Computer Science 477/677 Spring 2019

University of Nevada, Las Vegas Computer Science 477/677 Spring 2019

Answers for Practice Final Examination Spring 2019

The entire practice test is 655 points.

1. True or False. Write "O" if the answer is not known to science at this time. [5 points each]

   (a) **F** Computers are so fast today that complexity theory is only of theoretical, but not practical, interest.

   (b) **T** The inverse Ackermann function, $\alpha(n)$, grows so slowly that, from a practical (as opposed to theoretical) point of view, it might as well be constant.

   (c) **O** If a problem is $\mathcal{NP}$-complete, there is no polynomial time algorithm which solves it.

   (d) **F** Quicksort takes $\Theta(n \log n)$ time on an array of size n.

   (e) **T** Planar graphs are sparse.

   (f) **T** Acyclic graphs are sparse.

   (g) **F** Acyclic directed graphs are sparse.

2. Fill in the blanks. [5 points each blank.]

   (a) If a planar graph $\mathcal{G}$ has 20 edges, then the number of vertices of $\mathcal{G}$ cannot be less than 9. (You must give the best possible answer, exactly. No partial credit.)

   (b) A directed acyclic graph with 5 vertices cannot have more than 10 arcs, and a directed acyclic graph with 6 vertices cannot have more than 15 arcs. A directed acyclic graph with 10 vertices cannot have more than 45 arcs. (You must give the best possible answer, exactly. No partial credit.)

   (c) A directed acyclic graph with 20 arcs cannot have fewer than 7 vertices.

   (d) The height of a binary tree with 50 nodes is at least 6. (You must give the best possible answer, exactly. No partial credit.)

   (e) In perfect hashing, there are no collisions.

   (f) If separate chaining is used to resolve collisions in a hash table with n items and n places in the array and if the hash function is pseudo-random, then approximately _____% of the places will have more than two items. Pick the best answer from among these choices: (0%, 1%, 2%, 4%, 8%, 16%, 32%) The best of these answers is 8%.

   (g) The time complexity of every comparison-based sorting algorithm is $\Omega(n \log n)$. (Your answer should use Ω notation.)

   (h) Radix, or bucket, sorting is not comparison-based.

   (i) The infix expression $(x+y)*z$ is equivalent to the prefix expression $*+xyz$ and the postfix expression $xy+z*$.

(j) What is the **only** difference between the abstract data types *queue* and *stack*?

Stacks are last in first out (LIFO), while queues are first in first out (FIFO).

(k) The items stored in a priority queue (that includes stacks, queues, and heaps) represent unfulfilled obligations.

(l) Name a divide-and-conquer searching algorithm.

binary search

(m) Name two divide-and-conquer sorting algorithms.

mergesort

quicksort

(n) The following is pseudo-code for which sorting algorithm we've discussed?

```
------------------------------
int x[n];
obtain values of x;
for(int i = n-1; i > 0; i--)
 Find the largest element of x[0], ... x[i] and swap it with x[i]
```

Selection sort.

(o) The following is pseudo-code for which sorting algorithm we've discussed?

```
------------------------------
int x[n];
obtain values of x;
bool finished = false;
for(int i = n-1; i > 0 and not finished; i--)
 {
  finished = true;
  for(int j = 0; j < i; j++)
    if(x[j] > x[j+1])
     {
      swap(x[j],x[j+1]);
      finished = false;
     }
 }
```

Bubblesort.

3. Give the asymptotic complexity, in terms of `n`, of each of the following code fragments. [10 points each]

(a) `for(int i = n; i > 1; i = i/2)`

```
  cout << "hello world" << endl;
```

$\Theta(\log n)$.

(b)
```
for(int i = 1; i < n; i++)
  for(int j = 1; j < i; j = 2*j)
    cout << "hello world" << endl;
```

$\Theta(n \log n)$.

(c)
```
for(int i = 1; i < n; i++)
  for(int j = i; j < n; j = 2*j)
    cout << "hello world" << endl;
```

$\Theta(n)$

(d)
```
for(int i = 2; i < n; i = i*i)
  cout << "hello world" << endl;
```

$\Theta(\log \log n)$

4. [30 points] Name two problems which are known to be $\mathcal{NP}$-complete, and one problem that is known to be undecidable.

