Recursion, Divide and Conquer

19 - 3 - 2016 *Alex Poon*

Outline of Lecture

1. Function & Procedure
2. Recursion
3. Exhaustion
4. Divide & Conquer

Function

Function in Maths :

Input -> Process -> Output

e.g. F(x) = x^2 + 2x + 1

x is the input, F() is the process, y is the Output

Function in OI :

Similar meaning in Math

A program segment that perform : (Input) -> Process -> Output

Function in OI

```
int f(int x) {
   int y = x * x + 2x + 1;
   return y;
}

int main ()
{
        int val = f(5);
        cout << val;
        return 0;
}
```

Define a function (function name & type/no. of input)

Process of the function

Output of the function

Call the function with parameter x = 5

Function

```
int f(int x)
{
        int y = x * x + 2x + 1;
        return y;
}

int main ()
{
        printf("%d\n", f(3));
        return 0;
}
```

```
int main ()
{
        y = 3 * 3 + 2 * 3 + 1;
        printf("%d\n", y);
        return 0;
}
```

This two program yield the same output

Function

Similar to Math, there can be more than 1 input value in function

e.g.

```
int dist(double x, double y, double x2, double y2)
{
        double distance = (x - x2) * (x - x2) + (y - y2) * (y - y2);
        return sqrt(distance);
}
```

Procedure

Procedure : A sub-program

Similar to a function but does not give an output

i.e. Input -> Process

Procedure

```
void read() {
        cin >> a >> b;
}
void solve() {
        c = a + b;
        cout << c;
}
int main () {
        read();
        solve();
        return 0;
}
```

```
int main () {
        cin >> a >> b;
        cout << a + b;
        return 0;
}
```

These two program are same

Function & Procedure

Use of Function & Procedure:

1. Make your code cleaner & more understandable
2. Perform recursion

Recursion

Recursion is a function or procedure call itself

e.g.

```
int f(int x)
{
        return f(x - 1) + f(x - 2);    // We call function f in function f  -> recursion
}
```

Recursion

What is the meaning of a program calling itself?

Consider the following program:

```
void f(int x) {
        ans += x * x;
        if (x > 0) f(x - 1);
        ans++;
}
int main () {
        ans = 0;
        f(3);
        cout << ans;
}
```

Recursion

How this program run ?

```
void f(int x) {
        ans += x * x;          // line 1
        if (x > 1) f(x - 1);   // line 2
        ans += ans;            // line 3
}
int main () {
        ans = 0;
        f(3);
        cout << ans;
}
```

1. Call f(3)
2. Run line 1 of f(3)
3. Run line 2 of f(3) -> call f(2)
4. Run line 1 of f(2)
5. Run line 2 of f(2) -> call f(1)
6. Run line 1 of f(1)
7. Run line 2 of f(1) -> won't call f(0) as not (x > 1)
8. Run line 3 of f(1) -> end of f(1)
9. Run line 3 of f(2) -> end of f(2)
10. Run line 3 of f(3) -> end of f(3)

Program does not run sequentially with recursion.

Recursion

With recursion :

Our program can run in a way that is not sequentially

We can “jump” to run the first statement of the sub-program at anytime

Therefore, recursion help us to solve problem with following properties:

1. The problem can divide into same problem with smaller parameter (I call this sub-problem)
2. We need information of sub-problem to solve the current problem

The above problem-solving-method call divide and conquer

We use examples to explore recursion & divide and conquer now

1. Factorial

Problem statement : Find factoral of x (x! = x * (x - 1) * … 1)

Instead of use for loop to solve, we can use recursion to solve it.

Analysis : x! = x * (x - 1)! where (x - 1)! is same problem with smaller parameter

```
int factorial(int x) {
        return x * factorial(x - 1);
}
```

done……(?)

1. Factorial

```
int factorial(int x) {
        return x * factorial(x - 1);
}
```

According to this program segment:

f(2) = 2 * factorial(1)

f(1) = 1 * factorial(0)

f(0) = 0 * factorial(-1) …

Seems I miss something

1. Factorial

We should also tell our program when to stop recurring

```
int factorial(int x) {
        if (x > 0) return x * factorial(x - 1);
        else if (x == 0) return 1;          // 0! = 1
}
```

We call the critiria of stop recursion the base case : if (x == 0) return 1;

We call the relation of problem & sub-problem the recurrance relation : x * factorial(x - 1);

2. Fibonacci number

Problem : Find the x-th fibonacci number

Analysis : f(x) = f(x - 1) + f(x - 2) where f(x - 1) and f(x - 2) is same problem with smaller parameter

```
int f(int x) {
        if (x > 2) return f(x - 1) + f(x - 2);   // recurrance relation
        else if (x == 1 || x == 2) return 1;   // base caseS
}
```

