

ICDS Spring 2025

# Algorithm: Part III

**Algorithm Design Paradigms**

# Recap: Computational Complexity

- One task often has multiple solutions
  - To analyze algorithms:
    - **Time complexity:** #elementary operations with respect to the input size
    - **Space complexity:** total memory space taken with respect to the input size
  - **Big-O**/asymptotic notation is used: two assumptions

## Exercise: Complexity of Searching Algo.

What is the time complexity of linear, binary search & hash table?

- A.  $O(n)$ ,  $O(n)$ ,  $O(n)$
- B.  $O(n)$ ,  $O(\log n)$ ,  $O(n)$
- C.  $O(n)$ ,  $O(\log n)$ ,  $O(\log n)$
- D.  $O(n)$ ,  $O(\log n)$ ,  $O(1)$



Scan to answer!

## Exercise: Complexity of Sorting Algo.

What is the time complexity of bubble & merge sort at **best/worst case** scenarios?

- A.  $O(n^2)$  &  $O(n \log n)$  /  $O(n^2)$  &  $O(n^2)$
- B.  $O(n^2)$  &  $O(n \log n)$  /  $O(n^2)$  &  $O(n \log n)$
- C.  $O(n)$  &  $O(n \log n)$  /  $O(n^2)$  &  $O(n \log n)$
- D.  $O(n)$  &  $O(n)$  /  $O(n^2)$  &  $O(n \log n)$

Scan to answer!



## Exercise: Complexity of Sorting Algo.

What is the **space complexity** of insertion & merge sort?

- A.  $O(1)$ ,  $O(1)$
- B.  $O(1)$ ,  $O(\log n)$
- C.  $O(1)$ ,  $O(n)$
- D.  $O(n)$ ,  $O(n)$



Scan to answer!

# Agenda

- Solving NP problems with a decision making process
  - Trees and Recursions
  - Traversing a tree (Pre-order, In-order, Post-order)
    - Fibonacci number / Powerset
- Understanding design paradigms with a tree
  - Backtracking
  - Divide and conquer (quick sort)
  - Dynamic programming (fibonacci number, min bills)
- Other paradigms (Appendix)
  - Induction
  - Reduction
  - Greedy

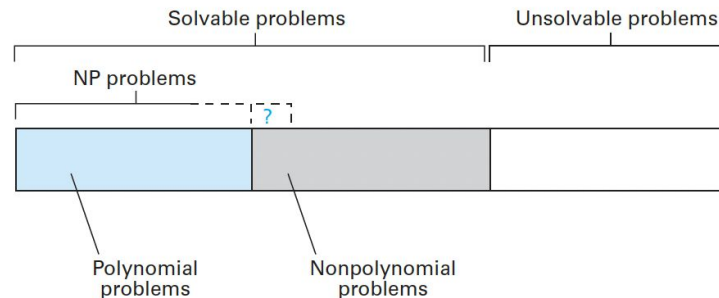
**Solving NP Problems  
by  
making decisions**

# Describing NP problems' solutions as decision processes

- Many real-world problems (NP) can be reduced to the task of answering a few **yes/no** questions about a certain property of the input.
  - e.g., The travelling salesman problem: we can guess a path then verify it by comparing its length with another known path.
  - e.g., In sorting a list, we choose an element and compare it with the others. If it is the smallest one, take it out.
- This decision process (yes/no) can be represented by a tree.

In other words, we can solve a problem by reducing it to sub-problems;

- yes/no questions can be a way to reduce it.





# Application: Binary search

```
def bSearch(L, e):  
    """Assume L is a list, the  
    Returns True if e is in L and False otherwise
```

```
    if len(L) == 1:  
        if L[0] == e:  
            return True  
        return False
```

```
    mid = len(L)//2
```

```
    if L[mid] == e:  
        return True
```

```
    elif L[mid] > e:  
        return bSearch(L[:mid], e)
```

```
    else:  
        return bSearch(L[mid+1:], e)
```

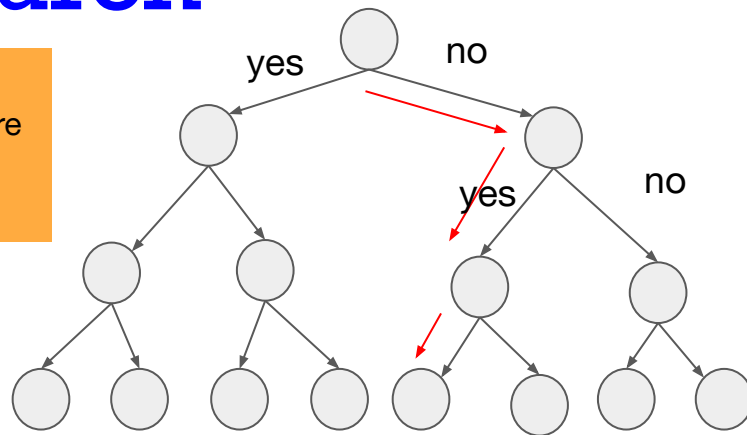
When there is only one element left, compare it with  $e$  and return true if they are equal; otherwise, false. (this is called the stop condition)

We set the  $i$  to be the middle of the list.

stop (return) if  $L[i] = e$

Go left if  $L[i] > e$

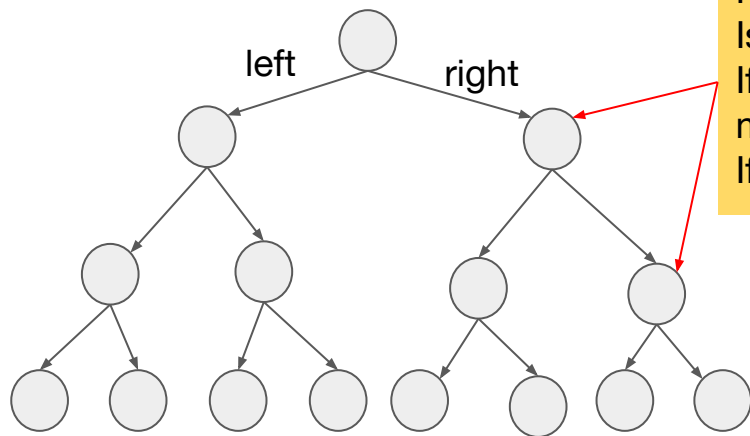
Go right if  $L[i] < e$



The search can be represented by a tree diagram. Each node is a list of numbers. For each node, we **always** do the following,

- a parent node splits to two children nodes by an index  $i$
- the value at  $i > e$  ?  $\rightarrow$  go left if yes; go right if no; stop if equal;

# Application: Merge sort



For each node:

Is the node (list) sorted?

