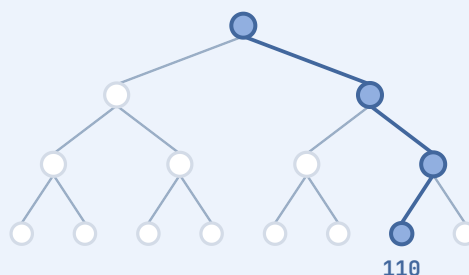


Recursion and backtracking

Chapter 2.4 · Overall level: Core



From the numbers 2, 5 and 6, *which ones can you pick so that they add up to exactly 7?* You could guess. Or you could decide about each number in turn, take it or skip it, and let a short recursive function try every combination for you, without missing one or trying one twice.

That is backtracking: build a solution one choice at a time, go back when a choice is finished, and try the next. It rests on recursion, a function that calls itself on a smaller problem. This chapter starts with how recursion runs, then uses it to list every word, subset and arrangement, and finally shows how to cut away branches that cannot succeed.

2.4.1 A function that calls itself

How many binary words of length n are there? A word of length n is a first bit, 0 or 1, followed by a word of length $n - 1$. So the count doubles with every extra position, and a word of length 0 (the empty word) is exactly one word:

$$\begin{aligned} \text{count}(0) &= 1 \\ \text{count}(n) &= 2 \times \text{count}(n - 1) \quad \text{for } n \geq 1 \end{aligned}$$

Every recursive function has the same two parts. The **base case** answers the smallest problem directly. The **recursive case** reduces the problem to a smaller one and calls the function again. Because n decreases by 1 on every call, it must reach 0, so the recursion always stops.

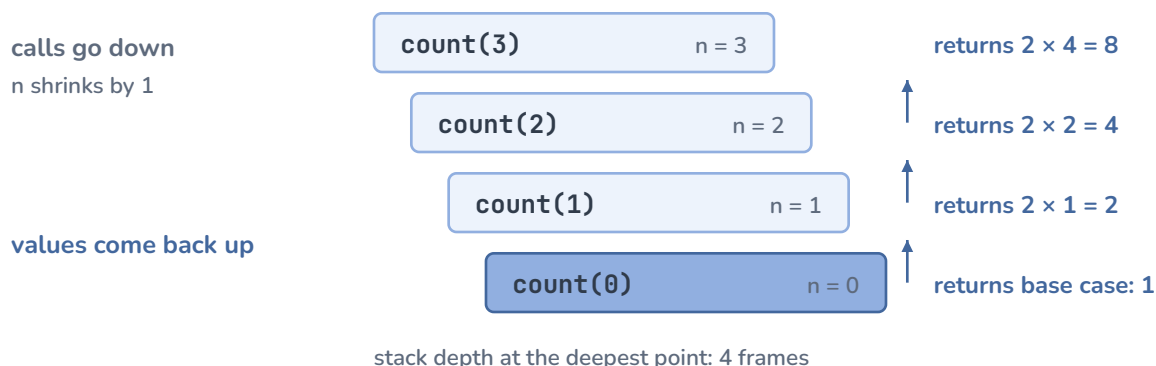
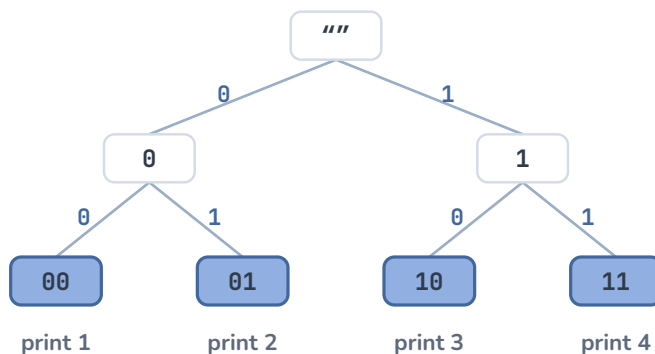


Figure 2.4.1 — Each call waits on the call stack until the smaller call returns. The values travel back up: 1, 2, 4, 8.

