20240227-Week2-时间复杂度

Updated 2145 GMT+8 Feb 24, 2024

2024 spring, Complied by Hongfei Yan

说明：平行班，笔试占比较大，所以，每次课件增加“笔试题目”部分。

TODO本周发布作业：

Assignment2, https://github.com/GMyhf/2024spring-cs201

作业评分标准

标准	等级	得分
按时提交	1 得分提交，0.5 得分请假，0 得分未提交	1 分
源码、耗时（可选）、解题思路（可选）	1 得分4或4+题目，0.5 得分2或2+题目，0 得分无源码	1 分
AC代码截图	1 得分4或4+题目，0.5 得分2或2+题目，0 得分无截图	1 分
清晰头像、pdf、md/doc	1 得分三项全，0.5 得分有二项，0 得分少于二项	1 分
学习总结和收获	1 得分有，0 得分无	1 分
总得分：		5 ，满分 5

前言

https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=outind

What is DSA?

DSA is defined as a combination of two separate yet interrelated topics – Data Structure and Algorithms. DSA is one of the most important skills that every computer science student must have. It is often seen that people with good knowledge of these technologies are better programmers than others and thus, crack the interviews of almost every tech giant.

What is Data Structure?

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

What is Algorithm?

Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

How to start learning DSA?

The first and foremost thing is dividing the total procedure into little pieces which need to be done sequentially. The complete process to learn DSA from scratch can be broken into 4 parts:

1. Learn about Time and Space complexities
2. Learn the basics of individual Data Structures
3. Learn the basics of Algorithms
4. Practice Problems on DSA

1. Learn about Complexities

Here comes one of the interesting and important topics. The primary motive to use DSA is to solve a problem effectively and efficiently. How can you decide if a program written by you is efficient or not? This is measured by complexities. Complexity is of two types:

1. Time Complexity: Time complexity is used to measure the amount of time required to execute the code.
2. Space Complexity: Space complexity means the amount of space required to execute successfully the functionalities of the code. You will also come across the term **Auxiliary Space** very commonly in DSA, which refers to the extra space used in the program other than the input data structure.

Both of the above complexities are measured with respect to the input parameters. But here arises a problem. The time required for executing a code depends on several factors, such as:

- The number of operations performed in the program,
- The speed of the device, and also
- The speed of data transfer if being executed on an online platform.

So how can we determine which one is efficient? The answer is the use of asymptotic notation.

> **Asymptotic notation** is a mathematical tool that calculates the required time in terms of input size and does not require the execution of the code.

It neglects the system-dependent constants and is related to only the number of modular operations being performed in the whole program. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

- **Big-O Notation (Ο)** – Big-O notation specifically describes the worst-case scenario.
- **Omega Notation (Ω)** – Omega(Ω) notation specifically describes the best-case scenario.
- **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.

Rate of Growth of Algorithms

The most used notation in the analysis of a code is the **Big O Notation** which gives an upper bound of the running time of the code (or the amount of memory used in terms of input size).

To learn about complexity analysis in detail, you can refer to articles on the **Analysis of Algorithms**.

3. Learn Algorithms

Once you have cleared the concepts of Data Structures, now its time to start your journey through the Algorithms. Based on the type of nature and usage, the Algorithms are grouped together into several categories, as shown below:

Sorting Algorithm

Here is one other most used algorithm. Often we need to arrange or sort data as per a specific condition. The sorting algorithm is the one that is used in these cases. Based on conditions we can sort a set of homogeneous data in order like sorting an array in increasing or decreasing order.

一、Big-O notation

https://stackoverflow.com/questions/44421828/is-this-sieve-really-on

https://www.geeksforgeeks.org/sieve-eratosthenes-0n-time-complexity/

Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

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. For most of this book, we shall assume a generic oneprocessor, **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. Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we couldsort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by c lg n bits for some constant $c \ge 1$. We require $c \ge 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

> 计算机字长（Computer Word Length）是指计算机中用于存储和处理数据的基本单位的位数。它表示计算机能够一次性处理的二进制数据的位数。
>
> 字长的大小对计算机的性能和数据处理能力有重要影响。较大的字长通常意味着计算机能够处理更大范围的数据或执行更复杂的操作。常见的字长包括 8 位、16 位、32 位和 64 位。
>
> 较小的字长可以节省存储空间和资源，适用于简单的计算任务和资源有限的设备。较大的字长通常用于处理更大量级的数据、进行复杂的计算和支持高性能计算需求。
>
> 需要注意的是，字长并不是唯一衡量计算机性能的指标，还有其他因素如处理器速度、缓存大小、操作系统等也会影响计算机的整体性能。

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constanttime instruction? In the general case, no; it takes several instructions to compute xy when x and y are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a “shift left” instruction, which in constant time shifts the bits of an integer by k positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by k positions to the left is equivalent to multiplication by 2^k. Therefore, such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of 2^k as a constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because 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.

Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

1.Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTIONSORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for **input size** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time ci, where ci is a constant. This viewpoint is in keeping with the **RAM** model, and it also reflects how the pseudocode would be implemented on most actual computers.

In the following discussion, our expression for the running time of INSERTIONSORT will evolve from a messy formula that uses all the statement costs ci to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each j = 1, 3, ... , n-1, where n = len(A), we let $t_j$ denote the number of times the **while** loop test in line 8 is executed for that value of j . When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

> https://www.geeksforgeeks.org/insertion-sort/
>
> **Insertion sort** is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.
>
> ## Insertion Sort Algorithm
>
> To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.
>
> Illustrations:

Implementation of Insertion Sort Algorithm

```
def insertionSort(A):                                                                       # cost
    for j in range(1, len(A)):                                  # c1                  n
        key = A[j]

        # Insert A[i] into the
        # sorted sequence A[0..j-1]             # 0                                   n - 1
        i = j - 1
        while i >= 0 and A[i] > key:    # c5                \sum_{j=2}^{n} t_j
            A[i + 1] = A[i]                                                   # c6
            i -= 1
        A[i + 1] = key                                                                # c8


arr = [12, 11, 13, 5, 6]
insertionSort(arr)
```

```
print(' '.join(map(str, arr)))

# Output: 5 6 11 12 13
# Time Complexity: O(N^2)
# Auxiliary Space: O(1)
```

“\sum_{j=2}^{n} t_j” 是

$\sum_{j=2}^{n} t_j$ 的LaTex表示，

“\sum_{j=2}^{n} t_{j}-1\” 是 $\sum_{j=2}^{n} t_{j}-1$ 的LaTex表示。

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes $c_i$ steps to execute and executes n times will contribute $c_in$ to the total running time. To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining

