The Author
Mike Chyson (Li Mingming)

The Big Book of

Algorithms

Theory and practice of algorithms

First created: Dec, 10, 2020
Last modified: Friday 26th March, 2021

Dedication

Contents

Chapter 1

Theory

1.1 What is algorithm?

$$input \longrightarrow algorithm \longrightarrow output$$

An algorithm is a sequence of computational that transform the input into the output, it describe a specific computational procedure for achieving the input/output relationship.

1.2 Instance of a problem

An instance of a problem is the input needed to compute a solution to the problem.

1.3 Correct algorithm

For every input instance, a correct algorithm halts out the corrent output.

1.4 Data structure

Data structure is a way to store and organize data in order to faciliate access and modifications. No single data structure works well for all purpose.

1.5 The core technique

Learning an algorithm is to learn the technique of algorithm design and analysis.

1.6 Algorithm efficiency

Computers are not infinitely fast and memory is not free, thus the efficiency of a algorithm matters.

1.7 Algorithms use information

There is information in the problem, the general is, the more information you use, the more efficiency the algorithm is. Data structure is a way to use this information.

1.8 Loop invariant

There can be infinite input cases, so it is hard to say an algorithm is corrent. Loop invariant is a way to guarantee that the algorithm is correct.

There are 3 elements in a loop invariant:

initialization It is true prior to the first iteration of the loop.

maintanance If it is true before an iteration of the loop, it remains true before the next iteration.

termination When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

1.9 Analyzing an algorithm

Analyzing an algorithm is to predict the resources consumed. The most important resources is time and space.

In general, the time grows with the size of the input, so it is traditional to describe the running time as the function of the size of its input.

1.9.1 Worst-case analysis

Becuase the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas. The reason to analyze worst-case running time is as follows:

1. It give an upper bound on the running time.
2. Worst case ocurrs fairly often.
3. The "average case" is often roughly as bad as the worst case.

1.10 Resource model

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. However, the focus is algorithm, not the tedious hardware detail. We shall assume a generic one-processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

1.11 Growth of functions

Althoght we can sometimes determine the exact running time of an algorithm, the extra procision is not usually worth the effort of computing it. When we look at input sizes large enought to make only the order of growth of the running time relevant, we are studying the **asymptotic efficiency** of algorithms.

1.11.1 Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constant } c_1, c_2 \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a memeber of $\Theta(g(n))$. However, we will usually write "$f(n) = \Theta(g(n))$"

to express the same notion.

Θ-notation indicates the function is bounded with a asymptotically tight upper bound and a asymptotically tight lower bound.

1.11.2 O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

O-notation indicates the function is bounded with a asymptotically tight upper bound.

1.11.3 Ω-notation

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Ω-notation indicates the function is bounded with a asymptotically tight lower bound.

1.11.4 Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.11.5 o-notation

$$O(g(n)) = \{f(n) : \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

or

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

o-notation indicates the function is bounded with a not asymptotically tight upper bound.

1.11.6 ω-notation

$$O(g(n)) = \{f(n) : \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq\leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

ω-nodation indicates the function is bounded with a not asymptotically tight lower bound.

Part I

Data Structure

Chapter 2

Array

2.1 What is an array?

An Array is a collection of items. The items are stored in neighboring (contiguous) memory locations. Thus elements can be accessed randomly since each element in the array can be identified by an array index.

For example in the Figure 2.1:

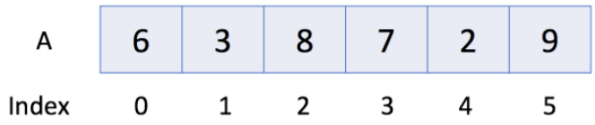

Figure 2.1: Array Index

In the above example, there are 6 elements in array A. That is to say, the length of A is 6. We can use A[0] to represent the first element in the array. Therefore, A[0] = 6. Similarly, A[1] = 3, A[2] = 8 and so on.

2.2 Capacity and length

The capacity is the items it can hold at most. It is specified when you create an array.

Length is the number of items it holds currently.

2.3 Operations

Array is a data structure, which means that it stores data in a specific format and supports certain operations on the data it stores.

There are three basic operations in array:

- insert
- delete
- search

2.4 String

A string is an array of characters.

