CS325 Midterm, Summer 2009, OSU

Forrest Briggs

July 29, 2009

1 Rules

You are required to **type your solution**, and turn it in printed at the beginning of class, Thursday, 7/23.

You are forbidden to discuss this test in any way with any person other than the course instructor. This includes posting questions online. You are allowed to use any reference material or tools (i.e. text books, wolframalpha.com, calculators, compilers, etc.) to help you solve the problems.

2 Questions

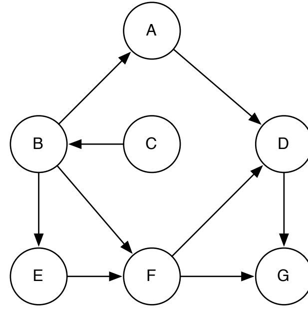
1. **(5 points)** Prove the following facts:
   (a) $3n^4 + 5n^2 + 2 \in O(n^4)$
   Solution: $3n^4 + 5n^2 + 2 \leq 3n^4 + 5n^4 + 2n^4 = 10n^4$, for all $n > 0$, thus, if we pick $c = 10$ and $n_0 = 0$, then we have shown $\exists c, n_0 > 0 : 0 \leq 3n^4 + 5n^2 + 2 \leq cn^4, \forall n > n_0$.
   (b) $\sqrt{n} \in O(n^2)$
   Solution: $lim_{n\to\infty} \frac{n^{1/2}}{n^2} = lim_{n\to\infty} \frac{1}{n^{3/2}} = 0$
   (c) $2^n \in O(4^n)$
   Solution: $lim_{n\to\infty} \frac{2^n}{4^n} = lim_{n\to\infty} \frac{2^n}{2^n 2^n} = lim_{n\to\infty} \frac{1}{2^n} = 0$
   (d) $\log_2 n \in O(\log_5 n)$
   Solution: By change of base, $\log_5 n = \frac{\log_2 n}{\log_2 5}$ Pick $c = \log_2 5$ and $n_0 = 10$, then $\log_2 n = c \log_5 n$, which implies $\log_2 n \leq c \log_5 n$. The value n_0 is chosen so that $\log_2 n$ and $\log_5 n$ are certainly positive.
   (e) For a complete graph $G = (V, E)$, $O(\log E) = O(\log V)$
   Solution: In a complete graph, $E = O(V^2)$, so $O(\log E) = O(\log V^2) = O(2 \log V) = O(\log V)$.

2. **(3 points)** For each of the following algorithms, write a recurrence describing its runtime, and solve it (in big-O notation).
   (a) Divide an input of size n into 5 subproblems of size $n/2$, recursively solve the subproblems, then do $O(n)$ work combining the solutions.
   Solution: The recurrence is $T(n) = 5T(n/2) + O(n)$. Using the master theorem, $a = 5, b = 2$ and $k = 1$, so $b^k = 2$ and $a > b^k$, and the solution is $O(n^{\log_b a}) = O(n^{\log_2 5}) \approx O(n^{2.321})$.
   (b) Solve a problem of size n by recursively solving 2 subproblems, each with size $n - 1$, then do $O(1)$ work to combine.
   Solution: The recurrence is $T(n) = 2T(n/2) + O(1)$. The master theorem does not apply here, but by plug and chug, or the recursion tree method, the solution is $O(2^n)$.

(c) Divide an input of size n into 9 subproblems of size $n/3$, recursively solve the subproblems, then do $O(n^2)$ work combining the solutions.

Solution: The recurrence is $T(n) = 9T(n/3) + O(n^2)$. Using the master theorem, $9 = 5, b = 3$ and $k = 2$, so $b^k = 9$ and $a = b^k$, and the solution is $O(n^k \log n) = O(n^2 \log n)$.

