Algorithms and Logical Methods Assignment 1

**Tadgh Wagstaff**

**15379556**

Bubble Sort

**Basis**

Bubble sort is a simple sorting algorithm which iterates through a list, comparing the value stored at each index position to the value stored at the next index position along. If the first value is greater than the second it swaps their positions, moving the larger value to the right. Bubble sort is so-called because the largest unsorted value in a list will always “bubble” to the top on each pass, as every time it is compared its neighbour it will be swapped to the right.

**Un-optimised Implementation**

Assuming that we want to list our values in ascending order, a single pass of an unoptimized bubble sort can be written as:

begin bubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

end bubbleSort

In order to ensure that enough passes are run to order any list we can force the algorithm to run one pass per index position in the array.

begin bubbleSort(list)

for all elements of list

for all elements of list

…

end for

end for

end bubbleSort

This approach is very inefficient. Regardless of how much the list actually needs to be sorted (there could only be a single value out of place) our un-optimised bubble sort will make n2 comparisons, where n is the number of index positions in the list.

**Optimisation**

We know that every pass of our bubble sort algorithm takes the largest number in our list and moves it to the end of the list. Therefore, we know that on pass *n*, *n-1* of the values in the list can be safely ignored. If the algorithm is starting its second run, we know that the largest value in the list has already been moved to the appropriate location. If it’s starting its third, we know that the two largest values are in the correct location – and so on. We can account for this by telling our program not to compare the final n-1 values.

begin bubbleSort(list)

int n = 1

for all elements of list

for (all elements of list – (n + 1))

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

n = n + 1

end for

end bubbleSort

Our revised bubble sort will now run only compare as many values as it has not already sorted when the inner loop is run.

Our second major inefficiency resides in the outer loop. Our bubble sort will continue running for as many times are there are values in a given list – even if the list is totally ordered at that point. If you pass it a list that is completely ordered aside from a small number of out of place values, it will continue to trigger the inner loop needlessly until it has done so a number of times equal to the length of the list. We can address that inefficiency by creating a flag that raises when the algorithm has detected that no more sorting is required

begin bubbleSort(list)

int n = 1

Boolean sortingComplete = false;

while sortingCompleted = false

for all elements of list

for (all elements of list – (n-1))

Boolean sortingRequired = true;

if list[i] > list[i+1]

swap(list[i], list[i+1])

else

sortingRequired = false;

end if

end for

n = n + 1

end for

end while

end bubbleSort

Despite bubble sort’s inefficient ordering of unordered lists, our bubble sort algorithm will now decrease in computational intensity according to how close our list is to being sorted. For that reason, bubble sort is often employed computer graphics applications where it is used to locate and fix errors in almost-sorted arrays. Bubble sort is also useful in applications where prohibitive wear and tear on the storage medium. Consider cassette tapes, where frequent winding and rewinding of a tape causes damage to both the tape and tape drive. Outside of these niche applications, bubble sort is rarely worth using due to how computationally expensive it becomes when sorting large, unordered lists.

Merge Sort

**Basis**

Merge Sort is a substantially more effective sorting algorithm that implements a “divide and conquer approach”. Essentially, it operates by splitting the computational workload of sorting an array into a series of simple array divisions, which can then be merged into a sorted array. Merge Sort has an average-case complexity of O(n log n), making it a much more efficient mechanism with which to sort an array than bubble sort or other O(n2) average-case sorting algorithms like Selection or Insertion Sort. ­­

**Implementation**

One way to sort a list using Merge Sort would be to use a recursive solution like the following:

begin mergesort(list)

int n = length of list

if (n == 1)

return list

end if

list1 = list[0] … list[n/2]

list2 = list[(n/2)+1] … list[n]

list1 = mergesort(list1)

list2 = mergesort(list2)

return merge (list1, list2)

end mergesort

Where the merge method operates like this:

begin merge(list1, list2)

int[] list3

while(list1 and list2 have values)

if (list1[0]) > list2[0])

append list2[0] to list3

remove list2[0] from list2

else

append list1[0] to list3

remove list1[0] from list1

end if else

end while

while(list1 has values)

append list1[0] to list3

remove list1[0] from list1

end while

while(list 2 has values)

append list2[0] to list3

remove list2[0] from list2

end while

return list3

end merge

Merge Sort can be used in a range of different applications as it is substantially faster than the O(n2) sorting algorithms discussed previously. Despite Merge Sort’s efficiency, many programmers default to using Quick Sort, another divide-and-conquer algorithm with marginally faster performance than Merge Sort[[1]](#footnote-1).

The Fibonacci Sequence

**Basis**

The Fibonacci sequence is a series of numbers where each number in a given sequence is the sum of the two numbers that precede it.

We can use the following equation to describe the Fibonacci Sequence:

We can implement the Fibonacci Sequence with and without the use of recursion. Below is one implementation using recursion, and one implementation without recursion.

**Recursive Implementation**

begin fibonacci()

Int n = user input

fib(n)

end fibonacci

begin fib(int n)

if (n <= 1)

return n;

end if

return fib(n-1) + fib(n-2)

end fib

This recursive implementation will take a user input and calculate the Fibonacci Sequence to the nth place, where n is a number input by the user. It does this by continuing to initialize instances of the fib() method where n = n-1 or n-2 until reaches an instance of the fib() method where n <= 1 (it’s base case).

Once is has initialized as many fib(n) methods as it needs to reach its base case, it can begin to resolve all of the fib() methods currently on the stack, calculating the Fibonacci sequence up until the nth position.

**Non-recursive implementation**

begin fibonacci()

int n = user input

int a = 3

int b = 1

int c = 0;

for (int i = 3; i < n; i++)

c = a + b

int x = a

a = c

b = x

end for

return b

end fibonacci

In this implementation we use iteration to the value of n to replace the recursive call we made earlier. In order to calculate later digits in the fibonacci sequence we can start our calculations at [0, 1, 1, 2], shortcutting (0 + 1) and (1 + 1). We use the integers a b and c to stare the most recent number calculated, the second most recent number calculated, and the outcome of the current calculation. I chose to initialise I at 3 as we’ve already calculated the first 2 digits of the sequence.

Exponentiation by Squaring

**Basis**

Exponentiation (xn) is an operation where the base (x) is multiplied by itself n times. A computationally efficient algorithm for exponentiation is exponentiation by squaring.

**Recursive Implementation**

Exponentiation by Squaring can be implemented like so:

begin exp(x, n)

int result = 0;

if (n<0)

result = exp(1/x, -n)

else if (n=1)

result = 1

else if (n%2=0)

result = exp(x\*x, n/2)

else

result = x \* exp(x\*x, (n-1) / 2)

end if

return result

end exp

**Tail Recursive Implementation**

In order to turn our Exponentiation by Squaring algorithm into a tail recursive algorithm we need to introduce an additional function, which could be done like this:

begin expInit(x, n)

exp(x, n)

end exp

1. Quick Sort and Merge Sort have the same average-performance complexity of O(n log n), but in real world usage well implemented Quick Sorts have been observed to be marginally faster / less computationally intensive. [↑](#footnote-ref-1)