2.5 Examples

2.5.1 Diagonal traverse

```
#!/usr/bin/env python3
"""
@project: leetcode
@file: 20210303_diagonal_traverse
@author: mike
@time: 2021/3/3

@function:
Given a matrix of M x N elements (M rows, N columns),
return all elements of the matrix in diagonal order as shown in the below image.



Example:

Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]

Output: [1,2,4,7,5,3,6,8,9]

Explanation:



Note:

The total number of elements of the given matrix will not exceed 10,000.
"""
from typing import List
import collections


```

```
37 class Solution:
38     def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:
39         dictionary = collections.defaultdict(list)
40         for i in range(len(matrix)):
41             for j in range(len(matrix[0])):
42                 dictionary[i + j].append(matrix[i][j])
43         print(dictionary)
44 
45         maximum = len(dictionary)
46         result = []
47         for i in range(maximum):
48             if i % 2 == 0:
49                 result.extend(list(reversed(dictionary[i])))
50             else:
51                 result.extend(dictionary[i])
52         return result
53 
54 
55 if __name__ == '__main__':
56     solution = Solution()
57     matrix = [
58         [1, 2, 3],
59         [4, 5, 6],
60         [7, 8, 9]
61     ]
62 
63     result = solution.findDiagonalOrder(matrix)
64     print(result)
```

There are two manners to solve the problem:

1. based on the procedure

2. based on overview properties

Chapter 3

Hash table

Hash Table is a data structure which organizes data using hash functions in order to support quick insertion and search.

There are two different kinds of hash tables:

hash set The hash set is one of the implementations of a set data structure to store no repeated values.

hash map The hash map is one of the implementations of a map data structure to store (key, value) pairs.

3.1 The principle of hash table

The key idea of Hash Table is to use a **hash function** to map **keys** to **buckets**. To be more specific,

1. When we insert a new key, the hash function will decide which bucket the key should be assigned and the key will be stored in the corresponding bucket;

2. When we want to search for a key, the hash table will use the same hash function to find the corresponding bucket and search only in the specific bucket.

3.2 Keys to design a hash table

There are two essential factors that you should pay attention to when you are going to design a hash table.

3.2.1 Hash function

The hash function is the most important component of a hash table which is used to map the key to a specific bucket. The hash function will depend on the **range of key values** and the **number of buckets**.

It is an open problem to design a hash function. The idea is to try to assign the key to the bucket **as uniform as you can**. Ideally, a perfect hash function will be a one-one mapping between the key and the bucket. However, in most cases a hash function is not perfect and it is a tradeoff between **the amount of buckets** and **the capacity of a bucket**.

3.2.2 Collision resolution

Ideally, if our hash function is a perfect one-one mapping, we will not need to handle collisions. Unfortunately, in most cases, collisions are almost inevitable. For instance, hash function (y = x % 5), both 1987 and 2 are assigned to bucket 2. That is a **collision**.

A collision resolution algorithm should solve the following questions:

1. How to organize the values in the same bucket?

2. What if too many values are assigned to the same bucket?

3. How to search a target value in a specific bucket?

These questions are related to **the capacity of the bucket** and **the number of keys** which might be mapped into **the same bucket** according to our hash function.

Let's assume that the bucket, which holds the maximum number of keys, has N keys.

Typically, if N is constant and small, we can simply use an array to store keys in the same bucket. If N is variable or large, we might need to use height-balanced binary search tree instead.

3.3 The principle of built-in hash table

The typical design of built-in hash table is:

1. The key can be any **hashable** type. And a key with belongs to a hashable type will have a **hashcode**. This code will be used in the mapping function to get the bucket index.

2. Each bucket contains **an array** to store all the values in the same bucket initially.

3. If there are two many values the same bucket, these values will be maintained in a **height-balanced binary search tree** instead.

The average time complexity of both insertion and search is still **O(1)**. And the time complexity in the worst case is **O(logN)** for both insertion and search by using height-balanced BST.

3.4 Designing the key

Sometimes you have to think it over to design a suitable key when using a hash table.

