●●●● Rare   0/10

Complete Search with Recursion

Author: Many
Contributors: Darren Yao, Sam Zhang, Michael Cao, Andrew Wang, Benjamin Qi, Dong Liu, Maggie Liu, Dustin Miao

Harder problems involving iterating through the entire solution space, including those that require generating subsets and permutations.

Language: Python   Edit This Page

Prerequisites

- Bronze - Basic Complete Search

TABLE OF CONTENTS

Warning!

Although knowledge of recursion is not strictly necessary for Bronze, we think that it makes more sense to include this module as part of Bronze rather than Silver.

Subsets

Apple Division
CSES - Easy

Focus Problem – try your best to solve this problem before continuing!

Resources

RESOURCES

CPH	★ **5.1 - Generating Subsets**	good explanation + code, no need to repeat

Solution - Apple Division

Since $n \leq 20$, we can solve this by trying all possible divisions of n apples into two sets and finding the one with the minimum difference in weights. Here are two ways to do this.

Generating Subsets Recursively

The first method would be to write a recursive function which searches over all possibilities.

At some index, we either add $\texttt{weight}_i$ to the first set or the second set, storing two sums $\texttt{sum}_1$ and $\texttt{sum}_2$ with the sum of values in each set.

Then, we return the difference between the two sums once we've reached the end of the array.

Copy PYTHON

```python
n = int(input())
weights = list(map(int, input().split()))


def recurse_apples(i: int, sum1: int, sum2: int) -> int:
    # We've added all apples- return the absolute difference
    if i == n:
        return abs(sum2 - sum1)

    # Try adding the current apple to either the first or second set
    return min(
        recurse_apples(i + 1, sum1 + weights[i], sum2),
        recurse_apples(i + 1, sum1, sum2 + weights[i]),
    )


# Solve the problem starting at apple 0 with both sets being empty
print(recurse_apples(0, 0, 0))
```

Generating Subsets with Bitmasks

⚠ Warning!

You are not expected to know this for Bronze.

A **bitmask** is an integer whose binary representation is used to represent a subset. In the context of this problem, if the i'th bit is equal to 1 in a particular bitmask, we say the i'th apple is in s_1. If not, we'll say it's in s_2. We can iterate through all subsets s_1 if we check all bitmasks ranging from 0 to $2^N - 1$.

Let's do a quick demo with $N = 3$. These are the integers from 0 to $2^3 - 1$ along with their binary representations and the corresponding elements included in s_1. As you can see, all possible subsets are accounted for.

Number	Binary	Apples In s_1
0	`000`	$\{\}$
1	`001`	$\{0\}$
2	`010`	$\{1\}$
3	`011`	$\{0, 1\}$
4	`100`	$\{2\}$

Number	Binary	Apples In s_1
5	`101`	$\{0, 2\}$
6	`110`	$\{1, 2\}$
7	`111`	$\{0, 1, 2\}$

With this concept, we can implement our solution.

You'll notice that our code contains some fancy bitwise operations:

- `1 << x` for an integer x is another way of writing 2^x, which, in binary, has only the x'th bit turned on.
- The `&` (AND) operator will take two integers and return a new integer. `a & b` for integers a and b will return a new integer whose ith bit is turned on if and only if the i'th bit is turned on for both a and b. Thus, `mask & (1 << x)` will return a positive value only if the x'th bit is turned on in $mask$.

If you wanna learn more about them, we have a **dedicated module** for bitwise operations.

Copy PYTHON

```python
n = int(input())
weights = list(map(int, input().split()))

ans = float("inf")
for mask in range(1 << n):
    sum1 = 0
    sum2 = 0
    for i in range(n):
        # Checks if the ith bit is set
        if mask & (1 << i):
            # If it is, the apple is included in sum1
            sum1 += weights[i]
        else:
            sum2 += weights[i]

    ans = min(ans, abs(sum1 - sum2))

print(ans)
```

Permutations

A **permutation** is a reordering of a list of elements.

Creating Strings I

CSES - Easy

Focus Problem – try your best to solve this problem before continuing!

Lexicographical Order

This term is mentioned quite frequently, ex. in **USACO Bronze - Photoshoot**.

Think about how are words ordered in a dictionary. (In fact, this is where the term "lexicographical" comes from.)

In dictionaries, you will see that words beginning with the letter `a` appears at the very beginning, followed by words beginning with `b`, and so on. If two words have the same starting letter, the second letter is used to compare them; if both the first and second letters are the same, then use the third letter to compare them, and so on until we either reach a letter that is different, or we reach the end of some word (in this case, the shorter word goes first).

Permutations can be placed into lexicographical order in almost the same way. We first group permutations by their first element; if the first element of two permutations are equal, then we compare them by the second element; if the second element is also equal, then we compare by the third element, and so on.

For example, the permutations of 3 elements, in lexicographical order, are

$$[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1].$$

Notice that the list starts with permutations beginning with 1 (just like a dictionary that starts with words beginning with `a`), followed by those beginning with 2 and those beginning with 3. Within the same starting element, the second element is used to make comparisons.

Generally, unless you are specifically asked to find the lexicographically smallest/largest solution, you do not need to worry about whether permutations are being generated in lexicographical order. However, the idea of lexicographical order does appear quite often in programming contest problems, and in a variety of contexts, so it is strongly recommended that you familiarize yourself with its definition.

Some problems will ask for an ordering of elements that satisfies certain conditions. In these problems, if $N \leq 10$, we can just iterate through all $N! = N \cdot (N-1) \cdot (N-2) \cdots 1$ permutations and check each permutation for validity.

