Recursion, Divide & Conquer

Jason

Table of contents

Function

- Function in Math:
- e.g. $f(x) = x^2 + 2x + 1$, $g(x)=\sin x + x$, …
- Process the input x and return the computed value

- Function in Programming:
- Similar to that in Math
- A subroutine that processes the input (or parameters) and return some outputs

Function in OI

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

int main(){
    int ans = f(10);
    cout << ans;
    return 0;
}
```

Define the function name and input

Process the given input

Output the required value

Call the function with x = 10

Function in OI

Functions can have several parameters:

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

Procedure

- Similar to functions
- A subroutine that processes the input (or parameters) but does not return any outputs

Procedure

```
void hello(int n){
    for (int i = 1; i <= n; i++)
        cout << “Hello world!\n”;
}

int main(){
    hello(10);
    return 0;
}
```

Define the procedure name and input

Process the given input

Call the procedure with n = 10

Recursion

- A function / procedure can be called inside itself

```
int factorial(int x){
	int y = factorial(x – 1) * x;
	return y;
}

void helloWorlds(int n){
	cout << “Hello world “ << n << endl;
	helloWorlds(n – 1);
}
```

Recursion

- Consider the following code:

```
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x – 1) * x;
    return y;
}

int main(){
    cout << fact(3);
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
```

```
    if (x == 1) return 1;
    int y = fact(x – 1) * x;
    return y;
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x – 1) * x;
    return y;
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 2, y = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = 1 * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

```
(x = 2, y = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = 1 * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3, y = 6)
int fact(int x){
    if (x == 1) return 1;
    int y = 2 * x;
    return y;
}
```

```
(x = 2, y = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = 1 * x;
    return y;
}
```

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

Recursion

```
int main(){
    cout << fact(3);
}
```

```
(x = 3, y = 6)
int fact(int x){
    if (x == 1) return 1;
    int y = 2 * x;
    return y;
}
```

```
(x = 2, y = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = 1 * x;
    return y;
}
```

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Each function call has their own local variables (x and y)
They are only used in their own subroutine

Recursion

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

```
(x = 3, y = 6)
int fact(int x){
    if (x == 1) return 1;
    int y = 2 * x;
    return y;
}
```

```
(x = 2, y = 2)
int fact(int x){
    if (x == 1) return 1;
    int y = 1 * x;
    return y;
}
```

```
(x = 1)
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x - 1) * x;
    return y;
}
```

Recursion

- Using recursion, some computations can be done with simpler code

- More importantly, recursion help us to solve problems with the following properties:
  - The problem can be divided into same problem with smaller parameter
  - We need information of the sub-problem(s) to solve the current one

Recursion

```
int fact(int x){
    if (x == 1) return 1;
    int y = fact(x – 1) * x;
    return y;
}
```

Base case: to stop the recursion

Recurrent relation: to further divide the problem into smaller one

Example – Fibonacci Numbers

- Find the n-th Fibonacci Number

- Base case: When n = 1 or 2, f(n) is 1
- Recurrence relation: Otherwise f(n) = f(n – 1) + f(n – 2)

Example – Fibonacci Numbers

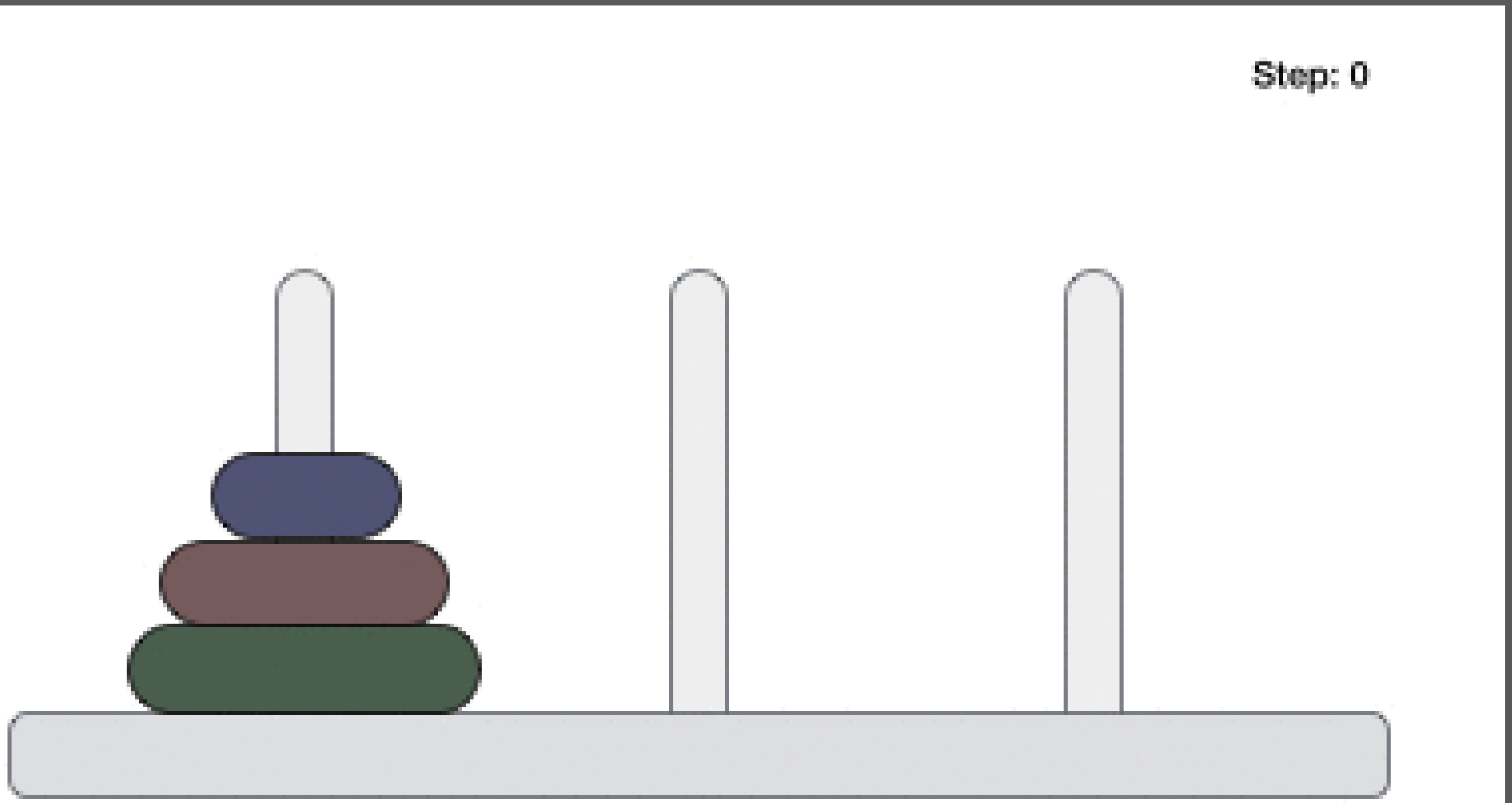
- Find the n-th Fibonacci Number

- Base case: When n = 1 or 2, f(n) is 1
- Recurrence relation: Otherwise f(n) = f(n – 1) + f(n – 2)

```
int f(int x){
    if (x == 1 || x == 2) return 1;
    else return f(x - 1) + f(x - 2);
}
```

Example – Tower of Hanoi

- There are three pegs numbered 0, 1 and 2.
- There are N disks. The number N is constant while working the puzzle.
- All disks are different in size.
- The disks are initially stacked on peg 0 so that they increase in size from the top to the bottom.
- The goal of the puzzle is to transfer the entire tower from the 0 peg to one of the others pegs.
- One disk at a time can be moved from the top of a stack either to an empty peg or to a peg with a larger disk than itself on the top of its stack.


Step: 0

Example – Tower of Hanoi

- Define the problem as `P(n, start, end, intermediate)`, meaning that the disk 1 to n are moved from the `start` peg to the `end` peg, using the `intermediate` peg for transition
  - No need to care about the positions of disk n + 1, n + 2, … for now
- Base case: when n = 1, simply move the disk 1 from `start` to `end`
- Recurrence Relation: when n > 1,
  - Move disk 1 to n – 1 from `start` to `intermediate`, using `end` for transition
  - Move disk n from `start` to `end`
  - Move disk 1 to n – 1 from `intermediate` to `end`, using `start` for transition

Example – Tower of Hanoi

```
int P(int n, int start, int end, int inter){
    if (n == 1){
        [Move peg n from start to end]
    }else{
        P(n – 1, start, inter, end);
        [Move peg n from start to end]
        P(n – 1, inter, end, start);
    }
}
```

Exhaustion

- Sometimes we don’t know a fast algorithm to solve the problem
- We can enumerate all possible solutions
- Check whether each of them satisfies the problem’s statement
- Find the best one among them

Example 0

- Given a list of integers and an integer M
- Choose a pair of integers such that the sum of them is equal to M
- E.g. A = {1,2,4,8,16}, M = 10 -> Choose 2 and 8

Example 0

- The number of possible solutions is small
  - Number of pairs = N(N-1)/2, where N is number of integers
- We can try to form all pairs and check whether they satisfy the requirement

```
For i = 1 to N-1
        For j = i+1 to N
                if A[i] + A[j] = M then
                        return {A[i], A[j]}
```

Example 1 - Subset

- Given a list of positive integers and an integer M
- Find a subset such that the sum of the integers is equal to M
- E.g. A = {2,9,15,16}, M = 27 -> Output {2, 9, 16}

Example 1 - Subset

- Number of possible solutions = 2^N
- {}
- {2}, {9}, {15}, {16}
- {2, 9}, {2, 15}, {2, 16}, {9, 15}, {9, 16}, {15, 16}
- {2, 9, 15}, {2, 9, 16}, {2, 15, 16}, {9, 15, 16}
- {2, 9, 15, 16}
- When N is small, we can just check all of them

Example 1 - Subset

- Method in example 1 can not solve the problem

```
if (n == 1){
    for (int choose_1 = 0; choose_1 <= 1; choose_1++)
        if (a[1] * choose_1 == m) return 1;
} else {
    for (int choose_1 = 0; choose_1 <= 1; choose_1++)
        for (int choose_2 = 0; choose_2 <= 1; choose_2++)
            if (a[1] * choose_1 + a[2] * choose_2 == m) return 1;
} else {
    for (int choose_1 = 0; choose_1 <= 1; choose_1++)
        for (int choose_2 = 0; choose_2 <= 1; choose_2++)
            for (int choose_3 = 0; choose_3 <= 1; choose_3++)
                if (a[1] * choose_1 + a[2] * choose_2 + a[3] * choose_3 == m) return 1;
} else …
```

