CS2223 Algorithms D Term 2009
Exam 2 Solutions
April 17, 2009

By Prof. Carolina Ruiz
Dept. of Computer Science
WPI

PROBLEM 1: Asymptoptic Growth Rates (30 points) Take the following list of functions and arrange them in ascending order of asymptotic growth rate. If function $g(n)$ immediately follows function $f(n)$ in your list, then **you must prove that** $f(n) = O(g(n))$. **Prove your answers decisively**, that is, provide formal, mathematical proofs using either the definition of big–O or the theorems stated in class or in the textbook.
$f_1(n) = n^4$; $f_2(n) = 8^n$; $f_3(n) = log(n)$; $f_4(n) = n^n$; $f_5(n) = \sqrt{n}$.

Solution:
Ascending order list:
$f_3(n) = log(n)$; $f_5(n) = \sqrt{n}$; $f_1(n) = n^4$; $f_2(n) = 8^n$; $f_4(n) = n^n$;

1. **(7 points)** Prove that your 1st function is O(your 2nd function):
   Solution:

$$f_3(n) = \log n,\ f_5(n) = \sqrt{n}$$
$$\lim_{n\to+\infty} \frac{f_5(n)}{f_3(n)} = \lim_{n\to+\infty} \frac{\sqrt{n}}{\log n} = \lim_{n\to+\infty} \frac{(1/2) * n^{-1/2}}{1/n} \text{ (by applying de l'Hôpital's rule)}$$
$$= \lim_{n\to+\infty} \frac{n * n^{-1/2}}{2} = \lim_{n\to+\infty} \frac{\sqrt{n}}{2} = +\infty$$
Hence, $f_5(n) = \Omega(f_3(n))$ and $f_3(n) = O(f_5(n))$

2. **(7 points)** Prove that your 2nd function is O(your 3rd function):
   Solution:

$$f_5(n) = \sqrt{n},\ f_1(n) = n^4$$
$$\lim_{n\to+\infty} \frac{f_1(n)}{f_5(n)} = \lim_{n\to+\infty} \frac{n^4}{\sqrt{n}} = \lim_{n\to+\infty} \frac{n^4}{n^{1/2}} = \lim_{n\to+\infty} n^{3.5} = +\infty$$
Hence, $f_1(n) = \Omega(f_5(n))$ and $f_5(n) = O(f_1(n))$

3. **(8 points)** Prove that your 3rd function is O(your 4th function):
   Solution:

$$f_1(n) = n^4,\ f_2(n) = 8^n = (e^{\ln 8})^n = e^{n \ln 8}$$
$$\lim_{n\to+\infty} \frac{f_1(n)}{f_2(n)} = \lim_{n\to+\infty} \frac{n^4}{8^n} = \lim_{n\to+\infty} \frac{n^4}{e^{n \ln 8}}$$
$$= \lim_{n\to+\infty} \frac{4 * n^3}{\ln 8 e^{n \ln 8}} \text{ (by applying de l'Hôpital's rule)}$$
$$= \lim_{n\to+\infty} \frac{4 * 3 * 2 * 1}{\ln 8^4 e^{n \ln 8}} \text{ (by applying de l'Hôpital's rule 3 more times)}$$
$$= \lim_{n\to+\infty} \frac{24}{\ln 8^4 8^n} = 0$$
Hence, $f_2(n) = \Omega(f_1(n))$ and $f_1(n) = O(f_2(n))$

4. **(8 points)** Prove that your 4th function is O(your 5th function):

   Solution:

$$f_4(n) = n^n,\ f_2(n) = 8^n$$

$$\lim_{n\to+\infty} \frac{f_4(n)}{f_2(n)} = \lim_{n\to+\infty} \frac{n^n}{8^n} = \lim_{n\to+\infty} (n/8)^n = \lim_{n\to+\infty} (n/8)^n = +\infty$$

$$\text{Hence, } f_4(n) = \Omega(f_2(n)) \text{ and } f_2(n) = O(f_4(n))$$

   Alternate Solution: To illustrate how to write a proof from the $O(.)$ definition.

   There exist constants $c = 1$ and $n_0 = 8$ such that for all $n \geq n_0$, $8^n \leq c * n^n$. Here is why: If $n \geq 8$, then for any positive $a > 0$, $8^a \leq n^a$. In particular, since $n \geq 8 > 0$, then $8^n \leq n^n$.