$T(n) = c_1n + c_2n(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n} t_j-1 + c_7\sum_{j=2}^{n} t_j-1 + c_8(n-1)$

Even for inputs of a given size, an algorithm’s running time may depend on which input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each j = 1, 3, ... , n-1, we then find that $A[i] \le key$ in line 8 when $i$ has its initial value of $j - 1$. Thus $t_j = 1$ for j = 1, 3, ... , n-1, and the best-case running time is

$T(n) = c_1n + c_2n(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

$\quad = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

We can express this running time as `an + b` for constants a and b that depend on the statement costs $c_i$; it is thus a **linear function** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element A[j] with each element in the entire sorted subarray A[0..j-1], and so tj = j for j = 1, 2, ..., n-1. Noting that

$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

$\sum_{j=2}^{n} j-1 = \frac{n(n-1)}{2}$

we find that in the worst case, the running time of INSERTION-SORT is

$T(n) = c_1n + c_2n(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} -1) + c_6(\frac{n(n-1)}{2}) + + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$

$\quad = (\frac{c_5}2 + \frac{c_6}2 + \frac{c_7}2)n^2 + (c_1 + c_2 + c_4 + \frac{c_5}2 - \frac{c_6}2 - \frac{c_7}2 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs ci; it is thus a **quadratic function** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting “randomized” algorithms whose behavior can vary even for a fixed input.

2.Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the **worst-case running time**, that is, the longest running time for any input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm’s worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.
- The “average case” is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[0 .. j-1] to insert element A[j] ? On average, half the elements in A[0 .. j-1] are less than A[j] , and half the elements are greater. On average, therefore, we check half of the subarray A[0 .. j-1], and so t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the **average-case** running time of an algorithm; we shall see the technique of **probabilistic analysis** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a **randomized algorithm**, which makes random choices, to allow a probabilistic analysis and yield an **expected** running time.

3.Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTIONSORT procedure. First, we ignored the actual cost of each statement, using the constants ci to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction: it is the **rate of growth**, or **order of growth**, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term’s constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared").

We usually consider one algorithm to be more efficient than another if its worstcase running time has a lower order of growth. Due to constant factors and lowerorder terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

4.O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n) (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

$O(g(n) = \{f(n): there \space exist \space positive \space constants \space c \space and \space n_0 \space such \space that$

$\quad\quad\quad\quad\quad\quad\quad 0 \le f(n) \le cg(n) for \space all \space n \ge n0\}$

We use O-notation to give an upper bound on a function, to within a constant factor.

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm immediately yields an $O(n^2)$ upper bound on the worst-case running time.

Since O-notation describes an upper bound, when we use it to bound the worstcase running time of an algorithm, we have a bound on the running time of the algorithm on every input.

Sorting Algorithm

Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

There are a lot of different types of sorting algorithms. Some widely used algorithms are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- ShellSort

There are several other sorting algorithms also and they are beneficial in different cases. You can learn about them and more in our dedicated article on Sorting algorithms.

1.Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm

> In Bubble Sort algorithm,
>
> - traverse from left and compare adjacent elements and the higher one is placed at right side.
> - In this way, the largest element is moved to the rightmost end at first.
> - This process is then continued to find the second largest and place it and so on until the data is sorted.

```python
# Optimized Python program for implementation of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break


# Driver code to test above
if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]

    bubbleSort(arr)
    print(' '.join(map(str, arr)))
```

Complexity Analysis of Bubble Sort:

Time Complexity: O(N2) Auxiliary Space: O(1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Some FAQs related to Bubble Sort:

Q1. What is the Boundary Case for Bubble sort?

Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(N2) time complexity.

Q2. Does sorting happen in place in Bubble sort?

Yes, Bubble sort performs the swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

Q3. Is the Bubble sort algorithm stable?

Yes, the bubble sort algorithm is stable.

Q4. Where is the Bubble sort algorithm used?

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

2.Selection Sort

> **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

```
A = [64, 25, 12, 22, 11]
```

```
# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

        # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print(' '.join(map(str, A)))

# Output: 11 12 22 25 64
```

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is **O(N2)** as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Frequently Asked Questions on Selection Sort

Q1. Is Selection Sort Algorithm stable?

The default implementation of the Selection Sort Algorithm is **not stable**. However, it can be made stable. Please see the stable Selection Sort for details.

Q2. Is Selection Sort Algorithm in-place?

Yes, Selection Sort Algorithm is an in-place algorithm, as it does not require extra space.

3.Quick Sort

> quickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

> The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
>
> Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

```
# Function to find the partition position
def partition(array, low, high):

        # Choose the rightmost element as pivot
        pivot = array[high]

        # Pointer for greater element
        i = low - 1

        # Traverse through all elements
        # compare each element with pivot
        for j in range(low, high):
                if array[j] <= pivot:

                        # If element smaller than pivot is found
                        # swap it with the greater element pointed by i
                        i = i + 1

                        # Swapping element at i with element at j
                        (array[i], array[j]) = (array[j], array[i])

        # Swap the pivot element with
        # the greater element specified by i
        (array[i + 1], array[high]) = (array[high], array[i + 1])

        # Return the position from where partition is done
        return i + 1


def quicksort(array, low, high):
        if low < high:

                # Find pivot element such that
                # element smaller than pivot are on the left
                # element greater than pivot are on the right
                pi = partition(array, low, high)

                # Recursive call on the left of pivot
                quicksort(array, low, pi - 1)

                # Recursive call on the right of pivot
                quicksort(array, pi + 1, high)


if __name__ == '__main__':
        array = [10, 7, 8, 9, 1, 5]
        N = len(array)

        quicksort(array, 0, N - 1)
        for x in array:
                print(x, end=" ")

# Output: 1 5 7 8 9 10
```

Complexity Analysis of Quick Sort:

Time Complexity:

- Best Case: $\Omega(N log (N))$ The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves. In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- Average Case: $\Theta (N log (N))$ Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- Worst Case: $O(N^2)$ The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Auxiliary Space: O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.

- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

4.Merge Sort

> Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

How does Merge Sort work?

> Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

```
def mergeSort(arr):
        if len(arr) > 1:
                mid = len(arr)//2

                L = arr[:mid]   # Dividing the array elements
                R = arr[mid:] # Into 2 halves

                mergeSort(L) # Sorting the first half
                mergeSort(R) # Sorting the second half

                i = j = k = 0
                # Copy data to temp arrays L[] and R[]
                while i < len(L) and j < len(R):
                        if L[i] <= R[j]:
                                arr[k] = L[i]
                                i += 1
                        else:
                                arr[k] = R[j]
                                j += 1
                        k += 1

                # Checking if any element was left
                while i < len(L):
                        arr[k] = L[i]
                        i += 1
                        k += 1

                while j < len(R):
                        arr[k] = R[j]
                        j += 1
                        k += 1


if __name__ == '__main__':
        arr = [12, 11, 13, 5, 6, 7]
        mergeSort(arr)
        print(' '.join(map(str, arr)))
# Output: 5 6 7 11 12 13
```

Complexity Analysis of Merge Sort

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

> $T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

- Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem: Inversion Count for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

Advantages of Merge Sort:

- Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

5.Shell Sort

Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.

Algorithm:

Step 1 – Start Step 2 – Initialize the value of gap size. Example: h Step 3 – Divide the list into smaller sub-part. Each must have equal intervals to h Step 4 – Sort these sub-lists using insertion sort Step 5 – Repeat this step 2 until the list is sorted. Step 6 – Print a sorted list. Step 7 – Stop.

```
def shellSort(arr, n):
    # code here
    gap = n // 2

    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap  # This will keep help in maintain gap value

            while i >= 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:

                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap  # To check left side also
            # If the element present is greater than current element
            j += 1
        gap = gap // 2


# driver to check the code
arr2 = [12, 34, 54, 2, 3]

shellSort(arr2, len(arr2))
```

```
print(' '.join(map(str, arr2)))

# Output: 2 3 12 34 54
```

Time Complexity: Time complexity of the above implementation of Shell sort is $O(n^2)$. In the above implementation, the gap is reduced by half in every iteration. There are many other ways to reduce gaps which leads to better time complexity. See this for more details.

Worst Case Complexity The worst-case complexity for shell sort is $O(n^2)$

Shell Sort Applications

1. Replacement for insertion sort, where it takes a long time to complete a given task.
2. To call stack overhead we use shell sort.
3. when recursion exceeds a particular limit we use shell sort.
4. For medium to large-sized datasets.
5. In insertion sort to reduce the number of operations.

https://en.wikipedia.org/wiki/Shellsort

The running time of Shellsort is heavily dependent on the gap sequence it uses. For many practical variants, determining their time complexity remains an open problem.

Unlike insertion sort, Shellsort is not a stable sort since gapped insertions transport equal elements past one another and thus lose their original order. It is an adaptive sorting algorithm in that it executes faster when the input is partially sorted.

Stable sort algorithms sort equal elements in the same order that they appear in the input.

6.Comparison sorts

Below is a table of comparison sorts. A comparison sort cannot perform better than $O(n \log n)$ on average.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Merge sort	$nlogn$	$nlogn$	$nlogn$	*n*	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Timsort	*n*	$nlogn$	$nlogn$	*n*	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	*n*	n^2	n^2	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.
Bubble sort	*n*	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
							Sort instead of swapping the two items.

Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heapsort), used (in variant forms) in some C++ sort implementations and in .NET.

二、编程题目

02810: 完美立方

bruteforce, http://cs101.openjudge.cn/practice/02810

02808: 校门外的树

implementation, http://cs101.openjudge.cn/practice/02808

04146: 数字方格

math, http://cs101.openjudge.cn/practice/04146

230B. T-primes

binary search/implementation/math/number theory, 1300

http://codeforces.com/problemset/problem/230/B

18176: 2050年成绩计算

http://cs101.openjudge.cn/2024sp_routine/18176/

作业

三、笔试题目

选择（30分，每题2分）

Q: 下列不影响算法时间复杂性的因素有（ C ）。 A：问题的规模 B：输入值 C：计算结果 D：算法的策略

Q: 有 $n^2$ 个整数，找到其中最小整数需要比较次数至少为（ C ）次。

A:$n$ B: $log_{2}{n}$ C:$n^2-1$ D:$n-1$

解释：假设有 $n^2$ 个整数，我们需要找到其中的最小整数。在最坏的情况下，最小整数可能位于数组的最后一个位置，因此我们需要比较 $n^2 - 1$ 次才能确定最小整数。

具体地说，我们可以进行以下步骤来找到最小整数：

1. 假设第一个整数为当前的最小整数。
2. 依次比较当前最小整数和数组中的其他整数。
3. 如果遇到比当前最小整数更小的整数，将其设为当前最小整数。
4. 重复步骤 2 和 3，直到遍历完所有的 $n^2$ 个整数。

在这个过程中，我们需要进行 $n^2 - 1$ 次比较才能找到最小整数。这是因为第一个整数不需要与自身进行比较，而后续的每个整数都需要与之前的最小整数进行比较。

Q: 用 Huffman 算法构造一个最优二叉编码树，待编码的字符权值分别为{3，4，5，6，8，9，11，12}，请问该最优二叉编码树的带权外部路径长度为（ B ）。（补充说明：树的带权外部路径长度定义为树中所有叶子结点的带权路径长度之和；其中，结点的带权路径长度定义为该结点到树根之间的路径长度与该结点权值的乘积）
A：58 B：169 C：72 D：18

解释：为了构造哈夫曼树，我们遵循一个重复的选择过程，每次选择两个最小的权值创建一个新的节点，直到只剩下一个节点为止。我们可以按照以下步骤操作：

1. 将给定的权值排序：{3, 4, 5, 6, 8, 9, 11, 12}。
2. 选择两个最小的权值：3 和 4，将它们组合成一个新的权值为 7 的节点。
   现在权值变为：{5, 6, 7, 8, 9, 11, 12}。
3. 再次选择两个最小的权值：5 和 6，将它们组合成一个新的权值为 11 的节点。
   现在权值变为：{7, 8, 9, 11, 11, 12}。
4. 选择两个最小的权值：7 和 8，将它们组合成一个新的权值为 15 的节点。
   现在权值变为：{9, 11, 11, 12, 15}。
5. 选择两个最小的权值：9 和 11，将它们合并成一个新的权值为 20 的节点。
   现在权值变为：{11, 12, 15, 20}。
6. 选择两个最小的权值：11 和 12，合并成一个新的权值为 23 的节点。

现在权值变为：{15, 20, 23}。
7. 选择两个最小的权值：15 和 20，合并成一个新的权值为 35 的节点。
   现在权值变为：{23, 35}。
8. 最后，合并这两个节点得到根节点，权值为 23 + 35 = 58。