Example 1 - Subset

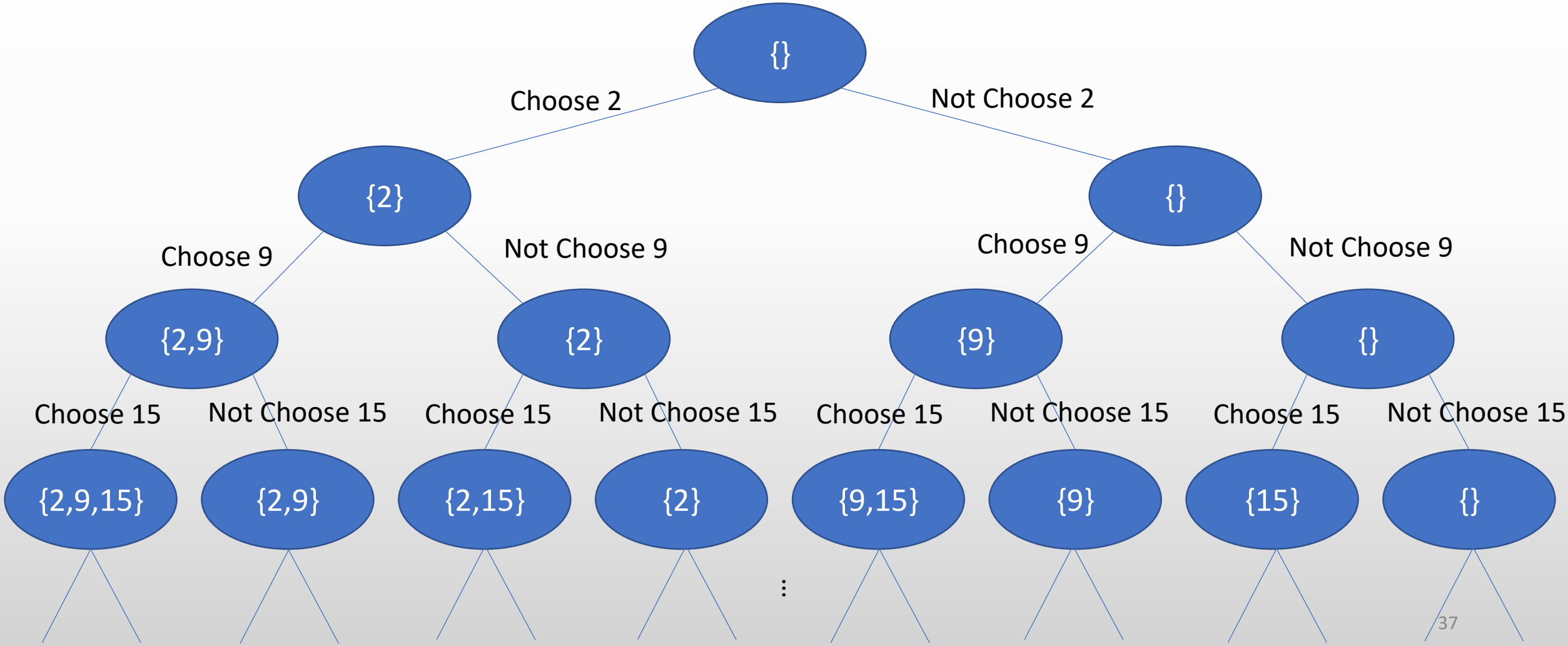
- 1 for loop to choose 1 integer, 2 for loops to choose 2 integers, 3 for loops to choose 3 integers, …
- Each integer requires its own for loop for the 2 branches
- Recursion provides a better way to generate all possible subsets

Example 1 - Subset

- Procedure Exhaustion(x, *current*) generates all subsets from a[x .. n] together with the set *current*

- Base case: When x > n, the only result is the set *current*
- Recurrence relation: Exhaustion(x, *current*) can lead to two cases:
  - Pick a[x] into the subset -> Exhaustion(x + 1, *current* + {a[x]})
  - Not pick a[x] into the subset -> Exhaustion(x + 1, *current*)

Example 1 - Subset



Example 1 – Pseudocode

```
Procedure exhaustion(integer x, set current)
    If x <= N then
        exhaustion(x + 1, current + {a[x]})
        exhaustion(x + 1, current)
    Else
        if sum of the integers in the set current = M then
                output the set current
```

Example 1 – Code

In reality, we can store the chosen subset outside the procedure (as global variables) instead of passing as parameters

```
void exhaustion(int x, int sum){          // we can store the sum instead
     if (x <= N){
       chosen[x] = true;
       exhaustion(x + 1, sum + a[x]);
       chosen[x] = false;
       exhaustion(x + 1, sum);
     } else {
       if (sum == M)
              [Output the chosen numbers]
     }
}
```

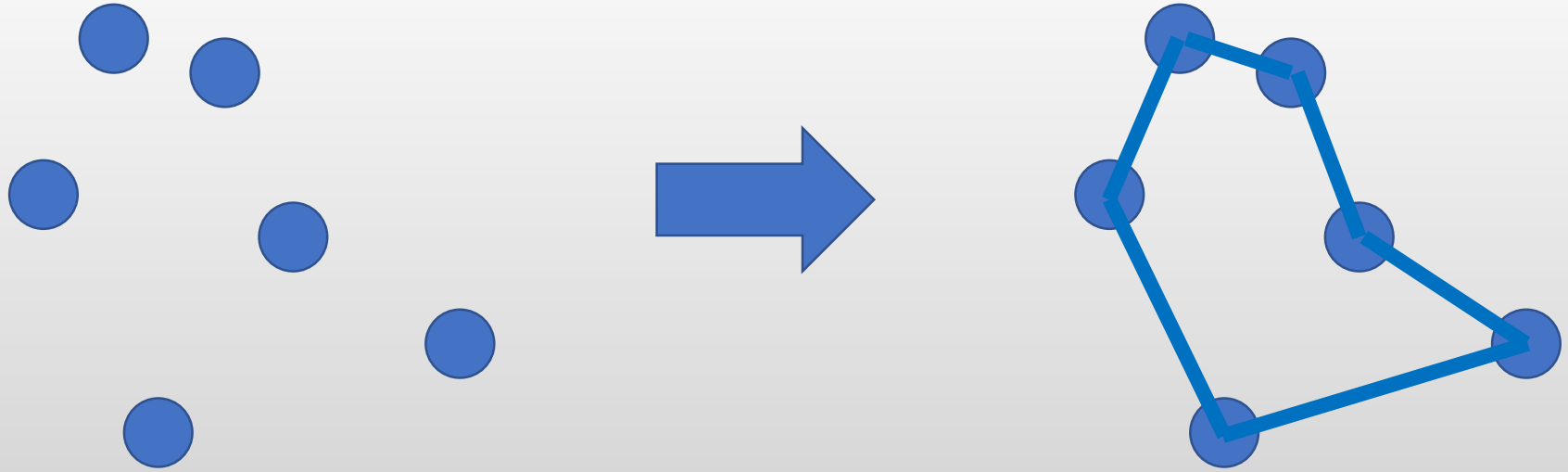
Example 1

- Time complexity: $O(2^N)$

- There exists other more efficient solutions
- Exhaustion is enough to solve the problem if N is small (e.g. N <= 20)

Example 2 - Permutation

- Given the coordinates of N points
- Find out the shortest possible route that visits each point exactly once and returns to the origin point
- Example:



Example 2 - Permutation

- The famous “Travelling salesman problem”
- Proven to be an NP-hard problem
- Cannot be solved in polynomial time (for now)

- The number of possible routes is N!
- Every permutation of the points gives a possible route
- Enumerate all of them to find the shortest one

Example 2 - Permutation

- Procedure Exhaustion(x, *current*) generates all permutation starting with the permutation *current* (which has the size of x – 1)

- Base case: When x > n, the permutation *current* contains all integers already
- Recurrence relation: In Exhaustion(x, *current*), we can pick any point that is not in *current* as the x-th point of the permutation
  - Calls Exhaustion(x + 1, current + [i]) for all i that i is not chosen yet

Example 2 – Pseudocode 1

```
Procedure exhaustion(integer x, list current)
    If x <= N then
        for i from 1 to N
                if the i-th point is not in current
                        exhaustion(x + 1, current + [i])
    Else
        Calculate the length of the route
        Check if it is the shortest one
```

Time complexity: O(N*N!)

Example 2 – Code

Similarly, we can store the permutation as global variables

```
void exhaustion(int x){
	if (x <= N){
		for (int i = 1; i <= N; i++)
			if (!chosen[i]){
				chosen[i] = true; per[x] = i; // add i to the list
				exhaustion(x + 1);
				chosen[i] = false; per[x] = 0; // revert the changes
			}
	}else{
		[Calculate and compare the lengths using the array per]
	}
```

Example 2 – Pseudocode 2

The procedure has time complexity of O(N), which can be optimized

Let P[i] denotes the i-th point

```
Procedure exhaustion(int x, double total)
    If x <= N then
        for i from x to N
                swap P[x] and P[i]
                exhaustion(x + 1, total + distance between P[x] and P[x-1])
    Else
        total = total + distance between P[N] and P[1]
        Check if it is the shortest one
```

Time complexity: O(N!)

Example 3 – Distinct Permutations

- HKOJ 01031
- Similar to the previous problem
- Given a string of alphabets
- Generate DISTINCT permutations

Example 3 – Distinct Permutations

- Methods from example 3 may fail
- If the input is ACBA, 2 AABC may be generated
- Because 2 ‘A’s are treated as different elements