If yes → return (when the length of list is one, it must be sorted)

If no → split ⇒ sort ⇒ merge (sorted) ⇒ return

```
# Merge sort definition
```

```
def merge_sort(m):
```

```
    if len(m) <= 1:
```

```
        return m
```

```
    middle = len(m) // 2
```

```
    left = m[:middle]
```

```
    right = m[middle:]
```

```
    left = merge_sort(left)
```

```
    right = merge_sort(right)
```

```
    return merge(left, right)
```

Check if the node is sorted

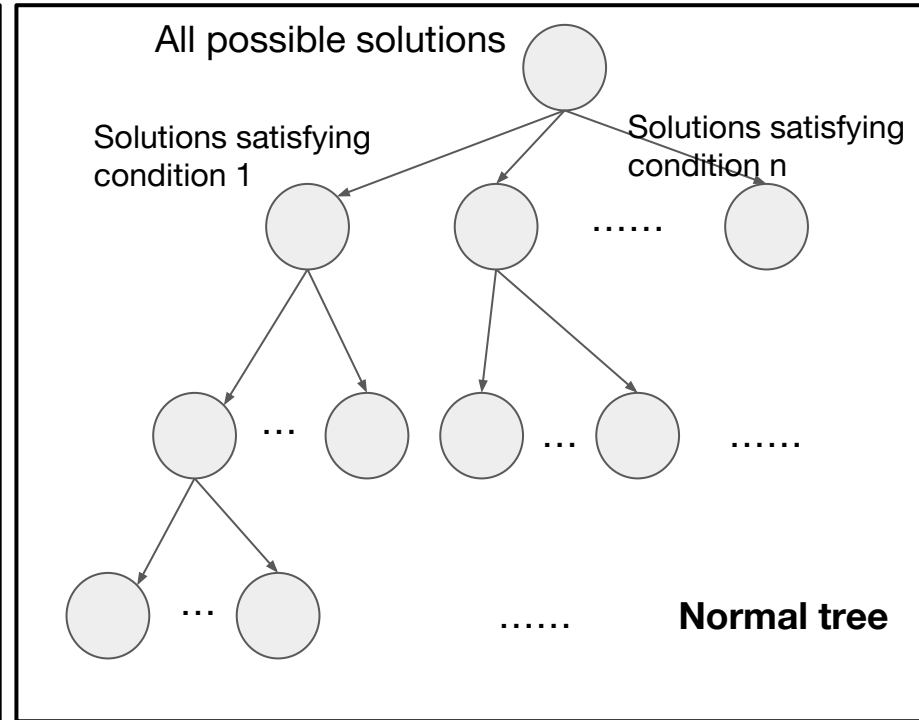
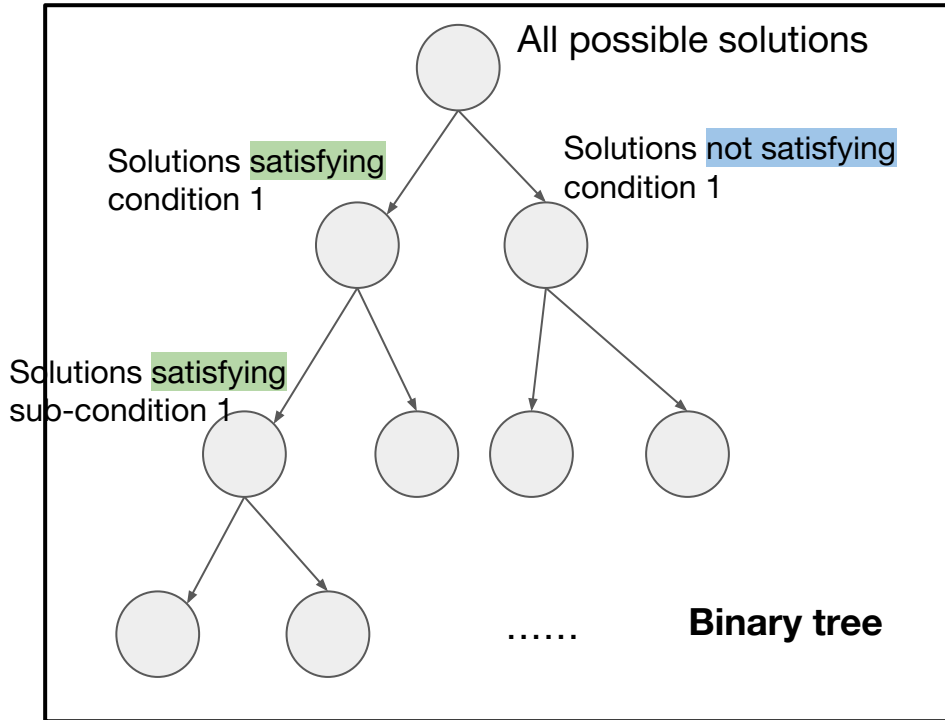
Split the list into left and right

Sort the left and right

merge the sorted left and right

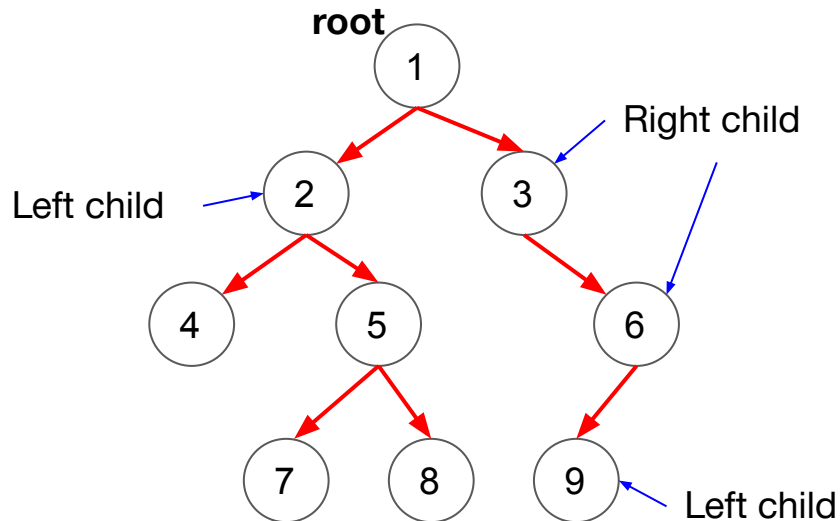
# A tree structure as a decision process

- We can often describe a decision process by a tree



# Binary tree

- A tree structure in which each node has at most two children.



