers: First-timers and returning students have different requirements for the problem set. You are considered a returning student if you participated in last year's team round. In general, returning students will be assigned harder problems and are required to earn more points.

Collaboration: Since nothing is officially graded, collaboration is allowed for most problems. Please discuss these problems only on the NOI.PH Training Discord server. Be mindful to use spoiler tags to avoid spoiling others who do not want hints.[6]

Even if you collaborate, **any submissions should be completely in your own words or your own code**. Make sure to take note of who you collaborated with. Acknowledge your collaborators for each non-coding problem somewhere in your write-up, and mention them in a comment in your code for each coding problem.

[6] In Discord, writing `||this is a spoiler||` hides the words "this is a spoiler" using spoiler tags.

5.1 Warmup problems

Some of these are mentioned as exercises in the text, or solved in the text itself. If the text gives the solution to one of these problems, you may want to rewrite the code in the text in your own style—personally, this makes it much easier for me to remember how to code something.

You are allowed to discuss these problems on the Discord.

W1 **Simple Equations:** UVa 11565

W2 **Lotto:** UVa 441

W3 Subtask 1 of **Yet Another Packing Problem:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/yet-another-packing-problem

W4 **Preparing Olympiad:** https://codeforces.com/problemset/problem/550/B

W5 **PFAST Inc.:** https://codeforces.com/problemset/problem/114/B

W6 **backtracking1:** https://gitlab.com/jddantes/the-code-project/tree/master/backtracking/backtracking1

W7 **backtracking2:** https://gitlab.com/jddantes/the-code-project/tree/master/backtracking/backtracking2

W8 **backtracking3:** https://gitlab.com/jddantes/the-code-project/tree/master/backtracking/backtracking3

W9 **8 Queens Chess Problem:** UVa 750

W10 **Tavas and SaDDas:** https://codeforces.com/problemset/problem/535/B

W11 Subtask 1 of **Any Problem You Didn't Score Any Points for During NOI.PH Finals**.

5.2 Non-coding problems

No need to be formal or rigorous; as long as you're able to convince me, it's fine.

First timers, solve at least [14★]. Returning students, solve at least [18★]. Problems in red are mandatory.

You are allowed to discuss the [2★] problems on the Discord.

N1 [2★] Exercise 3.6.

N2 [2★] Exercise 3.12.

N3 [2★] Exercise 3.13.

N4 [3★] Exercise 3.15.

N5 The following program attempts to enumerate all non-negative integers with k decimal digits when `backtrack(0)` is called.

```
1  vector<int> a;
2
3  void backtrack(int d) {
4    if (d == k) {
5      for (int i : a) cout << i;
6      cout << '\n';
7    } else {
8      for (int i = 0; i <= 9; i++) {
9        a.push_back(i);
10       backtrack(d + 1);
11       a.pop_back();
12     }
13   }
14 }
```

(a) [2★] Why is line 11 in the above code important?

(b) [3★] As above, the program is incorrect. What's wrong with it, and how do you fix it?

N6 [3★] The following program attempts to enumerate all tilings of a row of n cells with either 1×2 dominoes or 1×1 squares when `backtrack(n)` is called. What's wrong with it, and how do you fix it?

```
1  vector<int> t;
2
3  void backtrack(int i) {
4    if (i == 0) {
5      for (int j : t) {
6          if (j == 1) cout << "[ ]";
7          else cout << "[ | ]";
8      }
9      cout << '\n';
```

```
  } else {
    t.push_back(1);
    backtrack(i - 1);
    t.pop_back();

    t.push_back(2);
    backtrack(i - 2);
    t.pop_back();
  }
}
```

N7 [5★] In Exercise 3.4, we mentioned that your algorithm should run in $\mathcal{O}(n^{n+1})$. In fact, if you modify the algorithm in the right way, it should run in $\mathcal{O}(n \cdot n!)$. What's the right way to modify it? Can you prove that it runs in $\mathcal{O}(n \cdot n!)$?

N8 [9★] Exercise 3.16.

5.3 Coding problems

First timers, solve at least [65★]. I strongly recommend you prioritize problems in red over other problems. Some problems involve knowledge of graph theory; feel free not to answer these if you're unfamiliar with it.

Returning students, solve at least [80★]. Problems worth less than [4★] do not count towards this total. Problems in red that are worth more than [4★] are mandatory. You are required to solve one of the [12★] problems.

You are allowed to discuss the [2★] problems on the Discord.

C1 [2★] **The Next Number:** https://code.google.com/codejam/contest/dashboard?c=186264#s=p1

C2 [3★] Write a program that takes in an array and two integers l and r, then prints all of the array's subsets whose sum lies between l and r, inclusive, in lexicographical order. You can read the input however you want to.

C3 [5★] The first question I ever answered in Stack Overflow involved representing trees as 2D arrays: https://stackoverflow.com/a/53424019/1797728. In this problem, you'll do the same thing.