Example 3 – Distinct Permutations

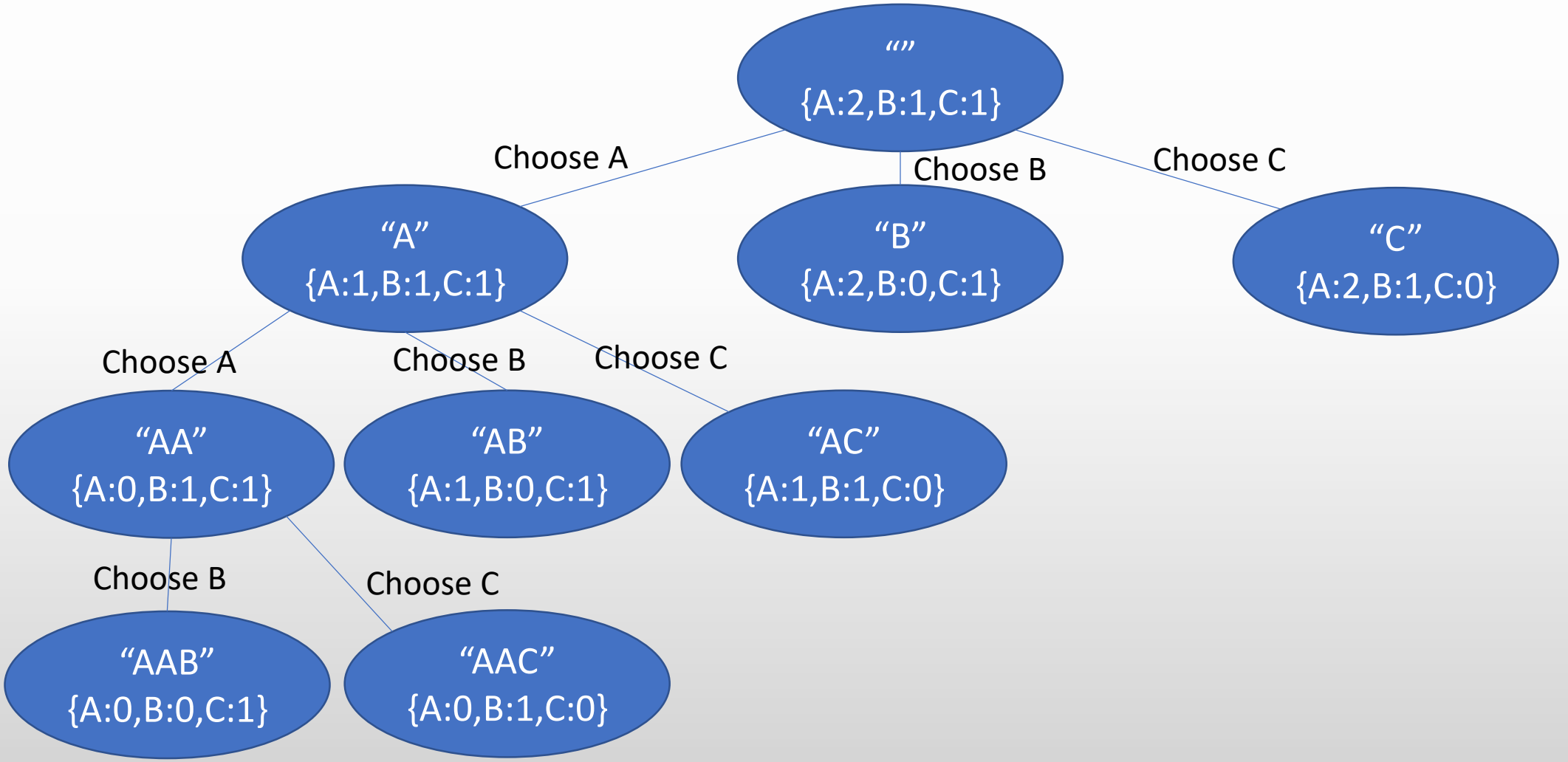
- Solution 1:
- After generate the permutations
- Delete the repeated ones by sorting (or using set/map in c++)

- Okay for this problem, but more time and memory is wasted

Example 3 – Distinct Permutations

- Solution 2:
- Count the frequency of each alphabet
- During the recursion, choose an alphabet that still has quotas

Example 3 – Distinct Permutations



Example 3 – Pseudocode

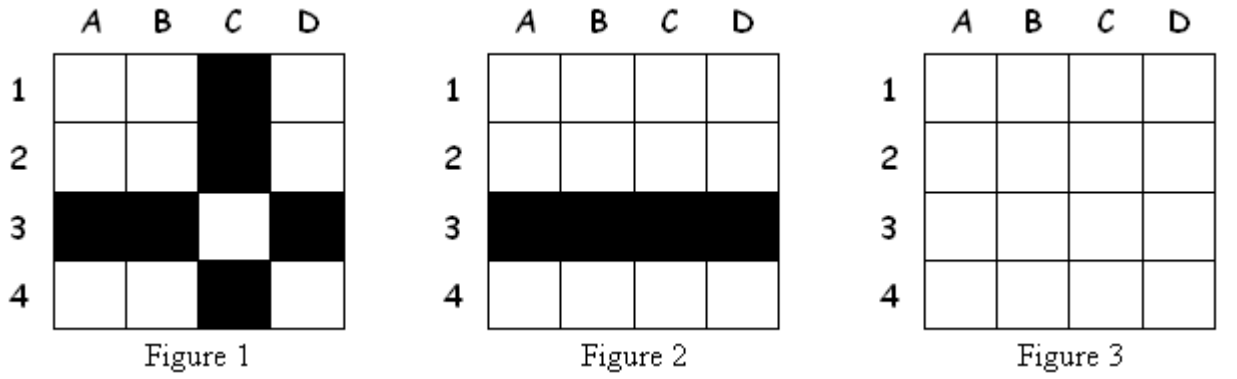
```
Let s[1..K] be the output string
Procedure exhaustion(int x)
    If x <= K then
        for i from ‘A’ to ‘Z’
                if frequency[i] > 0 then
                        frequency[i] = frequency[i] – 1
                        s[x] = i
                        exhaustion(x + 1)
                        frequency[i] = frequency[i] + 1
    Else
        print s
```

Exhaustion

- Sometimes the way we do exhaustion may affect the efficiency
- A part of the answer may determine the rest
- Smaller part is unknown -> less states to be searched

Example 4 – Toggle

- A board with N×N cells
- Each cell is colored either black or white initially
- In each move, you can either 'toggle' a row or a column so that the color of the cells on the entire row/column changes. If the color of a cell is black, it becomes white after it is toggled, and vice versa.
- Finds a sequence of moves that maximizes the number of white cells.



Example 4 – Toggle

- Observations:
  - The order of row/column to be toggled does not matter
  - Each row/column will be toggled at most once
- Try toggling all combinations of rows and columns
- See which one gives the most number of white cells

Example 4 – Toggle

- Number of combinations = $2^{2N} = 4^N$
- Time complexity : $O(N^2 * 4^N)$

- Works when N <= 4
- Too slow when N = 16

Example 4 – Toggle

- If the combination of rows to be toggled is fixed, we don’t need to try toggling all combinations of columns
- If a column has more white cells than black cells, we should not toggle it
- If a column has more black cells than white cells, we must toggle it

Example 4 – Toggle

- Try toggling all combinations of rows
- For each column, toggle it only if it has more black cells than white cells
- Check whether it has the most number of white cells

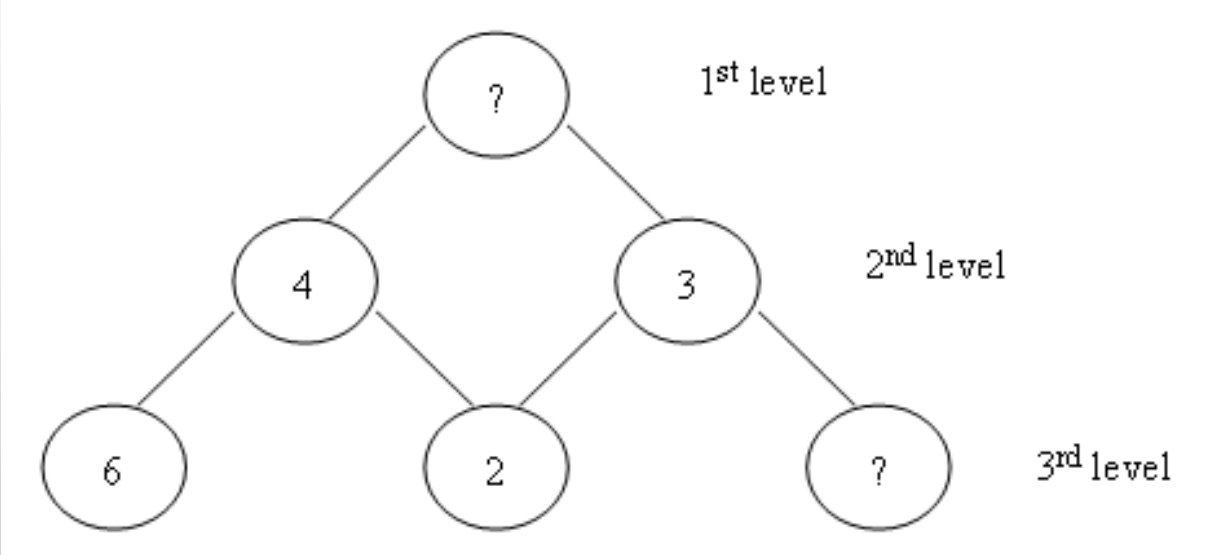
- Number of combinations = 2^N
- Time complexity : $O(N^2*2^N)$

Example 4 - Code

```
void exhaustion(int x){
	if (x <= N){
		flip_row[x] = true;
		exhaustion(x + 1);
		flip_row[x] = false;
		exhaustion(x + 1);
	}else{
		[Flip the rows according to flip_row]
		for (int i = 1; i <= n; i++)
			[Flip the column i if there are more black there]
		[Compare with the current answer]
	}
```

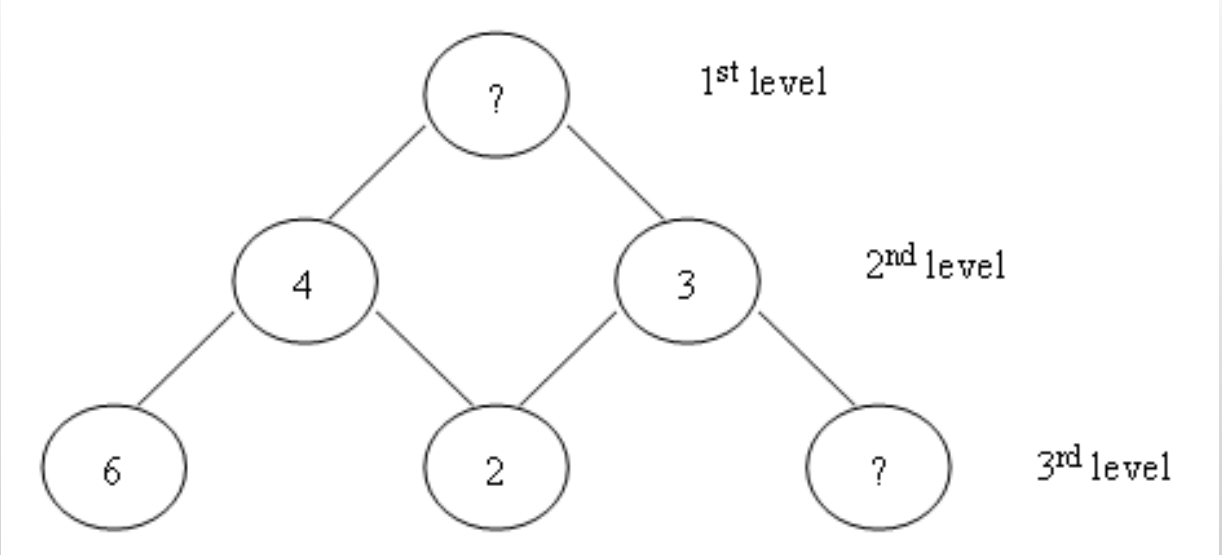
Example 5 – Magic Triangle

- There is a triangle with N levels (N <= 5)
- The i-th level has i nodes
- Each node except those in the Nth level has two children



Example 5 – Magic Triangle

- Assign 1 to N*(N+1)/2 into the nodes
- The value of some nodes are predetermined
- Output an arrangement such that the value of each node = The absolute difference of the value of its 2 children



Example 5 – Magic Triangle

- Solution 1 :
- Try all permutations of numbers
- Check whether they fulfill the condition
- Number of permutations = (N*(N+1)/2)!
- Time complexity : O((N*(N+1)/2)!)

