ECE-250 – Algorithms and Data Structures (Winter 2012)

Practice Problems for the Final

Disclaimer: Please do keep in mind that this problem set does not reflect the exact topics or the fractions of each of the topics or the amount of work that will appear in the final. It is supposed to help you prepare for the final, but I'm including here mostly the second part (after the midterm). For the first part, you have assignments, practice problems, and you have the midterm with posted solutions to review the material.

Sorting algorithms

2 – Suppose we're using quick sort to process data that we're receiving from a connection in a networked system. We want to cover our system from the possibility of being "sabotaged" by hostile connections — we could receive data that is specifically crafted to cause quick sort to have its worst-case performance and thus make our system consume excessive resources and time (rendering it unable to efficiently respond to other connections).

(a) Assuming that quick sort simply chooses the first element as the pivot (instead of the median of first, last, and middle), what is the arrangement of data that produces the worst-case performance in quick sort? (that is, if you were the attacker trying to sabotage the system, what data would you have to send?)

(b) Suggest a simple strategy (hopefully requiring no more than linear time) to avoid the problem. That is, a strategy to guarantee that quick sort will run in $O(n \log n)$ most of the time, regardless of input data, even if this input data is maliciously created. Notice that in the context of this question, you're not allowed to change the way quick sort selects the pivot (in fact, you will hopefully suggest a strategy that works regardless of how quick sort selects the pivot).

Solution:

(a) We recall that the worst-case ($\Theta(n^2)$) occurs when the pivot splits the values into two chunks of size 1 and $n-1$ *at every iteration* (in fact, if it splits into a given, fixed size c and $n-c$, it also leads to $\Theta(n^2)$ — and when you think about it, it does not have to be a fixed size c; if it always splits into $O(1)$ and the rest, we do get $\Theta(n^2)$... right?)

So, if the values are in order (or in reverse order), then when picking the first element as the pivot, at every iteration we either get the lowest or the highest (respectively), and so we split the set into 1 and the rest, leading to worst-case performance (do notice that no swaps occur).

(b) No matter what the pivot selection strategy may be (thus, no matter what the "hand crafted" data looks like), a simple solution that runs in linear time is to randomly permute the values before running quick sort. The actual order of values being sorted is now *independent* of whatever action the attacker took to manufacture the data. We're sorting an array of randomly ordered values, and thus we have a probability of falling in the worst-case in the order of $1/n!$ (since we have $n!$ possible permutations, and presumably a fixed number of them would lead to a worst-case execution).

Priority queues and Heaps

5 – Suppose that we do a depth-first traversal in a max heap, stopping the recursion when we either reach a leaf node or when we reach a node with a negative value.

Prove that the run time is $O(p)$, where p is the number of positive values in the max heap.

Solution:

Hopefully, you did notice that this problem is essentially question 3 from assignment 4 (yes?). Well, not exactly; in A4, you were asked to come up with the strategy, and you were expected to show that the run time was $O(m)$. This is a disguised version of that solution; the given value being 0, and here we have a max heap instead of a min heap.

Graphs

6 – Prove that a Directed Acyclic Graph (DAG) must have at least one vertex with in-degree 0.

Solution:

We did this one in class (though it's not on the slides). The idea being, we prove by contradiction: we assume that no vertex has in-degree 0, and based on this assumption, construct a cycle (which is a contradiction, since we're working with a DAG), so the assumption that no vertex has in-degree 0 must be false, proving the statement.

The idea being that in-degree $\neq 0$ means at least one edge coming in to each vertex. Pick one vertex to start, and follow any of the incoming edges back to another vertex; from that one, since it also has in-degree $\neq 0$, we can follow that edge back to another vertex, and so on. After at most n times repeating this process, we have no more unvisited vertices, so if we follow the incoming edge back to some other vertex, we will get to a vertex that is already part of our "path", meaning that we have a cycle.

8 – Why is Dijkstra's algorithm a greedy algorithm? Does it have characteristics that fall in some other algorithmic paradigm? If so, why?

Solution:

It is clearly a greedy algorithm, since at each iteration, the choice is made as the vertex with minimum distance among the available ones. No "looking ahead" of any kind is considered.

It could also be considered an instance of bottom-to-top dynamic programming, in that the shortest path to the various vertices is being stored, and from there, we continue to incrementally build our solution.

To visualize this, we notice that a solution to the problem could use the following reasoning: the shortest path to vertex v is composed by the shortest path to one of the vertices to which v is adjacent plus v. If we recursively work this way, given the fact that multiple paths and even cycles may exist

between two given vertices, we would end up solving the same sub-problem over and over again (i.e., sub-problems overlap).

9 – Suppose a DAG is implemented with a class template with the following declaration:

```
class Dag
{
public:
    Dag();
    void insert (int u, int v);   // inserts an edge from u to v
    bool adjacent (int u, int v); // checks if v is adjacent to u
    bool connected (int u, int v); // checks if there is a path from u to v
};
```

Assuming that we're using an adjacency matrix, show an implementation (it can be in pseudo-code — but at least make it C++-like pseudo-code) of the insert method, ensuring that no cycles are created.