For example:

```
Given an array of strings, group anagrams together.
```

As we know, a hash map can perform really well in grouping information by key. But we cannot use the original string as key directly. We have to design a proper key to present the type of anagrams. For instance, there are two strings "eat" and "ate" which should be in the same group. While "eat" and "act" should not be grouped together.

When you design a key, you need to guarantee that:

1. All values belong to the same group will be mapped in the same group.

2. Values which needed to be separated into different groups will not be mapped into the same group.

This process is similar to design a hash function, but here is an essential difference. A hash function satisfies the first rule but might not satisfy the second one. But your mapping function should satisfy both of them.

3.4.1 Summary

Here are some takeaways about how to design the key for you:

1. When the order of each element in the string/array doesn't matter, you can use the **sorted string/array** as the key.
2. If you only care about the offset of each value, usually the offset from the first value, you can use the **offset** as the key.
3. In a tree, you might want to directly use the **TreeNode** as key sometimes. But in most cases, the **serialization of the subtree** might be a better idea.
4. In a matrix, you might want to use **the row index** or **the column index** as key.
5. In a Sudoku, you can combine the row index and the column index to identify which **block** this element blongs to.
6. Sometimes, in a matrix, you might want to aggregate the values in the same **diagonal line**. $(i, j) \to i + j$, $(i, j) \to i - j$

3.5 Conclusion

A typical thinking process to solve problems by hash table flexibly show in Figure 3.1:

What's more, we will meet more complicated problems sometimes. We might need to:

- use several hash tables together
- combine the hash table with other data structure
- combine the hash table with other algorithms
- ...

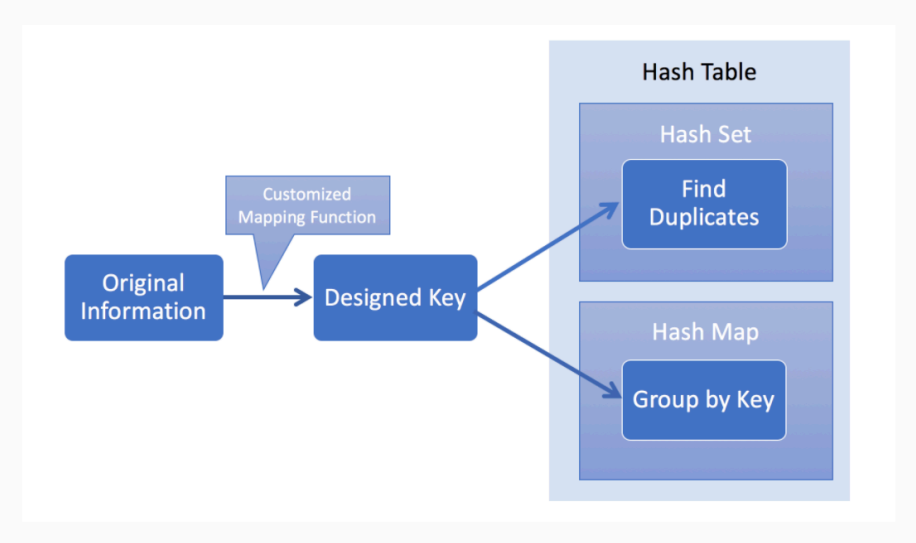

Figure 3.1: Thinking process by hash table

Part II

Algorithms

Chapter 4

Recursion

Recursion is an approach to solving problems using a function that calls itself as a subroutine.

The trick is that each time a recursive function calls itself, it reduces the given problem into subproblems. The recursion call continues until it reaches a point where the subproblem can be solved without further recursion.

A recursive function should have the following properties so that it does not result in an infinite loop:

1. A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer.

2. A set of rules, also known as recurrence relation that reduces all other cases towards the base case.

4.1 Memorization

Recursion is often an intuitive and powerful way to implement an algorithm. However, it might bring some undesired penalty to the performance, e.g. duplicate calculations, if we do not use it wisely. To eliminate the duplicate calculation, one of the ideas would be to store the intermediate results in the cache so that we could reuse them later without re-calculation. This idea is also known as **memoization**, which is a technique that is frequently used together with recursion. Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

4.2 Complexity analysis

4.2.1 Time complexity

Given a recursion algorithm, its time complexity $O(T)$ is typically the product of the number of recursion invocations (denoted as R) and the time complexity of calculation (denoted as $O(s)$) that incurs along with each recursion call:

$$O(T) = R \times O(s) \tag{4.1}$$

4.2.2 Space complexity

There are mainly two parts of the space consumption:

1. recursion related

2. non-recursion related space.

