香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Recursion, Divide and Conquer

Chau Lai Yin {happychau}

2023-03-11

Table of Content

Functions

- Functions in Math:
  - $f(x) = \sin(x)$, $f(x) = x^2+2x+1$
  - Substitute x (Input) into the f() (Process), return the calculated value (Output)

- Functions in OI
  - Similar to maths
  - As a subroutine that process the input/parameters and return the outputs

Function in OI

- There can be multiple input for a function

```
int f(int x) {                  // Define a function with parameter(s)
 int y = x * x + x + 2;        // Process the input
 return y;                     // Return the output
}

int main() {
 cout << f(7) << endl;         // Call the function with parameter x = 7
}
```

```
Output:
58
```

Function in OI

- There can be multiple input for a function

```
double dist(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}


int main() {
    cout << dist(0.0, 0.0, 1.1, 1.1) << endl;
}
```

```
Output:
1.55563
```

Procedure

- Similar to functions
- It can take inputs and process like functions
- But with no return values

Procedure

```
void hello(int n) {        // Define a procedure name with an int input
    for (int i = 1; i <= n; i++)     // Process the input
        cout << "HKOI" << endl;
    return;
}

int main() {
    hello(6);                         // Call the procedure with n = 6
}
```

```
Output:
HKOI
HKOI
HKOI
HKOI
HKOI
HKOI
```

Recursion

- A function or procedure that call itself
- But we have to beware of infinite loop
- Analyze base case and recurring case

Example - GCD

- Recursion!
- Base case and recurring case included by the formula.

```
int gcd(int a, int b) {
    if (b == 0) return a;    // Base case
    return gcd(b, a % b);    // Recurring case
}

int main() {
    cout << gcd(84, 36) << endl;
}
```

```
Output:
12
```

Example - GCD

- It is very common in OI to compute GCD of two numbers
- Most common way of computing GCD quickly is by Euclidean algorithm

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;   // false
    return gcd(18, 84 % 18 = 12);
}
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;   // false
    return gcd(18, 84 % 18 = 12);
}
```

```
int gcd(a = 18, b = 12) {
    if (12 == 0) return a;   // false
    return gcd(12, 18 % 12 = 6);
}
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;   // false
    return gcd(18, 84 % 18 = 12);
}
```

```
int gcd(a = 18, b = 12) {
    if (12 == 0) return a;   // false
    return gcd(12, 18 % 12 = 6);
}
```

```
int gcd(a = 12, b = 6) {
    if (6 == 0) return a;   // false
    return gcd(6, 12 % 6 = 0);
}
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;   // false
    return gcd(18, 84 % 18 = 12);
}
```

```
int gcd(a = 18, b = 12) {
    if (12 == 0) return a;   // false
    return gcd(12, 18 % 12 = 6);
}
```

```
int gcd(a = 12, b = 6) {
    if (6 == 0) return a;   // false
    return gcd(6, 12 % 6 = 0);
}
```

```
int gcd(a = 6, b = 0) {
    if (0 == 0) return 6;   // true
    // return gcd(6, 12 % 6 = 0);
}
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;   // false
    return gcd(18, 84 % 18 = 12);
}
```

```
int gcd(a = 18, b = 12) {
    if (12 == 0) return a;   // false
    return gcd(12, 18 % 12 = 6);
}
```

```
int gcd(a = 12, b = 6) {
    if (6 == 0) return a;   // false
    return 6;
}
```

```
// int gcd(a = 6, b = 0) {
//     if (0 == 0) return 6;   // true
//     return gcd(6, 12 % 6 = 0);
// }
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;  // false
    return gcd(18, 84 % 18 = 12);
}
```

```
int gcd(a = 18, b = 12) {
    if (12 == 0) return a;  // false
    return 6;
}
```

```
// int gcd(a = 12, b = 6) {
//   if (6 == 0) return a;  // false
//   return 6;
// }
```

```
// int gcd(a = 6, b = 0) {
//     if (0 == 0) return 6;  // true
//     return gcd(6, 12 % 6 = 0);
// }
```

Recursion

```
int main() {
    cout << gcd(84, 18) << endl;
}
```

```
int gcd(a = 84, b = 18) {
    if (18 == 0) return a;  // false
    return 6;
}
```

```
// int gcd(a = 18, b = 12) {
//   if (12 == 0) return a;  // false
//   return 6;
// }
```

```
// int gcd(a = 12, b = 6) {
//   if (6 == 0) return a;  // false
//   return 6;
// }
```

```
// int gcd(a = 6, b = 0) {
//    if (0 == 0) return 6;  // true
//    return gcd(6, 12 % 6 = 0);
// }
```

Recursion

```
int main() {
    cout << 6 << endl;
}
```

```
// int gcd(a = 84, b = 18) {
//   if (18 == 0) return a;   // false
//   return 6;
// }
```

```
// int gcd(a = 18, b = 12) {
//   if (12 == 0) return a;   // false
//   return 6;
// }
```