现在我们可以计算哈夫曼树的带权外部路径长度（WPL）。

```
            (58)
          /      \
       (23)       (35)
       /  \       /   \
     (11)(12)  (20)    (15)
               / \       / \
              (9)(11)   (7)(8)
                  / \   / \
                 (5)(6)(3) (4)
```

现在让我们计算每个叶子节点的带权路径长度：

- 权值 3 的节点路径长度为 4，WPL部分为 3 * 4 = 12。
- 权值 4 的节点路径长度为 4，WPL部分为 4 * 4 = 16。
- 权值 5 的节点路径长度为 4，WPL部分为 5 * 4 = 20。
- 权值 6 的节点路径长度为 4，WPL部分为 6 * 4 = 24。
- 权值 9 的节点路径长度为 3，WPL部分为 9 * 3 = 27。
- 权值 11（左侧）的节点路径长度为 3，WPL部分为 8 * 3 = 24。
- 权值 11（右侧）的节点路径长度为 2，WPL部分为 11 * 2 = 22。
- 权值 12 的节点路径长度为 2，WPL部分为 12 * 2 = 24。

将所有部分的 WPL 相加，我们得到整棵哈夫曼树的 WPL：

WPL = 12 + 16 + 20 + 24 + 27 + 24 + 22 + 24 = 169

Q：假设需要对存储开销 1GB (GigaBytes) 的数据进行排序，但主存储器（RAM）当前可用的存储空间只有 100MB (MegaBytes)。针对这种情况，（ B ）排序算法是最适合的。 A：堆排序 B：归并排序 C：快速排序 D：插入排序

解释：对于这种情况，最适合的排序算法是归并排序（B）。

归并排序是一种外部排序算法，它的主要思想是将数据分成较小的块，然后逐步合并这些块以获得有序的结果。由于主存储器的可用空间有限，归并排序非常适合这种情况，因为它可以在有限的主存中进行部分排序，并将排序好的部分写入外部存储（磁盘）中。然后再将不同部分进行合并，直到得到完全排序的结果。

堆排序（A）通常需要对整个数据集进行排序，因此不适合主存储器有限的情况。

快速排序（C）通常是一种原地排序算法，它需要频繁地交换数据，这可能导致频繁的磁盘访问，不适合主存储器有限的情况。

插入排序（D）的时间复杂度较低，但它需要频繁地移动数据，这可能导致频繁的磁盘访问，也不适合主存储器有限的情况。

Q: 若按照排序的稳定性和不稳定性对排序算法进行分类，则（ D ）是不稳定排序。 A：冒泡排序 B：归并排序 C：直接插入排序 D：希尔排序

解释：根据排序算法的稳定性，如果需要选择一个不稳定排序算法，选项D：希尔排序是正确的选项。

稳定排序算法是指，当有两个相等的元素A和B，且在排序前A出现在B的前面，在排序后A仍然会出现在B的前面。而不稳定排序算法则无法保证这种相对顺序。

冒泡排序（A）和直接插入排序（C）都属于稳定排序算法，它们在比较和交换元素时会考虑相等元素的顺序关系。

归并排序（B）是一种稳定排序算法，它通过分治的思想将待排序的序列划分为较小的子序列，然后逐步合并这些子序列并保持相对顺序。

希尔排序（D）是一种不稳定排序算法，它使用间隔序列来对数据进行分组，然后对每个分组进行插入排序。在插入排序的过程中，相等元素的顺序可能会发生变化。

Q: 以下（ C ）分组中的两个排序算法的最坏情况下时间复杂度的大 O 表示相同。 A：快速排序和堆排序 B：归并排序和插入排序 C：快速排序和选择排序 D：堆排序和冒泡排序

解释：选项C：快速排序和选择排序中的两个排序算法的最坏情况下时间复杂度的大 O 表示相同。

快速排序和选择排序都属于不同的排序算法，但它们的最坏情况下的时间复杂度都是O(n^2)。

快速排序的最坏情况下时间复杂度发生在每次选择的基准元素都划分出了一个很小的子序列，使得递归的深度达到了n，导致时间复杂度为O(n^2)。

选择排序的最坏情况下时间复杂度发生在每次选择最小（或最大）元素时，需要遍历未排序部分的所有元素，导致时间复杂度为O(n^2)。

Q: 给定一个 N 个相异元素构成的有序数列，设计一个递归算法实现数列的二分查找，考察递归过程中栈的使用情况，请问这样一个递归调用栈的最小容量应为（ C ）。 A：N B：N/2 C：$\lceil \log_{2}(N) \rceil$ D：$\lceil \log_{2}(N+1) \rceil$

解释：对于二分查找的递归实现，每次递归调用都会将问题规模减半，因此递归的深度就是问题规模的对数级别。在最坏情况下，递归的深度达到log2(N)。每次递归调用会占用栈空间，而栈的使用情况可以通过递归调用的最大深度来估计。因此，递归调用栈的最小容量应为最大递归深度的值。

根据给定的有序数列，共有N个相异元素，二分查找的递归深度为log2(N)。但是栈的容量必须能够容纳递归深度的最大值，所以栈的最小容量应为「上取整」，答案是C。

Q: 数据结构有三个基本要素:逻辑结构、存储结构以及基于结构定义的行为(运算)。下列概念中(B)属于存储结构。 A:线性表 B:链表 C:字符串 D:二叉树

解释：在这些选项中，有些描述的是数据的逻辑结构，而有些是存储结构。逻辑结构指的是数据对象中数据元素之间的相互关系，而存储结构是指数据结构在计算机中的表示（也就是内存中的存储形式）。

A: 线性表 - 这是一种逻辑结构，它描述元素按线性顺序排列的规则。 B: 链表 - 这是一种存储结构，它是线性表的链式存储方式，通过节点的相互链接来实现。 C: 字符串 - 这通常指的是一种逻辑结构，是一系列字符的集合。 D: 二叉树 - 这是一种逻辑结构，它描述每个节点最多有两个子节点的树状结构。

正确答案是 B: 链表，因为它指的是数据的物理存储方式，即内存中的链式存储结构。

Q: 回溯法是一类广泛使用的算法，以下叙述中不正确的是（ C ）。 A：回溯法可以系统地搜索一个问题的所有解或者任意解 B：回溯法是一种既具备系统性又具备跳跃性的搜索算法 C：回溯算法需要借助队列数据结构来保存从根结点到当前扩展结点的路径 D：回溯算法在生成解空间的任一结点时，先判断当前结点是否可能包含问题的有效解，如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向祖先结点回溯

