CMPS 256	**Fall 2004**
Exam 3: Final Exam	
C. BenAbdelkader	*01/02/05*

(This exam is for **Sections 2& 3 only**)

Name: .....

Section: .....

Time: 120 MINUTES

Important Notes ...

- This exam is closed-book and closed notes. You are allowed to use no more than TWO PAGES of notes (A4 paper), front and back.
- You may NOT use any calculators.
- Make sure there are 11 pages and 7 problems for a total of 104 points.
- Use **pseudo-code** notation discussed in class to write any algorithms you are asked to write.
- Write as legibly as you can, otherwise your answer will NOT be marked!
- Unless otherwise stated, assume the **adjacency list** representation for all graphs.

Question	Your Grade	Max. Grade
1		30
2		12
3		10
4		12
5		10
6		15
7		15
Total		104

Question 1: (30 Points) Short-Answer Questions

(a) (4 points)
Write an algorithm `Rearrange`(A) that takes an array of numbers (but not necessarily integers!), A, and rearranges it in-place such that all the negative numbers (< 0) precede all the nonnegative numbers (≥ 0). What is the running time of your algorithm? For full credit, your algorithm should use only **constant additional storage**, regardless of the size of A, and should be as efficient as possible. A naive $\Theta(n^2)$-time solution will get ZERO credit.

(b) (6 points)
Consider the following weighted graph $G = (V, E)$, where the edges are stored in the adjacency list in the following order: $(s, u), (s, v), (u, v), (v, x), (x, y), (y, u)$. Suppose path $p = \langle s, u, v, x \rangle$ is a **shortest path** from vertex s to vertex x. Assuming arbitrary edge weights, circle the correct answer in each of the following questions.

(i) **True or False:** if we run Dijkstra's algorithm on this graph, then $d[x]$ might converge to its true value *before* $d[v]$ converges to its true value.

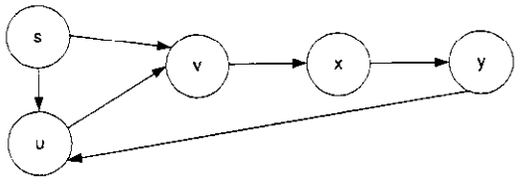
(ii) **True or False:** if we run Bellman-Ford algorithm on this graph, then $d[x]$ might converge to its true value *before* $d[v]$ converges to its true value.

(iii) **True or False:** if we run Dijkstra's algorithm on this graph, then $d[u]$ is always the *first* to converge to its true value (aside from $d[s]$ of course).

(iv) **True or False:** if we run Bellman-Ford algorithm on this graph, then $d[u]$ is always the *first* to converge to its true value (aside from $d[s]$ of course).

(v) **True or False:** if we run Dijkstra's algorithm on this graph, then $d[y]$ is always the *last* to converge to its true value.

(vi) **True or False:** if we run Bellman-Ford algorithm on this graph, then $d[y]$ is always the *last* to converge to its true value.



(c) (6 points)

Given a directed graph $G = (V, E)$ and a source vertex $s \in V$, **prove that** if at any point during the `Bellman-Ford` algorithm, $\pi[s]$ is set to a non NIL value, then G must contain a negative-weighted cycle. For partial credit, **illustrate** your answer with an small graph, clearly indicating vertex s and values of edge weights.

(d) (4 points)

Based on (c), consider the following modification to `Bellman-Ford` algorithm: instead of checking whether $d[v] > d[u] + w(u, v)$ for all edges (u, v) in the graph, we simply check if $\pi[s]$=NIL. Is this correct? why or why not?

(e) (4 points)

Compute the asymptotic time complexity of the following divide-and-conquer algorithm. You may assume that n is a power of 2. (NOTE: It doesn't matter what this does!).

```
Useless(A)
    n ← length[A]
    if n == 1
      then return A[1]
    let A1 and A2 be two arrays of size n/2
    for i = 1 to n/2
        do A1[i] ← A[i]
           A2[i] ← A[n/2 + i]
    for i = 1 to n/2
        do for j = i + 1 to n/2
           do if A1[i] == A2[j]
                 then A2[j] ← 0
    b1 ← Useless(A1)
    b2 ← Useless(A2)
    return max(b1, b2)
```

(f) (6 points)

Given a graph $G = (V, E)$ and a vertex $v \in V$, describe a $\Theta(V + E)$-time algorithm that outputs all vertices that are reachable from v in G. Your algorithm can call DFS, but it cannot call any other algorithm. Briefly discuss the running time of your algorithm.

Question 2: (12 Points)

(a) Suppose you wish to create a hash table of all the computer science students using as the key each student's AUB number (which as you know is a nine-digit number), and suppose you have decided to implement the hash table using an unsigned long integer array of size 4999 with quadratic probing to resolve conflicts. (by the way, both 4999 and 997 are prime numbers.)

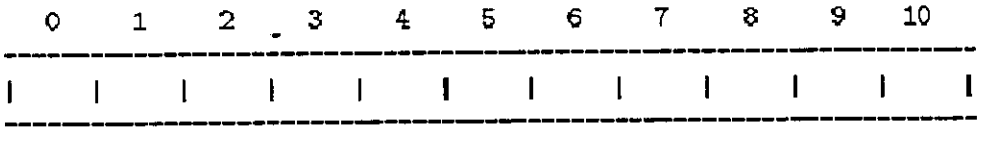
Consider the following three possible hash functions. Explain which one you think might be the best to use. Note that digit_i is the ith digit of the student's AUB number (and hence an integer value between 0 and 9).