```
// int gcd(a = 12, b = 6) {
//   if (6 == 0) return a;   // false
//   return 6;
// }
```

```
// int gcd(a = 6, b = 0) {
//     if (0 == 0) return 6;   // true
//     return gcd(6, 12 % 6 = 0);
// }
```

Recursion

- By using recursion, we can simplify our code

- Recursion can help us solve problem with following properties:
  - The problem can be divided / reduced into same problem with smaller parameter
  - We need informations of the sub-problems to solve the current one

Recursion

- The problem can be divided / reduced into same problem with smaller parameter
- We need informations of the sub-problems to solve the current one

```
int gcd(int a, int b) {
    if (b == 0) return a;    // Base case
    return gcd(b, a % b);    // Recurring case
}

int main() {
    cout << gcd(84, 36) << endl;
}
```

```
Output:
12
```

Example — Fibonacci number

- Find the n-th Fibonacci number
  - Fibonacci number: a sequence in which each number is the sum of the two preceding ones.
  - First few Fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Example — Fibonacci number

- Find the n-th Fibonacci number
  - Fibonacci number: a sequence in which each number is the sum of the two preceding ones.
  - First few Fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
- Warning: **DO NOT** use recursion to calculate the n-th Fibonacci number using recursion without memoization as it is very slow (this is just a demonstration :D).
- Exponential time complexity

Example — Fibonacci number

- Fibonacci number: a sequence in which each number is the sum of the two preceding ones.
- Formalizing the idea
  - $F_n = F_{n-1} + F_{n-2}$
- Base case
  - $F_0 = 0, F_1 = 1$

Example — Fibonacci number

- Fibonacci number: a sequence in which each number is the sum of the two preceding ones.
- Formalizing the idea
  - $F_n = F_{n-1} + F_{n-2}$ - (can be solved recursively!)
- Base case
  - $F_0 = 0, F_1 = 1$

Example — Fibonacci number

- Find the n-th Fibonacci number
- Base case: $F_0 = 0, F_1 = 1$
- Recurrence relation: $F_n = F_{n-1} + F_{n-2}$

```
int fib(int n) {
   if (n == 0 || n == 1) return n;      // Base case
   return fib(n - 1) + fib(n - 2);      // Recurrence relation
}


int main() {
   cout << fib(6) << endl;
}
```

```
Output:
8
```

Example — Tower of Hanoi

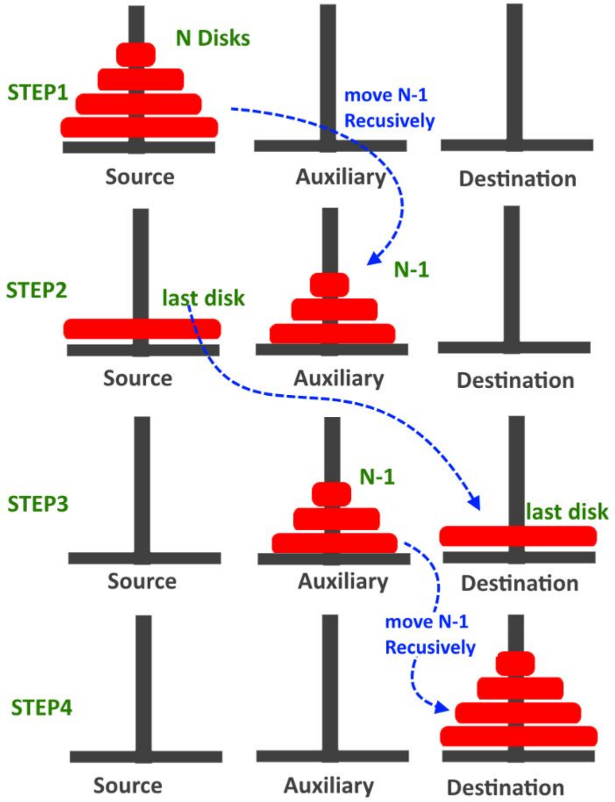
- There are 3 pegs, numbering 0, 1, 2 from left to right
- There are initially N disks of different size on the 0th peg.
- The disks increase in size from top to bottom
- The goal is to move entire tower of disks from 0th peg to the one of the other peg
- One disk can be moved from the top of one peg to another empty peg or peg with larger size topmost disk

Example — Tower of Hanoi

- Strategy:
- Name each peg by start, end and intermediate
- Move the topmost n-1 disks from start to intermediate
- Move the largest disk to end, now the start peg is empty
- Treat the start peg as intermediate peg, intermediate peg as start and repeat the step above.

Example — Tower of Hanoi

Image source:
https://medium.com/@jamalmaria111/tower-of-hanoi-js-algorithm-3f667fa46f0f



Example — Tower of Hanoi

- Base case: When n = 1, move the disk from start to end.
- Recurrence relation, when n > 1:
  - Move n-1 disk from start to intermediate
  - Move 1 disk from start to end
  - Move n-1 disk from intermediate to end

- Let p(n, start, inter, end) represents the problem we want to solve. Where we move n disks from start peg to end peg using the intermediate peg.

Pseudocode