   Hence, by the definition of $O(.)$, $f_2(n) = 8^n = O(n^n)$.

PROBLEM 2: Graph Algorithms (40 points + 5 bonus points)

This problem consists of 2 related parts:

Part I: Inverse of a Directed Graph The inverse of a directed graph G is another directed graph G^I which is identical to G except that the direction of each edge in G has been reversed. In other words, if $G = (V, E)$ is a directed graph, then $G^I = (V, E^I)$ is another directed graph where the edge (u, v) belongs to E^I if and only if the edge (v, u) belongs to E.

1. **Pseudo-code (10 points)**

   Write detailed pseudo–code for an algorithm that receives a graph $G = (V, E)$ as input, and produces $G^I = (V, E^I)$ as output. Assume that both G and G^I are represented using an adjacency list representation. (Do NOT assume that the neighbors of a node are organized in any particular order in the node's adjacency list). Your algorithm should run in linear time, that is it should be $O(n+m)$, where $n = |V|$ and $m = |E| = |E^I|$. Explain your work.

   Solution:

Instructions:	**Time per instruction:**	**Number of iterations:**	**Total per instruction:**
Invert-Graph(G): returns G^I			
Create an empty adjacency list for G^I	$c_1 n$	1	$c_1 n$
For each node u in G do {	c_2	n	$c_2 n$
For each edge (u, v) in the adjacency list of u in G do {	c_3	m	$c_3 m$
add an edge (v, u) in the adjacency list of v in G^I	c_4	m	$c_4 m$
}			
}			
TOTAL TIME:			$(c_1+c_2)n+$ $(c_3+c_4)m$

   Note that creating an empty adjacency list for G^I entails only to initialize an array of n positions (one for each node in the graph) each containing an empty list.

2. **Time Complexity Analysis (10 points)** Analyze the time complexity of your algorithm instruction by instruction in the table above. Produce a function $T(n, m)$ that measures the runtime of your algorithm for an input graph with n nodes and m edges. Prove in the space provided below that $T(n, m)$ is $O(n + m)$.

   Solution:
   $T(n, m) = (c_1+c_2)n+(c_3+c_4)m$ is $O(n+m)$ since for all $n > 0$ and $m > 0$, there exists a constant $k = (c_1+c_2+c_3+c_4)$ such that $T(n, m) = (c_1+c_2)n+(c_3+c_4)m \leq k(n+m)$.

Part II: Connectivity and Strong Connectivity of a Directed Graph

Connectivity of Directed Graphs We say that a directed graph $G = (V, E)$ is **connected** if for every pair of nodes u and v in G, there is a directed path between u and v. That is, there is a sequence of directed edges $(u, e_1), (e_1, e_2), \ldots, (e_k, v)$ in G that starts at u and ends at v, with $k \geq 0$.

Note that the same Breadth First Search (BFS) algorithm for undirected graphs discussed in class and in the textbook would work to determine connectivity in directed graphs. That is, given a directed graph G and a start node s in G, BFS would determine whether or not there is a directed path between s and any other node in G.

Strong Connectivity of Directed Graphs We say that a directed graph $G = (V, E)$ is **strongly connected** if for every pair of nodes u and v in G, there is a directed path between u and v, and there is a directed path between v and u. That is, there is a sequence of directed edges $(u, e_1), (e_1, e_2), \ldots, (e_k, v)$ in G that starts at u and ends at v, and there is another sequence of directed edges $(v, a_1), (a_1, a_2), \ldots, (a_q, u)$ in G that starts at v and ends at u, with $k \geq 0$, and $q \geq 0$.

1. **(8 points)** Let s be any node in a directed graph G. What property should the BFS tree of G starting at s, and the BFS tree of G^I (the inverse graph of G) starting at the same s satisfy to guarantee that G is strongly connected? Explain your answer in detail. Draw sketches of these trees to help you intuition and illustrate your answer. Remember that the edges in these trees are now directed.