We can represent a tree rooted at node 0 with n vertices as follows. Consider an array `children` of n `vector<int>`s. Then `children[i]` stores the children of the ith node. For example,

```
vector<int> children[8];
children[0].push_back(1);
children[1].push_back(5);
children[0].push_back(4);
children[4].push_back(6);
children[0].push_back(2);
children[1].push_back(3);
```

represents one such tree. We can represent this tree using a 2D array, like so:

$$\begin{array}{cccc} 0 & — & — & — \\ 1 & — & 2 & 4 \\ 3 & 5 & — & 6 \end{array}$$

Observe how the children of each vertex are sorted from left to right. Write a program that, supposing that the tree is already represented in `children`, outputs such an array. Represent blank entries with `-`.

C4 [7★] Write a program that takes in an integer n, then prints all combinations of $2n$ balanced parentheses. There are 5 such combinations when $n = 3$.

C5 [9★] **Parallel Lines:** Problem B in http://icpc.iisf.or.jp/past-icpc/regional2017/problems.pdf

C6 [12★] In subsection 4.2, we discuss `next_permutation()`. In this problem, you'll write a similar function called `next_combination()`. Similar to that function, your function should run in $\mathcal{O}(n)$.

The function takes an integer n and a (sorted) subset of $\{0, 1, \ldots, n-1\}$ as input. It should modify the subset to produce the next lexicographically larger subset of the same size. For example, if $n = 4$, the next subset of $\{0, 2\}$ is $\{0, 3\}$, then $\{1, 2\}$. Take in input however you want.

C7 [12★] Write a program that takes in an integer n, then prints all distinct binary trees with height n. Two binary trees are the same if the left children of the two roots are the same, and if the right children of the two roots are the same. Make sure to describe how the output is formatted. There are 21 such distinct trees when $n = 3$.

S1 [2★] **LLPS:** https://codeforces.com/problemset/problem/202/A

S2 [2★] **Sum It Up:** UVa 574

S3 [2★] Subtask 1 of **Sorting the Planets:** https://www.hackerrank.com/contests/noi-ph-2019-finals-2-mirror/challenges/sorting-the-planets (Your program should work by enumerating all possible partitions.)

S4 [2★] Subtask 1 of **The Room:** https://www.hackerrank.com/contests/noi-ph-2019-finals-1-mirror/challenges/the-room (Your program should work by enumerating all possible paths.)

S5 [2★] Subtask 1 of **Nyeh Heh Heh!:** https://www.hackerrank.com/contests/noi-ph-2018-practice/challenges/nyeh-heh-heh

S6 [2★] **Permutations:** https://codeforces.com/problemset/problem/124/B

S7 [3★] **Basic Subset Enumeration:** https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-subset-enumeration (Make sure your code runs in $\mathcal{O}(n^{k+1})$ and not $\mathcal{O}(n2^n)$.)

S8 [3★] **Basic String Enumeration:** https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-string-enumeration

S9 [3★] **Kirill And The Game:** https://codeforces.com/problemset/problem/842/A

S10 [3★] **CD:** UVa 624

S11 [3★] **Lucky Numbers (easy):** https://codeforces.com/problemset/problem/96/B

S12 [3★] **Undoubtedly Lucky Numbers:** https://codeforces.com/problemset/problem/244/B

S13 [3★] **Crossword Puzzle:** https://www.hackerrank.com/contests/master/challenges/crossword-puzzle

S14 [5★] **Sagheer, the Hausmeister:** https://codeforces.com/problemset/problem/812/B

S15 [5★] **Special Multiple:** https://www.hackerrank.com/contests/master/challenges/special-multiple

S16 [5★] **Mike and strings:** https://codeforces.com/problemset/problem/798/B

S17 [5★] Subtask 1 of **Sumbong Centers:** https://www.hackerrank.com/contests/noi-ph-2019-finals-1-mirror/challenges/sumbong-centers

S18 [5★] **Unique Factorization:** UVa 10858

S19 [5★] **New Year and Buggy Bot:** https://codeforces.com/problemset/problem/908/B

S20 [5★] **Back to the 8-Queens:** UVa 11085

S21 [7★] **Points and Powers of Two:** https://codeforces.com/problemset/problem/988/D

S22 [7★] **Circus:** https://codeforces.com/problemset/problem/1138/B

S23 [7★] **Network Mask:** https://codeforces.com/problemset/problem/291/C

S24 [7★] **Password:** UVa 1262

S25 [9★] **Beautiful IP Addresses:** https://codeforces.com/problemset/problem/292/C

S26 [9★] **Safe:** https://codeforces.com/problemset/problem/47/D

S27 [9★] **Restore Cube:** https://codeforces.com/problemset/problem/464/B

S28 [12★] **Xor:** https://codeforces.com/problemset/problem/193/B

S29 [12★] **Chain Reaction:** https://codeforces.com/problemset/problem/666/D

S30 [12★] **Twenty Four, Again:** ICPC Live Archive 8075