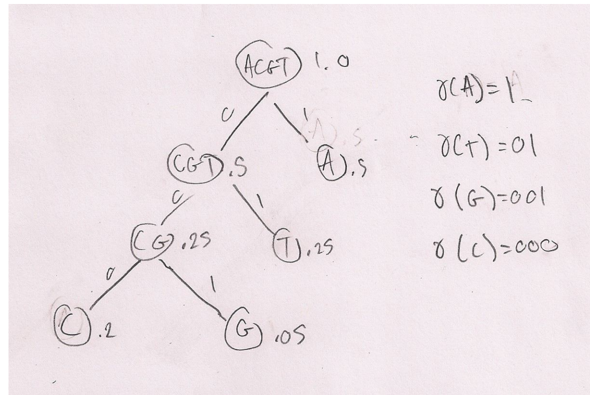
3. **(1 point)** Run topological sort on the following graph. Your answer should consist of a list of the vertices in a particular order.



Solution: One possible order is: C, B, A, E, F, D, G.

4. **(2 points)** You are working for genetics company, and they want you to compress a data file containing a DNA sequence consisting of 10 million letters (the letters are A, C, T and G). Suppose the letter A appears 5 million times, C appears 2 million times, T appears 2.5 million times, and G appears 500,000 times. Give a prefix code for the letters A, C, T and G that uses the minimum average bits per letter. Show how you got the code.

Solution: The letter frequencies are $f_A = .5, f_C = .2, f_T = .25, f_G = .05$. The prefix code and corresponding binary tree are listed below:



5. **(2 points)** Suppose you have an inefficient implementation of a priority queue where the operations take the following amounts of time:

$$insert - O(n) \qquad extractMin - O(n^2) \qquad decreaseKey - O(n \log n)$$

(a) What is the runtime of Dijkstra's algorithm using this data structure?

Solution: In the context of Dijkstra's algorithm n (i.e. the number of items in a priority queue) is at most V, so the operations here cost

$$insert - O(V) \qquad extractMin - O(V^2) \qquad decreaseKey - O(V \log V)$$

The runtime of Dijkstra's algorithm is

$$O(V \times insert + V \times extractMin + E \times decreaseKeyV)$$

Which comes out to

$$O(V \times V + V \times V^2 + E \times V \log V) = O(V^3 + EV \log V)$$

In the case of a complete graph, $E = O(V^2)$, so the runtime is $O(V^3 + V^2 V \log V) = O(V^3 \log V)$.

(b) What is the runtime of Huffman's algorithm using this data structure?

Solution: The while loop in Huffman's algorithm runs $O(n)$ times (where n is the number of letters in the alphabet). In each iteration, it calls *extractMin* twice and *insert* once, so the runtime is $O(n \times (extractMin + insert)) = O(n(n + n^2)) = O(n^3)$.

6. **(3 points)** Consider the following recursive formula:

$$f(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ min\{f(i-1,j) - f(i,j-1) + 3, 2f(i-1,j-1)\} & \text{otherwise} \end{cases}$$

Here i is in the range 0 to M, and j is in the range 0 to N.

(a) Write pseudocode for a dynamic programming algorithm to compute $f(m,n)$.

Solution:

```
allocate and 2D array F[0...m, 0...n]
for i = 0 to m: F[i,0] = i
for j = 0 to n: F[0,j] = j
for i = 1 to m:
    for j = 1 to n:
        F[i,j] = min { F[i-1, j] - F[i, j-1] + 3, 2F[i-1,j-1] }
return F[m,n]
```

(b) Analyze the runtime of your dynamic programming algorithm.

Solution: The algorithm does a constant amount of work mn times, so the runtime is $O(mn)$.

(c) Write pseudocode for a memoized algorithm to compute $f(M,N)$.