解释：选项C：回溯算法需要借助队列数据结构来保存从根节点到当前扩展节点的路径 是不正确的。

回溯算法通常使用递归来实现，而不是使用队列来保存路径。在回溯算法中，通过递归调用，在搜索过程中会不断地向下扩展当前节点，并在需要的时候回溯到上一层节点。

回溯算法的特点是在搜索过程中具有跳跃性，即可以根据问题的特点进行剪枝或跳过某些无效的搜索路径，以提高效率。选项B正确地描述了回溯算法的这一特点。

Q：假设线性表中每个元素有两个数据项 key1 和 key2，现对线性表按以下规则进行排序：先根据数据项 key1 的值进行非递减排序；在 key1 值相同的情况下，再根据数据项 key2 的值进行非递减排序。满足这种要求的排序方法是（ A ）。 A: 先按 key1 值进行冒泡排序，再按 key2 值进行直接选择排序 B: 先按 key2 值进行冒泡排序，再按 key1 值进行直接选择排序 C: 先按 key1 值进行直接选择排序，再按 key2 值进行冒泡排序 D: 先按 key2 值进行直接选择排序，再按 key1 值进行冒泡排序

解释：想要满足题目中所述的排序要求，需要使用稳定的排序算法。所谓稳定排序算法，是指排序过程中不会改变相同排序键值的元素之间的相对顺序。

在这个问题中，我们需要确保在按 `key1` 排序后，具有相同 `key1` 值的元素按 `key2` 的顺序排列。如果我们首先按 `key2` 排序，然后再按 `key1` 排序，我们必须使用稳定排序算法来保持 `key2` 的顺序。但是，如果我们先按 `key1` 排序，再按 `key2` 排序，即使 `key2` 的排序算法是不稳定的，也不会影响 `key1` 的排序结果。

在给定的选项中：

- 冒泡排序是稳定的。
- 直接选择排序通常是不稳定的。

根据以上信息，我们可以排除所有先使用直接选择排序的选项（因为它可能会破坏 `key1` 相等元素的相对顺序）。

唯一符合排序要求的方法是：

1. 首先按 `key1` 进行排序，这时可以使用任何排序算法，因为还没有其他排序顺序需要保持。

2. 然后按 `key2` 进行稳定排序，以确保 `key1` 相同的元素按 `key2` 保持正确的顺序。

因此，正确答案是：

A: 先按 `key1` 值进行冒泡排序，再按 `key2` 值进行直接选择排序

但这里需要注意的是，直接选择排序通常是不稳定的，这在某些情况下可能不会保持 `key1` 相等元素的 `key2` 排序。因此，理论上我们应该选择两次稳定排序，比如两次冒泡排序或者其他稳定排序的组合。但是，按照给出的选项，A是最接近正确的答案。如果这些选项是从一个考试或测验中提取的，并且没有提供稳定排序的组合，那么A可能是这个特定问题的“正确”答案，尽管在现实世界的排序中，我们通常不会这样做。

Q：下列选项中最适合实现数据的频繁增删及高效查找的组织结构是（ C ）。 A： 有序表 B： 堆排序 C： 二叉排序树 D： 快速排序

解释：在这些选项中，要实现数据的频繁增删及高效查找，我们应该选择一种数据结构，而不是排序算法。排序算法（如堆排序和快速排序）主要是对数据进行排序，并不提供数据结构的功能。

让我们分析给出的选项：

A：有序表 **-** 有序表可以提供高效的查找操作（如二分查找），但在有序数组中插入和删除操作可能会很慢，因为它们可能需要移动元素来维护顺序。

B：堆排序 **-** 这是一种排序算法，并不是一种数据结构。它使用的堆结构可以快速找到最大或最小元素，但不适合频繁的任意增删操作。

C：二叉排序树（也称为二叉搜索树）**-** 这是一种数据结构，对于增删查操作都可以提供对数时间复杂度的性能（即 O(log n)），这是基于树是平衡的假设。如果树保持平衡（如使用AVL树或红黑树），那么它可以提供高效的增删查操作。

D：快速排序 **-** 这也是一种排序算法，不是数据结构，不适合频繁的增删操作。

因此，最适合实现数据的频繁增删及高效查找的组织结构是：C：二叉排序树

如果是在一个平衡的二叉搜索树中，如 AVL 树或红黑树，那么插入、删除和查找操作的平均时间复杂度都是 O(log n)，这是非常高效的。

Q: 给定排序列表 [21,1,26,45,29,28,2,9,16,49,39,27,43,34,46,40]，在归并排序的第 3 次递归调用时，排序的是哪个子表？（ B ）

A： [16, 49,39,27,43,34,46,40] B： [21,1] C： [21,1,26,45] D：[21]

Q: 排序数据同上，归并排序中，哪两个子表是最先归并的?（ C ）

A： [21,1] and [26,45] B：[1, 2, 9,21,26,28,29,45] and [16,27,34,39,40,43, 46,49]

C： [21] and [1] D：[9] and [16]

判断（10分，每题1分）

对填写"Y"，错填写"N"

Q: （Y）考虑一个长度为 n 的顺序表中各个位置插入新元素的概率是相同的，则顺序表的插入算法平均时间复杂度为 $O(n) $。

Q: （N）希尔排序算法的每一趟都要调用一次或多次直接插入排序算法，所以其效率比直接插入排序算法差。

解释：实际上，希尔排序算法的效率通常比直接插入排序算法要高。虽然希尔排序确实在每一趟中使用了直接插入排序算法，但是这里有一个重要的区别：希尔排序在初期趟次使用较大的间隔（增量）对数据进行部分排序，这有助于快速减少大量的逆序对，从而将一些已经比较接近最终位置的元素更快地移动到相应位置。

随着算法的进行，这个间隔会逐渐减小，直到最后一趟的间隔为1，此时整个数组已经接近于有序，并且大多数元素已经处于或接近其最终位置。在这种情况下，直接插入排序的效率很高，因为它在几乎已排序的数组上运行时效率非常好。

希尔排序的主要优点在于通过初期的较大增量排序，它能够移动距离较远的元素，并且随着执行过程中增量的减小，数组变得越来越接近于有序，这减少了直接插入排序在最后几趟中的比较和移动次数。

简而言之，希尔排序算法通过允许非相邻元素的比较和交换，从而在大规模的数据移动方面比直接插入排序更加高效。这使得希尔排序的平均时间复杂度比直接插入排序的时间复杂度要低（直接插入排序的平均时间复杂度为 $O(n^2)$，而希尔排序的时间复杂度较低，但其确切的时间复杂度尚未解决，依赖于增量序列的选择）。因此，希尔排序在处理大量数据时通常比直接插入排序更加高效。