**Root:** 1

**Parent:** a node has some children (e.g., 2, 3, 5, 6)

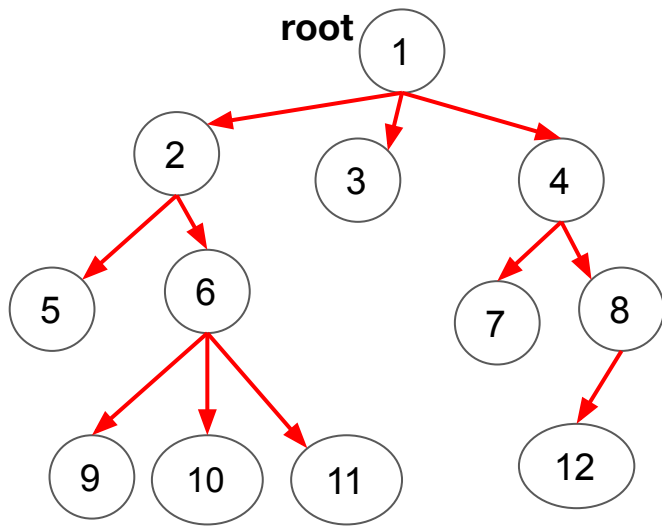
**Leaves:** nodes have no children. (e.g., 4, 7, 8, 9)

**Left children:** the child node on the left of the parent node

**Right children:** the child node on the right of the parent node

# Normal tree

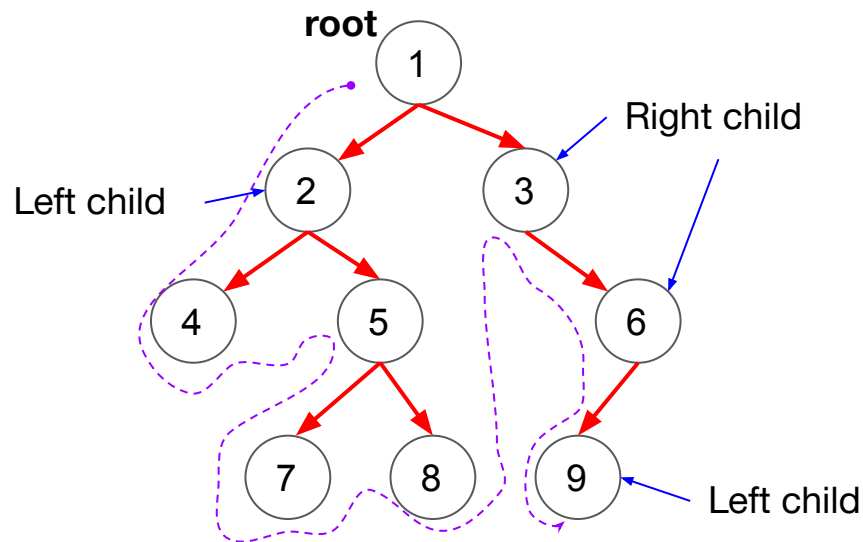
- A tree structure is a collection of nodes connected by directed (or undirected) edges.



Each node (except the root) is connected by an edge from exactly one other node.

- **Root:** 1
- **Parent:** a node has some children (e.g., 2 is the parent of 5 and 6; 6 is the parent of 9, 10, 11);
- **Child:** 5, 6 are children of 2;
- **Leaves:** node has no children (e.g., 5, 3, 7, 9, 10, 11, 12);

# Traversing a binary tree with recursion

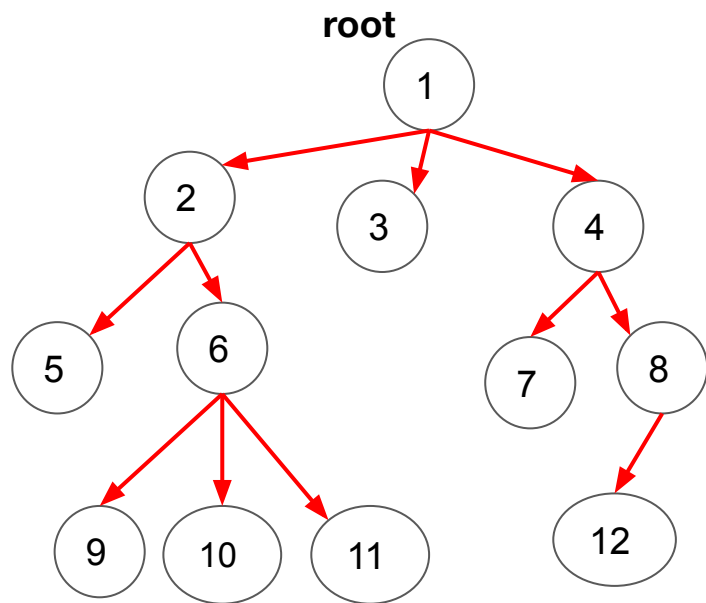


**Preorder** (node→left→right):  $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 3 \rightarrow 6 \rightarrow 9$  (as the purple line)

**Inorder** (left→node→right):  $4 \rightarrow 2 \rightarrow 7 \rightarrow 5 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 6$

**Postorder** (left→right→node):  $4 \rightarrow 7 \rightarrow 8 \rightarrow 5 \rightarrow 2 \rightarrow 9 \rightarrow 6 \rightarrow 3 \rightarrow 1$

# Traversing a normal tree with recursion



- **Preorder:**  
 $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 12$
- **Inorder:** no natural definition since the number of children varies
- **Postorder:**  $5 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 6 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 12 \rightarrow 8 \rightarrow 4 \rightarrow 1$

# The difference between the code framework

```
def bSearch(L, e):  
    """Assume L is a list, the elements of which are in ascending order.  
    Returns True if e is in L and False otherwise"""
```

```
    if len(L) == 1:  
        if L[0] == e:  
            return True  
        return False
```

Do something with  
the current node

```
    mid = len(L)//2
```

```
    if L[mid] == e:  
        return True  
    elif L[mid] > e:  
        return bSearch(L[:mid], e)  
    else:  
        return bSearch(L[mid+1:], e)
```

Then, go to the  
next node

```
# Merge sort definition
```

```
def merge_sort(m):
```

```
    if len(m) <= 1:  
        return m
```

Go to the child nodes (so get  
the information needed)

```
    middle = len(m) // 2  
    left = m[:middle]  
    right = m[middle:]  
    left = merge_sort(left)  
    right = merge_sort(right)  
    return merge(left, right)
```

Then, do something for the current node

- The two pieces of code process (i.e., visit) the nodes of the tree in different orders!

What traversal order do they follow?