The recursion related space refers to the memory cost that is incurred directly by the recursion, i.e. the stack to keep track of recursive function calls. In order to complete a typical function call, the system allocates some space in the stack to hold three important pieces of information:

1. The returning address of the function call. Once the function call is completed, the program must know where to return to, i.e. the line of code after the function call.

2. The parameters that are passed to the function call.

3. The local variables within the function call.

The non-recursion related space refers to the memory space that is not directly related to recursion, which typically includes the space (normally in heap) that is allocated for the global variables.

Recursion or not, you might need to store the input of the problem as global variables, before any subsequent function calls. And you might need to save the intermediate results from the recursive calls as well. The latter is also known as memoization.

4.2.3 Tail recursion

Tail recursion is a recursion where the recursive call is the final instruction in the recursion function. And there should be only one recursive call in the function.

```
def sum_non_tail_recursion(ls):
    """
    :type ls: List[int]
    :rtype: int, the sum of the input list.
    """
    if len(ls) == 0:
        return 0

    # not a tail recursion because it does some computation after the recursive call returned.
    return ls[0] + sum_non_tail_recursion(ls[1:])


def sum_tail_recursion(ls):
    """
    :type ls: List[int]
    :rtype: int, the sum of the input list.
    """
    def helper(ls, acc):
        if len(ls) == 0:
            return acc
        # this is a tail recursion because the final instruction is a recursive call.
        return helper(ls[1:], ls[0] + acc)

    return helper(ls, 0)
```

The benefit of having tail recursion is that it could avoid the accumulation of stack overheads during the recursive calls, since the system could reuse a fixed amount space in the stack for each recursive call.

4.3 Routine

Here is the general workflow to solve a recursion problem:

1. Define the recursion function;
2. Write donw the recurrence relation and base case;
3. Use memeorization to eliminate the duplicate calculation problem, if it exists;
4. Whenever possible, implement the function as tail recursion, to optimize the space complexisty;

4.4 Devide and conquer

A divide-and-conquer algorithm works by recursively breaking the problem down into two or more subproblems of the same or related type, until these

subproblems become simple enough to be solved directly. Then one combines the results of subproblems to form the final solution.

As you can see, divide-and-conquer algorithm is naturally implemented in the form of recursion. Another subtle difference that tells a divide-and-conquer algorithm apart from other recursive algorithms is that we break the problem down into two or more subproblems in the divide-and-conquer algorithm, rather than a single smaller subproblem.

There are in general three steps that one can follow in order to solve the problem in a divide-and-conquer manner.

1. Divide. Divide the problem S into a set of subproblems: $\{S_1, S_2, \ldots, S_n\}$ when $n \geq 2$, i.e. there are usually more than one subproblems.

2. Conquer. Sovle each subproblems recursively.

3. Combine. Combine the results of each subproblem.

Pseudocode template (Python):

```
1  def divide_and_conquer( S ):
2      # (1). Divide the problem into a set of subproblems.
3      [S1, S2, ... Sn] = divide(S)
4
5      # (2). Solve the subproblem recursively,
6      # obtain the results of subproblems as [R1, R2... Rn].
7      rets = [divide_and_conquer(Si) for Si in [S1, S2, ... Sn]]
8      [R1, R2,... Rn] = rets
9
10     # (3). combine the results from the subproblems.
11     # and return the combined result.
12     return combine([R1, R2,... Rn])
```

4.5 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems (notably Constraint satisfaction problems or CSPs), which incrementally builds candidates to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution.

Conceptually, one can imagine the procedure of backtracking as the tree traversal. Starting from the root node, one sets out to search for solutions that are located at the leaf nodes. Each intermediate node represents a partial candidate solution that could potentially lead us to a final valid solution. At each node, we would fan out to move one step further to the final solution, i.e.

we iterate the child nodes of the current node. Once we can determine if a certain node cannot possibly lead to a valid solution, we abandon the current node and backtrack to its parent node to explore other possibilities. It is due to this backtracking behaviour, the backtracking algorithms are often much faster than the brute-force search algorithm, since it eliminates many unnecessary exploration.