Q: （Y）直接插入排序、冒泡排序、 希尔排序都是在数据正序的情况下比数据在逆序的情况下要快。

解释：在排序算法分析中，通常讨论最好、平均和最坏情况的时间复杂度。对于直接插入排序和冒泡排序，数据的初始顺序确实会影响它们的执行速度：

1. **直接插入排序**：在最好的情况下，即数据已经是正序时，直接插入排序只需要进行 n-1 次比较和 0 次交换，这时的时间复杂度是 O(n)。然而，在最坏的情况下，即数据完全逆序，它需要进行大约 n^2/2 次比较和同样数量的交换，时间复杂度为 O(n^2)。
2. **冒泡排序**：在最好的情况下（即数据已经是正序），冒泡排序也只需要进行 n-1 次比较和 0 次交换，时间复杂度为 O(n)。在最坏的情况下（即数据完全逆序），冒泡排序需要进行大约 n^2/2 次比较和交换，时间复杂度同样为 O(n^2)。
3. **希尔排序**：希尔排序的情况比较特殊，它是基于插入排序的一种改进。希尔排序的性能并不像直接插入排序和冒泡排序那样严重依赖于原始数据的顺序。它通过设定不同的间隔对序列进行部分排序，随着间隔的减少，最终整个列表变得有序。希尔排序的最好情况时间复杂度可以达到 O(nlogn)，但最坏情况和平均情况的时间复杂度较难精确计算，一般认为是介于 O(nlogn) 和 O(n^2) 之间，依赖于间隔序列的选择。

综上所述，对于直接插入排序和冒泡排序，数据在正序的情况下确实比逆序情况下要快。而希尔排序的性能不是简单地由数据的初始顺序决定的，它的改进在于它能够让元素跳过一些位置快速接近最终位置，从而减少总的比较和移动次数。因此，虽然希尔排序在正序情况下可能比逆序情况下快，但这种差异通常没有直接插入排序和冒泡排序那么显著。

Q:（Y）用相邻接矩阵法存储一个图时，在不考虑压缩存储的情况下，所占用的存储空间大小只与图中结点个数有关，而与图的边数无关。

Q:（Y）构建一个含 N 个结点的（二叉）最小值堆，建堆的时间复杂度大 O 表示为 $O(Nlog_{2}{N})$。

解释：在最小值堆的构建过程中，我们需要将 N 个节点逐个插入堆中，并保持堆的性质。每次插入一个节点时，需要将它与已有的节点进行比较，然后根据堆的性质进行调整，以确保最小值堆的性质不被破坏。

当插入一个节点时，最坏情况下需要进行 log_2N 次比较和调整操作。因此，插入 N 个节点的时间复杂度为 $O(Nlog_{2}{N})$。

Q:（Y）队列是动态集合，其定义的出队列操作所移除的元素总是在集合中存在时间最长的元素。

解释：队列是一种遵循先进先出（First-In-First-Out, FIFO）原则的动态集合。在队列中，元素被添加到集合的一端（通常称为队尾）并从另一端（队首）移除。因此，当进行出队列（dequeue）操作时，移除的元素总是在队列中停留时间最长的元素，即最先被加入队列的元素。

Q:（Y）分治算法通常将原问题分解为几个规模较小但类似于原问题的子问题，并要求算法实现写成某种递归形式，递归地求解这些子问题，然后再合并这些子问题的解来建立原问题的解。

解释：分治算法的核心思想就是将一个难以直接解决的大问题，分割成若干个规模较小的同类型问题，递归地求解这些小问题，然后将这些小问题的解合并成原来大问题的解。

分治策略通常包括以下三个步骤：

1. **分解（Divide）**：原问题被分解为若干个规模较小的同类问题。
2. **解决（Conquer）**：递归地解决这些子问题。如果子问题的规模足够小，则直接求解。
3. **合并（Combine）**：将子问题的解合并成原问题的解。

一个经典的分治算法例子是归并排序（Merge Sort），它将一个数组分解为两个规模几乎相等的子数组，递归地对这两个子数组进行排序，然后将它们合并成一个有序的数组。其他分治算法的例子包括快速排序（Quick Sort）、二分搜索（Binary Search）等。分治算法通常用递归形式来实现，因为递归提供了一种自然且直观的方式来分解和解决问题。

Q:（Y）考察某个具体问题是否适合应用动态规划算法，必须判定它是否具有最优子结构性质。

解释：一个问题适合应用动态规划算法的一个重要条件是它具有最优子结构性质。最优子结构意味着问题的最优解包含其子问题的最优解。具体来说，一个问题的最优解可以通过组合其子问题的最优解来构造，而这些子问题可以独立求解。

动态规划的两个关键属性是：

1. **最优子结构（Optimal Substructure）**：如您所述，一个问题的最优解包含了其子问题的最优解。这意味着，可以通过组合子问题的最优解来构造整个问题的最优解。
2. **重叠子问题（Overlapping Subproblems）**：在解决问题的过程中，相同的子问题多次出现。这意味着一个子问题一旦被解决，它的解可以被存储和重复使用，避免了重复的计算工作。

动态规划算法通常是通过填表（通常是一维或二维的数组）的方式来解决问题，表中的每个条目对应一个子问题的最优解。动态规划的经典例子包括求解斐波那契数列、背包问题、最长公共子序列、最短路径问题（如Dijkstra算法）、编辑距离问题等。

值得注意的是，并不是所有具有最优子结构的问题都适合用动态规划来解决。如果问题的子问题不是重叠的，即每个子问题只被解决一次，那么可能使用分治算法而非动态规划会更加高效。

Q: （Y）考虑一个长度为 n 的顺序表中各个位置插入新元素的概率是相同的，则顺序表的插入算 法平均时间复杂度为 O(n) 。

解释：在顺序表（或者数组）中，插入一个新元素的时间复杂度取决于插入位置。插入操作需要将插入点之后的所有元素向后移动一位来腾出空间。如果在顺序表的末尾插入，则不需要移动任何元素，时间复杂度是 O(1)。但是，如果在顺序表的开始处插入，则需要移动全部的 n 个元素，时间复杂度是 O(n)。

如果每个位置插入新元素的概率相同，那么平均来看，插入新元素在顺序表的任何一个位置的概率是 1n1 1/(*n*+1)，因为有 n1*n*+1个可能的插入位置（考虑到列表初始时有 n 个元素，插入点可以在这些元素之间以及列表末尾，共 n1*n*+1个位置）。