# Example: Finding the n-th Fibonacci number

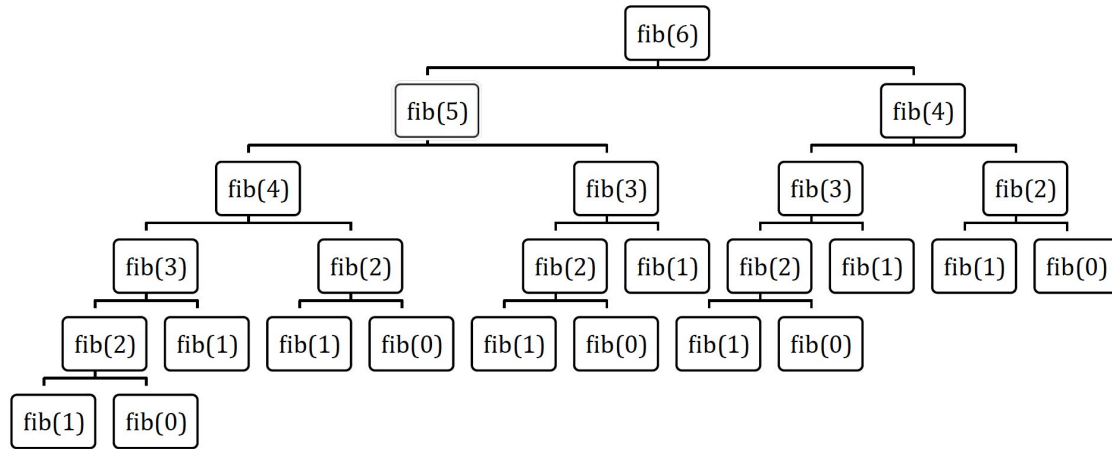
The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Each number (except the first two) is the sum of the two numbers that precede it.
- 1st  $\rightarrow$  0
- 2nd  $\rightarrow$  1

So, we know that

- 3rd Fibonacci number is  $0+1 = 1$
- 4th Fibonacci number is  $1+1 = 2$
- 5th ...
- .....

# The n-th Fibonacci number



**Tree of calls for recursive Fibonacci**

- To get the n-th Fibonacci number  $f(n)$ , we need to get  $f(n-1)$  and  $f(n-2)$ .
- To get  $f(n-1)$ , we need to get  $f(n-2)$ ,  $f(n-3)$ .
- ...
- We stop until we reach  $f(2)$ .

# The n-th Fibonacci number

```
16 def fib(n):  
17     """  
18     Assume n is an integer with n > 0  
19     """  
20     if n == 1:  
21         return 0  
22     if n == 2:  
23         return 1  
24  
25     n1_fib = fib(n-1)  
26     n2_fib = fib(n-2)  
27  
28     return n1_fib + n2_fib
```

Is the value of the current node is known?

Finding the values of the children nodes

Computing the value of the current node

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

What traversing order does it use?

What are the time & space complexities of this implementation?

## Example: finding the powerset

Given a set  $S$ , the power set of  $S$  is the set of all subsets of  $S$ . The power set of  $S$  is denoted by  $\mathcal{P}(S)$ .

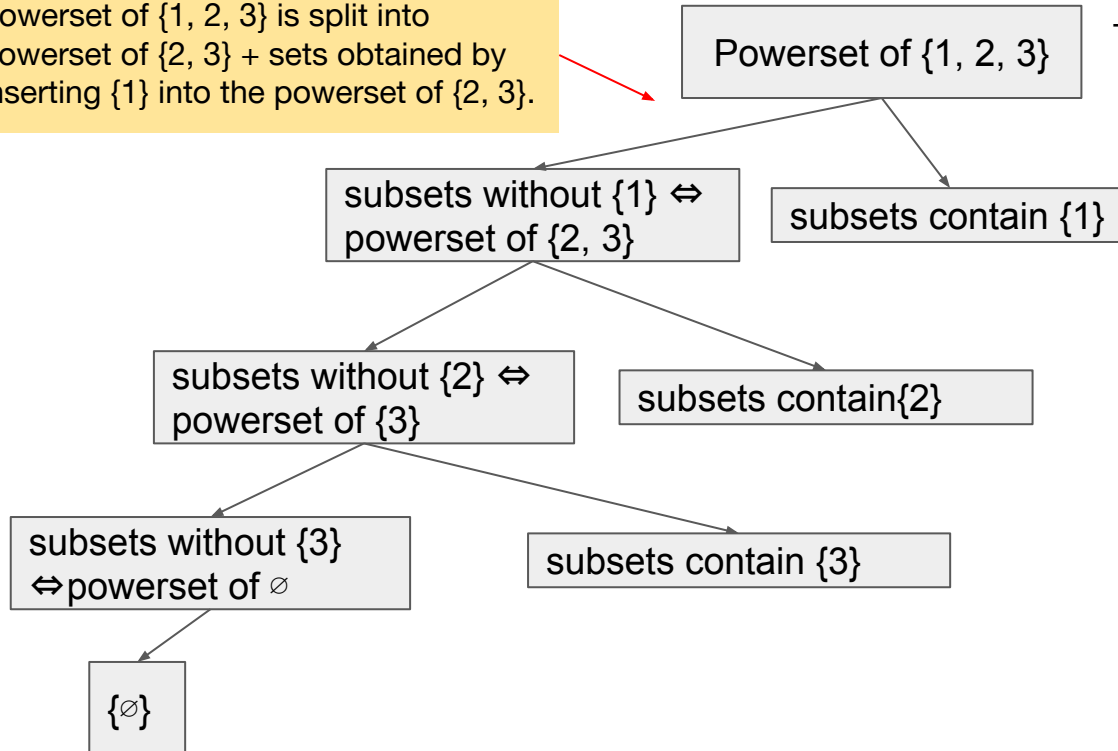
Example:

What is the power set of the set  $\{1, 2, 3\}$ ?

$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

# A tree based solution

Powerset of  $\{1, 2, 3\}$  is split into powerset of  $\{2, 3\}$  + sets obtained by inserting  $\{1\}$  into the powerset of  $\{2, 3\}$ .



To find the powerset of set  $S = \{1, 2, 3\}$ ,

- We need to find the power set of  $\{2, 3\}$ .
- Once we get the powerset of  $\{2, 3\}$ , we generate the sets that contains  $\{1\}$ .

To find the powerset of set  $S = \{2, 3\}$ ,

- We need to find the power set of the subset of  $\{3\}$ .
- Once we get the powerset of  $\{3\}$ , we generate the sets that contains  $\{2\}$ .

..... We will stop until  $S$  is empty.

# Powerset

Following the discussion in last page, we program the solution using recursion.

**Stop condition:** the set is empty

**Call the function itself:** Finding the powerset of the subset (left node)

**Code for getting the right node:** Inserting the element into each element of the left node.

**concatenate the left and right nodes:** the subset containing the element + the subset without the element

```
20 def powerset_recursive(lst):
21     if len(lst) == 0:
22         return [[]]
23     left = powerset_recursive(lst[1:])
24     right = deepcopy(left)
25     for subset in right:
26         subset.append(lst[0])
27     return left+right
```

What traversing order does it use?

