Ad hoc and complete search

Atli FF

September 2, 2024

School of Computer Science
Reykjavík University

Ad hoc

Ad hoc

- Not much to say about ad hoc problems
- Ad hoc means you should do what the statement says
- No tricks, just work
- Does *not* mean they are easy
- Read statements carefully

Complete search

Complete search

- We have a finite set of objects
- We want to find an element in that set which satisfies some constraints
    - or find **all** elements in that set which satisfy some constraints
- Simple! Just go through all elements in the set, and for each of them check if they satisify the constraints
- Of course it's not going to be very efficient...
- But remember, we always want the simplest solution that runs in time
- Complete search should be the first problem solving paradigm you think about when you're trying to solve a problem

Example problem: Closest Sums

- https://open.kattis.com/problems/closestsums

Complete search

- What if the search space is more complex?
    - All permutations of n items
    - All subsets of n items
    - All ways to put n queens on an $n \times n$ chessboard without any queen attacking any other queen
- How are we supposed to iterate through the search space?
- Let's take a better look at these examples

Iterating through permutations

- Already implemented in many standard libraries:
  - `next_permutation` in C++
  - `itertools.permutations` in Python

```
int n = 5;
vector<int> perm(n);
for (int i = 0; i < n; i++) perm[i] = i + 1;

do {
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");

} while (next_permutation(perm.begin(), perm.end()));
```

Iterating through permutations

- Even simpler in Python

- Remember that there are $n!$ permutations of length n, so usually you can only go through all permutations if $n \leq 10$
  - Otherwise you need to find a more clever approach than complete search

Example problem: Veci

- https://open.kattis.com/problems/veci

Iterating through subsets

- Remember the bit representation of subsets?
- Each integer from 0 to $2^n - 1$ represents a different subset of the set $\{1, 2, \ldots, n\}$
- Just iterate through the integers

```
int n = 5;
for (int subset = 0; subset < (1 << n); subset++) {
    for (int i = 0; i < n; i++) {
        if ((subset & (1 << i)) != 0) {
            printf("%d ", i+1);
        }
    }
    printf("\n");
}
```

Iterating through subsets

- Similar in Python
- Remember that there are 2^n subsets of n elements, so usually you can only go through all subsets if $n \leq 25$
  - Otherwise you need to find a more clever approach than complete search

Example problem: Exam Manipulation

- https://open.kattis.com/problems/exammanipulation

Backtracking

- We've seen two ways to go through a complex search space, but both of the solutions were rather specific
- Would be nice to have a more general "framework"
- Backtracking!

Backtracking

Backtracking

- Define states
  - We have one initial "empty" state
  - Some states are partial
  - Some states are complete
- Define transitions from a state to possible next states
- Basic idea:
  1. Start with the empty state
  2. Use recursion to traverse all states by going through the transitions
  3. If the current state is invalid, then stop exploring this branch
  4. Process all complete states (these are the states we're looking for)

Backtracking

- General solution form:

```
state S;

void generate() {
    if (!is_valid(S))
        return;

    if (is_complete(S))
        print(S);

    foreach (possible next move P) {
        apply move P;
        generate();
        undo move P;
    }
}

S = empty state;
generate();
```

Generating all subsets

- Also simple to do with backtracking:

```
const int n = 5;
bool pick[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            if (pick[i]) {
                printf("%d ", i+1);
            }
        }
        printf("\n");
    } else {

        // either pick element no. at
        pick[at] = true;
        generate(at + 1);

        // or don't pick element no. at
        pick[at] = false;
        generate(at + 1);
    }
}

generate(0);
```

Generating all permutations

- Also simple to do with backtracking:

```
const int n = 5;
int perm[n];
bool used[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            printf("%d ", perm[i]+1);
        }
        printf("\n");
    } else {

        // decide what the at-th element should be
        for (int i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;
                perm[at] = i;

                generate(at + 1);

                // remember to undo the move:
                used[i] = false;
            }
        }
    }
}

memset(used, 0, n);
```

n queens

- Given n queens and an $n \times n$ chessboard, find all ways to put the n queens on the chessboard such that no queen can attack any other queen

- This is a very specific set we want to iterate through, so we probably won't find this in the standard library
- We could use our bit trick to iterate through all subsets of the $n \times n$ cells of size n, but that would be very slow

- Let's use backtracking

n queens

- Go through the cells in increasing order
- Either put a queen on that cell or not (transition)
- Don't put down a queen if she's able to attack another queen already on the table

```
const int n = 8;
bool has_queen[n][n];
int queens_left = n;

// generate function

memset(has_queen, 0, sizeof(has_queen));
generate(0, 0);
```

n queens

```
void generate(int x, int y) {
    if (y == n) {
        generate(x+1, 0);
    } else if (x == n) {
        if (queens_left == 0) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    printf("%c", has_queen[i][j] ? 'Q' : '.');
                }
                printf("\n");
            }
        }
    } else {

        if (queens_left > 0 and no queen can attack cell (x,y)) {
            // try putting a queen on this cell
            has_queen[x][y] = true;
            queens_left--;

            generate(x, y+1);

            // undo the move
            has_queen[x][y] = false;
            queens_left++;
        }

        // try leaving this cell empty
        generate(x, y+1);
    }
}
```

Example problem: Lucky Numbers

- https://open.kattis.com/problems/luckynumber
- As a note, another classic use case is solving sudokus

Meet in the middle

Halving the exponent

- We now know at least two different ways we could solve the subset sum problem, i.e. given some numbers check if there is a subset summing to some given target value
- They would have time complexity $\mathcal{O}(2^n)$ or even $\mathcal{O}(n2^n)$, which is quite a lot.
- But there is a trick we can employ to get $\mathcal{O}(n2^{n/2})$ instead.

Halves

- Split the numbers into two groups A, B, then iterate over every subset of A and put it into a set. This takes $\mathcal{O}(n2^{n/2})$ time.
- Then iterate over every subset in B, and each time check if the target value minus the sum of B is in our set of sums. This also takes $\mathcal{O}(n2^{n/2})$ time.
- This way we do the problem one half at a time, and meet in the middle.

Middle

- This can be applied more generally, searching from both ends of a problem.
- It is a fairly common tactic in cryptography as well.
- Sometimes it can also be combined with backtracking, backtracking from two ends at once.

Not quite half

- Sometimes the gain isn't quite a square root of the base of the exponential run-time, but gains can still be large.
- Take the problem of having a set of numbers $\{x_1, \ldots, x_n\}$ and wanting to find two different non-empty subsets $\{x_{i_1}, \ldots, x_{i_r}\}$ and $\{x_{j_1}, \ldots, x_{j_s}\}$ with the same sum.
- The straight forward way would be $\mathcal{O}(4^n)$, but iterating over all subsets and counting how many get to any given sum is $\mathcal{O}(n2^n)$.

Not quite half

- But we can split $x_1, \ldots, x_n$ in two halves, then rewrite the condition

$$x_{i_1} + \cdots + x_{i_r} = x_{j_1} + \cdots + x_{j_s}$$

as

$$x_{i_1} + \cdots + x_{i'_r} - x_{j_1} - \cdots - x_{j'_s} = x_{j'_s+1} + \cdots + x_{j_s} - x_{i'_r+1} - \cdots - x_{i_r}$$

so that we move every term in the first half to the left hand side and every term in the second half to the right side.

- Then we can meet in the middle on all ways to ignore, add or subtract the element from the sum to get a $\mathcal{O}(n3^{n/2})$.