Benjamin Pritchard

217163073

# SIT221 Data Structures & Algorithms

## Assignment 1

### Task 1 Implementation of the Merge Sort algorithm

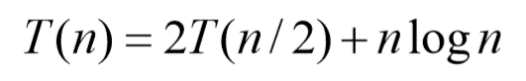
Part 1.2 - Master Theorem for Merge Sort

1. Divide - split the array in half (use floor and ceiling functions if array size is odd)

*a* subproblems of size *n/b*

1. Conquer - use recursion on the sub arrays
2. Combine – combine the result from step 2 – this is linear time





a = 2

b = 2

f(n) = n log n

n^log b a = n^log 2 2 = n^1 = n

j = 1

k = 1

This is case two of the Master Theorem as k >= 0



This shows that Merge sort is bounded by Big Theta of n log n

This is because the split happens log n times, followed by a combine (linear time)

### Task 2 Additional Optimization for Quick Sort

Part 2.1 – Quick Sort Analysis

1. Worst case program stack size is n-1 elements

A stack size of n-1 will occur when there is a list of distinct elements that are already sorted. Also the choice of a most sub optimal pivot – say the first or last element in the list, causes the three sub list to be as follows - one list (size n-1), the pivot (size 1) and the subsequent list (size 0). This is the least desired outcome and causes the stack to be size n-1. This happens because each recursive call is only one element less than the last call.

Quicksort can be a gamble, because without explicitly choosing the pivot (for example the median index) you don’t know how suitable the random choice will be. Another possibility would be to use a non-recursive sort with minimal swaps when the array becomes sufficiently small.

1. Stack depth of O (log n) is desired and expected?

A Log n stack is desirable as this is a reasonable size to compute for most inputs. When comparing this to the less optimal situation when the stack depth could possibly be size n. This could obviously cause overflow on larger sets.

Using a stack to process each sub problem

Pop the top sub problem and split it in two – then push the two sub problems.

It is desirable to push the larger of the two before the smaller.

1. How randomised Quick sort should be modified to guarantee stack size of O(log n) ?

With the use of in place sorting we use a small amount of extra memory on top of what is allocated for the input. The input sequence itself is used to store the splits for the recursive call. The two variables keep track of the leftmost index, the rightmost index. When the list is scanned for sorting the left and right indices swap the pair and advance towards each other. When the indices pass each other the algorithm is complete and recurrence occurs on the two sub lists again.

To also use space more efficiently we push the larger part to the stack and then do computations on the smaller portion. The result of this is that the larger part of each recursive call is smaller than the smaller part from the previous step.

Conversely if we were to push the smaller subset to the stack we may end up in the worst case with a stack depth of n/2.

### Optimised Quicksort Pseudocode

Quicksort(arr[], low, high)

if ( high <= low ) return

else

assign an index of low to pivot

Scan the list left to right until reach a value equal to the pivot or equal to high -> low++

Scan the list right to left until reach a value equal to the pivot or equal to low -> high--

If ( low <= high )

Swap the values -> low++ / high --

Quicksort (arr[], low, low-1 )

Quicksort (arr[], low + 1, high)

### Task 3 Comparing Sorting Algorithms

Part 3.1 – Algorithm analysis

Runtime – This is the period of time that the program is running. More specifically, the time between when the program is opened (executed) and when it is closed (quit). Runtime is synonymous with time complexity as they both measures the run time of the algorithm. Typically runtime will vary with the size of input, and is said to be a function of the input - f(n).

Stability – Makes sure that two elements with the same value are in the same order after the array is sorted. Stable algorithms may be slower than unstable as they can often have larger CPU and memory space complexity.

In place processing – an algorithm is said to be when there is no need for an auxiliary data structure. They usually occupy a constant amount of space O(1). Being in place also includes no need for extra function calls or pointers. With an in place algorithms the Input is usually overwritten by the outputs, but note there may be a small amount of memory used for auxiliary variables.

Dynamic Sorting – These algorithms receive the input in pieces and decisions are made with partial information. It Must serve requests in the same order as they arrive and once the input is received it cannot go back. A partial solution is provided at each step but its possible that the decision is be sub optimal. Within dynamic sorting there is a property called the “competitive ratio”, which simply tells us the cost of an optimal solution.

Insertion sort

Runtime – O(n^2)

Space complexity – O (1)

Stability - Yes

In-place processing - Yes

Possibility for dynamic sorting – Yes

Application – Good for small or partially sorted list

Notes – can also be used for small sub lists in recursive algorithms

Heap Sort

Runtime - worst case O (n log n)

Space complexity – O (1)

Stability - No

In-place processing - yes

Possibility for dynamic sorting

Application – Graph algorithms

Notes – similar in design but slower than quicksort

Quick sort

Runtime – worst case is O(n^2) but average case is O (n log n)

Space complexity – O (log n)

Stability - No

In-place processing – Yes, with O (log n) stack

Possibility for dynamic sorting

Application – when limited memory is a concern

Notes – may be slow on some inputs without randomized pivot

Merge sort

Runtime - worst case O (n log n)

Space complexity – O(n)

Stability - Yes

In-place processing – No, must allocate memory for sorted output

Possibility for dynamic sorting - No

Application – hotel booking websites -> source data from numerous sources

Notes – divide and conquer algorithm

May use insertion sort on sub arrays which can improve memory usage

Bucket & Radix sort

Runtime – O (d (n + N)) -> is better than O (n log n)

Space complexity – Bucket -> O(n) / Radix -> O( n + k )

Stability – Yes for both if secondary algorithm is stable

In-place processing – Bucket sort -> No / Radix -> No. must use auxiliary data structure for “buckets”

Possibility for dynamic sorting

Application – Radix -> used on dense arrays

Notes – Bucket sort -> fast when the data is distributed even among the buckets

Can apply recursion or alternate algorithm on buckets

Radix sort -> similar to bucket sort but uses integer keys as groups

### Task 5 Problem solving skills

### Reference:

GeeksforGeeks. (2018). *QuickSort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/quick-sort/ [Accessed 18 Aug. 2018].

Growing with the Web. (2018). *Bucket sort*. [online] Available at: https://www.growingwiththeweb.com/2015/06/bucket-sort.html [Accessed 20 Aug. 2018].

dumasia (2018). *Merge sort analysis and its real time applications*. [online] Slideshare.net. Available at: https://www.slideshare.net/hawkeye0/merge-sort-analysis-and-its-real-time-applications [Accessed 20 Aug. 2018].

GeeksforGeeks. (2018). *Applications of Heap Data Structure - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/applications-of-heap-data-structure/ [Accessed 20 Aug. 2018].

Techterms.com. (2018). *Runtime Definition*. [online] Available at: https://techterms.com/definition/runtime [Accessed 20 Aug. 2018].