```
void p(int n, int start, int inter, int end) {
    if (n == 1) {
        // [Move largest disk from start to end]
    }
    else {
        p(n - 1, start, end, inter);
        // [Move largest disk from start to end]
        p(n - 1, inter, start, end);
    }
}
```

Exhaustion

- Sometimes we don’t know a fast algorithms to solve a problem
- Exhaust all possible state/solutions
- Check whether the state is the one we want
- Output the best / first found

Example — Subset sum

- Given a list of N integers, find a subset of integers that sums to S.
- e.g. A = [1,2,4,8,16], S = 13
- The subset that gives S = 13 is [1,4,8]

Example — Subset sum

- Try subset of size 1, 2, 3, … , n
- Using for loop?

Example — Subset sum

- What a mess and we need so many lines of repetitive code.

```
for(int i = 0; i < n; i++)
    if (a[i] == S) cout << a[i] << endl;
for(int i = 0; i < n; i++)
    for(int j = i+1; j < n; j++)
        if(a[i] + a[j] == S) cout << a[i] << ' ' << a[j] << endl;
for(int i = 0; i < n; i++)
    for(int j = i + 1; j < n; j++)
        for(int k = j + 1; k < n; k++)
            if(a[i] + a[j] + a[k] == S) cout << a[i] << ' ' << a[j] <<  ' ' << a[k] << endl;
```

Example — Subset sum

- This can be solved by recursion
- For each number, we can decide whether or not to choose it.
    - Try both!
- Define F(i, X) denoting we have considered i number and the current sum is X
- Base case: i == n, if X = S, output, else this subset can't add up to the sum we want.
- Recurrence relation:
    - Try F(i+1, X + a[i]) // Include this number
    - and F(i+1, X) // Not to include this number

Example — Subset sum

```
bool choose[n]; // all initialized to false
void solve(int i, int X) {
    if (i == n && X == S) {
        for (int j = 0; j < n; j++)
            if(choose[j]) cout << a[j] << ' ';
    }
    else if (i != n){
        choose[i] = true; solve(i+1, X + a[i]);
        choose[i] = false; solve(i+1, X);
    }
}
```

Example — Subset sum

- Time complexity: $O(2^N)$
- Which can give solutions within 1s when N <= 20
- There exist solutions with pseudo-polynomial time by using DP
    - Check out Dynamic Programming (I)

Example — Permutation

- Generating sequences of permutation
- Can also be done using next_permutation in STL
- Time complexity: O(N!)
- Useful when N <= 10

Example — Permutation

- Suppose we want to permutate the string S = “ABCD”
- Define permutation(S, pos) as permuting S while the current position at pos
- Base case: When pos == S.size(), we can’t swap anymore so we just output
- Recurrence relation: For all i > pos, swap S[i] and S[pos], permutation(S, pos+1)

Example — Permutation

```
void permutation(string s, int pos) {
    if (pos == s.size()) cout << s << endl;      // Base case
    else {                                        // Recurrence case
        for (int i = pos; i < s.size(); i++) {
            swap(s[i], s[pos]);       // apply the change
            permutation(s, pos+1);    // recur to the next state
            swap(s[i], s[pos]);       // undo the change
        }
    }
}
```

Example — Permutation

- Using next_permutation in STL
- Very important to sort before using next_permutation or you may not get all the permutations

```
string s = "HKOI";
sort(s.begin(), s.end());
do {
   cout << s << endl;
}
while (next_permutation(s.begin(), s.end()));
int a[] = {9, 7, 1, 6};
sort(a, a + 4);
do {
   for (int i = 0; i < 4; i++)
       cout << a[i] << ' ';
   cout << endl;
}
while (next_permutation(a, a + 4));
```

Backtracking

- Most of the time when we are doing exhaustion, we do backtracking
- Which is using all the previous option except the current one
- The previous two examples also used backtracking so that the order of exhaustion is natural

Branch & Bound

- We may noticed that some state are invalid and recurring further won't make it valid again
- We can skip this state and not recur all the state below it which may saves a lot of time

Example — Subset sum revisited

- Current state: F(i, X)
- If X > S, adding any more number or not adding will not sum to S anyway
- Stop the recursion when X > S!

Example — Subset sum revisited

