Competitive Coding @ Berkeley

Decal Lecture #7 - Complete Search With Recursion

Announcements

Lecture outline for the rest of the semester:

- **March 13** - Complete Search with Recursion
- **March 20** - Dynamic Programming, Part 1
- **April 3** - Dynamic Programming, Part 2
- **April 10** - Segment Trees
- **April 17** - Rolling Hashes
- **April 24** - Wrap-up, Final Contest

Grading Stuff

- Plz make sure we have your codeforces handle in here!
- https://docs.google.com/spreadsheets/d/1Z-6Vjo-y3ZDD_YDUjWZ0WTy8AkcS2JAEYv83iGoFB_o/edit?usp=sharing
- If you're missing something, you can fill it out here
  - 

Last Week's Problemset!

- Seemed to be extremely difficult
  - We kinda expected this which is why it wasn't based on correctness
- Reminder that you can see solutions on the leaderboard

Motivating Problem

- https://cses.fi/problemset/task/1623

There are n apples with known weights. Your task is to divide the apples into two groups so that the difference between the weights of the groups is minimal.

Input

The first input line has an integer n: the number of apples.

The next line has n integers $p_1, p_2, \ldots, p_n$: the weight of each apple.

Output

Print one integer: the minimum difference between the weights of the groups.

Constraints

- $1 \leq n \leq 20$
- $1 \leq p_i \leq 10^9$

Example

Input:
```
5
3 2 7 4 1
```

Output:
```
1
```

Explanation: Group 1 has weights 2, 3 and 4 (total weight 9), and group 2 has weights 1 and 7 (total weight 8).

What is complete search?

- Not every problem will be solvable in polynomial time
- Sometimes we need to **brute force** a set of all possible options to calculate our answer
    - Checking all permutations or subsets

What is complete search?

- Not every problem will be solvable in polynomial time
- Sometimes we need to **brute force** a set of all possible options to calculate our answer
  - Checking all permutations or subsets
- Example: https://cses.fi/problemset/task/1623 (How would we do this?)
  - Observe that ***n***, the number of items, is very small (***n <= 20***)
  - We need the skill to be able to simulate all possibilities and then take the best answer

Complete Search over Subsets

- For a set ***a*** with ***n*** elements, there are 2^n subsets
    - We can decide to either include or exclude each element

Complete Search over Subsets

- For a set ***a*** with ***n*** elements, there are 2^n subsets
  - We can decide to either include or exclude each element
- A binary string ***s*** of length ***n*** can represent a subset
  - s_i = '1' if we should take the i^{th} element
  - s_i = '0' if we don't take it

Complete Search over Subsets

- For a set ***a*** with ***n*** elements, there are 2^n subsets
  - We can decide to either include or exclude each element
- A binary string ***s*** of length ***n*** can represent a subset
  - s_i = '1' if we should take the i^{th} element
  - s_i = '0' if we don't take it
- Example:
  - If ***a*** = {'A' , 'B', 'C' , 'D'} and ***s*** = "1001"...
  - Our subset will be {'A' , 'D'}

Complete Search over Subsets

- There are also 2^n different binary strings of length ***n***!
- We can form a ***bijection*** between the set of all length ***n*** binary strings and all subsets we want to make
    - All subsets can be represented by **exactly one** binary string

Complete Search over Subsets

- There are also 2^n different binary strings of length ***n***!
- We can form a ***bijection*** between the set of all length ***n*** binary strings and all subsets we want to make
  - All subsets can be represented by **exactly one** binary string
- How to make all the binary strings…

Complete Search over Subsets

- There are also 2^n different binary strings of length ***n***!
- We can form a ***bijection*** between the set of all length ***n*** binary strings and all subsets we want to make
  - All subsets can be represented by **exactly one** binary string
- How to make all the binary strings…
  - No need to! We can use the integers from 0 to $2^n - 1$
  - Use bitwise operations to see if the i^{th} bit is 0 or 1

Checking If A Specific Bit Is Set to 1

- To see if the i^{th} bit is set for an integer x
  - Check if the bitwise **AND** of x with 2^i is non-zero

Checking If A Specific Bit Is Set to 1

- To see if the i^{th} bit is set for an integer ***x***
  - Check if the bitwise **AND** of ***x*** with 2^i is non-zero
- Why?
  - 2^i will only have a bit set at index ***i***, everywhere else will be 0!
  - This forces the bitwise **AND** to only depend on the the value of the i^{th} bit in ***x***