4.5.1 Backtracking template

```
1  def backtrack(candidate):
2      if find_solution(candidate):
3          output(candidate)
4          return
5
6      # iterate all possible candidates.
7      for next_candidate in list_of_candidates:
8          if is_valid(next_candidate):
9              # try this partial candidate solution
10             place(next_candidate)
11             # given the candidate, explore further.
12             backtrack(next_candidate)
13             # backtrack
14             remove(next_candidate)
```

4.6 Unfold recursion

unfold the recursion: convert a recursion algorithm to non-recursion one.

Recursion could be an elegant and intuitive solution, when applied properly. Nevertheless, sometimes, one might have to convert a recursive algorithm to iterative one for various reasons:

Risk of stackoverflow The recursion often incurs additional memory consumption on the system stack. If not used properly, the recursion algorithm could lead to stackoverflow.

Efficiency The recursion could impose at least additional cost of function call, and in worse case duplicate calculation.

Complexity The nature of recursion is quite close to the mathematics, which is why the recursion appears to be more intuitive and comprehensive for many people. However, when we abuse the recursion, the recursive program could become more difficult to read and understand than the non-recursive one, e.g. nested recursion etc.

> We can always convert a recursion to iteration. In order to do so, in general, we use a data structure of stack or queue, which replaces the role of the system call stack during the process of recursion.

4.6.1 Example: same tree

Given two binary trees, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

recursive solution:

```
class Solution:
    def isSameTree(self, p, q):
        """
        :type p: TreeNode
        :type q: TreeNode
        :rtype: bool
        """
        # p and q are both None
        if not p and not q:
            return True
        # one of p and q is None
        if not q or not p:
            return False
        if p.val != q.val:
            return False
        return self.isSameTree(p.right, q.right) and \
               self.isSameTree(p.left, q.left)
```

iterative solution:

```
from collections import deque
class Solution:
    def isSameTree(self, p, q):
        """
        :type p: TreeNode
        :type q: TreeNode
        :rtype: bool
        """
        def check(p, q):
            # if both are None
            if not p and not q:
                return True
            # one of p and q is None
            if not q or not p:
                return False
            if p.val != q.val:
                return False
            return True

        deq = deque([(p, q),])
        while deq:
            p, q = deq.popleft()
            if not check(p, q):
                return False
            if p:
                deq.append((p.left, q.left))
                deq.append((p.right, q.right))
        return True
```

To convert a recursion approach to an iteration one, we could perform the following two steps:

1. We use a stack or queue data structure within the function, to replace the role of the system call stack. At each occurrence of recursion, we simply push the parameters as a new element into the data structure that we created, instead of invoking a recursion.

2. In addition, we create a loop over the data structure that we created before. The chain invocation of recursion would then be replaced with the iteration within the loop.

4.7 Divide and conquer vs. backtracking

1. Often the case, the divide-and-conquer problem has a sole solution, while the backtracking problem has unknown number of solutions. For example, when we apply the merge sort algorithm to sort a list, we obtain a single sorted list, while there are many solutions to place the queens for the N-queen problem.

2. Each step in the divide-and-conquer problem is indispensable to build the final solution, while many steps in backtracking problem might not be useful to build the solution, but serve as atttempts to search for the potential solutions. For example, each step in the merge sort algorithm, i.e. divide, conquer and combine, are all indispensable to build the final solution, while there are many trials and errors during the process of building solutions for the N-queen problem.

3. When building the solution in the divide-and-conquer algorithm, we have a clear and predefined path, though there might be several different manners to build the path. While in the backtracking problems, one does not know in advance the exact path to the solution. For example, in the top-down merge sort algorithm, we first recursively divide the problems into two subproblems and then combine the solutions of these subproblems. The steps are clearly defined and the number of steps is fixed as well. While in the N-queen problem, if we know exactly where to place the queens, it would only take N steps to do so. When applying the backtracking algorithm to the N-queen problem, we try many candidates and many of them do not eventually lead to a solution but abandoned at the end. As a

result, we do not know beforehand how many steps exactly it would take to build a valid solution.