因此，平均需要移动的元素数目是：

$(0 + 1 + 2 + … + n) / (n + 1) = n(n + 1) / 2 / (n + 1) = n / 2$

所以，平均时间复杂度是On2*O*(*n*/2)，在大 O 表示法中常数是可以忽略的，因此平均时间复杂度简化为 O(n)。

填空（20分，每题2分）

Q: 线性表的顺序存储与链式存储是两种常见存储形式；当表元素有序排序进行二分检索时，应采用哪种存储形式？顺序存储

Q: 如果只想得到 1000 个元素的序列中最小的前 5 个元素，在冒泡排序、快速排序、堆排序和归并排序中，哪种算法最快？ （ 堆排序 ）

Q: 目标串长是 n，模式串长是 m，朴素模式匹配算法思想为：从目标串第一个字符开始，依次与模式串字符匹配；若匹配失败，则尝试匹配的目标串起始字符位置往后移一位，重新开始依次和模式串字符匹配；……. ；直到匹配成功或遍历完整个目标串为止。则该算法中字符的最多比较次数是（使用大 O 表示法）？ O(nm)

解释：朴素模式匹配算法，也称为暴力模式匹配算法，在最坏的情况下，需要对每个目标串中的起始位置都尝试匹配模式串。在目标串中选取起始位置的选择有 $n - m + 1 $种（因为当目标串剩余字符少于模式串时，不可能匹配成功）。对于每个起始位置，你可能需要比较到模式串的最后一个字符，即最多比较 m 次。因此，最坏情况下的比较次数是$(n - m + 1) * m$。使用大 O 表示法，我们通常关注最主要的项并忽略系数，因此朴素模式匹配算法的时间复杂度为 O(nm)。

Q: 有 n 个数据对象的二路归并排序中，每趟归并的时间复杂度为? O(n)

解释：二路归并排序是一种分治算法，它将数组分成两半，递归地对它们分别进行排序，然后将排序好的两个半部分合并在一起。每一趟归并的时间复杂度是由合并这两个已排序数组的过程决定的。

具体来说，在每一趟归并中，你需要遍历两个子数组的所有元素以进行合并。无论这两个子数组的具体大小如何分配，它们的总大小是固定的，即 n 个数据对象。因此，合并两个子数组的过程需要对 n 个数据进行一次遍历。

所以，每趟归并的时间复杂度是 O(n)，因为你需要查看每个元素一次以确保它们合并在正确的顺序中。这个 O(n) 的时间复杂度适用于每一趟归并操作，无论归并发生在递归的哪一层上。

Q: 对一组记录进行降序排序，其关键码为（ 46， 70， 56， 38， 40， 80， 60， 22），采用初始步长为 4的希尔（ shell）排序，第一趟扫描的结果是？（46, 80, 60, 38, 40, 70, 56, 22）

而采用归并排序第一轮归并的结果是？（70, 46, 56, 38, 80, 40, 60, 22）

解释：**希尔排序第一趟扫描（初始步长为4）**

希尔排序的第一趟扫描涉及将数组分成以步长为间隔的多个子数组，然后对每个子数组进行直接插入排序。在这个例子中，初始步长为4，因此我们将列表划分为4个子数组，并对每个子数组进行直接插入排序。

原始数组：

```
46, 70, 56, 38, 40, 80, 60, 22
```

以步长4进行分组，我们得到以下子数组：

```
子数组1: 46, 40
子数组2: 70, 80
子数组3: 56, 60
子数组4: 38, 22
```

由于我们需要进行降序排序，我们将每个子数组按降序排列：

```
子数组1排序后: 46, 40（已经按降序排列）
子数组2排序后: 80, 70
子数组3排序后: 60, 56
子数组4排序后: 38, 22（已经按降序排列）
```

将排序后的子数组放回原数组对应的位置：

```
第一趟排序后的结果: 46, 80, 60, 38, 40, 70, 56, 22
```

归并排序第一轮归并

归并排序的第一轮归并通常将数组中相邻的单个元素视为已排序的子数组，然后两两归并。因为我们正在进行降序排序，所以在比较时，我们会选择两个元素中较大的那个放在前面。

原始数组：

```
46, 70, 56, 38, 40, 80, 60, 22
```

将它分成单个元素的对儿，然后两两归并：

```
归并1: (46, 70) => 70, 46
归并2: (56, 38) => 56, 38
归并3: (40, 80) => 80, 40
归并4: (60, 22) => 60, 22
```

第一轮归并后的结果：

```
70, 46, 56, 38, 80, 40, 60, 22
```

因此，希尔排序第一趟扫描的结果是 `46, 80, 60, 38, 40, 70, 56, 22` ，而归并排序第一轮归并的结果是 `70, 46, 56, 38, 80, 40, 60, 22` 。

简答（24分，每题6分）

Q: 哈夫曼树是进行编码的一种有效方式。设给定五个字符，其相应的权值分别为{4， 8， 6， 9， 18}， 试画出相应的哈夫曼树，并计算它的带权外部路径长度 WPL 。

A:

哈夫曼树（Huffman Tree）是一种根据字符出现频率或权重来构建的最优二叉树，它可以用于有效编码，最小化编码的总长度。构建哈夫曼树的步骤是：

1. 将所有字符根据它们的权值作为叶子节点，并将权值作为节点的权重。
2. 在森林中选择两个最小权重的树合并，将这两个树的权重相加形成新树的根节点权重。
3. 重复步骤2，直到森林中只剩下一棵树为止。

根据你给出的权值 {4, 8, 6, 9, 18}，我们可以开始构建哈夫曼树：

1. 首先列出所有权值：4，8，6，9，18。
2. 选择最小的两个数 4 和 6，合并为一个有权值为 10 的节点。
3. 新的权值列表为：10，8，9，18。
4. 再次选择最小的两个数 8 和 9，合并为一个有权值为 17 的节点。
5. 新的权值列表为：10，17，18。
6. 选择最小的两个数 10 和 17，合并为一个有权值为 27 的节点。

7. 新的权值列表为：27，18。
8. 最后，合并这两个节点，得到一个有权值为 45 的根节点。

现在我们可以画出哈夫曼树了：

```
          (45)
         /    \
       (27)  (18)
      /    \
    (10)   (17)
    /  \   /  \
  (4) (6) (8) (9)
```

