SECTION A

1 (a) **Suppose algorithm A has cost O(n) and algorithm B has cost O(n^2). Could there be circumstances when it would be rational to use algorithm B? Explain.** [10%]

Yes there might be. Big O is only about worst case cost, and B could be better on average than A. Or A might be horrid to implement but B simple, and all the cases you are interested in reasonably small. But **usually** A is liable to be better.

(b) **In an ideal world each time Quicksort selects a pivot this would lead to the data being partitioned into two equal sections. When that happens Quicksort can complete in time proportional to $n\log(n)$. Suppose instead that the pivot always splits the data at the 1% position, in other words the two sections after partitioning have sizes in the ratio 1:99. What are the cost consequences?** [10%]

Only a constant slow-down. For some fixed k (actually $k = 69$) we have $0.99^k < 0.5$ and that shows that after k passes you have certainly got down to half the original size, so the scheme described is at worst k times slower than the case with an ideal split.

(c) **You wish to find the item that is at rank $N/100$ from a set of N items. You have two possible methods to use. If you use the guaranteed linear-cost median finding algorithm you can assume the time taken will be around $10N$. As an alternative you can start using heap-sort on the N items and stop as soon as you have found the information you need. Which is liable to be better, and does you answer depend on N?** [10%]

This is comparing $10N$ against $N + (N/100)\log(N)$ where in the heapsort case the N is for heapify and the other bit os removing items. For all feasible N this is better than the linear cost median finder (even if you allowed say $2N$ for heapify). Of course theoretically for huge N the log term makes it worse.

(d) **Now you wish to identify the median, in other words the item whose rank is $N/2$. Is your answer still the same as in part (c) above? Explain.** [10%]

Now we have $N + (N/2)\log(N)$ and when $\log(N)$ gets to say 20 this becomes worst than the linear cost method. So at around a million items there is a turn-over.

(e) **What will be the worst case, and hence what will be the worst case cost for**

releasing a block of memory back into the control of a Buddy system allocator? [10%]

Log of heap-size, because the worst case will be deleting a unit size block from an otherwise empty heap.

(f) What advantages do Red-Black trees have over ordinary binary-search trees, and what disadvantages or extra costs do they have? [10%]

Guaranteed logarithmic cost for adding looking up items, against linear worst case. But Red-black trees need an extra bit to tag nodes as red or black, are a bit slower adding new data because of the need to maintain balance and are distinctly more complicated to implement.

(g) You have set up a hash table (which stores everything within the array that represents the table) and added items to it so it is now just two-thirds full. You now wish to remove one of the items that you inserted earlier. Explain how you do this and estimate the cost involved. [10%]

Find item in the table (at unit expected cost – more pedantically if load factor is 2/3 then at cost 3. Replace item there with a "tombstone" marker.

(h) One way of implementing Shell's Sort uses a sequence of strides using a recurrence $s_{k-1} = 2s_k + 1$ and arranging that the final stride in the sequence is just 1. Would it be reasonable instead to use a sequence of strides that was just powers of 3, as in one that finished $\ldots 27, 9, 3, 1$ instead of $\ldots 40, 13, 4, 1$? If not what disadvantage might arise from the slightly simpler sequence? [10%]

The use of mere powers of three would keep the subsequences at different multiples of 3 segregated until the very final step, so an initial sequence such as $0, 0, 9, 0, 0, 9, 0, 0, 9, \ldots$ would not have the zeros and nines shuffled at all until the very end, to bad effect.

(i) The word "engineers" has 3 instances of the letter "e", two of "n" and one of each other letter it uses. You will also need a code ω that will mark the end of file and which should be treated as being much less common than anything else. Using just the letters present in "engineers" plus ω with the relative frequencies indicated here create a Huffman coding tree that could be used to send streams of letters. [10%]

e:3, n:2, g,i,r,s:1,ω:0 so group (say) s+ω : 1,g+i:2, r+s+ω:2, then n+g+i:4, e+r+s+ω:5 and hence the full tree.

(j) Why is it useful to consider analysis in terms of amortised computing time when discussing garbage collection? Define any technical terms you need to use, and explain any limitations that balance the strengths that you identify. [10%]

Garbage collection happens as part of a storage management strategy, and you are often concerned with evaluating the cost overhead of this. An amortised analysis allows you to think of teh cost of a garbage collection as attributable to all the allocation (and perps any explicit deallocation) steps that led up to it. The key limitation of amortised analysis is that it does not address the effect to which a garbage collection can disrupt real time responsiveness.

2 (a) Explain how the data structure known as a Heap and as used in Heapsort treats a sequence on N items stored in an array as if they were arranged in a very well balanced binary tree. [15%]

(b) A programmer seeks to impose essentially perfect balance on all the binary search trees they use, and at the same time avoid the need for pointers. They decide to use the heap representation as in part (a). Explain in detail how to look up a key in a binary search tree stored in this manner (supposing that the tree already exists) and explain the costs involved. [15%]

The top of the tree lives in the array at position 1, and any item at position j has left and right offspring at locations $2j$ and $2j+1$. Given that lookup is exactly as for a normal binary tree. Note that what is being looked for here is that the pointerless and almost perfectly balanced representation of a TREE from the heap is now being used with the binary searct tree ordering of data rather than the heap one. Thus the binary tree concerned will be very nicely balanced - but very inflexible.

(c) Suppose N items are to be formed into a binary seach tree in this form. Can you tell which item will end up at the top of the tree (i.e. at the first position in the array)? If so which item will it be, or if there is flexibility comment about any items that could *not* end up in the top position. [15%]

There is no flexibility at all (supposing all keys are distinct). The easiest way to charactersise which item ends up at the top is to consider two cases. The first where the bottom row of the heap is less than half full and the other when it is more than half full. If less than half full then the tight hand side of the tree is a perfect tree with a power of two (less one) items in it. If more than half full then the left hand tree is perfect. So you have either $n-2^{h-1}$ or 2^h items to left of the root where h is the height of the tree.