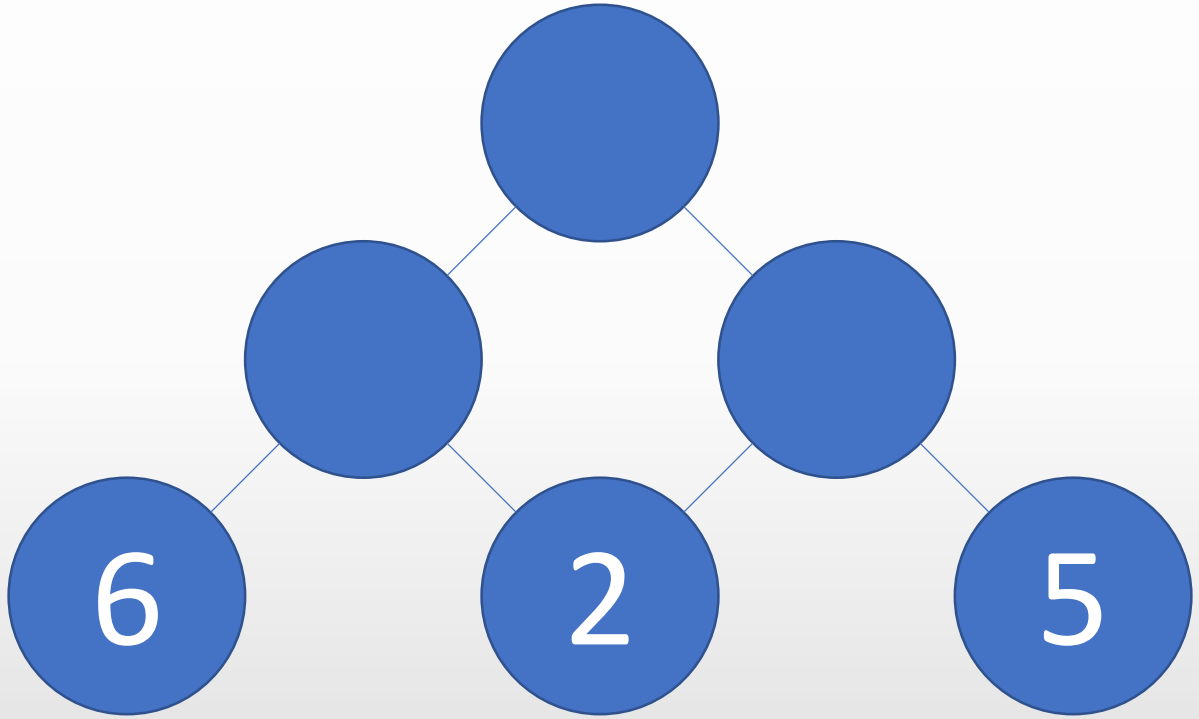
Example 5 – Magic Triangle

- When N = 5, number of permutations = 1307674368000
- Constant optimization is not enough
- Even by immediately discarding the permutations that does not match with the predetermined numbers, there are still too many
- A faster solution is needed

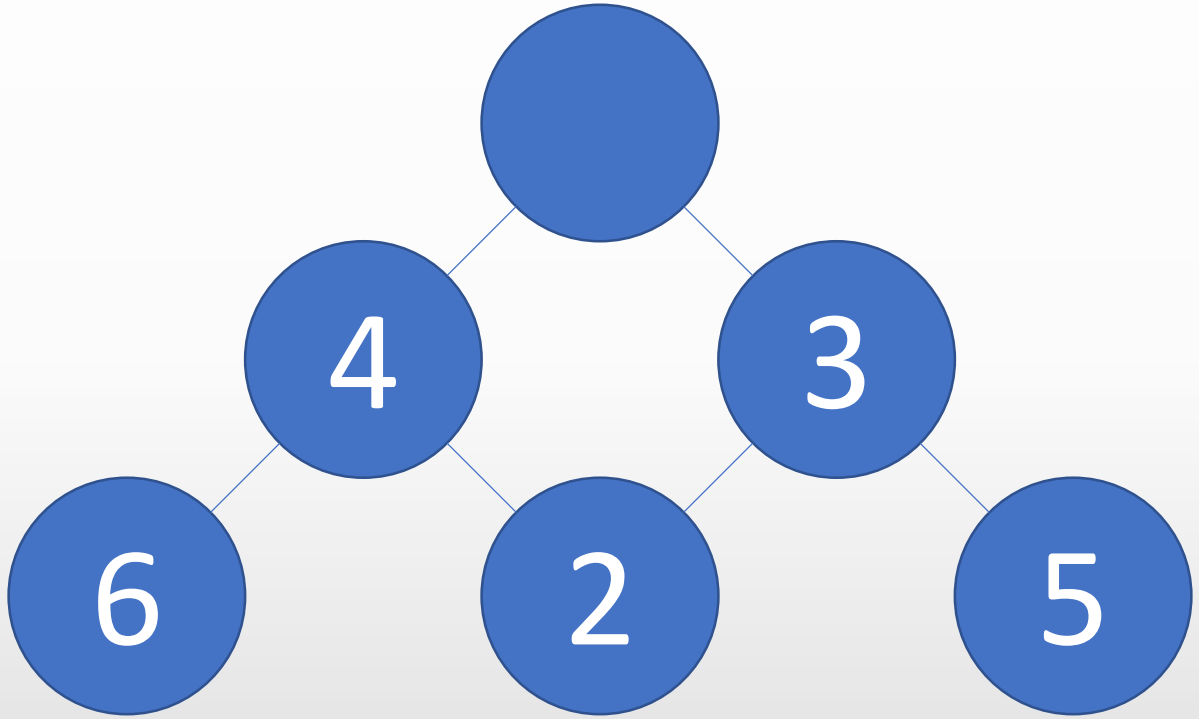
Example 5 – Magic Triangle

- Consider the values in the Nth level
- If these values are fixed, the values in the (N-1)th level can be calculated
- Use values in the (N-1)th level to calculate those in the (N-2)th level
- …
- …
- Use values in the 2nd level to calculate the value in the 1st level
- The whole arrangement is fixed!

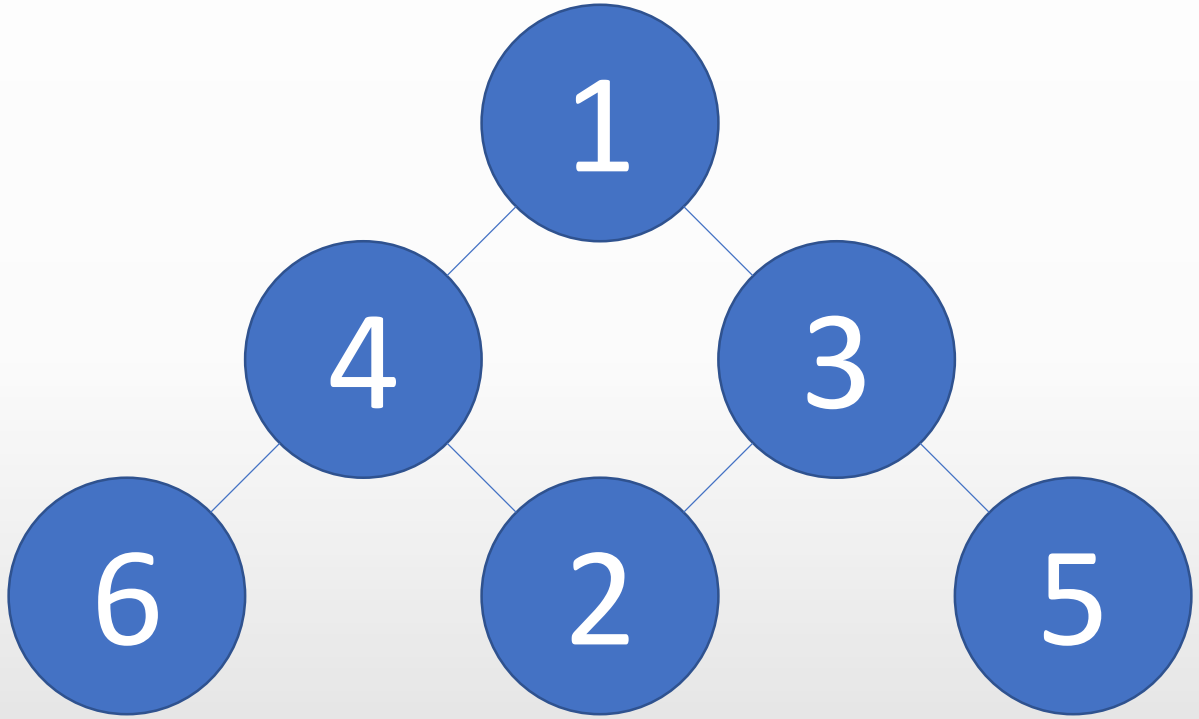
Example 5 – Magic Triangle



Example 5 – Magic Triangle



Example 5 – Magic Triangle



Example 5 – Magic Triangle

- Solution 2
- Exhaust all permutations of the Nth level
- Calculate the values of the whole triangle from the bottom to top
- Verify if the numbers are distinct

- Number of permutations = (N*(N+1)/2)PN
- When N is 5, the number of permutations = 360360
- Fast enough to solve the problem

Practice problem

- 01046 One-Step Tower of Hanoi
- 01014 Stamps
- 01031 Permutations
- 01035 Combinations
- 20296 Safecracker