The computer keeps a **call stack**: one frame per unfinished call, holding that call's own n and where to continue. `count(3)` cannot finish until `count(2)` returns, and so on down to the base case. Then the frames finish in reverse order. The deepest stack here has 4 frames; in general the depth equals the number of nested calls, $n + 1$.

2.4.2 Returning a value, or acting at the leaves

A recursive function can be used in two different ways. `count` *returns a value* that its caller combines. The other way is to build something on the way down and *act at the bottom*: to print every binary word, pass along the prefix built so far, append 0 or 1, and print only when the prefix has full length.



Each node is a call; its label is the prefix built so far. The edge says which bit was appended.

Only leaves (length n) print.

0 is tried before 1, so the leaves come out in lexicographic order.

Figure 2.4.2 — Every call extends the prefix by one bit. A call whose prefix has length n is a leaf: it prints and returns without calling again.

Each complete word corresponds to exactly one path from the top to a leaf, because the path spells the word bit by bit. So every word is printed, and printed once. The order of the output is the order of the branches: trying 0 before 1 at every position gives 00, 01, 10, 11, which is lexicographic order. If a problem only asks how many words there are, the order does not matter; when it asks for a specific order, the branch order must produce it.

Pseudocode

Generating binary words: $O(n \cdot 2^n)$ including output

```

build(prefix):
    if length(prefix) = n:           ▶ base case: a leaf
        output prefix
        return
    for bit in 0, 1:                 ▶ 0 first gives lexicographic order
        append bit to prefix
        build(prefix)
        remove the last bit          ▶ undo, so the sibling starts clean
  
```

2.4.3 Choose or skip: the subset tree

Back to the opening question. A subset of $[2, 5, 6]$ is fixed by three decisions: is 2 in it, is 5 in it, is 6 in it? Write 1 for “take” and 0 for “skip”, and each subset becomes a binary word of length 3. So the same recursion that prints words can visit all $2^3 = 8$ subsets. Each call needs only two pieces of state: the index i of the next value to decide, and the running sum of the values taken so far.

