CORE EXAM: ALGORITHMS PART

Department of Computer Science
New York University

Jan 19, 2001

- This is a *one and one-half hour* examination. It is the *Algorithms* part of the Core Exam for the M.S. program in Computer Science.
- Answer each question in the proper exam booklet; the booklets are pre-labeled with the question number.
- Write your name on the front of the exam envelope, in the blank space next to your exam number. Do not write your name *anywhere else*. Also print your SID underneath your name.
- Keep the slip of paper with your exam number. You will need to know this when your exam results are emailed.
- Please answer all three questions. Assume standard results, except where asked to prove them. Keep your answers brief but precise. Rewriting your solutions is recommended (but hand in any scratch work).

QUESTION 1

Suppose you can break a problem of size n into $b > 1$ subproblems, each of size n/b. Solving the subproblems recursively, you can finally solve the original problem by combining the solutions of the subproblems. The time to do this "combination" is n. **Simplifying assumption**: Note that problem size (e.g., n) and number of problems (e.g., b) are usually positive integers. But for the sake of simplifying our analysis, we pretend that these can be arbitrary real numbers. Otherwise, we would have to replace "n/b" by "$\lceil n/b \rceil$".

PART (i) If the worst case running time for solving a problem of size n is denoted $T(n)$, write a recurrence relation for $T(n)$.

PART (ii) Solve your recurrence in Part (i). Remember that your solution will depend on the value of the constant $b > 1$. For the base case, assume that $T(n) = 0$ when n is smaller than some constant $C \geq 0$. You are free to choose C. Try to give the exact solution; otherwise, you will get most of the credit if you give an O-bound and a Ω-bound on $T(n)$.

PART (iii) Only go on to Part (iii) if you have time and have solved Part (ii). Now suppose b is no longer a constant, but depends on n. So we write $b(n)$ for this value. Assume $b(n) > 1$ for $n \geq C$. Prove by induction that $T(n) \leq n \log\log n$ when $b(n) = \sqrt{n}$ and using a suitable base case. All logarithms are base 2.

SOLUTION TO QUESTION 1:

PART (i) The recurrence is $T(n) = bT(n/b) + n$.

PART (ii) This can be shown in several ways: the method of expansion or by induction. The solution is basically $T(n) = \Theta(n \log_b(n))$. One exact solution is

$$T(n) = n\lceil \log_b n \rceil$$

where we assume that $T(n) = 0$ for $n \leq 1$. In proof,

$$\begin{aligned} T(n) &= bT(n/b) + n && (1)\\ &= b(bT(n/b^2) + (n/b)) + n && (2)\\ &= b^2T(n/b^2) + 2n && (3)\\ &= \cdots && (4)\\ &= b^iT(n/b^i) + in && (5)\\ & && (6) \end{aligned}$$

for $i = 0, 1, 2, \ldots$. If we choose $i = \lceil \log_b n \rceil$, then $n/b^i \leq 1$ and the result follows.

PART (iii) Let $b(x) = \sqrt{x}$. We may choose the base case of $T(x) = 0$ for $x \leq 2$. For $x > 2$, we use the recurrence: $T(x) = \sqrt{x}T(\sqrt{x}) + x$. We expand the recurrence to get $T(x) \leq \sqrt{x}\,(\sqrt{x}\log\log(\sqrt{x})) + x$. This simplifies to $T(x) \leq x \log\log x$. NOTE: The above is only an upper bound. It is easily shown to be essentially tight. In fact, if you choose the base case to be $T(x) = x\log\log x$ for $2 \leq x \leq 4$, and apply the recurrence for $x > 4$, you will get the exact answer, $T(x) = x\log\log x$.

QUESTION 2

An important problem in web search engines is to rank a set of web pages by their "importance". One measure of importance of a page is the number of other pages that directly link to it. Suppose we want an algorithm called `RANK(url` x`, int` d`)` which, given a web page x as specified by its `url` (uniform resource locator or webpage address) and an integer $d \geq 1$, to return a list of the form

$$(y_1, n_1), (y_2, n_2), \ldots, (y_L, n_L)$$

where the y_i's ($i = 1, 2, \ldots, L$) are all the pages (in any order) that are reachable from x by following at most d links, and n_i is the number of links to y_i from other pages in this list. Note that x should be on this list, since it is reachable from x using zero links. You are to use the following primitive

`urlList getLinks(URL` x`)`

which returns a list of all the URL's that x directly points to. Note that n_i is not the number of elements in `getLinks`(y_i); that would be the number of links *from* y_i not the number of links *into* y_i.

PART (i) Describe an algorithm for `RANK` in high level pseudo-code. For instance, you may write

```
for ( y in list ) { . . . }
```

to iterate through a list. You may want to use a hash table to implement your algorithm.

PART (ii) Describe the running time for your algorithm as a function of L plus the number of links coming from $y_1, y_2, \ldots, y_L$.

NOTE: As you search through the web pages, you are not allowed to modify them in any way (e.g., no cookies). So you need some other means of keeping track of the pages you have visited.