1. $(\text{digit}_0 + \text{digit}_1 + \text{digit}_2 + \cdots + \text{digit}_8) \mod 4999$.
2. AUBNumber mod 4999.
3. $(\text{digit}_0 + \text{digit}_1 * 997 + \text{digit}_2 * 997^2 + ...\text{digit}_8 * 997^8) \mod 4999$.

(b) You are inserting the strings shown in the Table below with their corresponding hash values into a hash table of size 11.

String	Hash value
"pea"	12
"leek"	26
"bean"	20
"yam"	37
"corn"	33
"kale"	10
"okra"	31

Assuming the hash table is implemented with open addressing and linear probing, and that the strings are inserted in the order they appear in the Table above (i.e. "pea" first and "okra" last), fill in the slots of the hash table below.



(c) Recall that deletion can cause problems when a hash table is implemented with open addressing. Assuming the hash table of part (b), give an example of a delete operation followed by a search operation that illustrates one of these problems. What is the error that occurs in your example?

(d) Of the different hashing approaches that were discussed in class (simple array with linear probing, simple array with quadratic probing, simple array with double hashing, and chaining), which approach would be most appropriate for an application in which many equal keys (i.e. duplicates) are likely to be present? Explain your answer in a few sentences.

Question 3: (10 Points)

(a) Show that: for any integer k, there exists a directed graph G that has k strongly connected components and such that if we add one particular edge to G, it becomes strongly connected (i.e. the new graph has only one connected component). *Hint:* first try to construct such a graph for a small value of k, such as $k = 3$.

(b) Based on (a), is the following statement **True or False**? Justify your answer:
If a directed graph G has k strongly connected components, then by adding one more edge to G the number of strongly connected components can decrease by at most by 1 (i.e. the new graph obtained from G by adding one edge has at least $(k-1)$ strongly connected components).

Question 4: (12 Points)

A search-engine company has decided on the following method to rank web page quality for any given Web user: the user is asked to give a web page X that s/he considers to be a very high-quality page. Then, for every other web page Y, the quality of Y is determined as the minimum number of hyper-links that must be followed from page X to get to Y.

Suppose that the company's database currently contains p web pages and ℓ hyper-links between them. Answer the following questions:

(a) Formulate this problem as a graph problem. You need to specify the vertices, edges of the graph, and whether the graph is weighted/unweighted and directed/undirected.

(b) Select a graph representation under the assumption that p is roughly 1,000,000 and ℓ is roughly 10,000,000. Justify your answer.

(c) Design an algorithm to solve the problem as efficiently as possible, and analyze its running time. The **input** to your algorithm is a graph G (as specified in your answer (a)) and the preferred page X, and its **output** should be the p web pages in the database in *decreasing* order of their quality.

Question 5: (10 Points)

Consider the following modified version of the `DAG-Shortest-Paths` algorithm we have discussed in class (the changes start on line 5):

```
DAG-Shortest-Paths-Faster(G,w,s)
1  for each v ∈ V[G]
2      do d[v] ← ∞
3  d[s] ← 0
4  V' ← Topological-Sort(V[G])
5  flag ← FALSE
6  for each vertex u ∈ V'
7      do if u == s
8            then flag ← TRUE
9         if flag==TRUE
10           then for each vertex v ∈ Adj[u]
11                   do Relax(u,v,w)
```

First briefly explain in what way(s) this algorithm differs from `DAG-Shortest-Paths`, then answer each of the following two questions:

1. explain why, given a DAG G and a source vertex s, this algorithm still *correctly* computes the shortest paths from s.
2. derive its running time and explain why in practice it might be a little faster than `DAG-Shortest-Paths`.

Question 6: (15 Points)

Consider a **connected undirected** graph $G = (V, E)$. An edge $e \in E$ is called a **bridge** if removing e from E causes the graph to become disconnected.

Note: A graph is said to be connected if every two of its vertices are mutually reachable; a graph is said to be disconnected if it not connected.

(a) Give an example of a connected undirected graph containing at least one bridge (clearly label the bridges). Your graph must have **at least 4 vertices**.

(b) Show that: in any DFS of G, if an edge e is a bridge then e is a **tree edge**.

(c) Show that: an edge e is a bridge *if and only if* e does not lie on any simple cycle of G. (*Note:* A simple cycle is a cyclic path that does not contain any other cycles, i.e. sub-cycles.)

(d) Give a $O(V + E)$-time algorithm that computes all bridges of G. *Hint#1:* solution is based on (c). *Hint#2:* use a modified DFS.

Question 7: (15 Points) (* This problem is a bit challenging)

Solve **only one** of the following two questions:

1. A graph is said to be *semiconnected* if and only if for any pair of vertices u, v, there is a directed path from u to v **and/or** there is a directed path from v to u.

   Notice that if a graph is connected then it is also semiconnected (but the inverse is not always true).

   (a) Draw a **directed** graph with at least 4 vertices which is semiconnected but is **not** connected.

   (b) Design a $O(V^3)$-time algorithm that given a **directed** graph G determines whether the graph is semiconnected. As usual, briefly explain the running time of your algorithm. ***Hint:*** use BFS.

2. A *bipartite graph* is defined as follows: an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$, or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

   (a) Draw a graph with at least 5 vertices and which is bipartite; clearly indicate the two sets V_1 and V_2 on your graph.

   (b) Give a $\Theta(V + E)$-time algorithm to determine if a given graph $G = (V, E)$ is bipartite. ***Hint:*** use a modified version of BFS.