   Solution:
   Let $T_{G,s}$ be the BFS tree of G starting at s, and Let $T_{G^I,s}$ be the BFS tree of G starting at s. *The property that both $T_{G,s}$ and $T_{G^I,s}$ must satisfy in order for G to be strongly connected is that each and every node in G must appear in each of these two trees.*

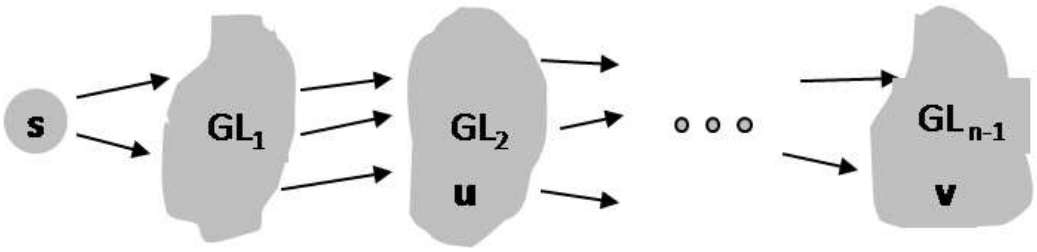
   This property as well as the implied algorithm to check strong connectivity of a graph are clearly explained in Section 3.5 of the textbook and in the corresponding slides (which you studied to be able to solve Problem 1 of HW3).

   Let u and v be two arbitrary nodes in the graph G. Assume that both u and v appear in $T_{G,s}$ and in $T_{G^I,s}$ (see graphical depiction on next page):

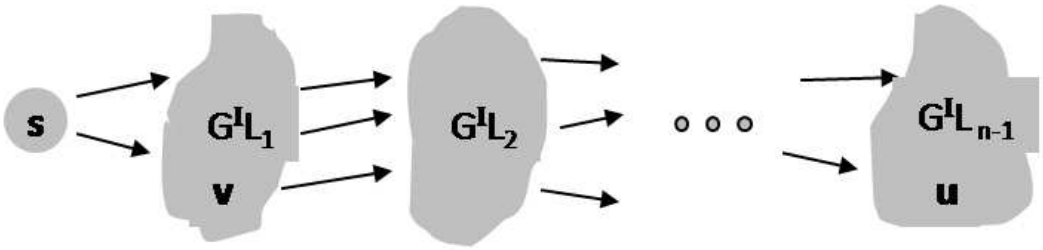
   (a) Since u appears in $T_{G,s}$, there is a directed path $s \rightsquigarrow u$ between s and u in G.
   (b) Since v appears in $T_{G^I,s}$, there is a directed path $s \rightsquigarrow^I v$ between s and v in G^I.
   (c) (b) above implies that there is a directed path $v \rightsquigarrow s$ between v and s in G.
   (d) (a) and (c) above imply that there is a directed path $v \rightsquigarrow u$ between v and u in G, namely the concatenation of $v \rightsquigarrow s$ and $s \rightsquigarrow u$.
   (e) Since v appears in $T_{G,s}$, there is a directed path $s \rightsquigarrow v$ between s and v in G.
   (f) Since u appears in $T_{G^I,s}$, there is a directed path $s \rightsquigarrow^I u$ between s and u in G^I.
   (g) (f) above implies that there is a directed path $u \rightsquigarrow s$ between u and s in G.
   (h) (e) and (g) above imply that there is a directed path $u \rightsquigarrow v$ between u and v in G, namely the concatenation of $u \rightsquigarrow s$ and $s \rightsquigarrow v$.
   (i) Since u and v are two arbitrary nodes in the graph, (d) and (h) above imply that the graph G is strongly connected.

   The picture below illustrate these arguments.

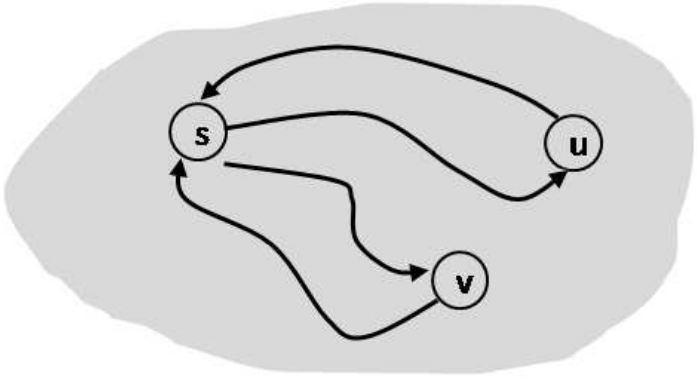
Images taken from textbook slides.



