
Recursion, Divide and Conquer

— 25 - 2 - 2017 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;  
}
```

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

Process of the function

Output of the function

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

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.

```
double 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

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

```
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;  
}
```

The program run in the following order:

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 factorial of x ($x! = x * (x - 1) * \dots * 1$)

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);  
}
```

1. Factorial

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

According to this program segment:

$f(2) = 2 * \text{factorial}(1)$

$f(1) = 1 * \text{factorial}(0)$

$f(0) = 0 * \text{factorial}(-1) \dots$

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 criteria of stop recursion the **base case** : if (x == 0) return 1;

We call the relation of problem & sub-problem the **recurrence relation** : $x * \text{factorial}(x - 1)$;

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); // recurrence relation  
    else if (x == 1 || x == 2) return 1; // base case  
}
```

3. Knapsack problem

1. There are N objects, each of them has its value $v[i]$ and weight $w[i]$ (in kg)
2. We can choose buying some of the objects
3. But the total weight of the things we buy cannot larger than W
4. We hope to maximize 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 (Buy object 3, 4)

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

i.e. 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 !!
6. In the examples above, we divide the problem $f(x)$ into $f(x - 1)$.

The parameter of sub-problem reduce by 1

Divide and Conquer

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 $f(x) = B^x$

Recurrence 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

4. Big mod

```
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 appropriately.

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 $N * N$ grid with a empty cell. Output a way to use L-piece to cover the whole grid where N is a power of 2

Input :	Output:
4	BBCC
0000	BAAC
0000	DAEo
0001	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. 4×4), can we divide it to same problem with smaller constraint?

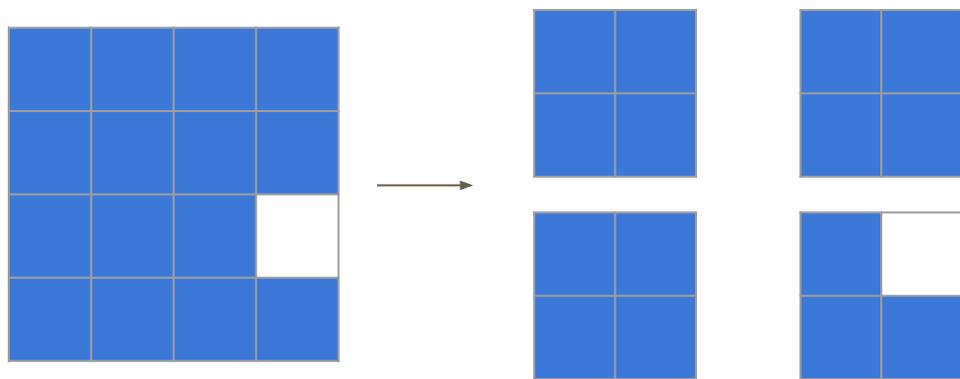
Note that in this problem: $N = 2^k$

So, 3×3 , $1 \times 2 \dots$ with 1 empty cell is **NOT** the same problem with smaller constraint as $N \neq 2^k$

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

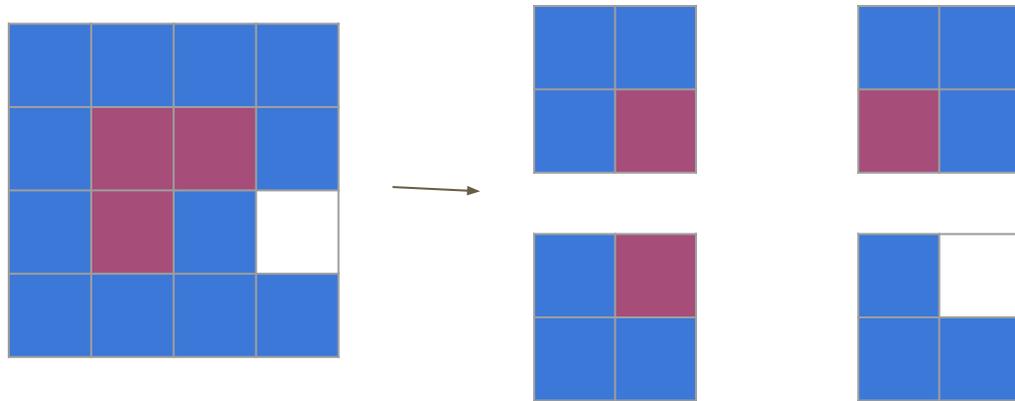
5. L-piece

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



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

5. L-piece



6. Merge sort

Given an array with N integer ($N \leq 100000$)

Output the sorted array

Input :

8

3 4 1 9 8 10 4 5

Output:

1 3 4 4 5 8 9 10

Bubble sort leads to time limit exceeded, we should use faster sorting algorithm

Merge sort can sort an array in $O(N \lg N)$!!!

6. Merge sort

What is merging ?

Given 2 SORTED array, can we combine the 2 array into 1 SORTED array in $O(N)$ time?

e.g.

Array A = {3, 8, 10, 10}

Array B = {2, 3, 7, 8}

We can solve this by pushing the smaller element in the FRONT of 2 array every time

6. Merge sort

Array A	Array B	Resulting Array
3, 8, 10, 10	2, 3, 7, 8	2
3, 8, 10, 10	3, 7, 8	2, 3
8, 10, 10	3, 7, 8	2, 3, 3
8, 10, 10	7, 8	2, 3, 3, 7
8, 10, 10	8	2, 3, 3, 7, 8
10, 10	8	2, 3, 3, 7, 8, 8
10, 10		2, 3, 3, 7, 8, 8, 10
10		2, 3, 3, 7, 8, 8, 10, 10

6. Merge sort

This show that we can merge two sorted array to 1 sorted array in $O(|A| + |B|)$ time

How can we use this skill to sort an array?

Think about divide and conquer !!!

6. Merge sort

Analysis : If we **split the sequence to two sequence**, and **sort each of them**, we can **merge them in $O(|A| + |B|)$**

Problem : sort the sequence

Divide : **We split the sequence to two**

Sub-problem : **Sort the splitted sequence**

Conquer : **Merge them in $O(|A| + |B|)$**

6. Merge sort

Problem : sort the sequence

Divide : We split the sequence to two

Sub-problem : Sort the splitted sequence

Conquer : Merge them in $O(|A| + |B|)$

Assume the procedure `sort(1, n)` sort the array A from position 1 to n, we can translate the above logic to code.

