ENGINEERING TRIPOS PART IIA 2004

Solutions to Module 3I1
Data Structures and Algorithms
Principal Assessor: Dr A Norman
Second Assessor: Professor A Hopper

SECTION A

Attempt all parts of this question

1. You should answer each of the following parts with a short-answer, quoting relevant results rather than proving them. Give names for algorithms you need to refer to and sketch methods or justifications of claims you make. Credit will be given for the clarity and succinctness of answers as well as for basic factual accuracy.

(a) Is it the case that every function of the form $f(n)=An^k$ (with A and k constant) satisfied $f(n)=O(2^n)$? [10%]

In brief, yes. Write out the definition of big-O and maybe consider r(n) = 2^n/(A*n^k).
Then r(n)/r(n-1) = 2 ((n-1)/n)^k = 2(1-1/n)^k. For any k first select n such that this ratio is > 1.5 then from there on 2^n catches up and overtakes An^k enthusiastically.

(b) G is a connected graph with n vertices and with a weight associated with each edge. F is a sub-graph of G consisting of the n-1 shortest edges in G. Is F necessarily a tree and does it necessarily span the vertices of G? Explain. [10%]

A spanning tree must have n-1 edges, but if the n-1 shortest edges form one of more loops you will have neither a tree nor a spanning set.

(c) A tree has a fan-out of 5. In other words each node in it has just five nodes as its immediate offspring. Every leaf in the tree is at the same distance, d, from the root, and d is fairly large. About what proportion of the nodes in the tree are at distance d from the root (the rest are of course at distances less than d)? [10%]

total nodes = 1 + 5 + 5^2 + 5^3 + ... + 5^d, while the number of leaves is just 5^d. Summing the GP we get (5^(d+1)-1)/4 so divide 5^d by this to get around 4/5 as leaves.

(d) Once a pivot has been selected a key part of quicksort involves partitioning the original data into a set of items less than (or equal to) the pivot and another set of items that are all greater than the pivot. Explain how to achieve this when the original values are

stored in an array and you do not have any additional workspace to use. Ensure that your method will work in extreme cases, such as when all the values are the same, or when the pivot value is the largest or smallest value in the array. [10%]

suppose array runs from 0 to n. set two pointers i=0, j=n.
While j>=i and item at j > pivot decrease j. Because the pivot is in the data somewhere this stops with j>=0. Now while i<=j and item at i <= pivot just increment i. Now i points to an item that is > pivot and j to one that is <= pivot. If i<j exchange these two values and continue. When you reach j<=i you have completed. See Cormen et al for the shortish program!

(e) Explain the concept of a "winding number" and show how it can be used to determine if a point is inside a polygon. Are there extra issues if the polygon might be concave? [10%]

take your point p and a vertices (in order) a1...an of the polygon. Add up the angles (a2,p,a1), (a3,p,a2),...,(a1,p,an). If the result is zero then the point is outside the polygon. Otherwise the result is a multiple of 2 pi indicating how many times the polygon "winds around" the point. Being concave is not an issue, but polygons that have edges that cross the polygon are an oddity and winding numbers give one possible interpretation of "insideness" in such cases.

(f) What is a "garbage collector" and what problem does it solve? Give a very brief sketch of the actions involved in some simple form of garbage collection. [10%]

A storage management strategy that allocates store in some simple way until there is no avaible memory, but which does nothing about released store. When memory one scheme scans from all roots and MARKS all memory blocks that can still be reached. Then then SWEEPS to collect all unmarked memory which I makes available for re-allocation.

(g) Why is it often said that sorting n values must necessarily cost $n \log(n)$ steps? Are there any cases where this assertion is false? [10%]

n distinct items can be in n! orders, and so it must take log<2>(n!) bits of information to decide

which. A pairwise comparison amounts to extracting just one bit. log(n!) is around n log(n).

If you sort by a method other than comparisons, or if you can exploit knowledge about which orderings of the input are probable, or if the items to be sorted have many repeats then the above analysis does not hold.