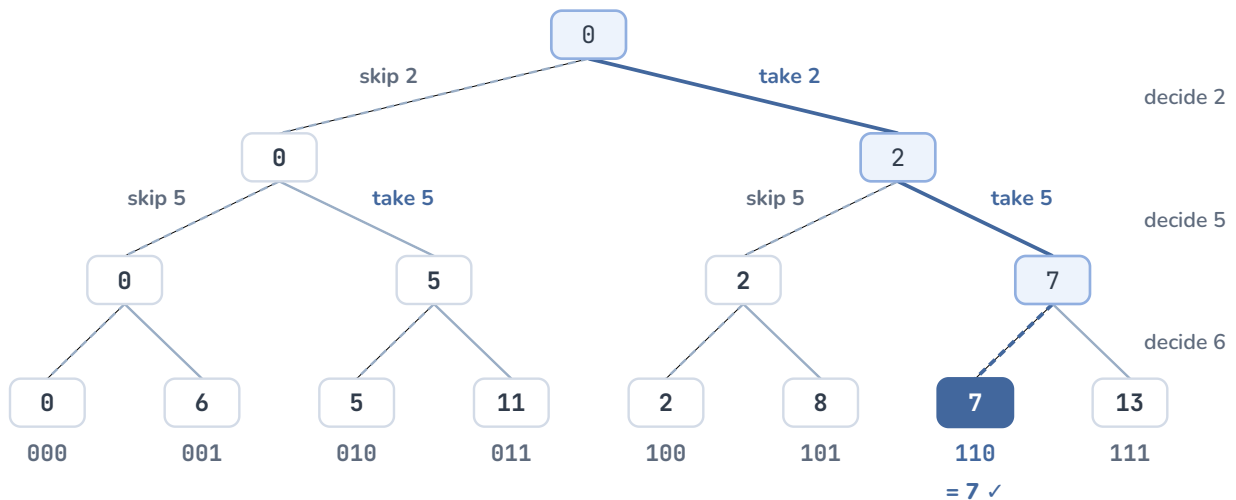


Figure 2.4.3 — Each bit of the decision path specifies whether one value is included. Numbers in the boxes are running sums; leaves show the sum of the whole subset. Dashed edges skip a value, solid edges take it.

Only the leaf for path 110, the subset $\{2, 5\}$, has sum 7. Every other leaf is still visited and checked; none is deleted. The leftmost leaf, path 000, is the **empty subset** with sum 0. It is a real subset: if the target were 0, it would count.

Pseudocode

Include or exclude: $O(2^n)$ leaves

```
search(i, sum):                                ▶ values 0..i-1 are decided
    if i = n:
        if sum = target: count ← count + 1
        return
    search(i + 1, sum)                          ▶ skip a[i]
    search(i + 1, sum + a[i])                  ▶ take a[i]
```

2.4.4 Undoing changes

In search the sum is passed as a parameter, so each call has its own copy and nothing needs undoing. Often the state is a shared object instead: a list holding the chosen values, or arrays of used columns. Then every change made before a recursive call must be **undone after it returns**, so that the next sibling branch starts from the same state as its parent saw. Here is the start of the search with a shared list `path` and a shared `sum`:

Step	Action	path after	sum after
1	take 2	[2]	2
2	take 5	[2, 5]	7
3	take 6, leaf 111: sum 13 $\neq$ 7	[2, 5, 6]	13
4	undo 6	[2, 5]	7
5	skip 6, leaf 110: sum 7, record {2, 5}	[2, 5]	7
6	undo 5, then skip 5	[2]	2
7	take 6, leaf 101: sum 8 $\neq$ 7	[2, 6]	8

This trace takes before it skips, so it starts at the right of Figure 2.4.3; either order visits the same eight leaves. Leave out step 4 and the bug is quiet but serious: the 6 would still be in `path` when the “skip 6” branch runs, and the leaf 110 would report the sum 13 instead of 7. A change that leaks into a sibling branch is the most common backtracking bug. The rule is simple: *every do has a matching undo, in reverse order*.

An alternative is to overwrite instead of undo. The Java program below keeps a fixed array `word` and writes position `len` before each call. The next choice at the same position overwrites it, and positions after `len` are never read before they are written again.

2.4.5 Code in four languages

Each program solves the first practice problem, **Binary words**. The input is one integer n ($1 \leq n \leq 12$). The program prints every binary string of length n , one per line, in lexicographic order, 0 before 1.

Sample input	Sample output
2	00 01 10 11

The recursion depth is only $n + 1$, so recursion is safe in every language here. The output has 2^n lines, at most 4,096, and the programs collect it and print it once.

Python 3

```
import sys

def main():
    n = int(sys.stdin.readline())
    word = []
    out = []

    def build():
        if len(word) == n:           # a leaf: the word is complete
            out.append("".join(word))
            return
        for bit in "01":           # 0 before 1 gives lexicographic order
            word.append(bit)
            build()
            word.pop()              # undo before trying the next bit

    build()
    print("\n".join(out))

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int n;
string word, out;

void build() {
    if ((int) word.size() == n) {    // a leaf: the word is complete
        out += word;
        out += '\n';
        return;
    }
    for (char bit : {'0', '1'}) {    // 0 before 1 gives lexicographic order
        word.push_back(bit);
        build();
        word.pop_back();             // undo before trying the next bit
    }
}

int main() {
    cin >> n;
    build();
    cout << out;
    return 0;
}
```

