NOTES

Quick Sort

The quick sort algorithm picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways. We are taking last element as the pivot element. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Say we have an array 'arr' with following elements-

{10, 80, 30, 90, 40, 50, 70}

Naturally the indexes are going to be from 0 to 6. Let us choose the **last element as the pivot** element and out target is place the pivot element such that all elements on one side of the pivot element are smaller and on other side all elements has to be larger. Say we are using variable low and high to maintain indexes. Initialize index of smaller element as i = -1

low = 0, high = 6, pivot = arr[high] = 70

Traverse elements from j = low to high-1

For First Iteration when j = 0
Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
So i = 0 and i, j are same so no effect on swapping so elements of array will be same

For Second Iteration when j = 1
Since arr[j] > pivot, do nothing

For Third Iteration when j = 2
Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
So i = 1 and swap 30 with 80 so the resultant array will be {10, 30, 80, 90, 40, 50, 70}

For Fourth Iteration when j = 3
Since arr[j] > pivot, do nothing

For Fifth Iteration when j = 4
Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
So i = 2 swap 40 with 80 so the resultant array will be {10, 30, 40, 90, 80, 50, 70}

For Sixth Iteration when j = 5
Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
So i = 3 and swap 90 with 50 so the resultant array will be {10, 30, 40, 50, 80, 90, 70}

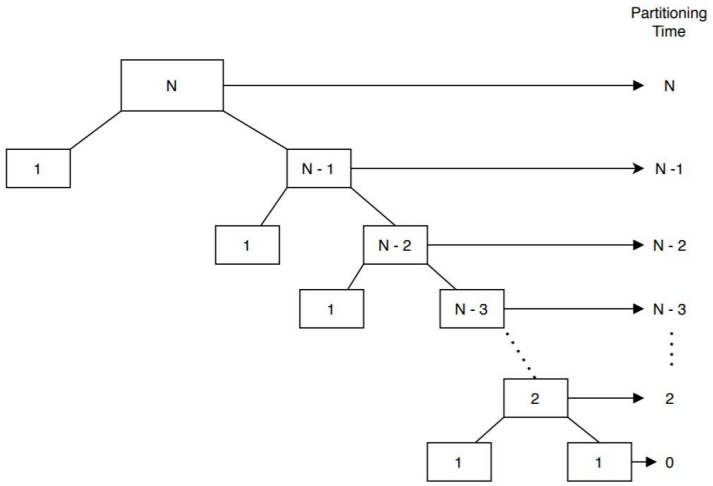
We come out of loop because j is now equal to high-1. Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot) so your array will be {10, 30, 40, 50, 70, 90, 80} Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

```
def partition(arr,low,high):
  i = (low-1)
  pivot = arr[high] # pivot element
  for j in range(low , high):
    # If current element is smaller
    if arr[j] <= pivot:
      # increment
      i = i+1
      arr[i],arr[j] = arr[j],arr[i]
  arr[i+1],arr[high] = arr[high],arr[i+1]
  return ( i+1 )
```

```
def quickSort(arr,low,high):
  if low < high:
    # index
    pi = partition(arr,low,high)
    # sort the partitions
    quickSort(arr, low, pi-1)
    quickSort(arr, pi+1, high)

arr = [2,5,3,8,6,5,4,7]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
  print (arr[i],end=" ")
```

The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.



Let's consider an input array of size N. The first partition call takes N times to perform the partition step on the input array. Each partition step is invoked recursively from the previous one. Given that, we can take the complexity of each partition call and sum them up to get our total complexity of the Quicksort algorithm. Therefore, the time complexity of the Quicksort algorithm in worst case is

$$[\mathbf{N} + (\mathbf{N-1}) + (\mathbf{N-2}) + (\mathbf{N-3}) + ... + \mathbf{2}] = [\tfrac{\mathbf{N(N+1)}}{\mathbf{2}} - \mathbf{1}] = \mathcal{O}(\mathbf{N^2})$$

We can avoid the worst-case in Quicksort by choosing an appropriate pivot element.

1. The first approach for the selection of a pivot element would be to pick it from the middle of the array. In this way, we can divide the input array into two sub-arrays of an almost equal number of elements in it.
2. In some cases selection of random pivot elements is a good choice. This variant of Quicksort is known as the randomized Quicksort algorithm.
3. Another approach to select a pivot element is to take the median of three pivot candidates. In this case, we'll first select the leftmost, middle, and rightmost element from the input array. Then we'll arrange them to the left partition, pivot element, and right partition. Each partition will be exactly half (±one element) of the problem and we will need exactly ceiling(log n) recursive calls.