(h) Suppose that a priority queue is to be implemented. The normal implementation would probably be to use a heap, but for some reason a programmer has tried to use red-black trees. How can the smallest item in the queue be found? Starting from an empty queue how much work is required to build a queue with n items in it? [Note that you are not required here to discuss the process of removing an item from the queue.] [10%]

The smallest item is the leftmost leaf of the tree and can be reached in log n steps. Building a tree with n steps takes log(current size) for each insertion and hence log(n!) steps in all. About n log(n).

(i) What problem do "splay trees" address and what particular advantages might they have over other competing technology? [10%]

Normal binary search trees are easy to implement and provide a convenient way to model a table. But they can in bad cases get ill-balanced and costs can grow badly. Splay trees eliminate this possibility (in an amortised sense). Two competing technologies: regular binary search trees: splay trees do not have the worst-case effect. Red-block or other balanced trees: splay trees are probably easier to implement and go faster. They also have good behaviour if the items touched in the tree are accessed with imbalanced probabilities – especially if some are never touched at all. Splay trees then have costs driven by the nodes that are touched.

(j) Why is it not possible to say that simple insertion insertion sort has cost $\Theta(n^2)$? Would it be safer to say it had cost $\Theta(n \log(n))$? [10%]

simple insertion sort is big-O(n^2) but in good cases it can have linear cost, so the big-Theta lower bound may not apply. It is neither big-O nor big-Theta of n log n.

SECTION B

2. (a) A complete compression algorithm must use some statistical or predictive model to detect expose redundancy in the original input, and some packing procedure that exploits this information to create a bit-compact representation of the data. Explain how the Burrows-Wheeler transform works and what it achieves in this context. [30%]

Take a block of text, and form all rotations of it. Sort these. Output the LAST character of each row in the sorted list. By the rotated-text property this can also be seen as the column at position (-1) and hence the character preceding the first column. Sorting brings substrings that are similar together, so the column (-1) amounts to a prediction (in backwards direction) of what comes before a string.

(b) The output from the Burrows-Wheeler transform is generally expected to be readily compressible using a combination of run-length and Huffman encoding. Give a brief explanation of these two techniques and how the corresponding decoders work. You do not need to give elaborate details of how to construct a Huffman encoding – just an overview. [20%]

Run length: send eg ABBBBABBBBAAAAACCC as 1A4B1A4B5A3C where runs of the same character are specified as a value/count pair. Decoding is obviously easy!

Huffman: build a binary tree from your symbols such that each branch tries to have the same probability to left and right. Encode symbol as bit-string of a path (left/right) down tree to the symbol. Decode the same way! Uncommon symbols will end up deep in the tree and get long strings of bits. Common symbols are at lower depths and get shorter strings of bits.

(c) Explain why it is possible to reverse the Burrows-Wheeler transform and hence recover the original data from its compressed form. [30%]

Write "column -1" of the sorted block. Then column 1 must have the same symbols but in sorted order, so is easy to fill in. Now we have 2 columns, -1 and 1. Columns 1&2 must be these pairs of symbols sorted alphabetically, so we can find column 2. And so on to find the rest.

(d) A normal explanation of Burrows-Wheeler treats it as working with data that is a block of characters. Discuss whether it would be possible to use the same procedure but to treat the blocks as blocks of bits rather than characters. Comment on whether this, if possible, would seems a good idea. You may wish to consider complexity of implementation, cost of performing a compression and any predictions about how the change would impact the ability of the algorithm to predict patters in the data. [20%]

Of course it would be possible. Blocks would feel 8 times longer so sorting them would be yet more work. But the "final column" would be just of bits and so run-length would maybe be especially easy to implement! But I guess mostly it would not make very good sense in that the probabilities you get when you compress a text file are based on the letters, and the rendering of letters to bit-strings ought not to reveal any new patterns so my feeling is that compression ought not to be much better or indeed not much worse buts costs are probably way higher. [this section provides scope for open ended discussion and there is no single right answer]

3.

(a) Explain the general idea of a hash function. If the data you are working with involves strings of characters, suggest a scheme that might be used to compute reasonable hash values. [25%]

have a function h(d) such that as d ranges over the data you have h(d) appears like a (for choice) uniformly distributed random number. But h(d) is deterministic. If a string is c1,c2,c3,... then having a sequence of randomly or arbitrarily selected weights w1,w2,... and computing
(w1*c1 + w2*c2 + ...) mod N
is known as "universal hashing" and tends to work well.