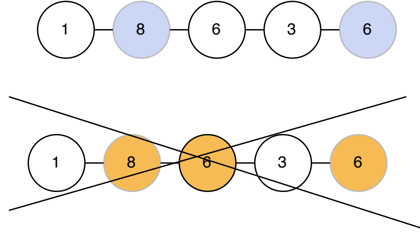
Solution: Use a map data structure with keys that are pairs (i,j), and values that are numbers

```
F(i,j):
    if i = 0 return j
    if j = 0 return i
    if map.exists( (i,j) ): return map.get( (i,j) )
    f = min { F(i-1, j) - F(i, j-1) + 3, 2F(i-1,j-1) }
    map.set( (i,j), f )
    return f
```

7. A path is a special kind of graph $G = (V,E)$, with vertices $v_1, v_2, ..v_n$, and edges between v_i and v_j if and only if i and j differ by exactly 1. In this problem, each node v_i has a corresponding weight w_i (unlike previous problems, where weights were associated with edges). A subset of nodes is independent if no two of them are joined by an edge. The goal of this question is to solve the following problem:

In a graph G, which is a path, find an independent subset of nodes whose total weight is maximized.

For example, consider the following path:



The nodes that are shaded have a total weight of 14, which is the maximum of any independent subset. The next example is not a valid independent subset, because two nodes in the subset share an edge:

(a) **(1 point)** Give an example that shows that the following greedy algorithm does not always find an independent set of maximum weight.

```
1  S ← {}
2  while some node remains in G:
3          Pick a node v_i of maximum weight
4          S ← S ∪ {v_i}
5          Delete v_i and its neighbors from G
6  return S
```

Solution: Use the weights $w_1 = 9, w_2 = 10, w_3 = 9$. The algorithm picks v_2, then deletes v_1 and v_3, which gives it a total weight of 9. However, picking v_1 and v_3 instead would give a total weight of 18.

(b) **(1 point)** Give an example that shows that the following algorithm does not always find an independent set of maximum weight.

```
1  Let S_1 be the set of all v_i where i is an odd number
2  Let S_2 be the set of all v_i where i is an even number
3  Determine which of S_1 or S_2 has the greater total weight and return this one.
```

Solution: Use the weights $w_1 = 1000, w_2 = 1, w_3 = 1, w_4 = 1000, w_5 = 1$. In this case, the value of S_1 is 1002, and the value of S_2 is 1001. However, the set $S = \{v_1, v_4\}$ has total weight 2000.

(c) **(3 points)** Let $W[i]$ be the maximum total weight that can be obtained using the first i nodes in a path. Write a recursive formula for $W[i]$.

Solution: Consider the last node in the path, i. Either we include node i in the subset, or we do not. If we include it, then node $i-1$ cannot be included, but node $i-2$ can be, so the maximum weight we can get for a subset containing node i is $W[i-2]+w_i$. If instead we do not include node i, then the best we can get is $W[i-1]$. Thus, the recursive formula is:

$$W[i] = \begin{cases} W[i-2] + w_i & \text{if we include node } i \\ W[i-1] & \text{if we do not include node } i \end{cases}$$

If all we care about is the value of $W[i]$ and not the actual subset of nodes, this can be simplified to $W[i] = max\{W[i-2] + w_i, W[i-1]\}$. We must also specify 2 base cases. They are $W[0] = 0$ (we can't get any weight with 0 nodes), and $W[1] = w_1$, i.e. the max weight we can get using just the first node is w_1, which is the weight of that node.

(d) **(1 point)** Write pseudocode for a dynamic programming algorithm to calculate $W[n]$, where n is the number of nodes in a path.

Solution:

```
allocate W[0,...,n]
W[0] = 0
W[1] = w_1
for i = 2 to n:
    W[i] = max {W[i-2] + w_i,   W[i-1]}
return W[n]
```

(e) **(1 point)** Analyze the runtime of your dynamic programming algorithm.

Solution: Just a basic for loop doing constant work each time around, so $O(n)$.

(f) **(3 points)** Modify your algorithm so it returns the actual independent subset of nodes with the maximum total weight, rather than just the maximum total weight.