What are the time & space complexities of this implementation?

# The code framework (tree-based solutions)

Two parts (with different traversing orders; usually, preorder or postorder):

- Visit the nodes (finding the nodes; in tree structures, this can be done by recursion)
- Do something with the nodes
  - e.g., checking the values (for search tasks)
  - e.g., adding up/comparing the values
  - .....

# Understanding design paradigms with tree structures

We can designing an algorithm in two steps:

1. Find a tree to represent the possible solutions to the problem
2. Find a way to visit the tree nodes to find the optimized solution

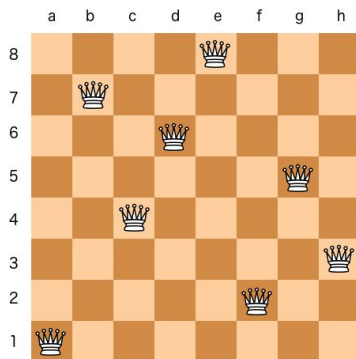
So, a faster algorithm means it can find the optimized solution by visiting less nodes:

- A good tree  $\Rightarrow$  using less nodes to represent the problem
- A good way to explore the tree  $\Rightarrow$  travelling less nodes

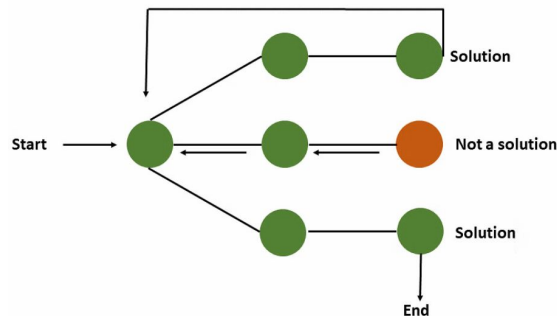


# Brutal force (Backtracking)

- Brutal force means we build a tree that contains every possible solution to the problem and visit all of them to solve the problem.
  - Linear search
  - Eight queen puzzle: placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other
- Backtracking is a programming trick to visit the nodes in a tree for brutal force
  - e.g., solving the eight queen puzzle  $\Rightarrow$  try all positions



try all positions

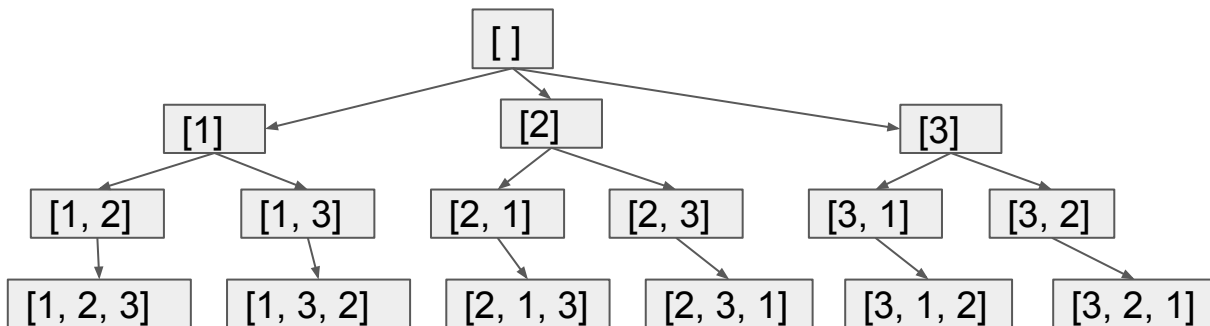


# Brutal force using Backtracking

**Example: find all permutations of [1, 2, 3]**

1. Start from an empty list
2. Pick a number from [1, 2, 3], add it into the list
  - a. If the number has been in the list, ignore the number
3. Repeat step 2 until the length of the list is 3
4. Repeat Step 1 to 3 until all permutations are found (note: 1, 2, 3 are added into the empty list in order in each repetition)

The solution is based on this tree.



# Finding the permutation

```
9  import copy
10
11  def permute(nums, leaves=[], current_node=[]):
12      if len(current_node) == len(nums):
13          leaves.append(current_node)
14          return
15      for n in nums:
16          if n in current_node:
17              continue
18          temp = copy.deepcopy(current_node)
19          temp.append(n)
20          permute(nums, leaves, temp)
21
22  return leaves
23
24
25  if __name__ == "__main__":
26      nums = [1, 2, 3]
27      p = permute(nums)
28      print(p)
```

Validation of  
the path.

nums: the list of numbers (all the numbers)  
leaves: the leaves of the tree (i.e., a collection of permutations)  
current\_node: the node currently visited (i.e., a subset of nums)

- It is a preorder traversal; We move to a node, and then, check if n is in the node or not;
- We use `deepcopy` to keep the content of the current node; once we visit a child (and its children), we will go back to the current node and visit another child (i.e., the backtracking)

Note: this piece of code only works for the list with no repetitive elements. Can you make it work for lists like [1, 1, 2, 3]? [Hint: you may change the validation and/or the stop condition.]

If n isn't in the current track, we will go on with the path; otherwise, we ignore(prune) the path.

# Divide and conquer

Divide and conquer is based on decomposing the problem in a way that improves performance.

⇒ It focuses on designing a tree that represents the problem with nodes as less as possible.

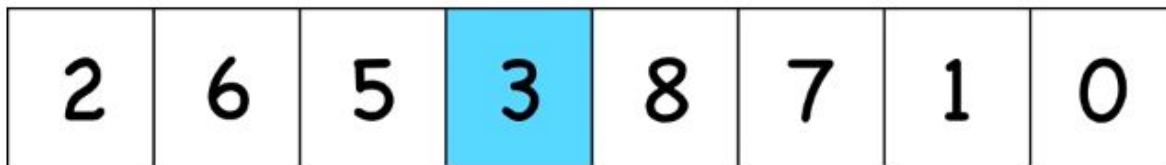
- The task of scale  $n$  is **divided** into  $k$  **independent subtasks** (i.e., reduced geometrically), each subtask can be solved in the same way as the origin task. ⇒ so we can solve the subtasks recursively
- Combine the results, and thereby conquer the problem.

## How to code:

- Traversing a binary tree
- Examples: merge sort, quick sort

# Quick sort

Step 1. Pick an element from the array; we call it as a pivot.



pivot

Pivot can be selected in many ways:

- From the most right/left
- Randomly choose one
- ...