Branch & Bound

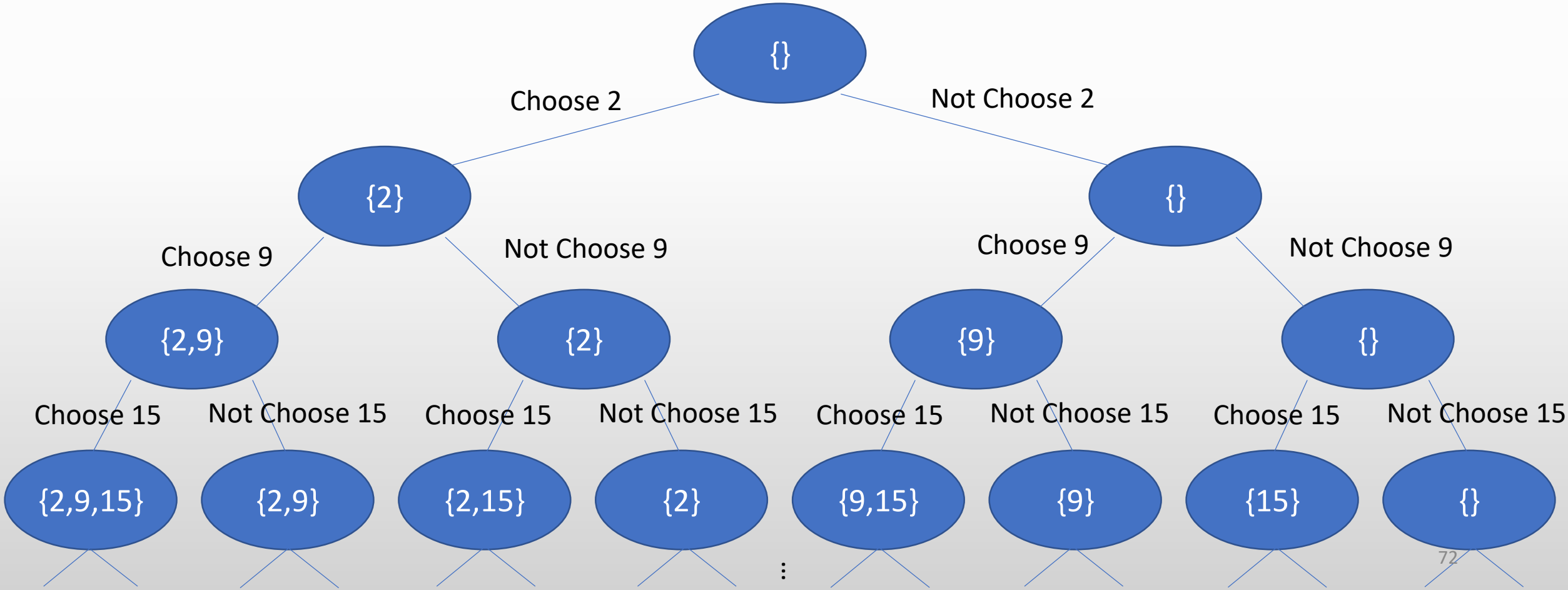
- During the searching, some invalid states may be visited
- When the intermediate state is known to be invalid, stop branching from it
- When the intermediate state gives an unsatisfied answer, stop branching from it

Example 1 – again

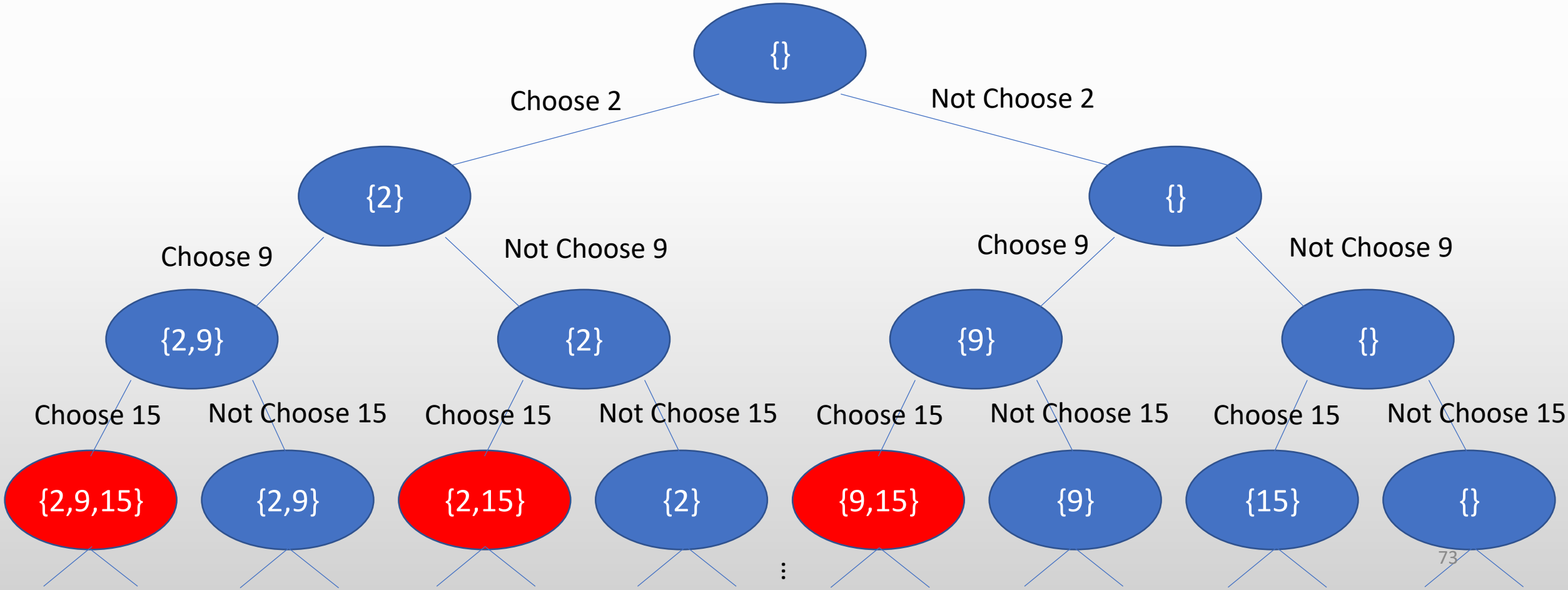
- Let’s go back to example 1
- Now A = {2, 9, 10, 3, 7, 2, 5, …} and M = 16
- Use the same technique again

Example 1 – again

- Some of the states are useless!

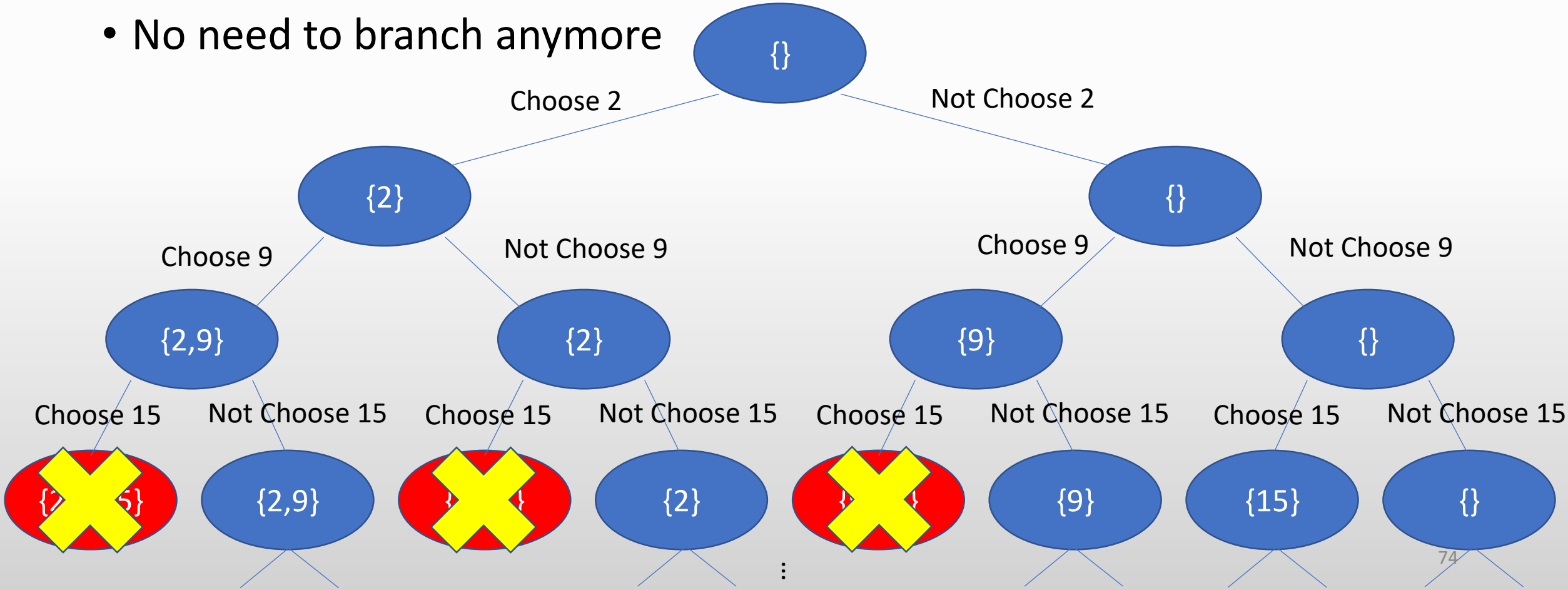


Example 1 – again



Example 1 – again

- Sum of the chosen numbers > 16
- No need to branch anymore



Example 1 – Code 2

```
void exhaustion(int x, int sum){
	if (sum > M) return;
	if (x <= N){
		choose[x] = true;
		exhaustion(x + 1, sum + A[x]);
		choose[x] = false;
		exhaustion(x + 1, sum);
	}else{
		if (sum == M)
			[Output the chosen numbers]
	}
}
```

Example 1 – Code 3

```
void exhaustion(int x, int sum){
	if (sum > M) return;
	if (solution_found) return;
	if (x <= N){
		choose[x] = true;
		exhaustion(x + 1, sum + A[x]);
		choose[x] = false;
		exhaustion(x + 1, sum);
	}else{
		if (sum == M){
			[Output the chosen numbers]
			solution_found = true;
		}
	}
}
```

Example 1 – again

- Constant optimization
- The time complexity is not affected much – still $O(2^N)$
- But much less operations are done
- Search meaningful states only

Example 6 - Chocolate

- HKOJ 01049
- Given the length of the divided pieces, find the minimum possible length of the original bars

Example 6 - Chocolate

- For all possible lengths, try to arrange the pieces to see if they can form complete bars

- Possible solutions:
- Each time we pick an unused piece that can fit into the current bar until a complete bar is formed
- Repeat until all piece is used

Example 6 – Code 1

```
void exhaustion(int x, int current){
	if (x > n){
		solution_found = true;
		return;
	}
	for (int i = 1; i <= n; i++)
		if (!used[i] && current + a[i] <= length){
			used[i] = 1;
			if (current + a[i] == length) exhaustion(x + 1, 0);
			else exhaustion(x + 1, current + a[i]);
			used[i] = 0;
		}
}
```

Example 6 - Chocolate

- The previous solution is not very efficient
- Different orders of pieces are generated
  - ({4, 2}, {3, 3}, …), ({2, 4}, {3, 3}, …) and ({3, 3}, {2, 4}, …) are all generated
- But the order doesn’t matter actually
  - They are all the same

- Solution 2:
- Each time we put the first unused piece into one of the incomplete bar, or start forming a bar by itself

Example 6 – Code 2

```
void exhaustion(int x, int m){                  // m denotes the current number of
       if (x > n){                                 incomplete/complete bars
              [check if all m pieces are complete]
              return;
       }
       for (int i = 1; i <= m; i++)
              if (current[i] + a[x] <= length){
                     current[i] += a[x];
                     exhaustion(x + 1, m);
                     current[i] -= a[x];
              }
       current[m + 1] = a[x];
       exhaustion(x + 1, m + 1);
       current[m + 1] = 0;
}
```

Example 6 - Chocolate

- Still not fast enough?