Solution: Add another array $S[i]$ that stores the set of nodes to include to get the max total weight with the first i nodes.

```
allocate an array of numbers W[0,...,n]
allocate an array of lists S[0,...,n]
W[0] = 0
W[1] = w_1
S[0] = []
S[1] = [1]
for i = 2 to n:
    take_node_i = W[i-2] + w_i
    dont_take_i = W[i-1]
    if take_node_i > dont_take_i:
        W[i] = take_node_i
        S[i] = S[i-2].append(i)
    else:
        W[i] = dont_take_i
        S[i] = S[i-1]
return S[n]
```

8. **(3 points)** Let G be an undirected graph with n vertices, where n is an even number. Prove the following theorem by contradiction:

   If every vertex of G has degree at least $n/2$, then G is connected.

   Recall that the degree of a vertex is the number of edges coming out of it, and a graph is connected if there is a path between all pairs of nodes. Hint: Think about connected components.

   Solution: By way of contradiction, assume that there is some graph G with n vertices, and every vertex has degree $n/2$, but this graph is not connected. Consider any vertex v in G. There is an edge from v to at least $n/2$ other vertices. Thus, the connected component that contains v consists of least $1 + n/2$ vertices. Let this component be C. Now consider some other vertex q which is not in the component C (by our assumption that G is not connected, such a q must exist). There are at most $n - (1 + n/2) = n/2 - 1$ vertices that are not in C, and there are at least $n/2$ edges from q, so there must be at least one edge from q to a vertex in C, which implies that r is in C, which is a contradiction.

9. Consider the following algorithm for sorting a list of numbers (MERGE is the same function from mergesort):

```
1  TRISORT(A[1 ... n])
2        if n ≤ 1 return A
3        L ← TRISORT(A[1....(1/3)n])
4        M ← TRISORT(A[(1/3)n + 1....(2/3)n])
5        R ← TRISORT(A[(2/3)n + 1....n])
6        LM ← MERGE(L, M)
7        LMR ← MERGE(LM, R)
8        return LMR
```

(a) **(2 points)** What is the runtime of this algorithm? Explain your answer.

Solution: The algorithm starts by dividing a problem of size n into 3 subproblems of size $n/3$ (basically), then solves them recursively, so the runtime is $T(n) = 3T(n/3)+$ the amount of work to merge. The number of iterations in all while loops in the first call to merge is (to within a constant), $\frac{1}{3}n + \frac{1}{3}n = \frac{2}{3}n$, then the second call to merge does $\frac{2}{3}n + \frac{1}{3}n = n$ iterations, for total of $\frac{5}{3}$n iterations. Thus, the merge process takes $O(n)$ time, and the recurrence is $T(n) = 2T(n/2) + O(n)$, and the solution is $O(n \log n)$.

(b) **(3 points)** Prove that TRISORT is correct by induction (you can assume MERGE is correct).

Solution: The proposition we will prove is $P(n) \equiv$ TRISORT correctly sorts a list of n items. First, the base case $P(1)$. If the algorithm is given a list with 1 (or 0) elements, it just returns that list, which is already sorted. Now the induction. Using strong induction, we assume the hypothesis $P(\frac{n}{3})$, i.e. that TRISORT correctly sorts a list of $\frac{n}{3}$ items. It follows from this assumption that L, M, and R are sorted correctly. Now, by the correctness of MERGE, since L and M are sorted, LM is sorted. Again using the correctness of MERGE, since LM and R are sorted, LMR is sorted, so now we have shown $P(\frac{n}{3}) \rightarrow P(n)$. □

3 A note on writing psuedocode in LaTeX

I recently found a nicer way to format psuedocode in LaTeX than the CLRS style. The alternative is to use the alltt package (the .sty file is included with this template), which allows you to mix verbatim (i.e. monospaced) type, with math type. Formatting pseudocode with this package looks like this (random meaningless code):

```
Here is some math  A_{B}  ∈  ∑_1^n  x
allocate an array A[1...m,  1...n]
for i  =  0 to n:
    for j  =  i to m:
        X_{i,j}  =  i  +  j
        if X_{i,j}  =  7: return 3
```