Checking If A Specific Bit Is Set to 1

- To see if the i^{th} bit is set for an integer ***x***
  - Check if the bitwise **AND** of ***x*** with 2^i is non-zero
- Why?
  - 2^i will only have a bit set at index ***i***, everywhere else will be 0!
  - This forces the bitwise **AND** to only depend on the the value of the i^{th} bit in ***x***

Checking If A Specific Bit Is Set to 1

- To see if the i^{th} bit is set for an integer ***x***
  - Check if the bitwise **AND** of ***x*** with 2^i is non-zero
- Why?
  - 2^i will only have a bit set at index ***i***, everywhere else will be 0!
  - This forces the bitwise **AND** to only depend on the the value of the i^{th} bit in ***x***
- Useful tip!
  - 2^n can be easily calculated as the integer 1 being bitshifted ***n*** times
  - 2^n == (1<<n)

Implementation/Apples Division Solution

```
1  n = int(input())
2  weights = list(map(int, input().split()))
3  total = sum(weights)
4  ans = 1e18
5  for subset in range(1<<n):
6      left = 0
7      for i in range(n):
8          if subset & (1<<i):
9              left += weights[i]
10     right = total - left
11     ans = min(ans, abs(right - left))
12
13 print(ans)
```

Generating Subsets With Recursion From CSES Book

- Maintain a vector/list of current subset
- Make a function that makes **two recursive calls**
  - One call chooses to add the current element
  - The other chooses not to add it

Generating Subsets With Recursion From CSES Book

- Maintain a vector/list of current subset
- Make a function that makes **two recursive calls**
  - One call chooses to add the current element
  - The other chooses not to add it
- After both recursive calls are complete, you will have finished processing all subsets that contain the current element
  - So you must remove it so that it isn't included in any other subset

Generating Subsets With Recursion From CSES Book

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 $\{0,1,\ldots,n-1\}$. 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();
    }
}
```

Apples Division With Recursive Solution

“ll” is a 64 bit integer, you can just think of it as int