- If the piece can form a complete bar with an incomplete bar, no capacity will be wasted
- Must be the best arrangement for that piece
- No need to consider putting it into other incomplete bars

Example 6 – Code 3

```
void exhaustion(int x, int m){
        if (x > n){
                [check if all m pieces are complete]
                return;
        }
        for (int i = 1; i <= m; i++)
                if (current[i] + a[x] == length){
                        current[i] += a[x]; exhaustion(x + 1, m); current[i] -= a[x];
                        return;
                }
        for (int i = 1; i <= m; i++)
                if (current[i] + a[x] <= length){
                        current[i] += a[x]; exhaustion(x + 1, m); current[i] -= a[x];
                }
        current[m + 1] = a[x]; exhaustion(x + 1, m + 1); current[m + 1] = 0;
}
```

Example 6 - Chocolate

- Still not fast enough???????

- Sorting the pieces in descending order may be useful, as the smaller pieces will have less choices at last
- Stop branching when too many bars are formed
- …

- Be creative :O)

Practice problems

- HKOJ 01049 Chocolate
- HKOJ 01050 Bin Packing
- HKOJ 20750 8 Queens Chess Problem
- UVA 307 Sticks
- UVA 524 Prime Ring Problem

Divide and Conquer

1. Divide the problem into smaller and independent sub-problems that are same as the original one
2. If the sub-problems are easy to solve, solve them directly; otherwise solve the sub-problems recursively
3. Combine the solutions of the sub-problems to form the solution for the original problem

Example 1 – Big Mod

- Given B, P and M, find B^P % M

- Naïve solution: multiply B for P times

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

or a recursive one using B^P % M = B * B^{P-1} % M

```
int f(int B, int P, int M){
       if (P == 0) return 1 % M;
       else return f(B, P – 1, M) * B % M;
}
```

- Time complexity: O(P)

Example 1 – Big Mod

- Given B, P and M, find B^P % M
- Notice that when P is even, a better way would be

$$B^P \% M = B^{P/2} \% M * B^{P/2} \% M$$

- When P is odd, we can still have B^P % M = B * B^{P-1} % M, or

$$B^P \% M = B^{P/2} \% M * B^{P/2} \% M * B \% M$$

Example 1 – Big Mod

- Base case: when P = 0, f(B, P, M) = 1 % M
- Recurrence relation:
  - If P is even, f(B, P, M) = f(B, P / 2, M) * f(B, P / 2, M) % M
  - If P is odd, f(B, P, M) = f(B, P / 2, M) * f(B, P / 2, M) % M * B % M
- Two calls f(B, P / 2, M) are the same so we can just call once

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

Example 1 – Big Mod

- Each time we reduce P to P / 2 until P become 0
- e.g. When P = 30, we only call f(30), f(15), f(7), f(3), f(1), which is around log(P) times
- So the time complexity is O(log(P))

- By dividing the problem properly, the time complexity may be reduced

Master Theorem

- In fact there are better ways to analyze the time complexity of the divide and conquer algorithms
- Master Theorem: For an algorithm run in time complexity $T(n)$
- If $T(n)$ can be written as a recurrence formula in form of $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$, where $a \geq 1, b > 1$,
- We have $T(n) = \begin{cases} O(n^d) & if\ d > \log_b a \\ O(n^d logn) & if\ d = \log_b a \\ O(n^{\log_b a}) & if\ d < \log_b a \end{cases}$
- $aT\left(\frac{n}{b}\right)$ denotes the time complexity for the divide part: we divide the original problem to a sub-problems, the parameter of each sub-problem reduces from n to $\frac{n}{b}$
- $O(n^d)$ denotes the time complexity for the combine part: we spend $O(n^d)$ time to combine the solutions to the one for the original problem

Example 1 – Big Mod

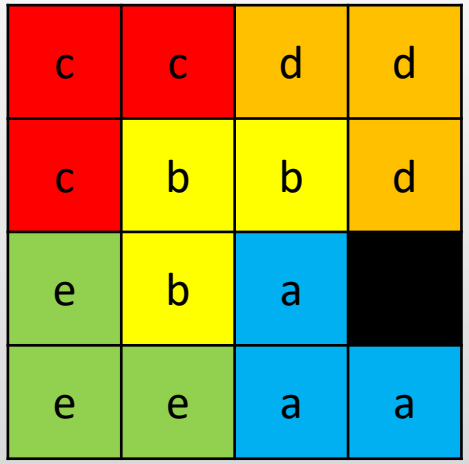
```
int f(int B, int P, int M){
        if (P == 0) return 1 % M;
        int tmp = f(B, P / 2, M);
        if (P % 2 == 0) return tmp * tmp % M;
        else return tmp * tmp % M * B % M;
}
```

- We divide the problem f(P) into ONE sub-problem f(P/2), and we takes O(1) (or $O(n^0)$) time in combining the answer
- Now $a = 1, b = 2, d = 0, T(n) \leq T\left(\frac{n}{2}\right) + O(1)$
- Since $\log_b a = 0 = d$, the time complexity is $O\left(n^d \log n\right) = O(\log n)$

Example 2 – L pieces

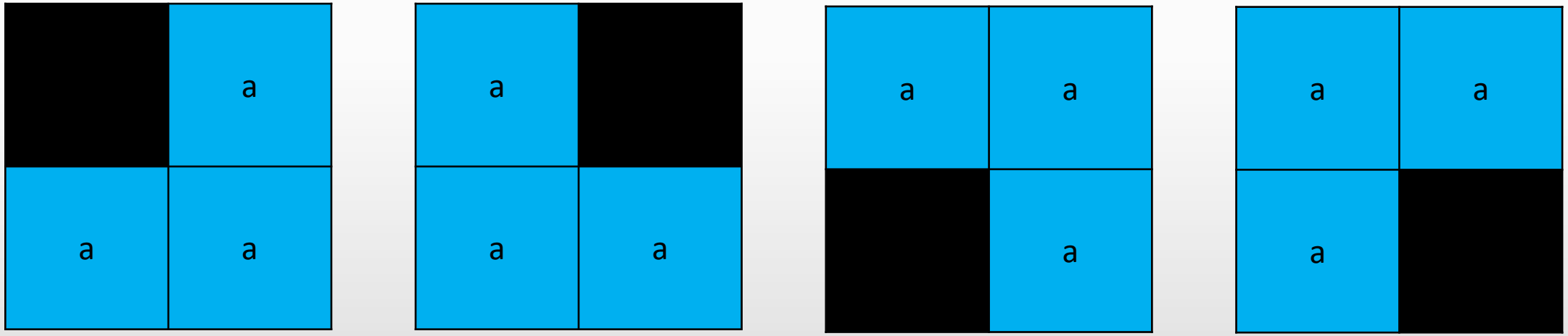
- Given a N * N grid with a empty cell. Output a way to use L-pieces to cover the whole grid where N is a power of 2

- e.g. N = 4, empty cell is (3, 4):



Example 2 – L pieces

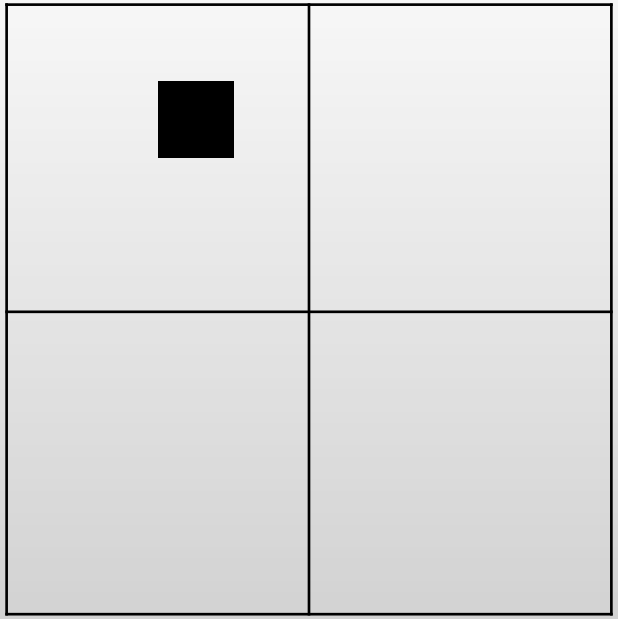
- Base Case: When N = 2, the solution is trivial
- Just put a L-piece on the remaining three cell



- How about the larger ones?

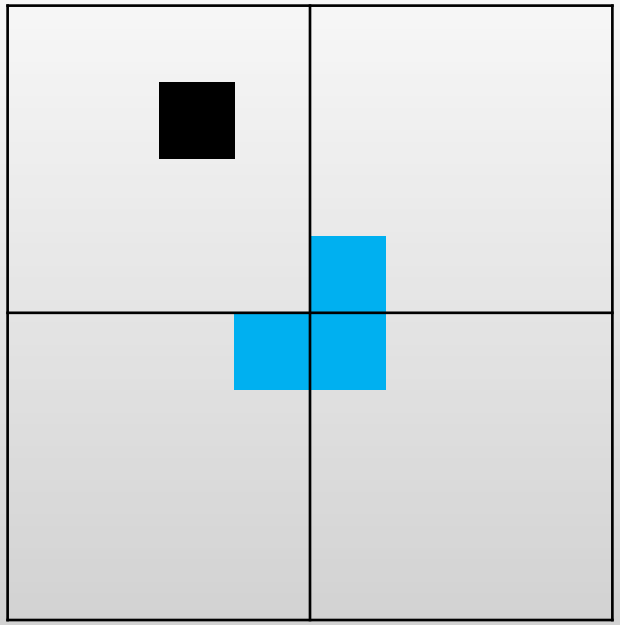
Example 2 – L pieces

- We can divide the N x N grid into 4 (N/2) x (N/2) smaller grids
- The empty cell must be in one of the smaller grids
- The problem is divided, but they are not the same problem
- The other three smaller grids does not have empty cell



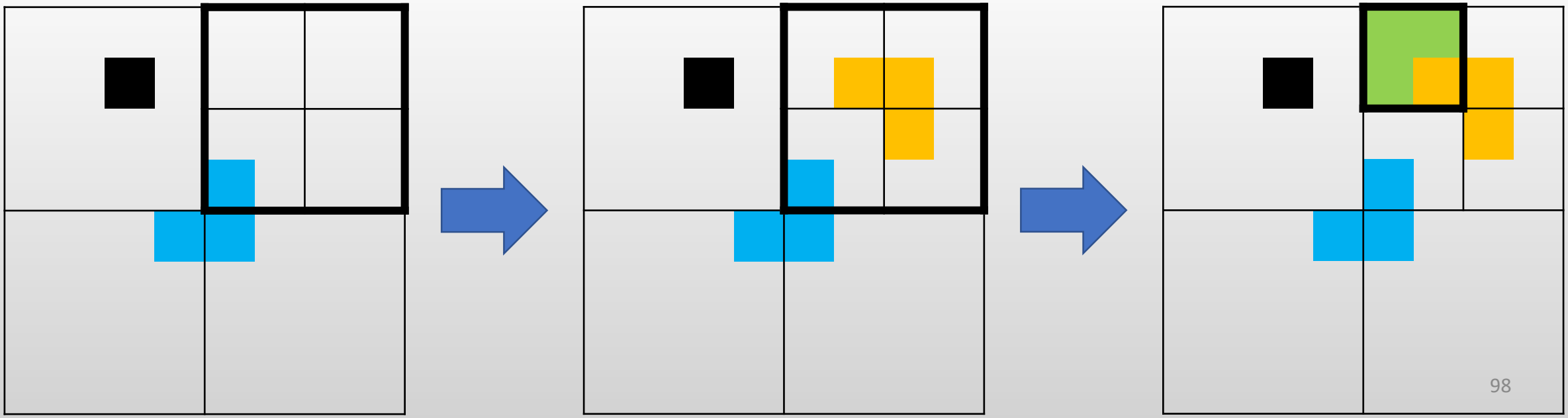
Example 2 – L pieces

- Add one L-piece at the center of the grid that covers the remaining three grids
- Now every smaller grid has one empty cell and becomes a sub-problem that is same as the original one