```
bool choose[n]; // all initialized to false
void solve(int i, int X) {
   if(i == n && X == S) {
       for (int j = 0; j < n; j++)
           if(choose[j]) cout << a[j] << ' ';
   }
   else if (i != n) {
       if (X + a[i] > S) return; // Branch cutting
       choose[i] = true; solve(i+1, X + a[i]);
       choose[i] = false; solve(i+1, X);
   }
}
```

Example — Subset sum revisited

- Constant optimization
- Time complexity: $O(2^N)$
- There is no guarantee we always cut the branch, but we can always do so when possible.

Example — Subset sum revisited

- More optimization can be done if you precompute suffix sum of array
- Branch cutting if choosing all number afterwards cannot even add to S
- Only works for all positive numbers

break;

Divide & Conquer

- Divide the problem into smaller and independent sub-problems that are the same as the original problem
- If the sub-problem is easy to solve, solve directly. Otherwise divide it into smaller sub-problems recursively.
- Combine the result/solutions from sub-problems to solve the original problem.

Master Theorem (Warning: Maths ahead)

- Master Theorem is used to calculate the time complexity of a divide-and-conquer algorithm
- Assume the time cost function of the problem is $T(n)$
- If $T(n) = aT(n / b) + O(n^d)$
- $O(n^d)$ can be regarded as the time cost of combining solutions
- $aT(n / b)$ can be regarded as dividing the problem into a sub-problems with parameters n/b.

Master Theorem

- $T(n) = aT(n / b) + O(n^d)$
- If $d > \log_b a$, $T(n) = O(n^d)$
- If $d = \log_b a$, $T(n) = O(n^d \log n)$
- If $d < \log_b a$, $T(n) = O(n^{\log_b a})$

- In real coding, you can just ignore all of the above calculations

Example — Big Mod

- Find B^P % M
- Naive solution: multiply B for P times, doing mod every time.
- Time complexity: O(P)

Example — Big Mod

- Find B^P % M
- When P is even, B^P % M = $B^{P/2}$ % M * $B^{P/2}$ % M
- When P is odd, B^P % M = B * ($B^{(P-1)/2}$ % M * $B^{(P-1)/2}$ % M)
- Base case: When P = 0, bmod(b, p, m) = 1
- Recurrence relation:
  - If P is even: bmod(b, p, m) = bmod(b*b%m, p/2, m)
  - If P is odd: bmod(b,p,m) = b * bmod(b*b%m, p/2, m), here p/2 is integer division so we can just skip the -1

Example — Big Mod

- Super useful, you can just add this to your code template

```
using ll = long long;

ll bmod(ll b, ll p, ll m) {
    if (p == 0)
        return 1;          // Base case
    else if (p % 2 == 0)
        return bmod(b * b % m, p / 2, m);
    else
        return b * bmod(b * b % m, p / 2, m) % m;
}

