ASD Advanced Algorithms:

Analysis (Task Part 2)

2.1 – Introduction

In the course, a number of algorithms were explained, implemented in pseudocode and analysed. This report makes some explanations as to the running time / correctness of 4 of these algorithms and their implementation, however the relative time differences are the same implemented in any programming language, as algorithmic complexity is being measured. In the report, mergesort, the closest pair algorithm, randomised quicksort and the randomised contraction algorithm (Karger’s minimum cut solution) will be analysed and compared in their running time.

2.2 – Mergesort

Mergesort is the fastest deterministic sorting algorithm in existence. It runs in O(n log n) time, with n as the input array length, this is much better than the quadratic running time of other, simpler sorting algorithms like bubble sort, insertion sort and selection sort.

The O notation supresses several constant factors – Mergesort produces a sorted output array for every input array of n numbers, using at most operations. This could be shown by using a recursion tree. The recursion tree would show the levels of recursion (of which there are exactly ). At any level j of the tree, there are distinct sub-problems, and the input size of each of these subproblems is . As mergesort calls itself twice every time it is called, the number of subproblems doubles at each level, however the input size of the subproblem halves at each subsequent recursive call.

The number of lines of code that is used in one invocation of mergesort is roughly 6 (can be varied to more or less), as mergesort does 3 things:

1. It calls itself with the left half of the array (not counted)
2. It calls itself with the right half of the array (not counted)
3. It calls a **Merge** subroutine to merge the products of 1 and 2 together, which takes about 6 lines of code.

So, if the input size of a single invocation of **Merge** is **m** it will use at most around 6m lines of code. So, the total number of operations at level j is the number of sub problems times the amount of work per sub problem, giving:

Simplifying this we get (the two s cancel each other out), a number independent of j. This means we do 6n operations at level 1, at level 2, etc. This gives an overall running time of:

(the amount of work per level times the number of levels). Expanding this we get , simplifying to O(n log n).

2.3 – Closest Pair Algorithm

The closest pair problem is a computational geometry problem: given n points in metric space, find the pair which has the smallest Euclidian distance between them. This can be solved using the divide and conquer approach in O(n log n) time. The algorithm is as follows:

1. Sort all given points according to their x co-ordinates
2. Split the set of points into two equal subsets (by a vertical line, x = the middle element)
3. Recuse on the left and right subsets, giving the minimum distance between pairs on each side
4. Find the minimum distance for all split pairs
5. Return the minimum of these three

Here is the first mathematical analysis made is that D(n), the number of pairwise distance calculations in the closest pair algorithm when run on n >= 1 points:

The second mathematical analysis is the number of comparisons between co-ordinates / distances in the algorithm when run on n >= 1 points:

To achieve a running time of O(n log n) the algorithm sorts by x-co-ordinate at the top level of recursion only, where each recursive call returns delta and a list of all points sorted by y-co-ordinate, and the points are sorted by merging the two lists. This gives us a running time of:

due to both the sorting and accessing all elements.

2.4 – Randomised Quicksort

Randomised quicksort is an algorithm that sorts an array by choosing a pivot at random and placing all elements less than it to the left of it and all elements greater than it to the right of it. This following analysis proves for every input array of length n, the **average** running time of the algorithm is O(n log n).

If C = #comparisons quicksort makes among pairs of elements in the input array, which is determined by how many times two elements i and j are compared, shown below.

We can say that if the outer sum i is fixed, the inner sum is

As the largest value of 1/(j-i+1) is if j is i+1, giving ½, and then all subsequent values are smaller. So, we can say that:

The sum in our new equation is only logarithmically large. This means that:

which gives us the number of comparisons, however the number of comparisons dominates the running time of quicksort. Also, as the implementation of quicksort can be in-place, no extra storage is needed to have the algorithm work at optimum level.

2.5 – Karger’s Minimum Cut Algorithm

In the minimum cut problem, we are given an undirected graph G = (V, E), where parallel edges are allowed, represented in an adjacency list, and the coal is to compute a cut with the fewest number of crossing edges (A, B). Let k = # edges crossing the two halves of the graph (sorted by x-coordinates). We can call these edges F. If an edge of F is contracted at some point during the iterations, the algorithm will not output (A, B) – it will be incorrect, and if only edges inside A or inside B are contracted, the algorithm will output (A, B). So, **Pr[output is (A, B)] = Pr[never contracts an edge of F]**, let this be Pr[]. Our goal is to compute this probability. If Si is the event that an edge of F is contracted in iteration i, we need to find .

The probability that an edge F is crosses the minimum cut is (call this k/m). In the first iteration of the algorithm: since , . And as **.** In the second iteration, we know that . To find #remaining edges, we know that all nodes contracted in the graph define cuts in G with at least k crossing edges. All possible numbers of incident edges are at least k. So:

Therefore,

For all iterations then, our success probability is:

This cancels out to give . This is a low success probability, however there are possible cuts!

2.6 – Comparison of Algorithms

Contraction Algorithm

Three of the four algorithms looked at here run in O(n log n) time, however Karger’s contraction algorithm for minimum cut runs in quadratic time – comparatively much slower than these other algorithms, and on top of its low success probability in producing the correct output makes it seem less desirable to use. However, in terms of its implementation and understanding it is relatively simple, especially compared to the closest pair problem which requires much more understanding. Additionally, the contraction algorithm deals with graphs, whereas the other algorithms involve arrays – generally speaking a “large” array has many more elements than a “large” graph.

Mergesort vs. Randomised Quicksort

Both these algorithms work to solve the sorting problem in the best time possible – and these are the two fastest algorithms to use. However, it is clear that mergesort is more practical algorithm to use in terms of runtime, having an upper bound of O(n log n) whereas randomised quicksort has a complexity of O(n log n) on average. So, with poor pivot choices, quicksort could run as slow as O(), but with mergesort, it will consistently be O(n log n). In terms of implementation, mergesort is more simple and does not require use of random functions. However, randomised quicksort in terms of its method of solving the problem has more ingenuity to it than mergesort.

2.7 – Sources

<https://courses.cs.washington.edu/courses/cse421/11su/slides/05dc.pdf>

Divide & Conquer, Sorting & Searching and Randomised Algorithms (University of Stanford, Coursera)

* Mergesort Analysis Lecture
* Quicksort Final Calculations Lecture
* Contraction Algorithm Analysis