Example 2 – L pieces

- The same method can be applied on the smaller grid as well
- Solve them recursively until it is divided into a 2 x 2 grid



Example 2 – L pieces

- Solution:
  - Divide the grid to 4 smaller grids: upper-left, upper-right, lower-right, lower-left
  - Put a L-piece in a suitable orientation in the center of the grid such that each smaller grid has exactly 1 empty cell
  - Repeat the above steps recursively on the smaller grids
- Time complexity:
  - $T(n) = 4T\left(\frac{n}{2}\right) + O(1), a = 4, b = 2, d = 0, \log_b a = 2 > d$
  - $T(n) = O\left(n^{\log_2 4}\right) = O(n^2)$

Example 3 – Number of Inversions

- Given a list of distinct integers a[1..n]
- Find the number of pairs (a[i], a[j]) such that i < j and a[i] > a[j]
- e.g. n = 6, a = [1, 2, 6, 3, 5, 4]
- Answer = 4, the pairs are (6, 3), (6, 5), (6, 4) and (5, 4)

Example 3 – Number of Inversions

- We can split the array a into two halves a_l and a_r
- $a = [1, 2, 6, 3, 5, 4]$ -> $a_l = [1, 2, 6]$ and $a_r = [3, 5, 4]$
- # of inversions of a = (# of inversions of a_l) + (# of inversions of a_r) + (# of inversions between a_l and a_r)
- To calculate the # of inversions between a_l and a_r, for each integer $a_r[i]$, we need to count the number of integers in a_l that is greater than $a_r[i]$

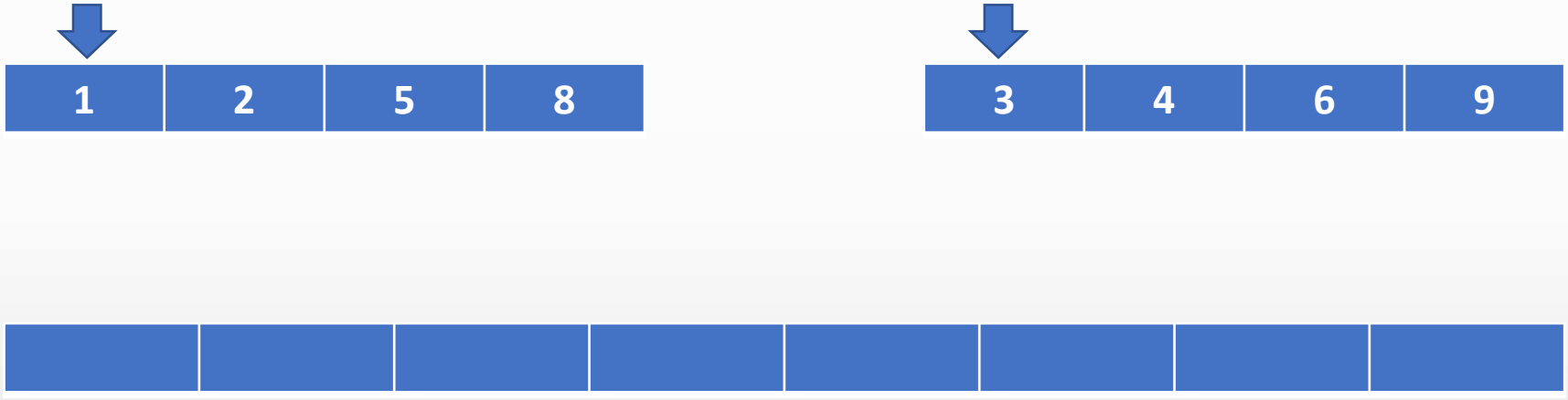
Example 3 – Number of Inversions

- Once the # of inversions of a_l is obtained, we do not care about the order in a_l anymore, and same for a_r
- Sorting a_l and a_r will not affect the answer now
- To count the number of inversions between a_l and a_r, we merge the sorted arrays a_l and a_r into one sorted array (like Merge Sort)

- Whenever an element from a_r is pushed into the array, count the number of elements from a_l that are not yet pushed into the array
  - These are the inversion pairs between a_l and a_r