int main() {
    cout << bmod(2, 10, 1023) << endl;
}
```

```
Output:
1
```

Example — Big Mod

- Here, we divide P by 2 in each recursion
- There are at most log(P) times before P becomes 0
- Time complexity: O(log(P))

- Way faster than O(P)

Example — Big Mod

- How can we analyze the time complexity using master theorem?
- At each recursion, we divide problem f(P) into **one** sub-problem f(P/2)
- It takes O(1) to combine solutions because it is simple multiplication

Example — Big Mod

- A brief idea on Master theorem:
- Let $T(n) = aT(n / b) + O(n^d)$
  - If $d > \log_b a$, $T(n) = O(n^d)$
  - If $d = \log_b a$, $T(n) = O(n^d \log n)$
  - If $d < \log_b a$, $T(n) = O(n^{\log_b a})$
- $T(P) = T(P / 2) + O(1)$, $a = 1$, $b = 2$, $d = 0$, $\log_b a = 0$
- We have the case $d = \log_b a$, substitute the number we have $O(\log P)$

Example — Big Mod

- Exact same problem on HKOJ
- 20374 Big Mod

Example — Minimum and Maximum element

- Given an array of N elements
- Find the minimum and maximum element using minimum number of comparisons
- {92, 65, 91, 25, 16, 12, 3, 32}

Example — Minimum and Maximum element

- {92, 65, 91, 25, 16, 12, 3, 32}
- Naive approach: Loop through the array, 2N comparison needed
- Can we do better?

Example — Minimum and Maximum element

- Divide and conquer!
- Consider the following cases if we divide into smaller problem (set of integers)
- 1 element: The minimum and maximum are both this number
- 2 element: The minimum and maximum can be determined by 1 comparison

Example — Minimum and Maximum element

- Assume we have solved the subproblem, how do we merge?
- Let m_A, m_b, be minimum of set A and B, and M_A, M_B be maximum of set A and B
- We need only 2 comparison for merging the result of two subproblem
- Total number of comparison:
- T(1) = 1, T(2) = 2, T(n) = 2T(n/2) + 2
    - If $d < \log_b a$, $T(n) = O(n^{\log_b a})$
- Solving the above relation gives T(n) = 3n/2 - 2 ~= 1.5n