Solution - Creating Strings I

RESOURCES

CPH ★ **5.2 - Generating Permutations** brief explanation + code for both of the methods below

Generating Permutations Recursively

This is just a slight modification of method 1 from CPH.

We'll use the recursive function **search** to find all the permutations of the string s. First, keep track of how many of each character there are in s. For each function call, add an available character to the current string, and call **search** with that string. When the current string has the same size as s, we've found a permutation and can add it to the list of **perms**.

Copy PYTHON

```python
s = input()
perms = []
char_count = [0] * 26


def search(curr: str = ""):
    # we've finished creating a permutation
    if len(curr) == len(s):
        perms.append(curr)
        return
    for i in range(26):
        # For all available characters
        if char_count[i] > 0:
            # Add it to the current string and continue the search
            char_count[i] -= 1
            search(curr + chr(ord("a") + i))
            char_count[i] += 1


for c in s:
    char_count[ord(c) - ord("a")] += 1

search()

print(len(perms))
for perm in perms:
    print(perm)
```

Generating Permutations Using `itertools.permutations`

Since `itertools.permutations` treats elements as unique based on position, not value, it returns all permutations, with repeats. Putting the returned tuples in a set can filter out duplicates, and since tuples are returned, we need to join the characters into a string.

Copy PYTHON

```python
from itertools import permutations

s = input()

# perms is a sorted list of all the permutations of the given string
perms = sorted(set(permutations(s)))

print(len(perms))
for perm in perms:
    print("".join(perm))
```

Backtracking

Chessboard & Queens

CSES - Normal

Focus Problem – try your best to solve this problem before continuing!

Resources

RESOURCES

CPH	★	**5.3 - Backtracking**	code and explanation for focus problem
CP2		**3.2 - Complete Search**	iterative vs recursive complete search

Solution - Chessboard & Queens

By Generating Permutations

A brute-force solution that checks all $\binom{64}{8}$ possible queen combinations will have over 4 billion arrangements to check, making it too slow.

We have to brute-force a bit smarter: notice that we can directly generate permutations so that no two queens are attacking each other due to being in the same row or column.

Since no two queens can be in the same column, it makes sense to lay one out in each row. It remains to figure out how to vary the *rows* each queen is in. This can be done by generating all permutations from $1 \cdots 8$, with the numbers representing which row each queen is in.

For example, the permutation $[6, 0, 5, 1, 4, 3, 7, 2]$ results in this queen arrangement:

	0	1	2	3	4	5	6	7
0		Q						
1				Q				
2								Q
3						Q		
4					Q			
5			Q					

6	Q							
7							Q	

Doing this cuts down the number of arrangements we have to check down to a much more manageable $8!$.

Easier Diagonal Checking

To make the implementation easier, notice that some bottom-left to top-right diagonal can be represented as all squares i, j (i being the row and j being the column) such that $i + j = S$ for some S. For example, all squares on the diagonal from $6, 0$ to $0, 6$ have their coordinates sum to 6.

Similarly, some bottom-right to top-left diagonal can be represented as the same thing, but with $i - j$ instead of $i + j$.

```python
from itertools import permutations

DIM = 8

blocked = [[False] * DIM for _ in range(DIM)]
for r in range(DIM):
    row = input()
    for c in range(DIM):
        blocked[r][c] = row[c] == "*"

valid_num = 0
for queens in permutations(range(DIM)):
    works = True

    # Check if any cells have been blocked off by the input
    for c in range(DIM):
        if blocked[queens[c]][c]:
            works = False
            break

    # Check the diagonals from the top-left to the bottom-right
    taken = [False] * (DIM * 2 - 1)
    for c in range(DIM):
        # Check if the diagonal with sum has been taken
        if taken[c + queens[c]]:
            works = False
            break
        taken[c + queens[c]] = True

    # Check the diagonals from the top-right to the bottom-left
    taken = [False] * (DIM * 2 - 1)
    for c in range(DIM):
        # queens[c] - c can be negative; the DIM - 1 offsets that
        if taken[queens[c] - c + DIM - 1]:
            works = False
            break
        taken[queens[c] - c + DIM - 1] = True

    if works:
        valid_num += 1

print(valid_num)
```

Using Backtracking

According to CPH:

> 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.

Since the bounds are small, we can recursively backtrack over all ways to place the queens, storing the current state of the board.

At each level, we try to place a queen at all squares that aren't blocked or attacked by other queens. After this, we recurse, then remove this queen and backtrack.

Finally, we increment the answer when we've placed all eight queens.

Copy PYTHON

```python
DIM = 8

blocked = [[False for _ in range(DIM)] for _ in range(DIM)]
for r in range(DIM):
    row = input()
    for c in range(DIM):
        blocked[r][c] = row[c] == "*"

rows_taken = [False] * DIM
# Indicators for diagonals that go from the bottom left to the top right
diag1 = [False] * (DIM * 2 - 1)
# Indicators for diagonals that go from the bottom right to the top left
diag2 = [False] * (DIM * 2 - 1)

valid_num = 0


def search_queens(c: int = 0):
    global valid_num

    if c == DIM:
        # We've filled all rows, increment and return
        valid_num += 1
        return

    for r in range(DIM):
        row_open = not rows_taken[r]
        diag_open = not diag1[r + c] and not diag2[r - c + DIM - 1]
        if not blocked[r][c] and row_open and diag_open:
            # A row and two diagonals have been taken
            rows_taken[r] = diag1[r + c] = diag2[r - c + DIM - 1] = True
            search_queens(c + 1)
            # And now they aren't anymore
            rows_taken[r] = diag1[r + c] = diag2[r - c + DIM - 1] = False


search_queens()
print(valid_num)
```

Problems