Hashing with Chaining

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

Let us consider string s. You are required to count the frequency of all the characters in this string

```
s = "ababcd";
```

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is O(26*N) where N is the size of the string and there are 26 possible characters.

```
def countFre(s):
    for i in range(97,123):
        ch = chr(i)
        fre = 0
        for j in range(len(s)):
            if s[j] == ch:
                fre+=1;
        print(ch,":",fre)

s = "ababcd"
countFre(s)
```

Output: a:2 b:2 c:1 d:1 … z:0

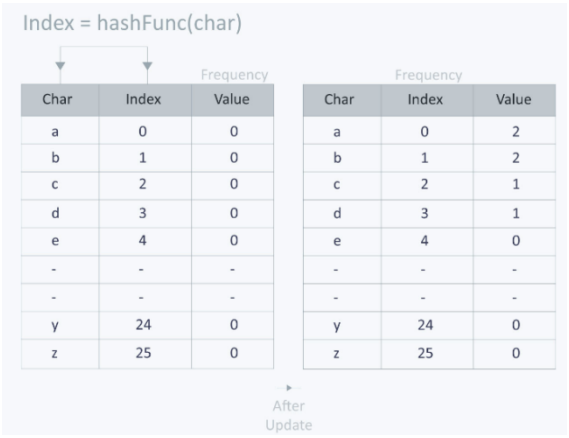
Let us apply hashing to this problem. Take an list frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is O(N) where N is the size of the string.

```
Frequency = [0]*26;

def hashFunc(ch):
        return (ord(ch)- ord('a'));

def countFre(S):
        for i in range(len(S)):
                index = hashFunc(S[i])
                Frequency[index]+=1
        for i in range(26):
                print(chr(i+97),":",Frequency[i])

countFre("ababcd")
```



Hashing

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

1. In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
2. In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, In cases where the keys are large and cannot be used directly as an index, you should use hashing. In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted. Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.
   hash = hashfunc(key)
   index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).
A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

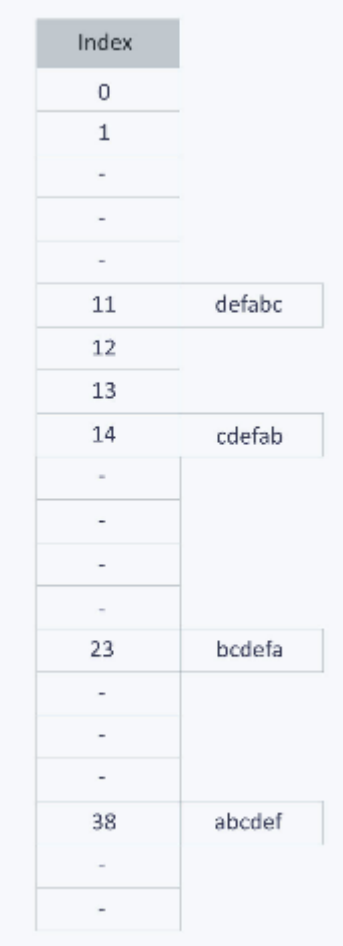
An Example
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function in which the index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599. The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

Here, it will take O(n) time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function. Let’s try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	2069%(97 x 1 + 98 x 2 + 99 x 3 + 100 x 4 + 101 x 5 + 102 x 6)	38
bcdefa	2069 %(98 x 1 + 99 x 2 + 100 x 3 + 101 x 4 + 102 x 5 + 97 x 6)	23
cdefab	2069 % (99 x 1 + 100 x 2 + 101 x 3 + 102 x 4 + 97 x 5 + 98 x 6)	14
defabc	2069 % (100 x 1 + 101 x 2 + 102 x 3 + 97 x 4 + 98 x 5 + 99 x 6)	11



Collision resolution techniques

1. Separate chaining (open hashing)

Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

2. Linear probing (Open addressing or closed hashing)

All entry records are stored in the array. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

and so on…

Applications of hashing

1. Associative arrays: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
2. Database indexing: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
3. Caches: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
4. Object representation: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.Hash Functions are used in various algorithms to make their computing faster