Example — Minimum and Maximum element

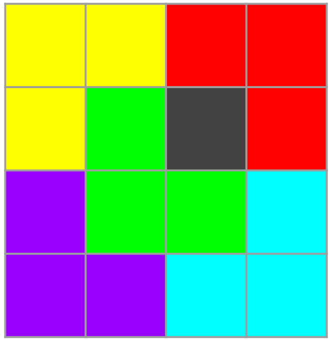
- {92, 65, 91, 25, 16, 12, 3, 32}
- Red is minimum, Blue is maximum
- {92, 65}, {91, 25}, {16, 12}, {3, 32}, cost = 4
- Merging {92, 65}, {91, 25} => {92, 25}, cost = 2
- Merging {16, 12}, {3, 32} => {3, 32}, cost = 2
- Merging {92, 25}, {3, 32} => {92, 3}, cost = 2
- Total cost = 4 + 2 + 2 + 2 = 10, better than 2n

Example — Minimum and Maximum element

- Exact same problem on HKOJ
- M1431 Comparing Game

Example — L-pieces

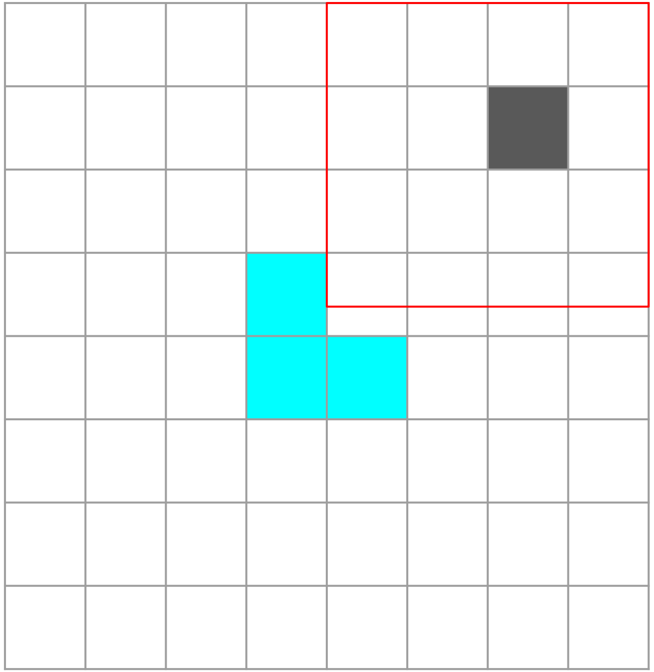
- Given a grid of N*N (N is a power of 2), all cells are initially empty except one of the cells.
- Fill the grid with L shape
- e.g., a 4*4 grid solution, the gray cell is originally filled



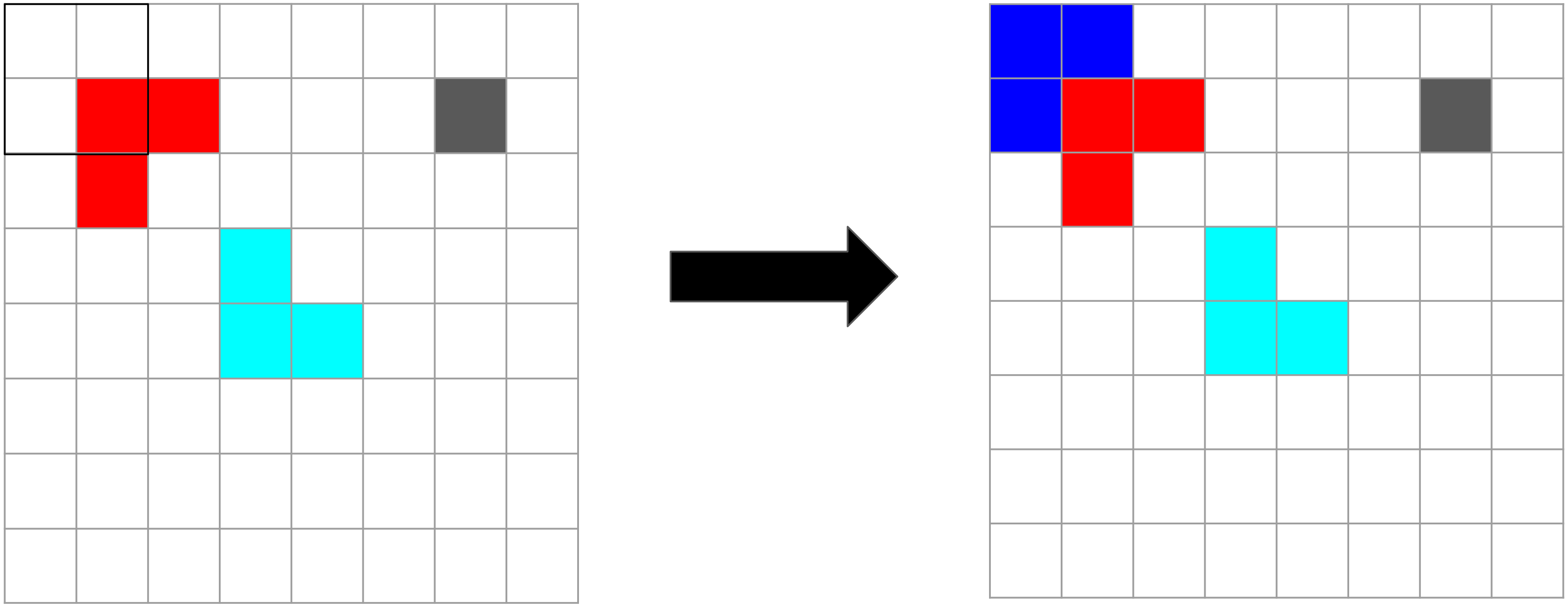
Example — L-pieces

- Idea:
- Divide the grid into 4 regions, each are N/2 * N/2
- Put a L piece right next to the corner of the region that contain an filled cell
- Solve recursively for each region
- Base case: 2*2 grid, just simply put a L piece

- The gray cell is the given cell
- The cyan L-piece is the one we are putting



Example — L-pieces



Example — L-pieces

- At each step, we divide our problem F(n) into 4 sub-problem F(n/2)
- Combining the sub-problem cost O(1) as we don't actually have to combine
- $T(n) = 4T(n/2) + O(1)$
- $a = 4$, $b = 2$, $d = 0$, $\log_b a = 2$
  - If $d < \log_b a$, $T(n) = O(n^{\log_b a})$
- $T(n) = O(n^{\log_2 4}) = O(n^2)$

Example — L-pieces

- Exact same problem on HKOJ
- 01003 L-pieces

Example — Merge sort & no. of inversions

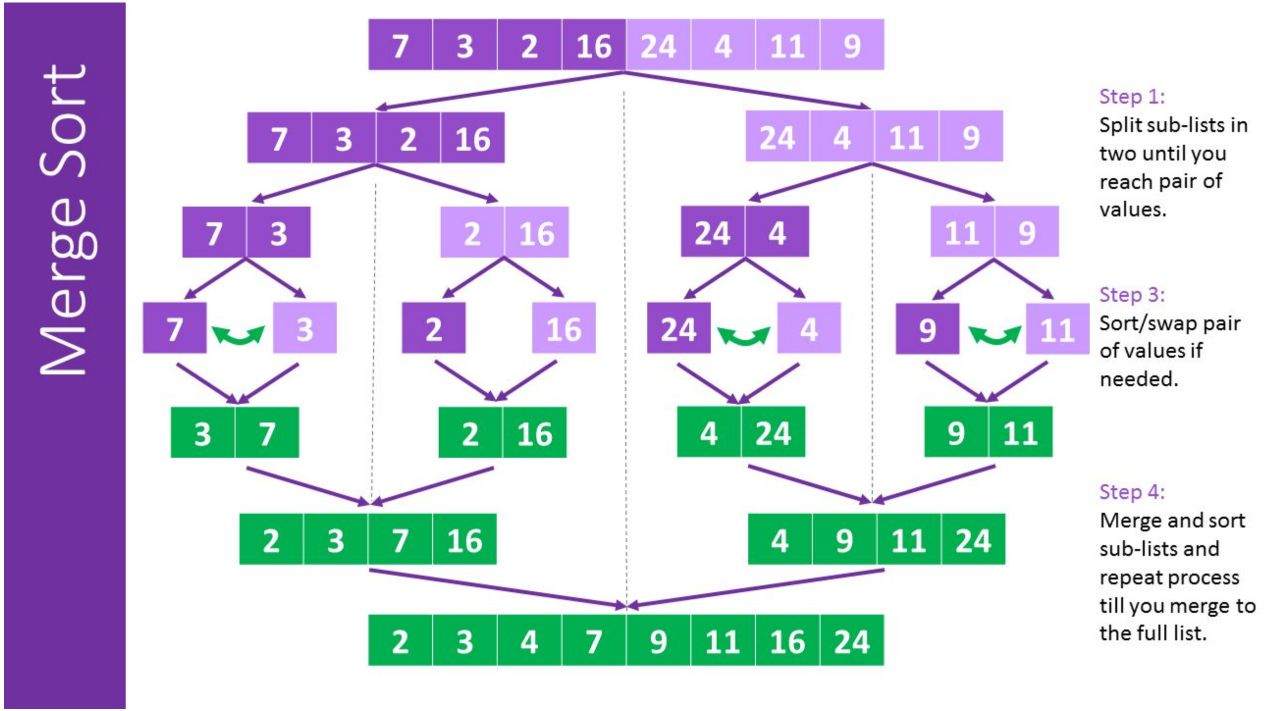
- Merge sort is a well known sorting algorithm that sort an array of length n in O(nlogn) time
- This can be extended to also count no. of inversions which will be discussed later

Example — Merge sort & no. of inversions

- Given the array A of length n
- Steps:
  - Divide A into two smaller array of length n/2