# Quicksort

## Step 2. Partitioning:

- reorder the array:
  - elements less than the pivot go to the left side of it;
  - elements greater than the pivot go to the right side of it;
- So, the pivot is in its final position after partitioning



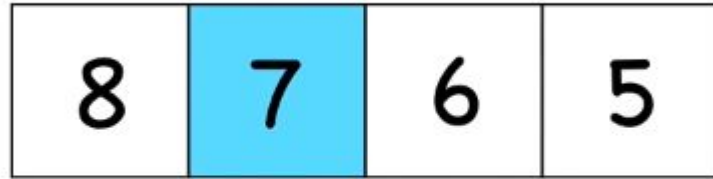
all elements with values less than the pivot come before the pivot

all elements with values greater than the pivot come after it  
(equal values can go either way)

# Quicksort

Step 3. Recursively apply Step1 and Step 2 to the sub-arrays.

- Sub-array: all the numbers on the left (or right) side of the pivot.



pivot

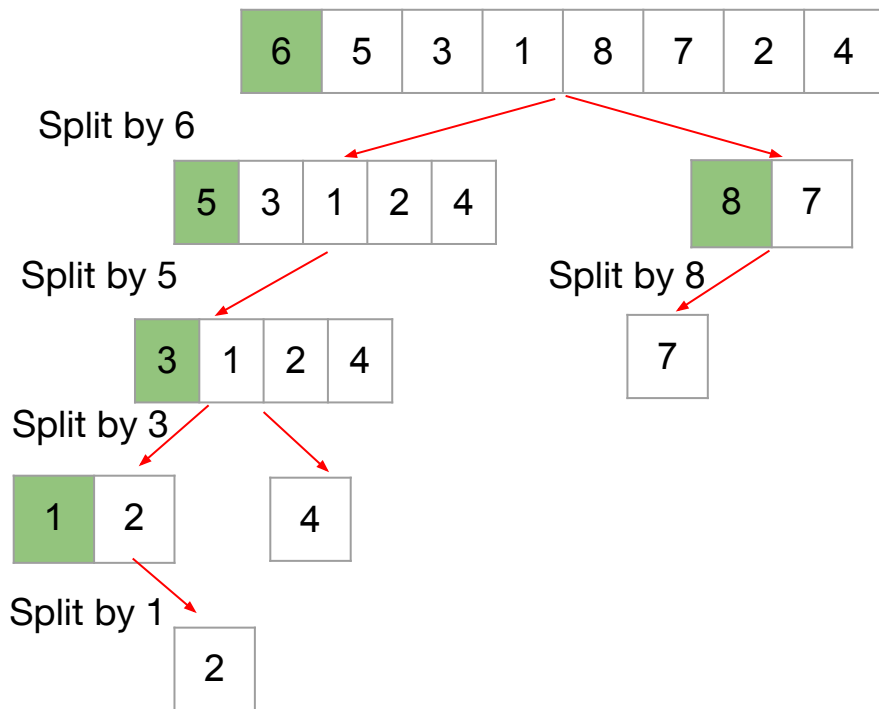
# Quick sort

- Quicksort divides a large array into two small sub-array and one pivot.
  - the low elements ( $<$  pivot)
  - the pivot
  - the high elements ( $>$  pivot)
- It sorts the sub-arrays recursively.

**Difference to merge sort:** Quick sort creates sublists that has a certain order (e.g., less than a specific value) instead of finding a sorted sublist.



# Quick sort



It has a binary tree structure.

- We move to a node → process it (splitting it into low and high, which are roughly sorted, and so we get the next nodes) → go to the next node
- Quick sort is the preorder traversal of the binary tree, because the nodes are generated during the partition (i.e., the node is processed then we visit it.).

# Python code

```
9  def quicksort(seq):
10     if len(seq) <= 1:
11         return seq
12     low, pivot, high = partition(seq)
13     return quicksort(low) + [pivot] + quicksort(high)
14
15
16
17 def partition(seq):
18     pivot, seq = seq[0], seq[1:] #pick the first element as pivot
19     low = [x for x in seq if x <= pivot]
20     high = [x for x in seq if x > pivot]
21     return low, pivot, high
```

- Divide the problem into sub-problems
- Combine the results of each sub-problem

Time & space complexities?

Exercise: Can you improve the given example, to make the quicksort being done in-place, so, in the partition process, the space complexity is  $O(1)$ ? (Hint: ask ChatGPT for quicksort in-place)

# Dynamic programming

Dynamic programming (DP) focuses on reducing the #nodes for travelling.

Briefly, the core of dynamic programming is to **memorize** the solutions of subproblems so that it avoids computing them again.

Problems can be solved by DP:

- A problem can be decomposed into overlapping subproblems, with optimal substructure

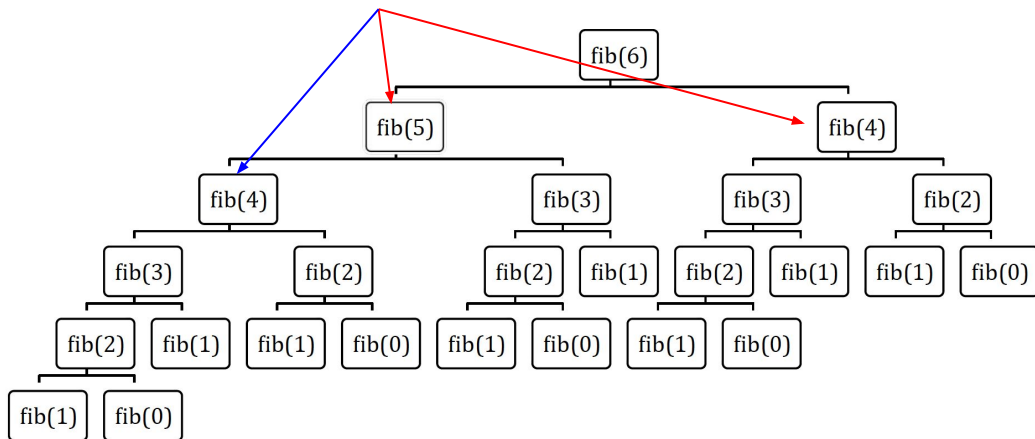
**How to code:**

- Recursion + “memory” (e.g., a dictionary)

## fast\_fib():

- Since there are overlaps between the subproblems, you can remember the past as you explore.

Lots of redundant computations!



**Tree of calls for recursive Fibonacci**

- Finding fib(6) can be done by finding fib(5) and fib(4).
- The two subproblem has some overlaps, i.e., the result of fib(5) depends on the result of fib(4).
- We can save (remember) the result of fib(4)

Complexity:  $O(n)$

# fast\_fib(): “trade the time by the space”

- Remember the past as you explore

```
def fast_fib(n, memo = {}):  
    """  
    Assume n is an int > 0, memo used only by recursive calls  
    returns Fibonacci of n.  
    """  
    if n == 0 or n == 1:  
        return 1  
    try:  
        return memo[n]  
    except KeyError:  
        result = fast_fib(n-1, memo) + fast_fib(n-2, memo)  
        memo[n] = result  
        return result
```

a Python dictionary plays the role of memory. (We often use a hash table as the “memory”.)