Solution:

The key idea is that for any two vertices u and v, if there is a path from u to v and there is a path from v to u, then u and v are part of a cycle. When inserting an edge from u to v, we're creating a path from u to v, so we verify whether there is a path from v to u — if there is, the insertion would create a cycle:

```
bool Dag::insert (int u, int v)
{
    if (connected(v,u))
    {
        throw std::invalid_argument ("Insertion would create a cycle");
    }
    else
    {
        adjacency_matrix[u][v] = true;
            // error fixed (was:  adj_m[u] = v; ...  sorry for the oversight!)
    }
}
```

Algorithm design techniques

12 – (a) Show that a naive recursive implementation of a function to obtain the n-th Fibonacci number (as $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ for $n > 2$, 1 otherwise) has non-polynomial run time.

(b) Write a C++ function that uses dynamic programming to make this run time linear (with respect to the value of n), using the top-to-bottom approach.

(c) Write a C++ function that uses the bottom-to-top approach and achieves linear run time.

Solution:

(a) The run time $T(n)$ of the above function is given by the following recurrence relation: $T(n) = T(n-1) + T(n-2) + \Theta(1)$

Because of the term $\Theta(1)$, we see that $T(n) \geqslant Fib(n)$, since it follows the same recurrence relation as the sequence of Fibonacci numbers plus some positive term. Since the sequence of Fibonacci numbers is known to grow exponentially, we have that $T(n)$ must grow at least exponentially (which is non-polynomial).

For (b) and (c), see course slides on Algorithm Design Techniques (it's *almost* explicitly answered there)

13 – Write a C++ function that solves the eight queens puzzle in a reasonably efficient way (i.e., using backtracking). You can assume that there is a function that checks whether the current configuration is valid (i.e., if in the given configuration, no queen is under threat of capture) — this function could have prototype `bool valid ( ··· )`, where the parameters are whatever you decide works well with your approach.

Solution:

The idea is to create a recursive function that returns true if a solution is found through this path, and false if it doesn't. At the previous level, if the function returns false, we try the next option; if at that level we exhaust all options and none lead to a solution, then we return false, so that we "backtrack" (at the previous level, the next option will be tried, and so on).

We'll use an array of 8 integers, where each position represents each column of the board (left to right), and the value is the row at which a queen is placed at that column. The parameter column indicates at which column we're attempting the placement (the main program will call the function passing 0 as that parameter).

For the function to validate, we pass the board, and the current column (we want to validate up to that column):

```
bool solve_8queens (int column, vector<int> & board)
{
    if (column > 8)
    {
        return true;
            // we only call the next level if everything up to the
            // current level was fine --- if we call for column 9,
            // it means that we have our solution
    }
    for (int row = 0; row < 8; row++)
    {
        board[column] = row;
```

```
        if (valid(board, column))
        {
            if (solve_8queens (column+1, board))
            {
                return true;
            }
        }
    }
        // we completed the loop and nothing worked;  backtrack
        // and have the previous level keep trying
    return false;
}
```

For your entertainment, here's a complete program, including an implementation of the validate function — it does run and output 0 4 7 5 2 6 1 3, which you can verify that is a valid solution to the puzzle: `https://ece.uwaterloo.ca/~cmoreno/ece250/8queens.c++`

For fun (*after* your finals are over, and if your notion of fun is anything like mine :-)), you could try to modify the above so that it finds other solutions (maybe find *all* solutions? or maybe find a different, random solution each time that it's run? By all means, do count on me to give you a hand if you want to go for it)

NP-completeness

15 – Show that the computational version of subset-sum reduces in polynomial time to the decision subset-sum problem. That is, show that obtaining a subset that adds to the given parameter can be implemented in terms of an algorithm that simply determines whether such a subset exists, with a polynomial time reduction.

Solution:

Given an algorithm $\mathcal{D}$ that tells us whether a subset exists that produces a given sum s, we obtain the subset (or *a* subset, since several could exist) as follows:

For each element, we remove it, and invoke algorithm $\mathcal{D}$ with the remaining elements; if $\mathcal{D}$ responds yes, then this element is not part of the subset, so we continue; and of course, if it responds no, then we know that this value is needed to produce the required sum, so we include the element again and continue to the next element. After checking all the elements (in linear time), we'll have the subset of elements that produce the required sum (.

Notice that a critical detail is that when algorithm $\mathcal{D}$ responds yes with an element removed, we need to leave that element removed for the rest of the processing! Otherwise, the procedure would fail because there could be several subsets, so only the elements that are part of *all* subsets would be flagged (since those are the only ones that, when removed, no solution can be found since all the solutions include that one). But the subsets could be disjoint, so no element would be flagged!

When algorithm $\mathcal{D}$ responds yes, that means that at most, the removed element would be part of

some subset(s), but we know that at least one other subset not containing this one is found in the remaining elements, so we restrict our search to those.