```
int main(){
    ll n;
    cin>>n;
    vector<ll> v(n);
    ll total = 0;
    for(ll &x: v){
        cin>>x;
        total += x;
    }
    vector<ll> subset;
    ll ans = INF;
    function<void(ll)> search = [&](ll index){
        if(index == n){
            ll sumA = 0;
            for(ll x: subset)
                sumA += x;
            ll sumB = total - sumA;
            ans = min(ans, abs(sumA - sumB));
            return;
        }
        search(index+1);
        subset.push_back(v[index]);
        search(index+1);
        subset.pop_back();
    };
    search(0);
    cout<<ans<<"\n";
}
```

Complete Search over Permutations

If we have a list, the permutations of the list will be all of the possible ways to shuffle the elements of the list. For example, the permutations of [1, 3, 5] are the following:

[1, 3, 5]	[3, 5, 1]
[1, 5, 3]	[5, 1, 3]
[3, 1, 5]	[5, 3, 1]

There are N! (N factorial) permutations for a list of length N

Complete Search over Permutations

In many problems, you're asked to find the best answer over all permutations of a list. Often, the constraints are high enough that iterating over and checking every permutation is too slow, and you'll have to something smarter.

However, sometimes the constraints are low enough that simple brute force works. This technique can also be useful for stress testing incorrect solutions.

Complete Search over Permutations

To find permutations, we can iterate over all possible elements we can add to the first position in the array, and recursively find the possible permutations of the remaining elements.

There are also reliable built-in functions for finding permutations: **itertools.permutations()** in Python, and **std::next_permutation** in C++.

Complete Search over Permutations

Since there are N! total permutations of length N, finding all permutations will run in O(N!) time.

This is very slow – even if N = 12, this solution will time out.

Recursive Complete Search with Backtracking

Example Problem:

Write a program to solve a Sudoku puzzle

Recursive Complete Search with Backtracking

For some problems, like the Sudoku solver problem, we need to find a way to place numbers/letters/etc in a grid.

We can do this by iterating over all possible numbers to put in the first unfilled space, and calling the function recursively.

Since we'll go back to an earlier state in the grid when we get stuck, this is called **backtracking**

N-Queens Problem

Time limit: 1.00 s **Memory limit:** 512 MB

Your task is to place eight queens on a chessboard so that no two queens are attacking each other. As an additional challenge, each square is either free or reserved, and you can only place queens on the free squares. However, the reserved squares do not prevent queens from attacking each other.

How many possible ways are there to place the queens?

Input

The input has eight lines, and each of them has eight characters. Each square is either free (`.`) or reserved (`*`).

Output

Print one integer: the number of ways you can place the queens.

Example

Input:
```
........
........
..*.....
........
........
.....**.
...*....
........
```

Output:
```
65
```

N-Queens Problem

- Each queen must be in a separate row and column
- Considering the rows as indices, we can think of the columns of each queen as a permutation
- We can either use backtracking, or iterate over these permutations of columns

Complete Towns (Practice Problem)

There are n towns in the land of Completeness. Because the land of Completeness is so perfect, the towns are also located perfectly such that each town i's position can be represented by an integer coordinate (x_i, y_i).

Since the land of Completeness is complete, it is also possible to go from any town to any other town in a straight line. So the distance traveled between town i and j will be $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

A perfect path in the land of Completeness is one that visits every town, but only once. What is the average length of all of the possible perfect paths you can take?

Input

The first line will contain $n(2 \leq n \leq 8)$, the number of towns in the land of completeness.

Then n lines will follow, each containing the coordinate of the i-th town that is at (x_i, y_i).

Output

Output in a single line the average length of all of the possible paths. Your output will be judged as correct when the absolute possible difference from the judge's output is at most 10^{-6}.

Complete Towns Solution

- Since N is very small, we can iterate over all possible permutations (i.e. visiting orders) of towns.
- For each visiting order, we can calculate the total distance traveled, and average this over all iterations.

Pruning

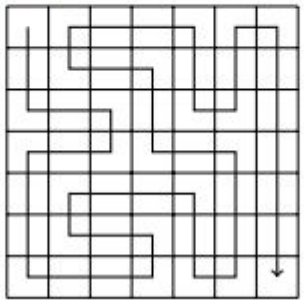
- **Pruning** is a complete search optimization that removes unnecessary and redundant cases
  - Greatly reduces the size of the decision tree formed by complete search

Pruning

- **Pruning** is a complete search optimization that removes unnecessary and redundant cases
  - Greatly reduces the size of the decision tree formed by complete search
- “Stopping bad cases early”
- Useful when complete search is **very close** to being fast enough

Example From CSES Book (Page 51)

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:



No Pruning

Basic algorithm

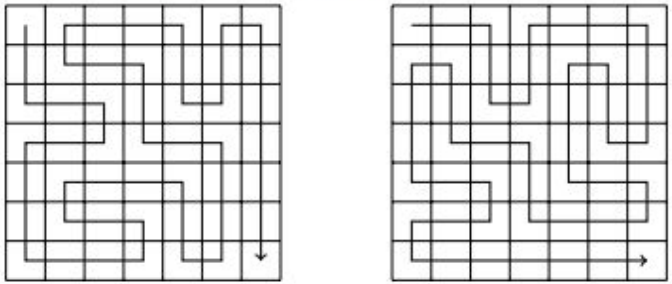
The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

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

Example From CSES Book

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:



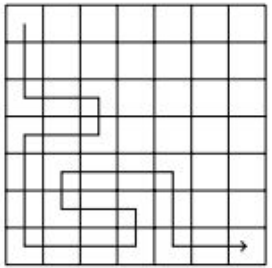
Hence, we can decide that we always first move one step down (or right), and finally multiply the number of solutions by two.

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

Example From CSES Book

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:



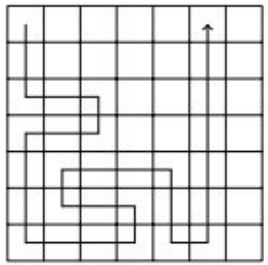
Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

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

Example From CSES Book

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:



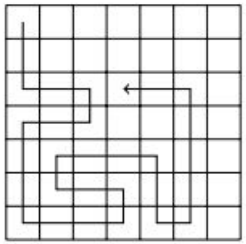
In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

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

Example From CSES Book

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:



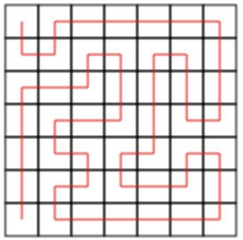
It is clear that we cannot visit all squares anymore, so we can terminate the search. After this optimization, the search is very efficient:

- running time: 0.6 seconds
- number of recursive calls: 69 million

CSES Grid Paths (Difficult Example)

There are 88418 paths in a 7×7 grid from the upper-left square to the lower-left square. Each path corresponds to a 48-character description consisting of characters `D` (down), `U` (up), `L` (left) and `R` (right).

For example, the path



corresponds to the description
`DRURRRRRDDDLUULDDDLDRRURDDLLLLLURULURRUULDLLDDDD`.

You are given a description of a path which may also contain characters `?` (any direction). Your task is to calculate the number of paths that match the description.