  - Sort each of them recursively with merge sort
  - Combine the two sorted array into one array
- Base case:
  - When n = 1, no sorting is needed.
  - When n = 2, easy comparison with an if statement

Example — Merge sort & no. of inversions

Source:
https://www.101computing.net/merge-sort-algorithm/



Example — Merge sort & no. of inversions

- How can we merge the two sorted array?

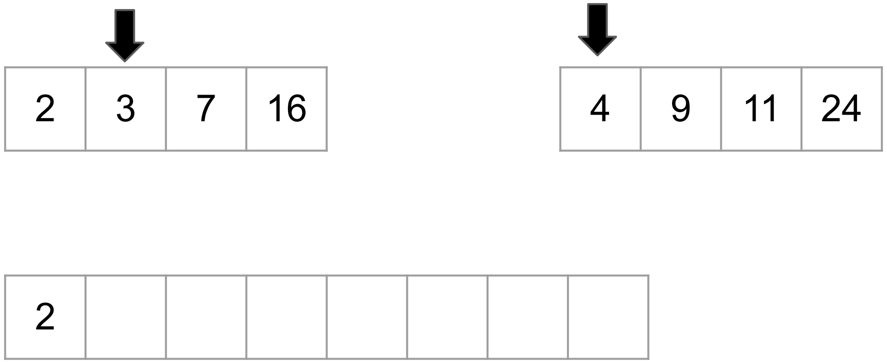
- Let i be the pointer for the first array A_L
- And j be the pointer for the second array A_R
- Compare $A_L[i]$ and $A_R[j]$
- If $A_L[i] <= A_R[j]$, put $A_L[i]$ into the back of the combined array, increment i
- Else put $A_R[j]$ into the back of the combined array, increment j
- Until either we exhaust all element in A_L or A_R, put all the remaining elements in the other array to the combined array directly

Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

Example — Merge sort & no. of inversions



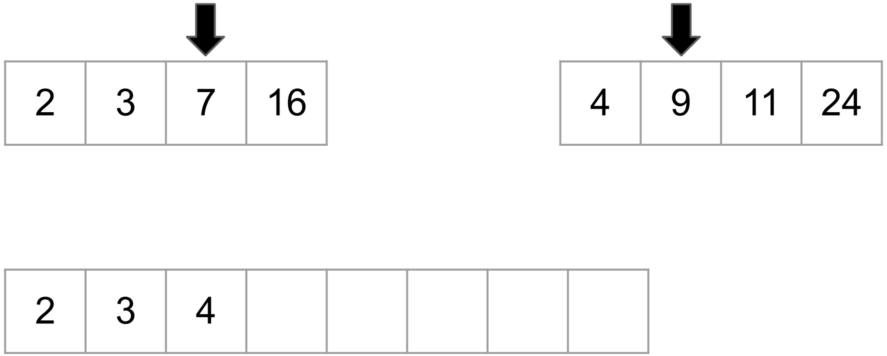
Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

2	3						

Example — Merge sort & no. of inversions



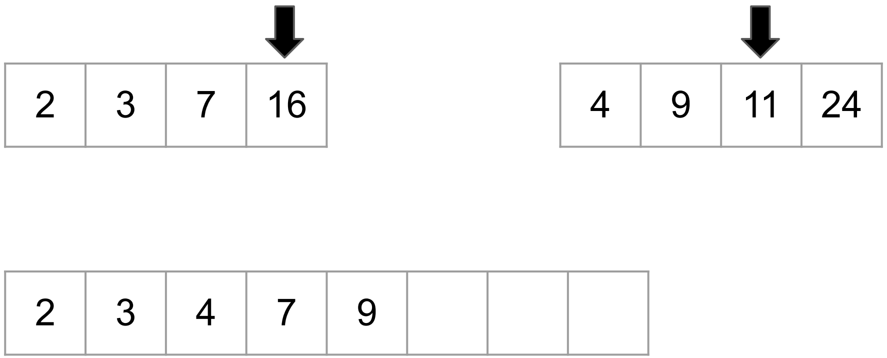
Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

2	3	4	7				

Example — Merge sort & no. of inversions



Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

2	3	4	7	9	11		

Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

2	3	4	7	9	11	16	

Example — Merge sort & no. of inversions

2	3	7	16

4	9	11	24

2	3	4	7	9	11	16	24

Example — Merge sort & no. of inversions

- At each step we divide each problem F(n) into 2 sub-problems F(n/2)
- Combining the solutions takes O(n)
- $T(n) = 2T(n/2) + O(n)$
- $a = 2, b = 2, d = 1, \log_b a = 1$
  - If $d = \log_b a$, $T(n) = O(n^d \log n)$
- $T(n) = O(n^d \log n) = O(n \log n)$

Example — Merge sort & no. of inversions

Code:

a is the original array

r is the temporary array

s is the starting position

t is the ending position