(b) How do you use hash functions to implement a table where you can record key-value pairs and access stored data efficiently? If your hash table has space for a maximum of n records in it but only k are in use at some stage estimate (assuming that the hash function you use has perfect statistical properties) the number of table accesses (probes) needed when inserting a new entry into the table. [25%]

h(d) is the first choice place to look for data d. Then have some sequence after that (simplest is just add one to location) to search after a clash. Let a=k/n be the loading of the hash table, then you get probability of (1-a) of insert OK with one probe, a(1-a) with 2,
a^2(1-a) of 3 so we have total cost to insert of
(1-a)(1a + 2a^2 + 3a^3 + ...)
and I think that sums to 1/(1-a).

(c) What is Larsen's method of dynamic hashing, and how can it be used to organise data stored on disc? Why might it be used? Explain how new data can be added and existing data retrieved using Larsen's method. [25%]

It can access existing data on disc with just 1 disc read. Uses an in-memory "signature table" with on integer entry per disc block. One trial to access data is: Compute h(d), s(d) two hash funs on the key, d. From h(d) derive a block number b which data might be in. If signature table(b) >= s(d) then it will be and so read from disc. Otherwise this probe failed so try another. Must use independent new h and s hashes each time.

To add data start as for retrieve. If block fetched has enough space in it just add data there – no trouble. If not eject from block item with largest signature, and decrease entry in signature table to reflect the new threshold. This leaves you with the ejected item to re-insert somewhere else, and if you are lucky even though that starts sort of from scratch you will manage after not too many tries.

(d) Describe a method for using hashing to speed up the search for a particular target string in a very long sequence of characters. Discuss its typical and worst-case costs. [25%]

Select a carefully suitable hash value of your pattern, and compute the same function at each character posn in the source string. If the hashes match then you have a probable match, so do a full check and if hashing is working a probable match will almost always be a real one. The key to making this work well is to have a hash function such that if you have computed h(c1 c2 c3 .. cn) you can derive h(c2 c3 c4 ... c(n+1)) efficiently, and universal hashing with weights that form a geometric progression (mod N) does this pretty well.

4. The normal implementation of a heap keeps data in an array, and a node at offset k in the array is interpreted as having its two child-nodes at offsets $2k$ and $2k+1$. The tree representation this gives is always very well balanced, and so if there are n items in the heap the height of the tree is $\log_2(n)$. The main operations on heaps have costs that are proportional to the height of the tree.

(a) Show how to interpret offsets in an array to represent a balances tree with fan-out 4 rather than 2. [10%]

Let the root node be at 0. Then the 4 children of the node at position n could be at 4n+1, 4n+1, 4n+3 and 4n+4.

```
                    0
        1           2           3           4
  5 6 7 8     9 10 11 12 ...
```

(b) Explain the relationship between the height of your new 4-way branching tree and the original 2-way branching one. [10%]

The 4-heap has height log<4>n where the 2-heap had height log<2>n, and so the new height is just half the old one.

(c) Since heap operation costs depend on the height of the tree, your new scheme might have different costs to the original binary heap. Describe algorithms to rearrange arbitrary initial data so it satisfies the heap property and to remove the top item from your 4-heap. Discuss the cost growth rates and implications of your procedures and compare them with the normal implementation of heaps. [80%]

Well actually they are just the same as an ordinary 2-heap except that where in a 2-heap you have 3 items to compare in each branch-point here you have 5. So this bit is just a re-description of how an ordinary heap works with the wider comparisons in there. Because this makes it an adaptation of bookwork I will not recount full details here.

Wrt costs, you have costs that are halved because of the halving of heights, but scaled up by the

comparison between 5 rather than 3 items at each stage. At each stage you are doing a "find smallest" and on 3 that takes 2 comparisons, on 5 it takes 4. So overall that doubles the number of comparisons. Sticking the two together I reckon that things cancel out and the new 4-heap has costs very similar to those of a simple 2-heap. There are certainly no major savings and cost differences will end up depending on minor details etc.

END OF PAPER