(d) Given your data as a sorted list, hence or otherwise develop a procedure that can form it into a properly arranged search tree filling just the N initial entries in the array. How long will the process take? [15%]

Pick the item to go at the top and put it where it has to go. Now recurse to distrute first items less than that and then items more. That should have linear cost.

(e) Now the programmer has established a perfectly balanced search tree, but wants to add a new item. Are there circumstances in which no other data will need

moving, and are there cases in which every other item must be moved? [15%]

Eg if the whole tree currently holds $2^n - 2$ items and the new item is bigger than all there so far you can merely tag it on the end. Often if you add a new smallest item you will need to move everything.

(f) Invent and describe an algorithm that inserts a new item such that the insertion procedure performs approximately $\log(N)$ comparisons. You may disregard all cost of data movement and other administration. [25%]

Easiest way is probably to "cheat" and flatten the tree into a sorted array, use binary search to find where to insert. Insert than re-form into the tree. Cost is logarithmic in comparisons but linear in data movement. Any scheme will have worst case linear data movement cost (and so reall this is very much like binary insertion sort!) so it is possible in many cases to traverse the tree in the order that corersponds to the sorting and shift stuff along as you go. But even that is messy and you need to watch out eg when the tree changes height. The simplest approach will suffice to get most of the marks here.

3 Data will be transmitted and the raw channel can cope with a set of 10 symbols, 0-9. The messages to be sent, however, use a much larger alphabet, so the engineer who is setting everything up designs a variable length encoding scheme very loosely based on the idea of Huffman Encoding. The digits 0-4 are used to represent the 5 most commonly used characters. Two digit codes of the form 50-59, 60-69 and 70-79 provide a less compact way of denoting the next 30 characters. Then 800-899 and 900-998 provide 199 more. The final code 999 is used to mark the end of the data.

The engineer now wishes to use a further layer of file compression technology to convert this stream of symbols into a stream of bits.

Two schemes are under consideration – Lempel Zif and Arithmetic Coding. In each case the compression will work with one symbol at a time, i.e. it is not considered proper to decode the existing stream into the string of items it represents. You must handle it digit by digit, but if relevant you may maintain some status or history information.

(a) For Lempel Zif comment on how the behaviour of compression is liable to be affected by the variable length encoding. Describe how the method works, giving sufficient detail to reveal any special behaviour which may arise. [40%]

L-Z will consolidate pairs of characters together to make its symbols. Its general working is bookwork to sketch. It will if you have input say "2,52,2" it will first record the pairs 25, 52 and 22. And if either 2,52 or 52,2 occurs again these will be useful. It seems probable that the odd input encoding will not have any significant adverse effect on it.

(b) For arithmetic coding, what other information would you need, if any, and how would you use it? Describe the coding process, commenting on how the statistical modelling involved is affected by the nature of the raw data. [40%]

For arithmetic coding you need an explicit probability model, and the probabilities here are liable to depend whether you are at the start of or within an extended symbol. If the probabilities in the input text perfectly matched the variable length coding scheme then Arithmetic coding would simplify to being merely a decimal to binary conversion.

(c) Which of these two compression methods would be preferable here, and why? [20%]

I think that in this case the simple automatic manner in which L-Z builds its own

statistical model make it look very attractive, but this section of the question provides an opportunity for fairly open ended commentary, and any good points made will gain credit.

4 You are given an undirected graph that has integer-valued weights associated with each edge.

(a) Define a *minimum spanning tree* for such a graph and explain an algorithm that will find one. Your answer should include discussion of particular bottlenecks or ways in which special sorts of graphs could impact costs. You should also provide an upper bound on the cost of your method in terms of the number of vertices and edges involved. It will be sufficient to use simple techniques for any sub-tasks that need handling. [35%]

Place all egdes in a priority queue ordered by their weights. Iterate a step that picks the next shortest edge and if it would not form a cycle with existing selected edges add it to the forest that will end up as your spanning tree. Sub-algorithms are the priority queue and testing if you have would create a cycle. Cost of priority queue (if a heap) = E to set up (V vertices and E edges in graph) + $V\log(E)$ to remove enough items. Simple way to check for look might have cost V giving total bounded by V^2. Very dense or very sparse graphs might be special cases where either extra optimisations or other techniques (ie Kruskall vs Prim) could end up best.

(b) In the same graph a user identifies two vertices, A and B, and your task is to find a shortest path from A to B through the graph, treating the weights on the edges as distances. You are expected to report both the length of this path and the chain of edges that make it up. Present and analyse an algorithm for solving the problem. [35%]

Dijkstra's ink-blot style algorithm is the canonical response here. Again the main sub-algorithm is some form of priority queue. Simplest implementation can have V^2 cost. Again sparse vs dense graphs may matter. Note that question does not demand detailed analysis of exactly how!

(c) You are now told that around two thirds of the edges in such a graph will have weight 1, while the remaining edges will have weights of 2 or 3. Discuss whether, and if so how, you would alter your algorithms of parts (a) and (b) to take advantage of this new information. [30%]

This new information allows you to replace a general priority queue with something much easier! Eg for MST you merely partition edges into those of lengths 1, 2 and 3 and process all the 1 edges first. For Dijkstra a neat trick is to add in new phantom vertices to divide every 2 or 3 edge into parts so you ende up with a graph with only 1-edges. Then the ink-blot becomes close to trivial to implement and the cost becomes linear.

END OF PAPER