为了计算这棵哈夫曼树的带权外部路径长度（WPL），我们需要计算每个叶子节点的权重乘以其路径长度（从根节点到叶子节点的边数）。

- 对于权重为 4 的叶子节点，路径长度为 3，所以其部分 WPL 为 4 * 3 = 12。
- 对于权重为 6 的叶子节点，路径长度也为 3，所以其部分 WPL 为 6 * 3 = 18。
- 对于权重为 8 的叶子节点，路径长度为 3，所以其部分 WPL 为 8 * 3 = 24。
- 对于权重为 9 的叶子节点，路径长度为 3，所以其部分 WPL 为 9 * 3 = 27。
- 对于权重为 18 的叶子节点，路径长度为 1，所以其部分 WPL 为 18 * 1 = 18。

将所有部分的 WPL 相加，我们得到整棵哈夫曼树的 WPL：

WPL = 12 + 18 + 24 + 27 + 18 = 99

因此，这棵哈夫曼树的带权外部路径长度 WPL 为 99。

算法（16分，每题8分）

阅读下列程序，完成图的深度优先周游算法实现的迷宫探索。已知图采用邻接表表示，Graph 类和 Vertex 类基本定义如下：

```
import sys
sys.setrecursionlimit(10000000)

class Graph:
    def __init__(self):
        self.vertices = {}

    def addVertex(self, key, label):    # #添加节点，id 为key，附带数据 label
        self.vertices[key] = Vertex(key, label)

    def getVertex(self, key):   # 返回 id 为 key 的节点
        return self.vertices.get(key)

    def __contains__(self, key):        # 判断 key 节点是否在图中
        return key in self.vertices
    def addEdge(self, f, t, cost=0):    # 添加从节点 id==f 到 id==t 的边
        if f in self.vertices and t in self.vertices:
            self.vertices[f].addNeighbor(t, cost)

    def getVertices(self):      # 返回所有的节点 key
        return self.vertices.keys()

    def __iter__(self): # 迭代每一个节点对象
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, key, label=None):        # 缺省颜色为"white
        self.id = key
        self.label = label
        self.color = "white"
        self.connections = {}

    def addNeighbor(self, nbr, weight=0):       # 添加到节点 nbr 的边
        self.connections[nbr] = weight

    def setColor(self, color):  # 设置节点颜色标记
        self.color = color

    def getColor(self): # 返回节点颜色标记
        return self.color

    def getConnections(self):   # 返回节点的所有邻接节点列表
        return self.connections.keys()

    def getId(self):    # 返回节点的 id
        return self.id

    def getLabel(self): # 返回节点的附带数据 label
        return self.label


mazelist = [
    "++++++++++++++++++++++",
    "+   +   ++ ++        +",
    "E     +     ++++++++++",
    "+ +    ++  ++++ +++ ++",
    "+ +   + + ++    +++  +",
    "+          ++  ++  + +",
    "+++++ + +      ++  + +",
    "+++++ +++  + +  ++   +",
    "+         + + S+ +   +",
    "+++++ +  + + +     + +",
    "++++++++++++++++++++++",
]


def mazeGraph(mlist, rows, cols):       # 从 mlist 创建图，迷宫有 rows 行 cols 列
    mGraph = Graph()
    vstart = None
    for row in range(rows):
        for col in range(cols):
            if mlist[row][col] != "+":
                mGraph.addVertex((row, col), mlist[row][col])
                if mlist[row][col] == "S":
                    vstart = mGraph.getVertex((row, col))   # 等号右侧填空（1分）

    for v in mGraph:
        row, col = v.getId()
        for i in [(-1, 0), (1, 0), (0, -1), (0, +1)]:
            if 0 <= row + i[0] < rows and 0 <= col + i[1] < cols:
                if (row + i[0], col + i[1]) in mGraph:
                    mGraph.addEdge((row, col), (row + i[0], col + i[1])) #括号中两个参数填空（1分

    return mGraph, vstart       # 返回图对象，和开始节点


def searchMaze(path, vcurrent, mGraph): # 从 vcurrent 节点开始 DFS 搜索迷宫
    path.append(vcurrent.getId())
    #print(path)
    if vcurrent.getLabel() != "E":
        done = False
        for nbr in vcurrent.getConnections(): # in 后面部分填空（2分）
            nbr_vertex = mGraph.getVertex(nbr)
            if nbr_vertex.getColor() == "white":
                done = searchMaze(path, nbr_vertex, mGraph) # 参数填空（2分）
                if done:
                    break
        if not done:
            path.pop()  # 这条语句空着，填空（2分）
            vcurrent.setColor("white")
    else:
        done = True
    return done  返回是否成功找到通路


g, vstart = mazeGraph(mazelist, len(mazelist), len(mazelist[0]))
path = []
searchMaze(path, vstart, g)
print(path)
```

参考

Introduction to Algorithms, 3rd Edition (Mit Press) 3rd Edition, by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein

https://dahlan.unimal.ac.id/files/ebooks/2009%20Introduction%20to%20Algorithms%20Third%20Ed.pdf

or

https://github.com/calvint/AlgorithmsOneProblems/blob/master/Algorithms/Thomas%20H.%20Cormen,%20Charles%20E.%20Leiserson,%20Ronald%20L.%20Rivest,%20Clifford%20Stein%20Introduction%20to%20Algorithms,%20Third%20Edition%20%202009.pdf

Complexity of Python Operations 数据类型操作时间复杂度

https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt

This is called "static" analysis, because we do not need to run any code to perform it (contrasted with Dynamic or Empirical Analysis, when we do run code and take measurements of its execution).

Learn Data Structures and Algorithms | DSA Tutorial

https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=outind

附录

Puzzle

22507:薛定谔的二叉树

http://cs101.openjudge.cn/practice/22507/

假设二叉树的节点里包含一个大写字母，每个节点的字母都不同。 给定二叉树的前序遍历序列和后序遍历序列(长度均不超过20)，请计算二叉树可能有多少种

前序序列或后序序列中出现相同字母则直接认为不存在对应的树

输入

多组数据 每组数据一行，包括前序遍历序列和后序遍历序列，用空格分开。 输入数据不保证一定存在满足条件的二叉树。

输出

每组数据，输出不同的二叉树可能有多少种

样例输入

```
ABCDE CDBEA
BCD DCB
AB C
AA AA
```

样例输出

```
1
4
0
0
```

来源: 刘宇航

坑

01035: 拼写检查

http://cs101.openjudge.cn/routine/01035/

字典会覆盖，有时候得避免（一个常见的坑）

y第一次输出正确，第二次为什么不输出了