Tutorial 7: Sorting and Graphs

(1) Suppose we have an array A with n distinct comparable keys. We define the *median* to be the key m such that $\lfloor n/2 \rfloor$ keys in A are strictly less than m and $\lceil n/2 \rceil$ of them are greater than or equal to m. (Intuitively speaking m is in the middle of the keys if ordered.)

(a) Suppose we have an algorithm findMedian that finds the median of an array in time $M(n)$. We use this to choose the partition element of quickSort, otherwise the algorithm is as before. Call this variant quickSortM.

i. What are the sizes of the two partitions produced at top level?

ii. Write down the recurrence for the runtime $T(n)$ of quickSortM.

iii. Suppose we have a version of findMedian that runs in time $\Theta(n)$ does this help with the worst case runtime of quickSort?
Hint: Use the Master Theorem.

iv. [Hard.] Describe an algorithm for finding the median in time $\Theta(n)$.
Note: Do not spend a great deal of time on this. There is a recursive algorithm that is discussed in [CLRS]. It is worthwhile spending 10 or 15 minutes thinking about how you might begin to produce such an algorithm. If you have time and feel you are succeeding then go further.

(2) Design an algorithm that sorts any 4 inputs using only 5 comparisons.

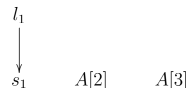
Is this the best possible? Generalise the argument for this part to show that sorting n items requires $\Omega(n \lg n)$ comparisons (if your reasoning for the first part is on the right lines then the general version is quite easy).

NOTE: We are given nothing about the inputs other than the means to compare any two and decide their relative order.

For the first part use partial order diagrams to guide your design. At the start we know nothing about the relative ordering of the inputs so our diagram is:

$$A[0] \qquad A[1] \qquad A[2] \qquad A[3]$$

(we have assumed that the inputs are given in an array A). Each comparison we make tells us something about the ordering of the given elements. We might as well compare $A[0]$ with $A[1]$. After this we have a partial order



Where l_1 is the larger of the two compared elements and s_1 the smaller. The remaining problem is to choose three more comparisons that get us to the

total order.

where the dots stand for the appropriate elements $A[0], A[1], A[2], A[3]$ with the largest at the top then the next largest etc.

For the second part consider the following:

(a) Each comparison leads to one of two outcomes, so the sequence of comparisons along with the possible decisions at each stage can be pictured as a complete binary tree, the *decision tree*. A path from the root to a leaf for a given input gives us the total ordering of that input. (At the root we know nothing about the partial order as we move towards a leaf we know more and more till we have the total order at the leaf.)

(b) Deduce that the number of leaves of the decision tree must be at least as large as the number permutations of the input (the tree picks out the correct one for any given input).

(c) How many permutations of the input are there?

(d) Now put the two preceding parts together to deduce the lower bound.

(3) For this question we work with undirected graphs. A graph G is said to be *bipartite* if its vertices can be put into two disjoint sets V_1, V_2 such that no edge has both endpoints in V_1 or both in V_2.

(a) Draw a simple example of a graph that is bipartite and one that is not.

(b) Describe an algorithm that takes a graph G and:

- if G is bipartite the algorithm assigns the vertices to V_1 or V_2;
- if G is not bipartite the algorithm reports this.

Your algorithm should run in time $\Theta(n + m)$ where n is the number of vertices of the graph and m is the number of edges.

Take care to choose an appropriate representation of the graph (you may assume that it allows each vertex to be marked with V_1 or V_2 as appropriate.

Note: You are not required to prove the correctness of your algorithm, of course it must be correct! It is probably best to describe your algorithm in clear English (using any appropriate ones form the notes as sub-algorithms).

(4) Consider the following part of an exam question and an "answer" given.

Question: Explain how to check if a binary search tree T contains a vertex with a given key k.

Answer: Use DFS or BFS. Each time you visit a vertex check if its key is equal to k. If it is equal then return TRUE otherwise return FALSE once the search finishes (all the vertices have been visited).

The algorithm given in the "answer" certainly returns the correct result but in fact it is inept. Putting that aside, it fails badly in a very important sense. Explain this by reference to the runtime; it will be helpful to prove that a tree with $n \geq 1$ vertices has exactly $n - 1$ edges (use induction). Note that we do not consider leaves as vertices and so there are no edges going to them. (Sometimes in diagrams we show the leaves explicitly and join them to the relevant internal vertices just to be clear but in a computer implementation the leaves are just null pointers so a search does not follow any edges to leaves).