SOLUTION TO QUESTION 2:

You can use depth first search (DFS) with two modifications. First, the search needs to be bounded by depth from `url` x, (you can't search the whole web). Secondly, the nodes to be visited are not known in advance, so you cannot begin the DFS with the usual "initialize all nodes to be UNVISITED".

To handle the first modification, we augment the call to DFS with a depth parameter indicating the "remaining depth". The search is stopped when the remaining depth reaches 0. Thus, we initially call `DFS`(d), and in general make a recursive call with `DFS`$(d-1)$.

To handle the second modification, we can use a hash table to keep track of the visited `url`'s. For each `url` returned by `getLinks`, we check if is already in the hash table. If not, we insert it with the integer value 1 as its associated data, to indicate that one link to it was found, and recursively call DFS on this node.. If the node is already in the hash table, the associated count is incremented. If closed hashing is used, and the hash table space is exceeded, a new hash table with double the size can be created, with the same average complexity. (A binary tree or other dictionary data type can also be used, with different time complexity).

At the end of the bounded-depth DFS search, we list all the `url`'s in the hash table, with their associated integer counts.

The time to do this is L plus the number of links from the `url`'s. Each edge is explored once, and leads to one update in the hash table. The `getLinks` routine is called once per `url`. Each of these operations has expected constant time. A common mistake was recursively calling DFS for each `url` node even if it were previously visited. REMARK: This can also be done using a Breadth First Search instead of DFS, with similar modifications.

The pseudo-code might look like this:

```
Algorithm:

proc main{
  initialize hash table;
  insert x in hash table with count_x = 0;
  dfs(x,d);
}

proc dfs(url page, int depth){
     list=getLinks(page);
     for (y in list){
```

```
        if (y in hash table)
          increment count_y;
        else {
          insert y in hash table;
          count_y = 1;
          if (depth > 0) dfs(y, depth-1);
        }
    }
}
```

QUESTION 3

Dijkstra's algorithm solves the single-source shortest path problem for a weighted directed graph $G(V, E)$ with a designated source $s \in V$ and with weight function W. If $i, j \in V$, then $W(i, j)$ denotes the weight of the edge from i to j; if the edge does not exist, then $W(i, j) = \infty$. It is assumed that weights are non-negative, $W(i, j) \geq 0$. The algorithm maintains two data structures:

- An array d of size n. The entries of d are indexed by vertices: for each $i \in V$, $d[i]$ is a number (possibly ∞).
- A priority queue Q which stores pairs of the form (x, p) where x is any element and p is a number representing the priority of x. The operations supported by Q are `Insert`(x, p, Q), `Delete`(x, Q) and `ExtractMin`(Q). The latter operation removes from Q any element with the minimum priority. In our algorithm, x will be a vertex and p will be the value $d[x]$.

The pseudo-code is as follows:

DIJKSTRA(G, W, s):
    OUTPUT: the array d in which $d[i]$ is the shortest distance from s to i.
    INITIALIZATION:
    1. let $d[i] = \infty$ for each $i \in V$
    2. $d[s] = 0$
    3. Create an empty priority Q
    4. For each $i \in V$,
        `Insert`$(i, d[i], Q)$
    MAIN LOOP:
    5. While Q is non-empty,
    6. $i \leftarrow$ `ExtractMin`(Q)
    7. For each j that is adjacent to i,
    8. `RELAX`(i, j, W)
    9. Return(d)

The `RELAX` procedure is standard: if $d[j] \leq d[i] + W(i,j)$ then there is nothing to do. Otherwise, we update $d[j]$ to $d[i] + W(i,j)$. Furthermore, we must delete j from Q (since its priority is now wrong), and re-insert j with the new priority $d[j]$.

PART (i) Assume the priority queue is stored as a linear array. Briefly say how you will implement the priority queue operations. Give the complexity of Dijkstra's algorithm using this implementation, expressed as a function of $n = |V|$ and $m = |E|$.

PART (ii) Assume the priority queue is stored as a binary heap. Give the complexity of Dijkstra's algorithm using this implementation, expressed as a function of $n = |V|$ and $m = |E|$. Why is this solution better than that in Part (i) for sparse graphs?

SOLUTION TO QUESTION 3: **PART (i)** Assume that the array Q stores all the element-priority pairs (x, p) in an arbitrary order. [An alternative is to store them in sorted order, but this has different complexity behavior.] In this case, inserting takes constant time, deletion takes constant time (we just mark an item as deleted). But `ExtractMin` would take linear time (you have to search the whole array, skipping past deleted entries, etc). The overall algorithm now takes time $O(n^2)$ since you need to perform n ExtractMin's, at most $m = O(n^2)$ deletes and inserts (within the `RELAX` procedure).

PART (ii) If we use a binary heap for Q, then insert, delete and Extract-Min each takes $O(\log n)$ time. The overall time is seen to be $O((n+m)\log n)$. Note that Relax will make calls (visit) only once for each edge. Clearly, if $m/(n^2/\log n) \to 0$ as $n \to \infty$, this solution is asymptotically faster, which is the case for sparse graphs.