$T_{G,s}$: BFS tree of G starting at node s



$T_{G^I,s}$: BFS tree of G^I starting at node s



Directed paths between u and v, and between v and u, via s

2. **Pseudo-code (10 points)**

Based on your answers above, write pseudo–code for an algorithm that receives a directed graph $G = (V, E)$ as input, and returns *true* if G is strongly connected, and *false* if it is not. Assume that G is represented using an adjacency list representation. Your pseudo-code can invoke a rutine BFS(G, s, T) that receives a directed graph G, a node s in G, and returns the BFS tree of G starting at s. You don't have to write the pseudo-code for BFS(G, s, T). You just need to invoke it. You can also invoke the **Invert-Graph(G)** procedure you wrote above.

Your algorithm should run in linear time, that is it should be $O(n+m)$, where $n = |V|$ and $m = |E|$. Explain your work.

Solution:

Instructions:	**Time per instruction:**	**# of iterations:**	**Total per instruction:**
Is-Strongly-Connected?(G): returns a boolean {			
Pick any node in G. Call it s.	$O(1)$	1	$O(1)$
Call BFS(G, s, T) to construct the BFS tree T of G.	$O(n+m)$	1	$O(n+m)$
/* Check if T contains all the nodes in G */ /* See procedure below */ If Tree-contains-all-graph-nodes(G, T) then {	$O(n)$	1	$O(n)$
Call BFS(G^I, s, T^I) to construct the BFS tree T^I of G^I	$O(n+m)$	1	$O(n+m)$
If Tree-contains-all-graph-nodes(G^I, T^I) then {	$O(n)$	1	$O(n)$
return(*true*)	$O(1)$	1	$O(1)$
} Else return(*false*)	$O(1)$	1	$O(1)$
} Else return(*false*)	$O(1)$	1	$O(1)$
}			
TOTAL TIME Is-Strongly-Connected?:			$O(n+m)$
Tree-contains-all-graph-nodes($G = (V, E), T$): returns a boolean {			
Create an array **node-present**$[1...n]$ of integers	$O(n)$	1	$O(n)$
For $u := 1$ to n	$O(1)$	n	$O(n)$
node-present$[u] := 0$	$O(1)$	n	$O(n)$
Traverse the tree T level by level. For each node u in the tree	$O(1)$	$O(n)$	$O(n)$
node-present$[u] := 1$	$O(1)$	$O(n)$	$O(n)$
For $u := 1$ to n {	$O(1)$	$O(n)$	$O(n)$
If node-present$[u] == 0$ then	$O(1)$	$O(n)$	$O(n)$
return(*false*)	$O(1)$	1	$O(1)$
}			
return(*true*)	$O(1)$	1	$O(1)$
}			
TOTAL TIME Tree-contains-all-graph-nodes:			$O(n)$

3. **Time Complexity Analysis (7 points)** Analyze the time complexity of your algorithm instruction by instruction in the space provided above. Prove in the space provided below that your algorithm $O(n+m)$.

   Solution:
   As shown on the table above, **Tree-contains-all-graph-nodes** runs in $O(n)$. **Is-Strongly-Connected?** consists of two major parts: Constructing 2 BFS trees ($O(n+m)$), and checking that each of them constains all the nodes in the graph ($O(n)$), plus a fixed number of constant time instructions. Hence, the full **Is-Strongly-Connected?** runs in time $O(n) + O(n+m) = O(n+m)$.

PROBLEM 3: Greedy Algorithms (30 points + 5 bonus points)

You have a list of songs $S_1, S_2, S_3, \ldots, S_n$ that you would like to download onto your MP3 player. The available disk space in our MP3 player would allow you to download all of these songs, but your budget would not. Assume that you can spend no more than D dollars, that the i^{th} song S_i costs d_i dollars, and that $\sum_{i=1}^{n} d_i > D$.

1. **Greedy Strategy**

   Assume that you want to maximize the number of songs that you download from that list within your D dollar budget. Describe a greedy strategy to select what songs to download. Prove that your greedy strategy does indeed produce an optimal solution.