```
int a[200005], r[200005];
void mergesort(int s, int t) {
  if (s == t)
    return;
  int mid = (s + t) / 2;
  mergesort(s, mid);
  mergesort(mid + 1, t);
  int i = s, j = mid + 1, k = s;
  while (i <= mid && j <= t) {
    if (a[i] <= a[j])
      r[k++] = a[i++];
    else
      r[k++] = a[j++];
  }
  while (i <= mid)
    r[k++] = a[i++];
  while (j <= t)
    r[k++] = a[j++];
  for (int i = s; i <= t; i++)
    a[i] = r[i];
}
```

Example — Merge sort & no. of inversions

- Number of inversion of an array of size n is defined as:
- The number of pair (i, j) such that $1 <= i < j <= n$ and $a[i] > a[j]$
- e.g. A = [1, 8, 6, 5]
- The inversions are (2, 3), (2, 4) and (3, 4)
- Note that they are index

Example — Merge sort & no. of inversions

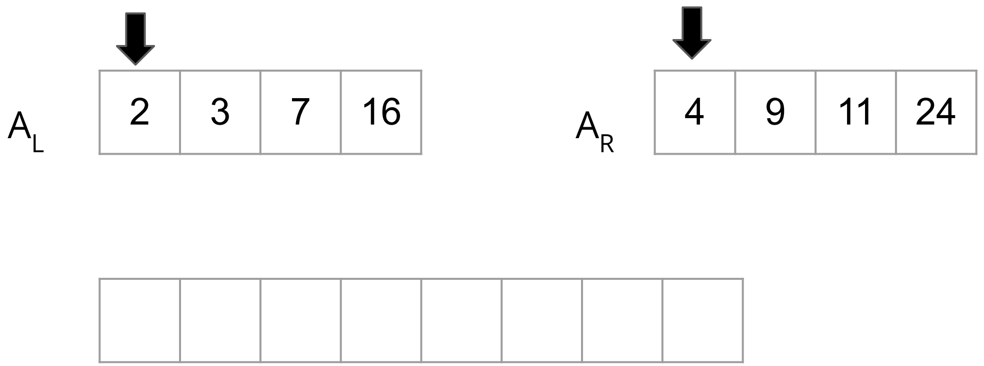
- We can count this number naively using nested for-loop
- Time complexity: $O(n^2)$
- Too slow
- Instead we can modify our merge sort so count this number much faster

Example — Merge sort & no. of inversions

Note that element in A_L is always on the left of A_R.

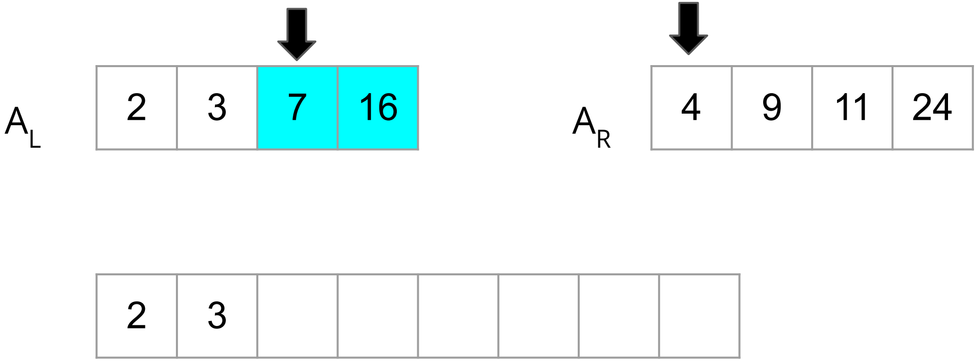
If at some point $A_L[i] > A_R[j]$, there is an inversion.

Not only one inversion, but actually all elements after $A_L[i]$ including itself are inversions with $A_R[j]$



Example — Merge sort & no. of inversions

Here we encounter the first inversion, there are two elements after.



\# of Inversion: 2

Example — Merge sort & no. of inversions

Here we encounter the second inversion, there are only one element after.

A_L

2	3	7	16

A_R

4	9	11	24

# of Inversion: 3

2	3	4	7				

Example — Merge sort & no. of inversions

- During the merging process we can count no. of inversions directly in O(1)
- Doing a merge sort takes O(nlogn)
- So counting no. of inversions also takes O(nlogn)

Example — Merge sort & no. of inversions

Code:

invcnt = # of inversions

Beware that maximum # of inversions = n * (n + 1) / 2

Use **long long** if needed!

```
ll invcnt = 0;
void mergesort(int s, int t) {
  if (s == t)
    return;
  int mid = (s + t) / 2;
  mergesort(s, mid);
  mergesort(mid + 1, t);
  int i = s, j = mid + 1, k = s;
  while (i <= mid && j <= t) {
    if (a[i] <= a[j])
      r[k++] = a[i++];
    else
      invcnt += 1LL * mid - i + 1; r[k++] = a[j++];
  }
  while (i <= mid)
    r[k++] = a[i++];
  while (j <= t)
    r[k++] = a[j]++;
  for (int i = s; i <= t; i++)
    a[i] = r[i];
}
```

Suggested Tasks

- 01046 One-Step Tower of Hanoi
- 01014 Stamps
- 01031 Permutations
- 01035 Combinations
- 20296 Safecrackers
- 30098 Generating Fast Permutations
- 01049 Chocolate
- 01050 Bin packing
- 20750 8 Queens Chess Problem
- T183 Exam Anti Cheat (You can get at least 50 marks by cutting branch)

Reference

https://assets.hkoi.org/training2022/rdc.pdf