Java 17

```
import java.io.*;

public class Main {
    static int n;
    static char[] word;
    static StringBuilder out = new StringBuilder();

    // Fill positions len..n-1 of word; positions before len are fixed.
    static void build(int len) {
        if (len == n) {                // a leaf: the word is complete
            out.append(word).append('\n');
            return;
        }
        for (char bit = '0'; bit <= '1'; bit++) {
            word[len] = bit;           // overwritten by the next choice,
            build(len + 1);             // so no separate undo is needed
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        n = Integer.parseInt(in.readLine().trim());
        word = new char[n];
        build(0);
        System.out.print(out);
    }
}
```

JavaScript (Node)

```
const n = parseInt(require("fs").readFileSync(0, "utf8"));
const word = [];
const out = [];

function build() {
    if (word.length === n) {          // a leaf: the word is complete
        out.push(word.join(""));
        return;
    }
    for (const bit of ["0", "1"]) {   // 0 before 1 gives lexicographic order
        word.push(bit);
        build();
        word.pop();                  // undo before trying the next bit
    }
}

build();
console.log(out.join("\n"));
```

2.4.6 Pruning: balanced brackets

Pruning means abandoning a branch before it reaches the leaves, because no completion of the current prefix can succeed. The reason must hold for every completion. In the subset search it is tempting to stop as soon as $\text{sum} > \text{target}$. With positive values that is safe: adding more can only increase the sum. With negative values it is wrong: for $[1, 3, -2]$ and target 1, the partial sum 3 is already above 1, yet $\{3, -2\}$ works. Brackets give a pruning rule that is always correct.

A string of n opening and n closing brackets is balanced if no prefix has more closing than opening brackets. Listing all 2^{2n} strings and testing each wastes most of the work: for $n = 2$ only 2 of the 16 strings are balanced. Instead, track two counts while building, `open` and `close`, and only make choices that can still lead to a balanced string:

- add "(" only if `open < n`, since a string with more than n opening brackets cannot be completed;
- add ")" only if `close < open`, since otherwise the new prefix would have more closing than opening brackets.