```
void sort(int start, int end) {  
    sort(start, mid);  
    sort(mid + 1, end);  
    merge(start, mid, mid + 1, end);  
}
```

How about base case?

6. Merge sort

Analysis : If the sequence we are sorting have length = 1, we can stop dividing as we need not to sort an array with length = 1

```
void sort(int start, int end) {  
    int mid = (start + end) / 2;  
    if (start != mid) sort(start, mid);  
    if (mid + 1 != end) sort(mid + 1, end);  
    merge(start, mid, mid + 1, end);  
}
```

Done.

Implementation of procedure merge:

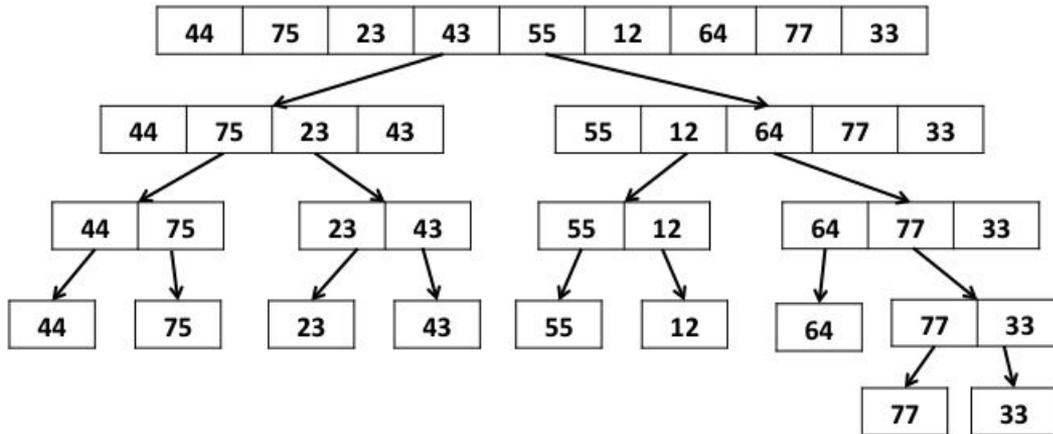
```
void merge(int st1, int ed1, int st2, int ed2) {  
    int n = 0, m = 0;  
    for(i, st1, ed1) tmpA[n++] = A[i];  
    for(i, st2, ed2) tmpB[m++] = A[i];  
    int x = 0, y = 0;  
    for(i, st, ed2) {  
        if (y == m || tmpA[x] <= tmpB[y]) {  
            A[i] = tmpA[x]; x++;  
        }  
        else {  
            A[i] = tmpB[y]; y++;  
        }  
    }  
}
```

6. Merge sort

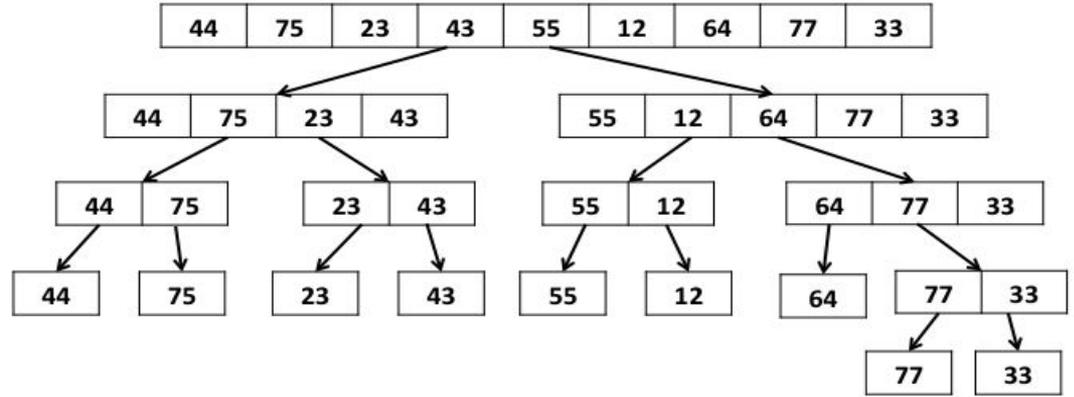
What is the time complexity of quick sort?

Assume our array have the length = 9 which is {44, 75, 23, 43, 55, 12, 64, 77, 33}

We need to divide the sequence like the following



6. Merge sort



We can see that

1. The number of element in each layer equal to N (except the last layer)
2. There are at most $\lg(N) + 1$ layer totally
3. We need $O(|A| + |B|)$ = the number of element in the sequence we sort for 1 merging procedure
4. i.e. The time complexity = number of element in the above picture = $N * (\lg N + 1) = O(N \lg N)$

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

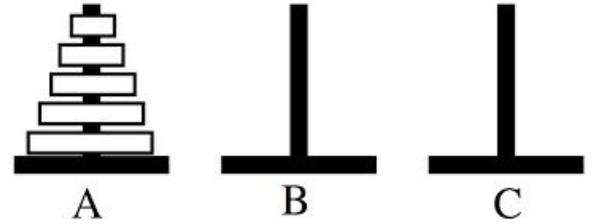
Advance divide & conquer

Some advance d&c technique will (may) be covered in HK-team training.

1. Divide and conquer on tree
2. DP optimized by divide and conquer
3. Closest pair

Extra : Tower of Hanoi

1. We have 3 towers and N pieces of disks