# Picking the fewest bills

Suppose you want to count out a certain amount of money, say \$123, using the fewest bills and coins. (Assume we are in China, we have 7 bills: 100, 50, 20, 10, 5, 2, and 1, and each bill is sufficient supply. )

There are a number of ways to collect \$123.

- \$1 x 123
- \$20 x 6 + \$1 x 3
- \$100 x 1 + \$10 x 2 + \$1 x 3
- .....

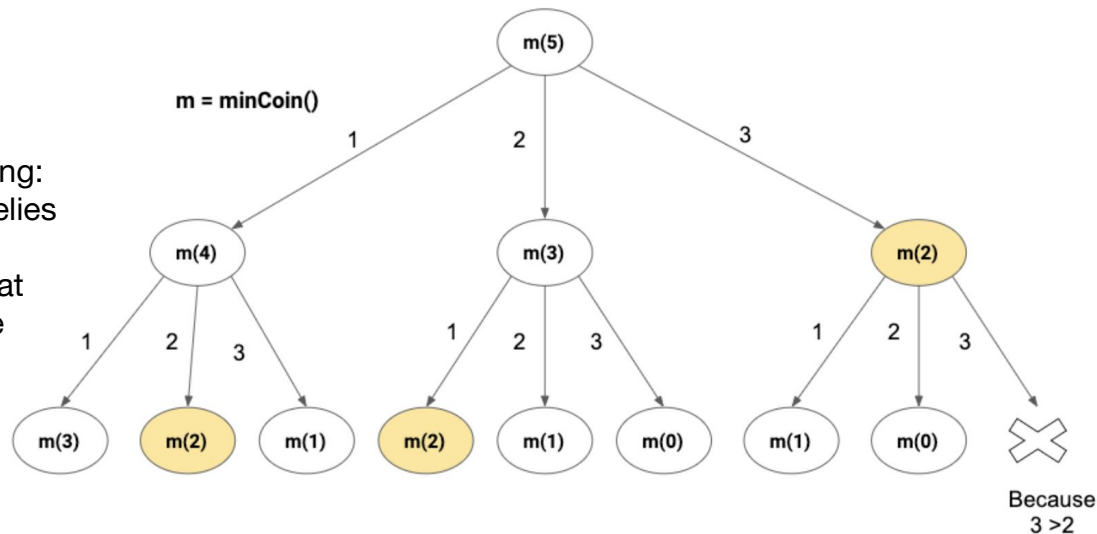
One way to find the optimized collection is to list all possible collections and then find the one with fewest bills and coins. (but the number of possible collections is large)

# Solving Picking the fewest bills

- We start by brutal force  $\Rightarrow$  building a tree showing every possible combinations
- Then, finding the overlaps among the nodes; designing a way to memorize them.

The problem can be solved by dynamic programming:

- Substructure: each round, the bill you pick relies on the bills you picked in the past.
- Overlapping: each round, we pick one bill that minimize the number of bills to count out the current value.

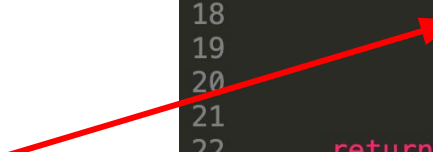


# Brutal force: Picking the fewest bills

This is the recursive version based on our previous analysis.

To avoid repeatedly calculating the (p, num\_last\_picked) for the (b, value-b) pair, we shall keep them in a memo.

```
12 def minBills(bills, value):
13     if value == 0:
14         return [], 0
15     num_current_picked = float("inf")
16     for b in bills:
17         if b <= value:
18             p, num_last_picked = minBills(bills, value-b)
19             if num_last_picked + 1 < num_current_picked:
20                 num_current_picked = num_last_picked + 1
21                 pk = p + [b]
22     return pk, num_current_picked
23
24
25 if __name__ == "__main__":
26
27     bills = [9, 6, 5, 1]
28     value = 11
29     picked, n = minBills(bills, value)
30     print(picked, n)
```



Time & space complexities?



# DP: Picking the fewest bills

This is the fast\_minBills(), very similar to the fast\_fib() we written.

```
24 def fast_minBills(bills, value, memo={}):
25     if value == 0:
26         return [], 0
27     num_current_picked = float("inf")
28     for b in bills:
29         if b <= value:
30             try:
31                 p, num_last_picked = memo[(b, value-b)]
32             except KeyError:
33                 p, num_last_picked = fast_minBills(bills, value-b, memo)
34                 memo[(b, value-b)] = (p, num_last_picked)
35             if num_last_picked + 1 < num_current_picked:
36                 num_current_picked = num_last_picked + 1
37                 pk = p + [b]
38     return pk, num_current_picked
```

A Python dictionary plays the role of memory.

The key you designed should be unique.

# Summary: coding solutions with a tree

- Represent the problem by a tree (a node is a subproblem)
  - A good representation often has less nodes than a normal one.
- Visit the node to solve the problem
  - take use of the relations between the subproblems to reduce the number of nodes need to visit (e.g. dynamic programming)
- Program your solution with the tree framework
  - Preorder: do something first, then, go to the next node
  - Inorder: go to the left node first, do something, then, go to the right node
  - Postorder: go to the next node first, then, do something

# **Appendix: Other paradigms**

## Other strategies

Of course, there are algorithm design paradigms that are not strongly connected with tree structures:

- **Induction:** from the simplest to the complex
- **Reduction:** transform the problem to another problem
- **Greedy algorithm:** taking the best they can get at this moment, not worrying about the future

# Induction

Induction is often used in math proofs:

- First, showing a solution is feasible for a base case (basis step)
- Then, showing it carries over from one object to the next (inductive step)
- Way to code it:
  - Loops

## Bottom-up: Fibonacci number

```
func fib_bottom_up(n) #n>2
    var f as an array of size n
    f[0] = 0
    f[1] = 1
    for i from 2 to n-1:
        f[i] = f[i-1] + f[i-2]
    return f[n-1]
end func
```

# Bottom-up: generating Fibonacci number inductively

```
def fib(n):  
    """  
    n > 0  
    This function can return the  
    fib sequence.  
    """  
    if n == 1:  
        return 0  
    f = [0] * n  
    f[1] = 1  
    for i in range(2, n):  
        f[i] = f[i-1] + f[i-2]  
    return f[n-1]
```

```
def fib(n):  
    """  
    n > 0  
    return the n-th fib number  
    """  
    a = 0  
    b = 1  
    if n == 1:  
        return a  
    if n == 2:  
        return b  
    for i in range(2, n):  
        c = b + a  
        a = b  
        b = c  
    return c
```

# The powerset

Given a set  $S = \{s_1, s_2, \dots, s_n\}$ , how many subsets does  $S$  have?