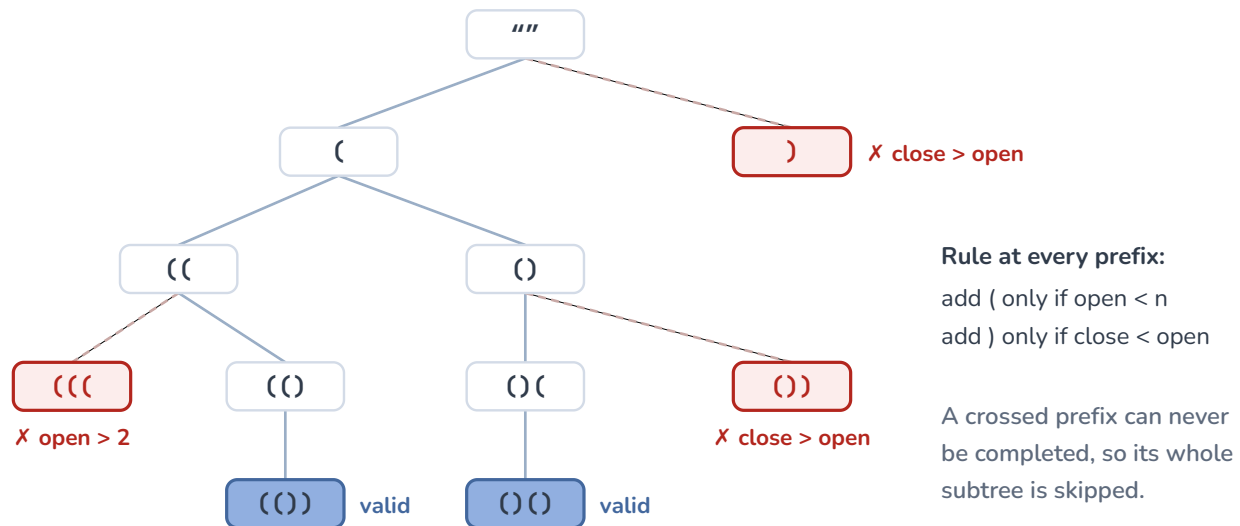


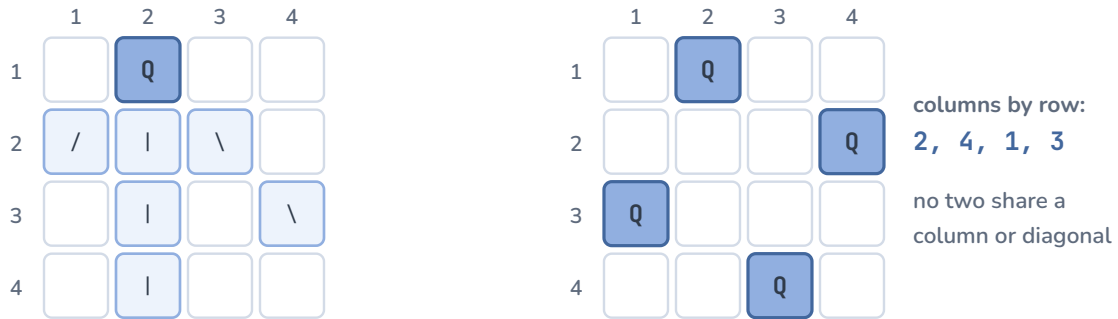
Figure 2.4.4 — Pruning removes a branch only after proving none of its completions can be valid. Red prefixes are cut with the reason shown.

Every legal string appears exactly once. A balanced string spells out one path from the root, so it can appear at most once. It does appear, because each of its prefixes satisfies both rules: it never has more than n opening brackets, and it never has more closing than opening brackets. And every leaf the search reaches is balanced: it has length $2n$ with `open = n`, so `close = n`, and the second rule held at every step. So the pruned search lists exactly the balanced strings. For $n = 3$ there are 5 of them.

2.4.7 Queens: tracking conflicts

Place n queens on an $n \times n$ board so that no two share a row, a column or a diagonal. Any legal placement has exactly one queen in each row, so decide row by row: the choice at row r is the column c of that row's queen. Row conflicts are then impossible by construction.

To test a square in $O(1)$, keep three sets of used lines. A square (r, c) lies on column c , on the down-right diagonal with constant $r - c$, and on the up-right diagonal with constant $r + c$. Two queens attack each other diagonally exactly when they share one of these numbers.



queen at (1, 2): column 2, $r - c = -1$, $r + c = 3$ are now used

Figure 2.4.5 — Left: after a queen is placed on (1, 2), marks |, \ and / show the squares ruled out by its column and its two diagonals. Right: one of the two complete placements on the empty 4×4 board; the other is 3, 1, 4, 2.

```
place(r):                                ▶ rows 0..r-1 already hold queens
    if r = n: count ← count + 1; return
    for c from 0 to n - 1:
        if square (r, c) is blocked or col[c] or diag1[r - c] or diag2[r + c]:
            continue
        col[c], diag1[r - c], diag2[r + c] ← true           ▶ do
        place(r + 1)
        col[c], diag1[r - c], diag2[r + c] ← false         ▶ undo
```

After trying one column, all three marks must be cleared, otherwise the next column in the same row would see conflicts that no longer exist. In code, $r - c$ can be negative; add $n - 1$ to use it as an array index. A blocked square is simply a choice that is never allowed.