4.8 Examples

4.8.1 Skyline

```
#!/usr/bin/env python3
"""
@project: leetcode
@file: 20210227_the_skyline
@author: mike
@time: 2021/2/27

@function:
A city's skyline is the outer contour of the silhouette formed by
all the buildings in that city when viewed from a distance.
Given the locations and heights of all the buildings,
return the skyline formed by these buildings collectively.

The geometric information of each building is given
in the array buildings where buildings[i] = [left_i, right_i, height_i]:

left_i is the x coordinate of the left edge of the ith building.
right_i is the x coordinate of the right edge of the ith building.
height_i is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points"
sorted by their x-coordinate in the form [[x_1,y_1],[x_2,y_2],...].
Each key point is the left endpoint of some horizontal segment in the skyline
except the last point in the list, which always has a y-coordinate 0 and
is used to mark the skyline's termination where the rightmost building ends.
Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline.
For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable;
the three lines of height 5 should be merged into one in the final output
as such: [...,[2 3],[4 5],[12 7],...]



Example 1:


Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Explanation:
Figure A shows the buildings of the input.
Figure B shows the skyline formed by those buildings.
The red points in figure B represent the key points in the output list.
Example 2:

Input: buildings = [[0,2,3],[2,5,3]]
Output: [[0,3],[5,0]]


Constraints:

1 <= buildings.length <= 10^4
0 <= left_i < right_i <= 2^31 - 1
1 <= height_i <= 2^31 - 1
buildings is sorted by left_i in non-decreasing order.
"""
from typing import List
```

```
60
61
62  class Solution:
63      def getSkyline(self, building: List[List[int]]) -> List[List[int]]:
64          # Base case
65          if not building:
66              return []
67          if len(building) == 1:
68              return [[building[0][0], building[0][2]], [building[0][1], 0]]
69
70          mid = len(building) // 2
71          left = self.getSkyline(building[:mid])
72          right = self.getSkyline(building[mid:])
73          return self.merge(left, right)
74
75      def merge(self, left, right):
76          h1, h2 = 0, 0
77          i, j = 0, 0
78          result = []
79
80          while i < len(left) and j < len(right):
81              if left[i][0] < right[j][0]: # x
82                  h1 = left[i][1] # height
83                  corner = left[i][0] # x
84                  i += 1
85              elif right[j][0] < left[i][0]:
86                  h2 = right[j][1]
87                  corner = right[j][0]
88                  j += 1
89              else:
90                  h1 = left[i][1]
91                  h2 = right[j][1]
92                  corner = right[j][0]
93                  i += 1
94                  j += 1
95
96              if self.is_valid(result, max(h1, h2)):
97                  result.append([corner, max(h1, h2)])
98          result.extend(right[j:])
99          result.extend(left[i:])
100         return result
101
102     def is_valid(self, result, new_height):
103         return not result or result[-1][1] != new_height
104
105
106 if __name__ == '__main__':
107     solutino = Solution()
108     buildings = [[0, 2, 3], [2, 5, 3]]
109     buildings = [[2, 9, 10], [3, 7, 15], [5, 12, 12], [15, 20, 10], [19, 24, 8]]
110     solutino.getSkyline(buildings)
```

There are only exact one answer, so this is may be a divide and conquer problem, not the backtrack problem.

For the first time, this problem seems very hard if we combine several buildings at a time. This can be simplified by combining only two buildings at a time. The rule of combiningg buildings can be simplified if we convert the building into only two points.

Chapter 5

Binary search

Binary Search is one of the most fundamental and useful algorithms in Computer Science. It describes the process of searching for a specific value in an ordered collection.

In its simplest form, Binary Search operates on a contiguous sequence with a specified left and right index. This is called the Search Space. Binary Search maintains the left, right, and middle indicies of the search space and compares the search target or applies the search condition to the middle value of the collection; if the condition is unsatisfied or values unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with an empty half, the condition cannot be fulfilled and target is not found.

5.1 Thress parts of a successful binary search

Binary Search is generally composed of 3 main sections:

Pre-processing - Sort if collection is unsorted.

Binary Search - Using a loop or recursion to divide search space in half after each comparison.

Post-processing - Determine viable candidates in the remaining space.