2. We would like to move all N pieces of disks to tower C
3. Each time we should only move 1 disk
4. A larger disk cannot put on the top of a smaller disk



Given n , output a sequence of moves to move all N disks from tower A to C

E.g. $N = 2$

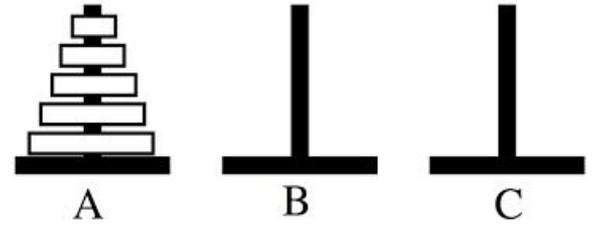
Output:

Move disk 1 from A to B

Move disk 2 from A to C

Move disk 1 from B to C

Extra : Tower of Hanoi

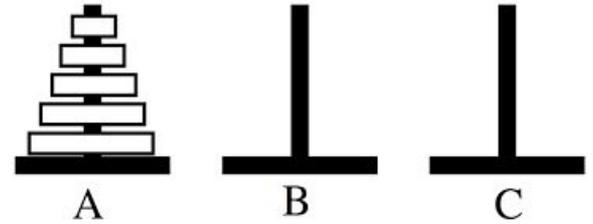


Analysis : If we want to move all N disks from A to C

1. We need to move the Nth (largest) disks from A to C
2. In order to move the Nth disks from A to C, we should move the 1 - N-1 disks from A to B
3. After we move the Nth disks from A to C, our tasks become : move N-1 disks from B to C

Let solve(N, A, B, C) means we need to move ALL N disks from A to C, using B as a temporary storage

Extra : Tower of Hanoi

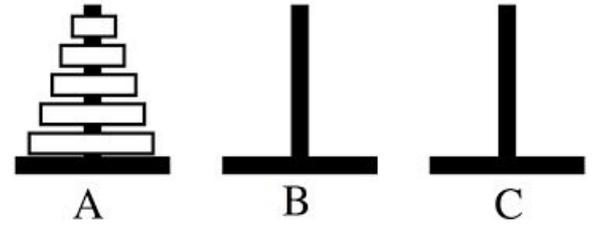


1. We need to move the Nth (largest) disks from A to C
2. In order to move the Nth disks from A to C, we should move the 1 - N-1 disks from A to B
3. After we move the Nth disks from A to C, our tasks become : move N-1 disks from B to C

```
void solve(int n, char A, char B, char C) {  
    solve(n - 1, A, C, B); // move n-1 disks from A to B  
    printf("Move disk %d from %c to %c\n", n, A, C); // move the Nth disks from A to C  
    solve(n - 1, B, A, C) // move n-1 disks from B to C  
}
```

How about base case?

Extra : Tower of Hanoi



Base case analysis : If $N = 1$, we can directly move it to the destination

```
void solve(int n, char A, char B, char C) {  
    if (n == 1) printf("Move disk %d from %c to %c\n", n, A, C);  
    else  
    {  
        solve(n - 1, A, C, B); // move n-1 disks from A to B  
        printf("Move disk %d from %c to %c\n", n, A, C); // move the Nth disks from A to C  
        solve(n - 1, B, A, C) // move n-1 disks from B to C  
    }  
}
```

Done :)