ENSF 338 Assignment Report

|  |  |
| --- | --- |
| Assignment # | 2 |
| Group # | 51 |
| Lab Section | B02 |
| Names | Sergiy Redko (30151178), Nelson Thompson (30163519) |

# Repository Link

<https://github.com/SergiyRedko/338Assignment2>

# Work Performed By Each Member

It is difficult to clearly break down work we performed, since we adopted pair programming approach for this assignment. Below is our best shot at it:

|  |  |
| --- | --- |
| Task | Person Responsible |
| Exercise 1 | Sergiy Redko |
| Exercise 2 | Nelson Thompson |
| Exercise 3 | Sergiy Redko |
| Exercise 4 | Sergiy Redko |
| Exercise 5 | Nelson Thompson |

The work distribution is 50-50 among the team members as we did the whole assignment sitting next to each other.

# Exercise 1

## 1.1

Memoization is the style of writing functions in which the results of previous iteration are recorded so as to not repeat the same work again. So far memoization has been used in recursive functions in this course, but similar technique can be applied to other programming methods as well.

Memoization is a good technique to use where the memory loss is far outweighed by performance gain. For instance Fibonacci sequence calculation.

## 1.2 & 1.3

The function recursively calculates nth number in Fibonacci sequence.

## 1.4

This is an example of divide-and-conquer algorithm as far as any recursive functions is. It is difficult to argue the extent of encapsulation of such a limited function though.

## 1.5

Each time we increment the position of Fibonacci number we have to double the amount of work. This gives us time complexity of O(2n).

## 1.6

# Declare dictionary to store fibonacci numbers

fib\_dict = {0:0, 1:1}

def fib(n):

    if n in fib\_dict:

        return fib\_dict[n]

    else:

        fib\_dict[n] = fib(n-1) + fib(n-2)

        return fib\_dict[n]

## 1.7

Best case scenario, we have already computed the value of fib(n), in which case its already in fib\_dict. The complexity of that is Ω(1).

Worst case scenario, no entries exist in fib\_dict other than the default ones. Each entry has to be calculated once, this gives a linear relationship between n and complexity. Therefore, the time complexity is O(n).

## 1.8

def func(n):

    if n == 0 or n == 1:

        return n

    else:

        return func(n-1) + func(n-2)

# Declare dictionary to store fibonacci numbers

fib\_dict = {0:0, 1:1}

def fib(n):

    if n in fib\_dict:

        return fib\_dict[n]

    else:

        fib\_dict[n] = fib(n-1) + fib(n-2)

        return fib\_dict[n]

if \_\_name\_\_ == "\_\_main\_\_":

    import timeit

    import matplotlib.pyplot as plt

    slow\_fib = []

    for i in range(0, 36):

        slow\_fib.append(timeit.timeit("func({})".format(i), setup="from \_\_main\_\_ import func", number=1))

    fast\_fib = []

    for i in range(0, 36):

        fast\_fib.append(timeit.timeit("fib({})".format(i), setup="from \_\_main\_\_ import fib", number=1))

    # Plot slow\_fib and fast\_fib in plt

    plt.plot(slow\_fib, label="Slow Fibonacci")

    plt.plot(fast\_fib, label="Fast Fibonacci")

    plt.legend()

    plt.title("Fibonacci Time Comparison")

    plt.xlabel("nth Fibonacci Number")

    plt.ylabel("Time (s)")

    plt.show()

## A picture containing chart Description automatically generated1.9

The improved Fibonacci code remains fast regardless of the sample size, while the original implementation slows down logarithmically with the size of n. This supports our findings in parts 1.5 and 1.7 of this exercise.

# Exercise 2

## 2.1

This code is an implementation of the QuickSort algorithm. Func1 will take an array to be sorted along with both a high, and a low index which will represent the bounds of the part of the array which needs to be sorted. The function is recursive with each recursion being used to smaller sub arrays of one element each. Func2 selects a pivot element from the array and arranges elements based on whether they are larger or smaller than the pivot.

The QuickSort has a time complexity of O(n \* log(n)), where “n” is the number of elements in the array. This is because portioning the array will, on average, divide the array into roughly equal heights. therefore, each level of recursion will process roughly half of the elements with log(n) levels of recursion. In each recursion the program preforms O(n) operations. This makes the overall time complexity O(n \* log(n)

# Exercise 3

## 3.1

Interpolation search is not necessarily better than binary search. It is, however, better for uniformly distributed data (or any other type of known “interpolatable”, sorted data) since it jumps to where it expects data to be, rather than to the middle of remaining zone. The benefits of using interpolation search for uniformly distributed data:

* Less time complexity (works faster)
* Allows us to make an intelligent guess about data.

## 3.2

Yes, performance would be affected, usually for the worst. The closer the data is to uniform distribution the less steps would interpolation search have to perform to get to the result. Take 2 arrays for instance:

1. abcdefghijklmnopqrstuvwxyz
2. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbccccccddddeeeeeefffffffggggggghhhhhiiiiijjjjjkkkkklmnopqrstuvwxyz

Let us assume we are using our knowledge about English alphabet to find element b in this list. In both cases we would go close to the beginning of the array (and beginning of each following iteration), since that is where we expect to find the element. In case 1 it would take us just a few (likely 2) iterations to get to b. In case 2 it would take us around a dozen iterations.

