Problem Solving Paradigms

David Jacobo
jguillen@cimat.mx

May 8, 2015

Paradigm:

Comes from Greek (paradeigma), "pattern, example, sample"

"If all you have is a hammer, everything looks like a nail"

Abraham Maslow, 1962.

Outline

Complete search

Also called backtracking or brute force, tries all the possible combinations within the search space. A bug-free implementation ensures you will always get the correct answer!

Pros

- Easy to code.
- Always gives the correct answer.
- No complex data structures needed.
- Should be your first option.

Cons

- Pretty bad complexity.
- Cannot handle large inputs.
- People tend to think your ki is low.

Coin change

Having a known subset of coins (i.e. 1,3,4,10) and an infinite amount of each of those, determine what is the minimum amount of coins used to form N.

```
int coins[] = {1,3,4,10};

int get_coins_complete_search(int N){
    int min = MAX_N;

    for(int w=0;w < MAX_N;++w){
        for(int x=0;x < MAX_N;++x){
            for(int y=0;y < MAX_N;++y){
                for(int z=0;z < MAX_N;++z){
                    if(w*coins[0] + x*coins[1] + y*coins[2] + z*coins[3] == N \
                      && w + x + y + z < min){
                      min = w + x + y + z;
                    }
                }
            }
        }
    }

    return min;
}
```

E.T. calls 'homie' 1/2

Despite E.T. being part of a super advanced alien race... he has serious troubles to call his friends to pick him up when drunk.

He can remember that his friend telephone number has 7 digits. As he starts getting lucid again, he remembers the sum of the digits in the number is 42.

You have to help E.T. printing a list of all the possible numbers for which both restrictions are valid (not because you like E.T., but he is at your home and he is a real jackass in his current state).

E.T. calls 'homie' 2/2

```
void list_numbers(int state[],int k,int digits,int must_sum,int sum){
    if(k==digits){
        if(valid(must_sum, sum)){
            for(int i=1;i<=digits;i++){
                cout<<state[i];
            }
            cout<<endl;
        }
    } else {
        ++k;
        int temp_sum;

        for(int i=0;i<10;++i){
            state[k] = i;
            temp_sum = sum + i;
            if(temp_sum > must_sum) continue;
            list_numbers(state, k, digits, must_sum, temp_sum);
        }
    }
}
```

The complexity for this is $O(n^m)$ where n is the amount of different symbols [0-9] and m is the length of the telephone number.

Outline

Divide and conquer

Based on three steps:
1) Separate the problem in smaller cases
2) Solve those smaller cases
3) Mix the solutions

Pros

- Easy to code as a recursion.
- A lot of efficient algorithms are written this way!
- Search space is reduced by half every time.

Cons

- You need to be careful about base cases (infinite loop).

Problem 3.- Range Sum Query 1/3

Given an static array of N values A = $[2, 3, 5, 1, 3, 7, 3, 8]$ you are asked to respond to queries like "What is the sum from i to j?", where i and j are between [0,N-1] and $i <= j$.

Problem 3.- Range Sum Query 2/3

In order to answer the queries it is necessary to build a segment tree first:

```
int build_tree(int ind,int p,int q){
    if(p==q){
        return st[ind] = A[p];
    } else {
        int m = (p+q)>>1;
        int left = ind<<1;
        int right = left+1;

        int sum = build_tree(left  ,p  ,m);
        sum +=    build_tree(right ,m+1,q);

        return st[ind] = sum;
    }
}
```

$O(nlog_2n)$ is a great building complexity, but how do we make a query? ...

Problem 3.- Range Sum Query 3/3

The query code could look like this:

```
int query(int ind,int p,int q,int i,int j){
    if(p==i && q==j) return st[ind];

    int m = (p+q)>>1;
    int left = ind<<1;
    int right = left+1;

    if(i> m) {
        return query(right, m+1, q, i, j);
    } else if(j<= m) {
        return query(left ,   p, m, i, j);
    } else {
        return query(right, m+1, q, m+1, j)\
             + query(left ,   p, m,   i, m);
    }
}
```

$O(log_2 n)$ per query, isn't that cool enough? Well this data structure supports $O(log_2 n)$ updates, for the case where A is a dynamic array.

Outline

Dynamic Programming

Paradigm focused on constructing the answer of a problem by solving the smaller instances first. Some people just call it memoization.
A problem needs to exhibit two properties to be approached this way: **optimal substructure and overlapping sub-problems**.

Pros

- Efficient, never computes the same case twice.
- Correct, it evaluates all the possible solutions.
- 2 flavours!, bottom-up and top-down.

Cons

- Memory costly, is it feasible to store the answers for the N-1 sub-problems?

DP(Dynamic programming) based problems are frequently described in terms of a recurrence formula, the one for our coin change problem can be written as:

$$f(N) = \begin{cases} 0, & \text{if } N == 0. \\ 1 + min(f(N - coin_i)) \forall coin_i \in \mathbf{S}, & \text{otherwise.} \end{cases} \tag{1}$$

where **S** is a set containing the different coins.

Base cases are those for which you already know the answer, for this scenario, we know we need 0 coins to give 0 pesos back. But it has a potential problem. What would happen if N is negative?

This version uses 1 coin at a time, it builds the optimal solution for the [0, N-1] cases before calculating N.

```
int get_coins_dp_bottom_up(int N){
    int memo[N+1];
    int coin_types = sizeof(coins) / sizeof(int);

    for(int i=1;i <= N;++i) memo[i] = INF;
    memo[0] = 0;

    for(int i=0;i < coin_types;++i){
        for(int j=1;j <= N;++j){
            if(j>=coins[i]) {
                memo[j] = min(memo[j] , memo[j-coins[i]]+1);
            }
        }
    }

    return memo[N];
}
```

Being N the amount of change to give, and M the different types of coins, $O(MN)$ describes the running time complexity and $O(N)$ the space complexity.

This version perfectly reflects the formula we described before, so its easier to implement if you are comfortable with recurrences.

```
int top_down(int N,int *memo,int coin_types){
    if(N<0) return INF;
    if(INF!=memo[N]) return memo[N];

    int min = INF;
    for(int i=0;i < coin_types;++i){
        int temp = top_down(N-coins[i], memo, coin_types)+1;
        if(temp < min)
            min = temp;
    }

   return memo[N] = min;
}

int get_coins_dp_top_down(int N){
    int coin_types = sizeof(coins) / sizeof(int);
    int memo[N+1];

    for(int i=0;i <= N;++i)
        memo[i] = INF;
    memo[0] = 0;

    top_down(N, memo, coin_types);
    return memo[N];
}
```

Outline

Greedy algorithms

Technique based on making the best local decision every time expecting to obtain the optimal solution. A greedy algorithm need to exhibit an **optimal substructure** too.

Pros

- Faster than DP.
- Optimal, if the problem exhibits an optimal substructure.
- Easier code most of the times.

Cons

- Not easy to prove (correctness proof).
- Heuristics are needed sometimes.

The code for the coin change problem could look like this:

```
int get_coins_greedy(int N){
    int coins_counter = 0;
    int coin_types = sizeof(coins) / sizeof(int);

    for(int i=--coin_types;i >= 0;--i){
        while(N >= coins[i]){
            ++coins_counter;
            N-= coins[i];
        }
    }

    return coins_counter;
}
```

As silly as it looks, this code runs in $O(N)$ and needs no extra saving space, but, does it really works for any coins set?

Q & A

References

- Competitive Programming site
- Algorists' repository