3. Knapsack problem

Problem : There are N objects, each of them has its value v[i] and weight w[i] (in kg)

We can choose buying some of the objects

but the total weight of the things we buy cannot over W

We hope to maximum the total value of the things we buy

Sample input:

4 10.5 (n, W)

7.5 100 (w[i], v[i])

3.1 10

5 50

5.5 51

Sample output: 101

3. Knapsack problem

Greedy approach cannot give us the optimal answer

We can try all combination of find the optimal answer

All combination = {Y, Y, Y, Y}, {Y, Y, Y, N}, {Y, Y, N, Y}, {Y, Y, N, N} ..... {N, N, N, N}

How to code?

3. Knapsack problem

Code :

```
if (n == 1) for (bool a = false; a <= true; a++)
if (n == 2) {
        for (bool a = false; a <= true; a++)
        for (bool b = false; b <= true; b++)
}
if (n == 3) {
        for (bool a = false; a <= true; a++)
        for (bool b = false; b <= true; b++)
        for (bool c = false; c <= true; c++)
}
.....
```

3. Knapsack problem

Think about recursion

Origin problem: Decide whether to buy object 1..N

Assume we buy the first object

new problem: Decide whether to buy object 2..N (same problem with smaller parameter)

Assume we do not buy the first object

new problem: Decide whether to buy object 2..N (same problem with smaller parameter)

We get the recurrence relation!!

3. Knapsack problem

```
void knapsack(int x) {                  //   decide whether to buy object x to N
        // assume we buy object x
                curweight += w[x];
                curvalue += v[x];
                knapsack(x + 1);        // decide whether to buy object x + 1 to N
        // assume we do not buy object x
                curweight -= w[x];
                curvalue -= v[x];
                knapsack(x + 1);
}
```

Don’t forget the base case

3. Knapsack problem

Base case analysis: When we will stop recursion ?

We stop after we had decided whether to buy all of the N objects

knapsack(1) represent we are deciding whether to buy object 1 to N

knapsack(N) represent we are deciding whether to buy object N to N

…

therefore

when x = N + 1 represent we had decided whether to buy all N objects

3. Knapsack problem

```
void knapsack(int x) {                  //   decide whether to buy object x to N
        if (x == N + 1) {
                if (curweight <= W && curvalue > ans) ans = curvalue;
        }
        else {
                curweight += w[x];
                curvalue += v[x];
                knapsack(x + 1);        // decide whether to buy object x + 1 to N
                curweight -= w[x];
                curvalue -= v[x];
                knapsack(x + 1);
        }
}
```

Exhaustion

In the previous example, we use recursion to Try All Combination which is also call exhaustion
This algorithm is slow but intuitive and correct all the time

We can get partial score by exhaustion in OI contest all the time.
When we encounter difficult problem, use exhaustion to get some score is a good strategy !!

Summary

We can think the problem in the following way :

1. The problem can divide into same problem with smaller parameter
2. If yes, it can be solved by divide & conquer
3. What is the recurrence relation?
   3.1. How to divide it to smaller-parameter-problem

   3.2 How to solve if we get the information of smaller-parameter-problem
4. What is the base case?
5. Use recursion to code it !!

In the examples above, we divide the problem f(x) into f(x - 1).

The parameter of sub-problem reduce by 1

In some case we can divide the problem f(x) into sub-problem with even smaller paramter

4. Big mod

Problem : Given B, P, M. Find (B^P) % M

Naive solution :

```
ans = 1;
for (int i = 0; i < P; i++) { ans *= B; ans %= M; }
cout << ans;
```

Time complexity: O(N)

4. Big mod

In the solution above, we use B^P = B^(P - 1) * B

i.e. f(x) = f(x - 1) * B

We divide f(x) to f(x - 1) every time

A better way is to divide f(x) to f(x / 2) every time !!!

4. Big mod

Analysis : Let f(x) = B^x

Recurrance relation: if (x is even) f(x) = f(x / 2) * f(x / 2)

if (x is odd) f(x) = f(x / 2) * f(x / 2) * B

How about base case? When we stop recurring?

When x = 1, we are calculating B^1. We can direct return the answer B

```
int f(int x) {
    if (x == 1) return B % M;
    int tmp = f(x / 2) % M;
    if (x % 2 == 0) return (tmp * tmp) % M;
    else return (tmp * tmp * B) % M;
}
```

4. Big mod

Every time we reduce f(x) to f(x / 2) until x = 1

Let N = 30, we only need to call f(30), f(15), f(7), f(3), f(1) which is around lg(N) call

So, time complexity is lg(N)

By divide & conquer, we sometime can reduce the time complexity if we divide the problem appropreiately.