For case 2 binary search would clearly be better.

## 3.3.

We would change the way pos is calculated each iteration, since it is the “brains” of the search algorithm. Binary search is a good default (pos = (high + low)/2), and should only be changed if we are confident that data would be presented in a specific case.

## 3.4

### a

Any case of non-sorted data. Binary search (and interpolation search, which is a type of binary search) are reliant on data being sorted. If they are run on unsorted data, the result would be unpredictable and likely wrong.

### b

First, if the data is unsorted, linear search would perform better than sorting data and then searching it.

Second, if the item we are searching for has a tendency of being close to the beginning of the end of data (it would have to be one of these exclusively, and programmer would have to know which it is) linear search would have to go through fewer iterations to get there. Also, an iteration of linear search is faster than that of binary or interpolation search since there is less “thinking” involved.

Third, similar to second, if the data is heavily skewed to beginning or the end, and binary/interpolation search are not well-suited to traverse it.

Interpolation search might be improved, by modifying positioning algorithm to better reflect the distribution of data. It would become rather situational in this case though.

# Exercise 4

## 4.1

Advantages and disadvantages of arrays and linked lists are dependent on situation. Below we discuss some advantages and disadvantages.

### Lookup by index

Lookup by index has complexity O(1) in an array and O(n) in a linked list. This favors arrays in applications with frequent data lookup of specific members.

### Growing/Shrinking

Growing/shrinking is O(n) for both linked list and array.

NOTE: insertion/deletion of element is O(1) for linked list if the position is known, but traversing to find that position is O(n).

Arrays, however, are disadvantaged for this operation. Insertion or deletion of an element must be followed by shifting the data for all pos ≥ n. This can be resource expensive, when compared to linked-list. Especially for operations at the beginning of array/list.

On top of that, array is not a mutable object. This means that in case array has run out of space a new array would have to be created and the old one would have to be copied into the new one in its entirety. That is another O(n) operation.

### Search

Search complexity depends on implementation and appearance of data. If data is unsorted or linear search is implemented complexity is O(n) for both array and linked list.

If data is sorted, or a sort more intelligent than linear sort is implemented array would typically have a better performance, since it is easier to traverse and index.

### Sort

Will be compared in detail in part 3.

## 4.2

Both insertion and deletion are O(n) operations in array.

In order to optimize the process, both operations may be lumped together to prevent shifting of data. Easiest way to implement it:

Array[atThisLocation] = data\_we\_are\_deletin\_and\_replacing\_with\_new

During lab 3 Maan Khedr mentioned that the data in the array is to be treated as if it is sorted. This changes the situation a little. Still, in order to optimize performance, both operations should be lumped together to minimize data shifting. In this case algorithm looks like the following:

1. Find location where we need to delete the data. Let us call this location *del*.
2. Find location where we need to insert the data. Let us call this location *ins*.
3. Shift all the data between *ins* and *del* (inclusive of *ins* location, but exclusive of *del* location; *del* location is to be overridden, but not shifted) towards *del* location.
4. Array[*ins*] = data\_to\_insert

This way we only need to shift |*del* *– ins*| number of elements, rather than 2 ∙ num\_elements – *ins* – *del*.

Though, the complexity (strictly) remains O(n).

When lumping these operations together, or even just performing delete before insert we always prevent the necessity for growing the array.

## 4.3

Each of these sorts can be implemented for both array and linked list. Some performance penalties may apply.

### Selection Sort

In both array and linked list, it has O(n2) complexity. The algorithm would be the same for both, but implementation differs.

In case of linked list there would be a lot of memorization involved for keeping track of the pointers to:

* current element,
* element preceding current element,
* element to swap the current element with,
* element preceding the element to sap the current element with.

Overall, implementation of this sort in singly linked list would have slightly lower performance.

### Insertion Sort

Here is how I would implement insertion sort in linked list:

1. First, it doesn’t need to be an in-place sort. Create a temporary list.
2. Take out elements from the beginning of the original list and insertion-sort them (via injection) into the temporary list.
3. Copy the head from temporary list to head of original list.

Since there are up to n comparisons for up to n elements the complexity is still O(n2); much like the array implementation.

Unlike in the array implementation, there is no need to shift elements for every insertion, which saves resources for every iteration. Insertion sort in linked list would be much faster than one in array.

### Merge Sort

Merge sort can be implemented in linked lists with the same complexity as in arrays. Fast pointer would have to be used to find middle. Once middle is found, list can be unlinked and sorted by recursive call to merge sort method.

The complexity is O(n∙log(n)) for both array and list implementations.

### Bubble Sort

Bubble sort is similar in both array and linked list implementation both are O(n2), both would take the same number of iterations to sort identical data.

It is a terrible sort in both arrays and lists.

Traversing the linked list would not come with a performance penalty since the direction of movement is always ascending.

### Quick Sort

Similar to merge sort, quick sort can be implemented in linked list using slow and fast pointers. It can be split in two similarly to merge sort as well, and then call itself recursively on the partitions.

Much like the quick sort in array it has O(n2) complexity in worst case scenario.

# Exercise 5