2.4.8 How big is the search?

Backtracking is exhaustive, so its cost is the size of the tree. Estimate it before you write the code:

Search	Leaves or bound	n = 10	n = 20	n = 40
Binary words, subsets	2^n	1,024	about 10^6	about 10^{12}
Balanced brackets of $2n$ characters	grows like 4^n	16,796	about 6.6×10^9	far too many
Queens, one per row	at most $n!$	3,628,800	—	—

Around 10^6 to 10^7 leaves is comfortable; 10^{12} is hopeless however fast each step is. Pruning helps a lot in practice: the queens search on an empty 10×10 board visits only about 35,000 calls instead of $10!$ paths. But pruning does not change the worst case in general, so plan for the full tree.

When the answer is a list, the output is a lower bound on the work. Printing all binary words of length n writes $n \cdot 2^n$ characters, so no program can be faster than $O(n \cdot 2^n)$. Even counting without printing still visits every leaf: counting the subsets with a given sum inspects all 2^n choices unless more structure is used.

Recursion depth

Each pending call uses stack memory. Here the depth is at most $n + 1$, a few dozen frames. Later chapters walk graphs and trees with 100,000 nodes or more (Chapters 3.1 and 4.1). A recursion that deep can overflow the stack, and Python stops at about 1,000 nested calls by default, so those chapters use an explicit stack instead.

2.4.9 Common mistakes

- **Missing the empty subset.** The all-skip path is a real subset with sum 0. A search that only records sets when something is added misses it.
- **Mutated state leaking into sibling branches.** A list, a counter or a used-column mark that is changed before a call must be restored after it.
- **Pruning that assumes positive values.** “Stop when the sum exceeds the target” is wrong if later values can be negative.
- **A base case that is never reached.** The parameter must change towards the base case on every call, or the recursion never ends.
- **Choosing by value instead of by position.** With equal values at different positions, decide each position separately; they are different choices.

2.4.10 When exhaustive search is the wrong tool

Backtracking is the right first idea when n is small, around 20 for subsets or 10 for permutations, or when every solution must be listed anyway. When the same smaller question is asked again and again, for example “how many ways can the last k values make the sum s ?”, remembering those answers turns an exponential search into dynamic programming (Chapter 3.4). When the goal is one best choice that can be made locally, a proven greedy rule (Chapter 2.5) avoids the search altogether.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● What state must be undone after trying one queen position?
2. ● Why can counting all subsets still take exponential time, even when nothing is printed?
3. ● In what order does the program print the binary words of length 3? Which word is printed sixth?
4. ● How many children does the bracket prefix $(())$ have when $n = 3$? Which are they?
5. ●● List the subsets of $[1, 3, -2]$ with sum 1. Which one does the rule “stop when the sum exceeds the target” lose?
6. ●● A queen stands at row 2, column 3 of an empty 4×4 board. Which squares of row 3 are still allowed?

Answers and explanations

Read these after attempting the exercises.

1. The marks it set: its column and its two diagonals ($r - c$ and $r + c$), and the square itself if the path of chosen squares is stored.
2. There are 2^n different choices of which values to take, and without more structure each one has to be inspected to see whether it counts.
3. 000, 001, 010, 011, 100, 101, 110, 111. The sixth is **101**.
4. **Two:** `open = 2 < 3` allows `((()`, and `close = 1 < 2` allows `(())`.
5. {1} and {3, -2}. The partial sum 3 is already above the target 1, so the rule cuts the branch before -2 is added and loses {3, -2}.
6. Column 3 is used; diagonal $r - c = -1$ rules out (3, 4); diagonal $r + c = 5$ rules out (3, 2). Only **(3, 1)** remains.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Binary words

[Apply](#)

Print every binary string of length n in lexicographic order (0 before 1).

Target subsets

[Connect](#)

Count subsets of array positions whose values sum to x . The empty subset is allowed. Equal values at different positions are distinct choices.

Balanced bracket count

[Reason](#)

Count strings containing n opening and n closing parentheses such that every prefix has at least as many openings as closings.

Blocked queens

[Stretch](#)

Place n queens on an n by n board with blocked squares. No two queens may share a row, column or diagonal. Count all placements using only free squares.