Let's start with some simple case:

- When  $S = \{\}$ , we have  $P(S) = \{\{\}\}$ , so the size of  $P(S)$  is 1.
- When  $S = \{a\}$ , we have  $P(S) = \{\{\}, \{a\}\}$ , so the size of  $P(S)$  is 2.
- When  $S = \{a, b\}$ , we have  $P(S) = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$ , so the size of  $P(S)$  is 4.
- When  $S = \{a, b, c\}$ , we have  $P(S) = \{\{\}, \{a\}, \{b\}, \{a, b\}, \{c\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$ , so the size of  $P(S)$  is 8.



# Bottom-up: generating a powerset inductively

The initial powerset has an empty set.

```
def powerset_add(l):  
    pset = [[]]  
    for item in l:  
        new_subset = deepcopy(pset)  
        print(pset)  
        for subset in new_subset:  
            subset.append(item)  
        print(new_subset)  
        pset.extend(new_subset)  
        print(pset)  
    return pset
```

New subset = Adding an element into each existing set.

Current powerset  
= the last powerset + new\_subset

# Reduction

Reduction means transforming one problem to another.

- We often reduce an unsolved problem to one we know how to solve.

# Finding two closest but not identical numbers in a list

Given a list of numbers( $n > 2$ ), you need to find the two (nonidentical) numbers that are closest to each other.

e.g., Given a list like this: [3, 5, 1, 2, 9, 12, 3, 30, 1, 6, 6, 9, ...]; you need to find the two closest numbers.

One intuitive solution:

- When  $n = 3$ , [a, b, c] → We calculate the absolute value of  $|a-b|$ ,  $|a-c|$ ,  $|b-c|$ , the closest numbers have a smallest absolute value.
- When  $n = 4$ , [a, b, c, d] → We calculate the absolute value of  $|a-b|$ ,  $|a-c|$ ,  $|a-d|$ ,  $|b-c|$ , ..., the closest numbers have a smallest absolute value.

Our solution works for any list whose length is greater than 2.

Now, let's code it.

# Finding two closest but not identical numbers in a list


Following the intuitive solution, we can write the code like the right.

- Two nested loop:  $O(n^2)$
- Stores the values in a dictionary.
  - It wastes memory space since we only need to find the first closest pair.
- Returns the first detected closest pair.

```
def find_first_closest(lst):  
    d = {}  
    for x in lst:  
        for y in lst:  
            if x == y:  
                continue  
            try:  
                d[abs(x-y)].append((x, y))  
            except KeyError:  
                d[abs(x-y)] = [(x, y)]  
    keys = sorted(d)  
    return d[keys[0]][0]
```

There is a better solution, and we will talk about it later.

A memory saving version



```
def find_first_closest(lst):  
    dd = float('inf')  
    for x in lst:  
        for y in lst:  
            if x == y:  
                continue  
            d = abs(x-y)  
            if d < dd:  
                xx, yy, dd = x, y, d  
    return (xx, yy)
```

## Reduction: convert it to another problem

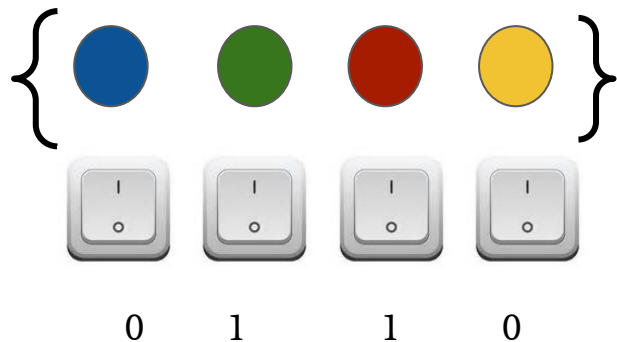
We reduce the original problem to the problem of sorting a sequence, then comparing the neighbors.

- The complexity is  $O(n\log(n) + n) \sim O(n\log(n))$

```
def find_first_closest(lst):  
    dd = float("inf")  
    lst.sort()  
    for i in range(len(lst)-1):  
        x, y = lst[i], lst[i+1]  
        if x == y:  
            continue  
        d = abs(x-y)  
        if d < dd:  
            xx, yy, dd = x, y, d  
    return xx, yy
```

# Powerset: The combinatorial proof

Assume there is a set  $S$  of  $n$  items, we can imagine that each item is connected to a switch.



We can use the switches to represent the subset of  $S$ .

- If a switch is on, the corresponding item is in the subset.
- If a switch is off, the corresponding item is not in the subset.


If we use 1 and 0 to indicate on and off, counting the number of subsets is to count the number of distinguishable bit strings of length  $n$ .

- There are  $2^n$  distinguishable bit strings of length  $n$ ; therefore,  $S$  has  $2^n$  subsets.

# Reduction: find the powerset

Recall each subset can be represented by a (binary) number from 0 to  $2^n$ , we need to convert the decimal to binary to find its corresponding subset.

Remember to invert the string, because `bin()` can't generate string like '0000101'.



```
def powerset_comb(l):  
    pset = []  
    total_items = len(l)  
    for i in range(2 ** total_items):  
        subset = []  
        code = bin(i).split('b')[-1]  
        code = code[::-1]  
        for j in range(len(code)):  
            if code[j] == '1':  
                subset.append(l[j])  
        pset.append(subset)  
    return pset
```

# Greedy algorithms

Greedy algorithms are characterized by how they make decisions.

- making each choice in isolation, doing what looks good right here, right now
- taking what they can get at this moment, not worrying about the future
- designing and implementing a greedy algorithm is usually easy



# Picking the fewest bills

Suppose you want to count out a certain amount of money, say \$123, using the fewest bills and coins. (Assume we are in China, we have 7 bills: 100, 50, 20, 10, 5, 2, and 1, and each bill has sufficient supply. )

Using a greedy algorithm: At each step, take the largest possible bill that does not overshoot. So, we will choose:

- a 100 yuan
- a 20 yuan
- a 2 yuan
- a 1 yuan

# Picking the fewest bills

But greedy algorithms **cannot always** provide the optimized solutions.

For example, the bills you have is 9, 6, 5, 1, and the value you need is 11.

- If you follow a greedy algorithm, you solution is {9, 1, 1}
- But the optimized solution is {6, 5}

Anyway, a greedy algorithm is always an option, especially when it is hard to find an optimized solution. Then, a greedy algorithm can at least provide you with a suboptimal solution.