   - **Describe your greedy strategy (8 points)**
     Solution:
     Sort the songs in increasing order of cost, and select songs one by one from this list stated with the cheapest song until you cannot afford any more songs. That is, select the first k songs from this sorted list, where k is such that $\sum_{i=1}^{k} d_i \leq D$ but $\sum_{i=1}^{k+1} d_i > D$.
   - **Prove the optimality of your greedy strategy (10 points)**
     Solution:
     This greedy strategy will produce an optimal solution. Here is the proof (this proof is an adaptation of the proof I wrote in my solutions to Problem 1 HW4 CS2223 D term 2009):
     The algorithm starts by sorting the songs in increasing cost order. Assume that the resulting sorting is: $S_{g1}, S_{g2}, S_{g3}, \ldots, S_{gn}$. Now, the algorithm will select the first k songs on this resulting list, $S_{g1}, S_{g2}, S_{g3}, \ldots, S_{gk}$, where k is such that $\sum_{i=1}^{k} d_{gi} \leq D$, but $\sum_{i=1}^{k+1} d_{gi} > D$.
     Assume by way of contradiction, that the solution produced by this greedy algorithm ($S_{g1}, S_{g2}, S_{g3}, \ldots, S_{gk}$) is not optimal. Hence there must exist a different solution $S_{o1}, S_{o2}, S_{o3}, \ldots, S_{oq}$ where $q > k$ (that is, this solution contains more songs than the one produced by the greedy algorithm). Assume that the different solution is sorted in increasing cost order: that is, $d_{o1} \leq d_{o2} \leq d_{o3} \leq \ldots \leq d_{oq}$.
     Let's compare these two solutions position by position and let $j+1$ be the first position where the two sequences differ:

S_{g1},	S_{g2},	S_{g3},	$\ldots$,	S_{gj},	$S_{g(j+1)}$,	$S_{g(j+2)}$,	$\ldots$,	S_{gk}		**greedy**
S_{o1},	S_{o2},	S_{o3},	$\ldots$,	S_{oj},	$S_{o(j+1)}$,	$S_{o(j+2)}$,	$\ldots$,	S_{ok},	$\ldots$, S_{oq}	**different**

     The greedy algorithm selected $S_{g(j+1)}$ because it was the cheapest song not yet selected (i.e., cheapest song not in $S_{g1}, \ldots, S_{gj}$). The different solution selected a different song $S_{o(j+1)}$ and hence it must hold that $d_{g(j+1)} \leq d_{o(j+1)}$, and also that $\sum_{i=1}^{j+1} d_{gi} \leq \sum_{i=1}^{j+1} d_{oi}$. So the greedy solution "stays ahead" of the different solution. We can continue this reasoning by induction and show that for each $x, 1 \leq x \leq q, d_{gx} \leq d_{ox}$, and $\sum_{i=1}^{x} d_{gi} \leq \sum_{i=1}^{x} d_{oi}$.
     But now, the greedy algorithm stopped selecting songs after the k-th one. As stated above, this means that $\sum_{i=1}^{k} d_{gi} \leq D$, but $\sum_{i=1}^{k+1} d_{gi} > D$. Nevertheless, the different solution picked more songs. However, $\sum_{i=1}^{k+1} d_{oi} \geq \sum_{i=1}^{k+1} d_{gi} > D$. Hence, the different solution is not even a correct solution as it picked songs that exceeded the budget limit D. This is a contradiction with the assumption that the different solution was indeed a solution better than the greedy one, and hence the greedy solution is optimal in terms of the number of songs selected.

2. **Greedy Algorithm** Write down your greedy algorithm below **(10 points)**. You can invoke a sorting function without having to write the code that implements sorting. Analyze the time complexity of your algorithm in the table below **(7 points)**.

Solution:

Instructions:	**Time per instruction:**	**# of iterations:**	**Total per instruction:**
Sort the songs in increasing order of cost let the resulting sorting be: $S_{g1}, S_{g2}, S_{g3}, \ldots, S_{gn}$ with corresponding costs: $d_{g1} \leq d_{g2} \leq d_{g3} \leq \ldots \leq d_{gq}$	$O(nlogn)$	1	$O(nlogn)$
$A := \emptyset$ /* A is the solution set */	$O(1)$	1	$O(1)$
$k := 1$	$O(1)$	1	$O(1)$
While $d_{gk} \leq D$ do {	$O(1)$	n	$O(n)$
$A := A$ union S_{gk} /* Add S_{gk} to A */	$O(1)$	n	$O(n)$
$D := D - d_{gk}$ /* pay for S_{gk} */	$O(1)$	n	$O(n)$
$k := k + 1$	$O(1)$	n	$O(n)$
}			
Return $(k-1)$ and A			
TOTAL TIME:			$O(nlogn)+$ $O(n) =$ $O(nlogn)$

Since in the worst case k can reach n, then the while loop above can be iterated up to n times.