8.3 Big-O Notation

After learning all the searching and sorting algorithms, we need to analyze their performance so that we know which one to choose for a particular problem. The performance can be quantified in terms of time and space complexity, i.e. how much runtime and how large memory space each algorithm takes.

However, the running time for a searching algorithm definitely increases with the size of the list to be searched and also depending on some condition of the list (e.g. whether it has been sorted). So the common practice is that we study how the time cost changes with respect to its input size n in the worst case performance, and this is called Big-O Notation. Before moving onto searching and sorting algorithms, let’s look at the Big-O Notation for some simple programs.

Constant Complexity: O(1)
Complexity of the program remains constant regardless of the input size.

```
def get_last(List):
     return List[-1]
```

Linear Complexity: O(n)
The time cost grows linearly and proportionally with the input size.

```
def get_sum(List):
     total = 0
     for item in List:
          total = total + item
     return total
```

Quadratic Complexity: O(n^2)

```
def multi_table(n):
#generate the multiplication table
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            print( i * j, end = ' ')
        print()
```

Clearly, linear search requires O(n) comparisons of the items in the list. Binary search halves the list in each iteration, so it requires O(log n) comparisons. But don’t forget that binary search requires input to be an ordered list. For hash table, in ideal circumstances without collision, we found the item in one step, i.e. O(1). When collision occurs, it requires O(n). Hence a good hash function is very important. On the other hand, hash table may require more spaces as well. Each searching algorithm has its own pros and cons.

Sorting algorithms are more complicated and the nested loops makes both bubble sort and insertion sort quadratic complexity with $O(n^2)$. This is not a problem with small data sets, but with hundreds or thousands of elements, this becomes very significant.

Bubble sort does perform better for partially sorted lists because it is able to detect when a list is sorted and does not continue making unnecessary passes through the list. As a general sorting scheme, however, it is very inefficient because of the large number of interchanges that it requires. In fact, it is the least efficient of the sorting schemes.

Insertion sort also is too inefficient to be used as a general-purpose sorting scheme. However, the low overhead that it requires makes it better than bubble sort.

While Quick sort partitions and usually makes less comparisons than Bubble sort and Insertion sort, in the worst case scenario the time complexity is still $O(n^2)$.

Merge sort has a time complexity of O(n log n) and is a very efficient general-purpose sorting schemes and especially for large lists.

In summary:

Algorithm	**Time Complexity (worst case)**
Linear search	O(n)
Binary search	O(log n)
Hash Table search	O(n)

Algorithm	**Time Complexity (worst case)**
Bubble sort	$O(n^2)$
Insertion sort	$O(n^2)$
Quick sort	$O(n^2)$
Merge sort	O(n log n)