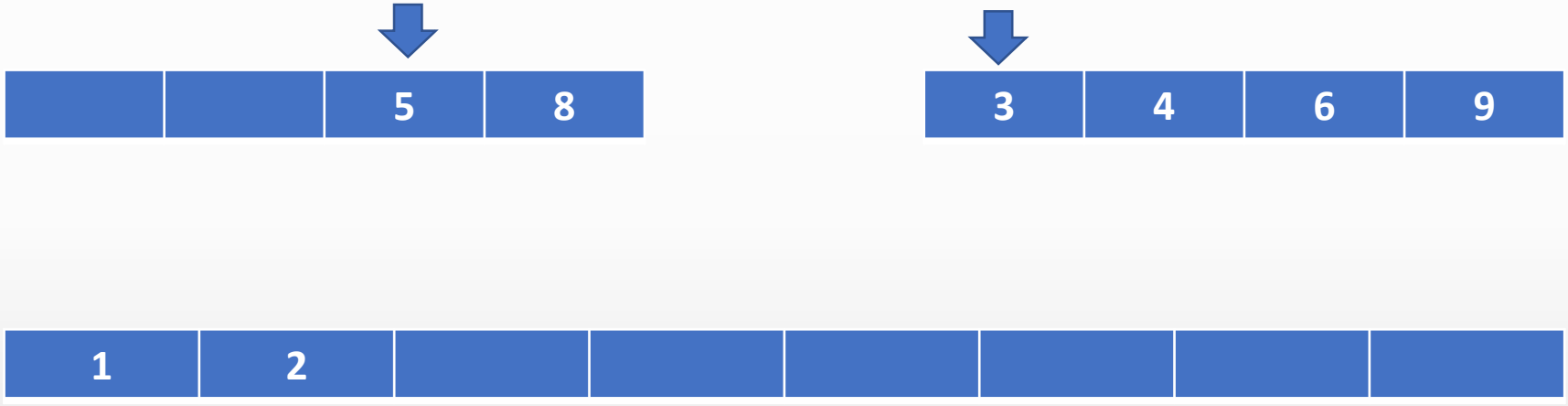
Example 3 – Number of Inversions

- Answer = 0



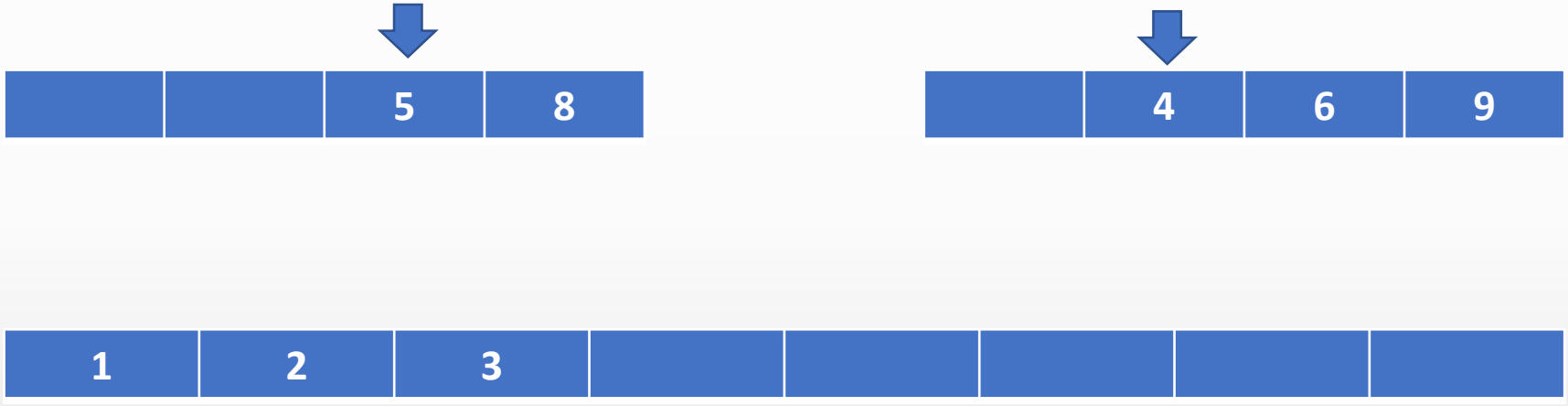
Example 3 – Number of Inversions

- Answer = 0



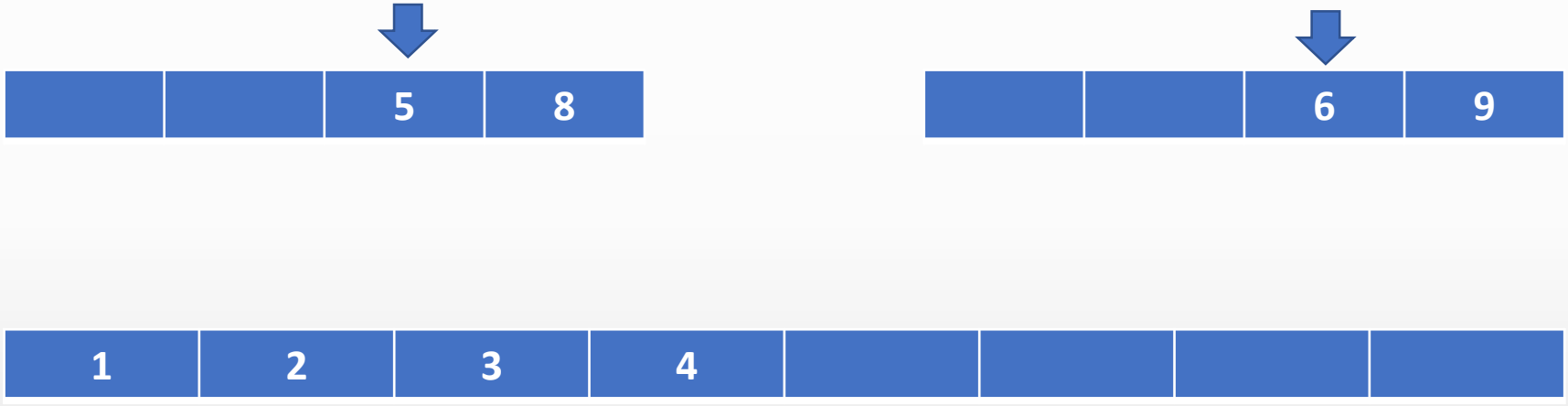
Example 3 – Number of Inversions

- Answer = 2, as there are 2 remaining integers in a_l



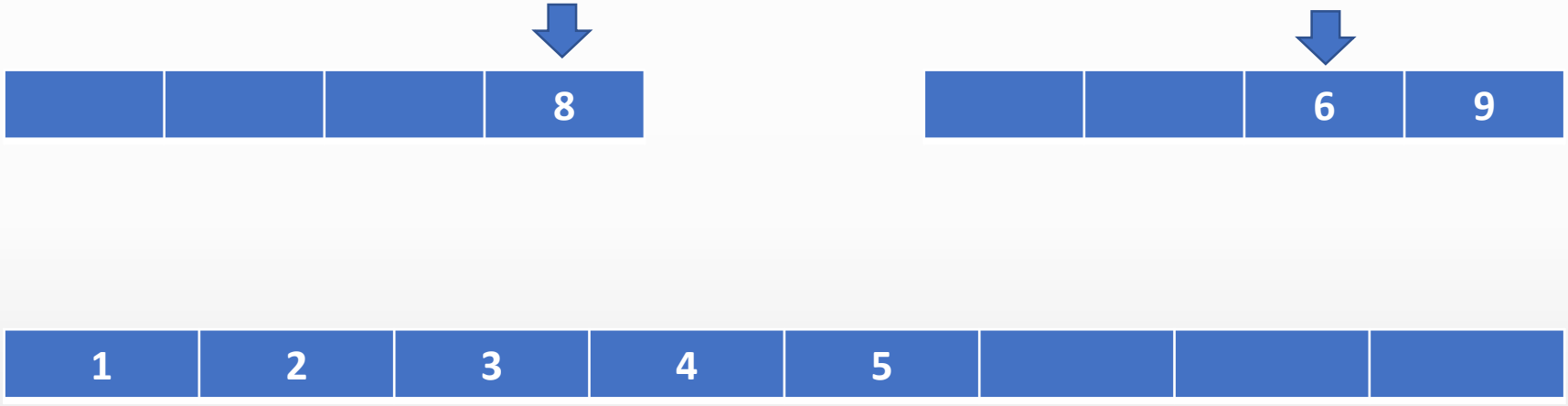
Example 3 – Number of Inversions

- Answer = 4, as there are 2 remaining integers in a_l



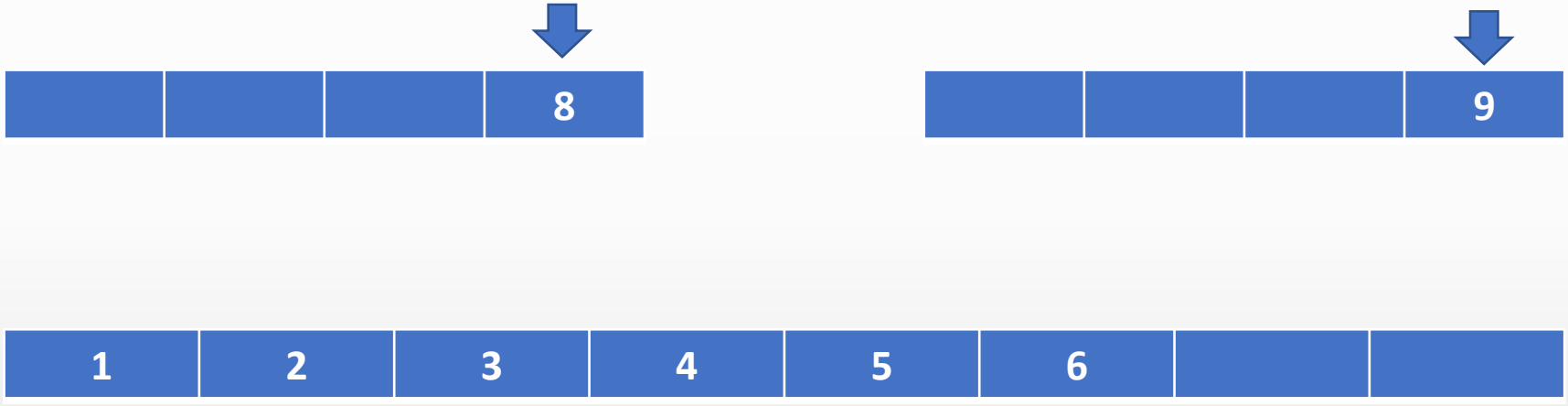
Example 3 – Number of Inversions

- Answer = 4



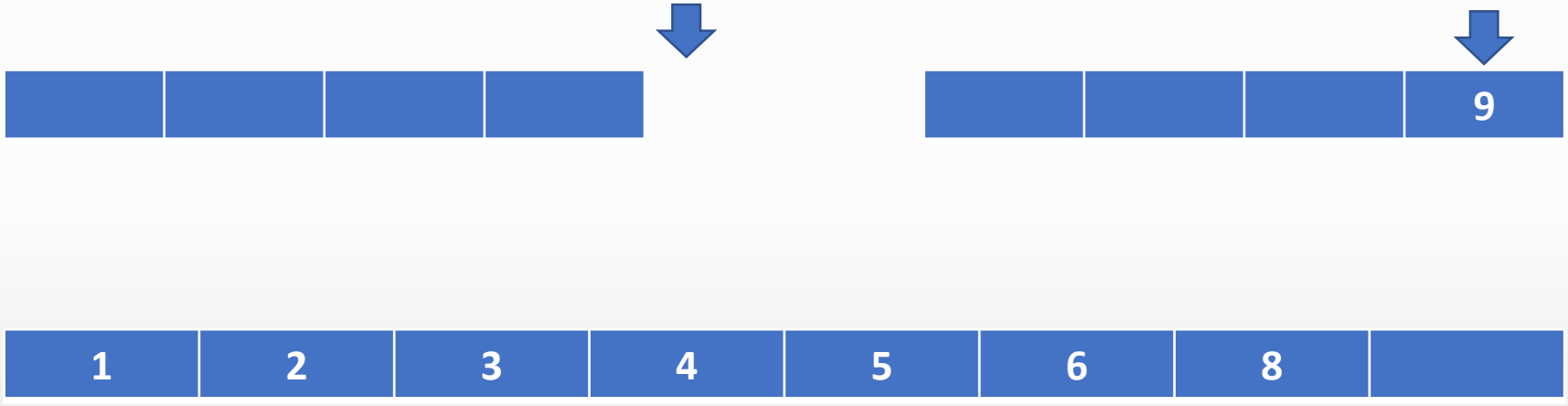
Example 3 – Number of Inversions

- Answer = 5, as there are 1 remaining integer in a_l



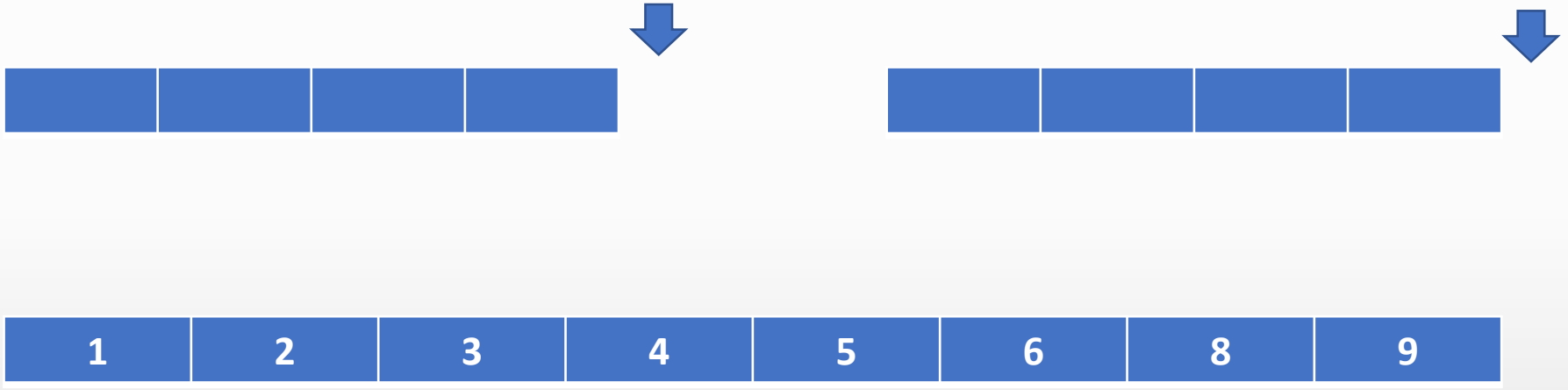
Example 3 – Number of Inversions

- Answer = 5, as there are 1 remaining integer in a_l



Example 3 – Number of Inversions

- Answer = 5, as there are 0 remaining integers in a_l



Example 3 – Number of Inversions

- Solution:
  - Split the array into two halves a_l and a_r
  - Compute the number of inversions of the two halves and sort them recursively
  - Merge the two halves and compute the number of inversions between two halves
  - Return the sum of the three parts as the answer
- Time complexity:
  - $T(n) = 2T\left(\frac{n}{2}\right) + O(n), a = 2, b = 2, d = 1, \log_b a = 1 = d$
  - $T(n) = O\left(n^d \log n\right) = O(n \log n)$

Example 3 – Number of Inversions

```
function compute_and_sort(integer x, y){          // processing the subarray a[x .. y]
        if x = y then return 0
        ans1 = compute_and_sort(x, (x + y) / 2)
        ans2 = compute_and_sort((x + y) / 2 + 1, y)
        ptr_l = x, ptr_r = (x + y) / 2 + 1
        ans3 = 0
        for i from x to y
                if ptr_l > (x + y) / 2 then                b[i] = a[ptr_r++]
                else if ptr_r > y then                     b[i] = a[ptr_l++]
                else if a[ptr_l] < a[ptr_r] then           b[i] = a[ptr_l++]
                else
                        b[i] = a[ptr_r++]
                        ans3 += (x + y) / 2 + 1 – ptr_l
        a[x .. y] = b[x .. y]
        return ans1 + ans2 + ans3
}
```

Summary

- Recursion is useful and common
- Don’t hesitate to use exhaustion to get some scores or find patterns
- Don’t ignore it and spend all the time on other subtasks
- Divide and Conquer is essential for many data structures, and is often related to other algorithms
- Practice more :O)

Practice Problems

- 01003 L-pieces
- 01046 One-Step Tower of Hanoi
- 01047 Partially-Solved Tower of Hanoi
- M0713 Cream Soda
- J173 Fibonacci Word
- S134 Unfair Santa Claus
- S163 Arithmetic Sequence