5. L-piece

Constructive problem can often be solved by Divide and Conquer

As constructive problem with small constraint are usually intuitive and it can often be divided into sub-problem with smaller constraint

Problem : Given a N * N grip with a empty cell. Output a way to use L-piece to cover the whole grip where N is a power of 2

Sample input	Sample Output
4 4	AABB
0000	ACoB
0010	DCCE
0000	DDEE
0000	

5. L-piece

When we encounter constructive problem, according to Alex Tung, we should try small example

When N = 2, there are 4 cases

10 01 00 00

00 00 10 01

Just put a L-piece on the non-empty cells, done.

5. L-piece

For larger cases (e.g. 4 x 4), can we divide it to same problem with smaller constraint?

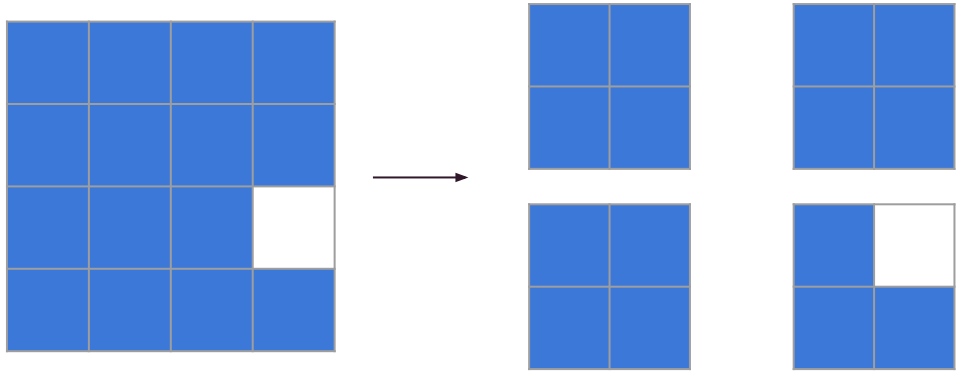
Note that in this problem: N = 2 ^ k

So, 3 x 3, 1 x 2, 4 x 2... with 1 empty cell is NOT the same problem with smaller constraint as N != 2 ^ k

So, we can just consider can 4 x 4 grip with 1 empty cell divide into 2 x 2 grip with 1 empty cell

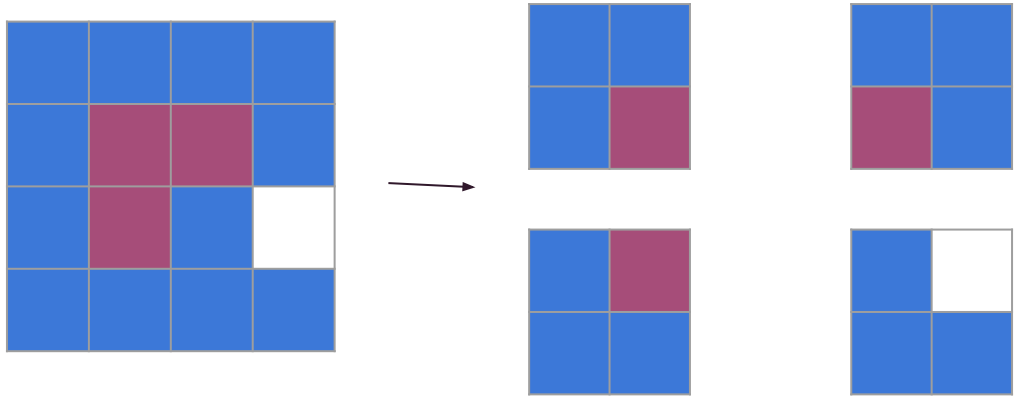
5. L-piece

As we need to divide a 4 x 4 grip to a 2 x 2 grip, let try dividing it into 4 sub-grip



We hope that each sub-grip has exactly 1 empty cell, how can we attain this?

5. L-piece



Yeah!!! We reduce it to smaller-parameter-same-problem!!!

Summary

Be careful what the meaning of SAME Problem with smaller constraint

Sometimes we do not divide the problem f(x) to f(x - 1), but f(x / 2) or other

You should at least familiar with how to use recursion to perform exhaustion (example 3) as it is useful and common

Practise problem:

Exhaustion problem: HKOJ 01031, 01037, 01048, J092, [01049, 01050 (some optimization required)]

Divide and Conquer problem: HKOJ : 01003, 01046, 01047, S134, S163

Other Example or practise