---

5. Solve the recurrences. Give asymptotic answers in terms of n, using either O, Ω, or Θ, whichever is most appropriate.

(a) [10 points] $F(n) = 2F\left(\frac{n}{2}\right) + n$

$F(n) = \Theta(n \log n)$

(b) [10 points] $F(n) \geq 4F\left(\frac{n}{2}\right) + n^2$

$F(n) = \Omega(n^2 \log n)$

(c) [10 points] $F(n) = F(n-1) + \frac{n}{4}$

$F(n) = \Theta(n^2)$

(d) [10 points] $F(n) \leq F\left(\frac{n}{2}\right) + F\left(\frac{n}{4}\right) + F\left(\frac{n}{5}\right) + n$

$F(n) = O(n)$

(e) [10 points] $F(n) = F\left(n - \sqrt{n}\right) + n$

$F(n) = \Theta(n^{3/2})$

(f) [10 points] $F(n) = F(\log n) + 1$

$F(n) = \Theta(\log^* n)$

6. [30 points] Use dynamic programming to compute the length of the longest common subsequence of the strings "011011001" and "1010011001."

		1	0	1	0	0	1	1	0	0	1
	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1
1	0	1	1	2	2	2	2	2	2	2	2
1	0	1	1	2	2	2	3	3	3	3	3
0	0	1	2	2	3	3	3	3	4	4	4
1	0	1	2	3	3	3	4	4	4	4	5
1	0	1	2	3	3	3	4	5	5	5	5
0	0	1	2	3	4	4	4	5	6	6	6
0	0	1	2	3	4	5	5	5	6	7	7
1	0	1	2	3	4	5	6	6	6	7	8

The answer is 8. There is a unique maximum length common subsequence, namely 01011001

7. Solve each of the following recurrences, giving the answer in terms of O, Θ, or Ω, whichever is most appropriate [10 points each].

(a) $T(n) < T(n-2) + n^2$

$T(n) = O(n^3)$

(b) $F(n) \geq F(\sqrt{n}) + \lg n$

$F(n) = \Omega(\log n)$

(c) $G(n) \geq G(n-1) + n$

$G(n) = \Omega(n^2)$

(d) $F(n) = 4F(n/2) + n^2$.

$F(n) = \Theta(n^2 \log n)$

(e) $H(n) \leq 2H(\sqrt{n}) + O(\log n)$.

$H(n) = O(\log n \log\log n)$

(f) $K(n) = K(n - \sqrt{n}) + 1$.

$K(n) = \Theta(\sqrt{n})$

(g) $F(n) = 4F\left(\frac{3n}{4}\right) + n^5$ (No, you don't need a calculator.)

$F(n) = \Theta(n^5)$

8. Find the asymptotic complexity, in terms of n, for each of these fragments, expressing the answers using O, Θ, or Ω, whichever is most appropriate.

   (a)
   ```
   for(i = 0; i < n; i = i+1);
   cout << "Hi!" << endl;
   ```

   $\Theta(n)$

   (b)
   ```
   for(i = 1; i < n; i = 2*i);
   cout << "Hi!" << endl;
   ```

   $\Theta(\log n)$

   (c)
   ```
   for(i = 2; i < n; i = i*i);
   cout << "Hi!" << endl;
   ```

   $\Theta(\log \log n)$

   (d) The following code models the first phase of heapsort.

   ```
   for(int i = n; i > 0; i--)
    for(int j = i; 2*j <= n; j = 2*j)
     cout << "swap" << endl;
   ```

   $\Theta(n)$

   (e) The following code models the second phase of heapsort.

   ```
   for(int i = n; i > 0; i--}
    {
     cout << "swap" << endl;
     for(int j = 1; 2*j <= i; j = 2*j)
      cout << "swap" << endl;
    }
   ```

   $\Theta(n \log n)$

   (f) The following code models insertion of n items into an AVL tree.

   ```
   for(int i = 1; i < n; i++)
    for(int j = n; j > 0; j = j/2)
      cout << "check AVL property and possibly rotate" << endl;
   ```

   $\Theta(n \log n)$

9. Solve each of the following recurrences, expressing the answers using O, Θ, or Ω, whichever is most appropriate. [10 points each]

   (a) $F(n) = F(n/2) + 1$

   $F(n) = \Theta(\log n)$

(b) $F(n) = F(n-1) + O(\log n)$

$F(n) = O(n \log n)$

(c) $F(n) = F\left(\frac{n}{2}\right) + 2F\left(\frac{n}{4}\right) + n$

$F(n) = \Theta(n \log n)$

(d) $F(n) = F\left(\frac{3n}{5}\right) + F\left(\frac{4n}{5}\right) + n^2$

$F(n) = n^2 \log n$

Use the same method you used for the previous problem. Hint: $3^2 + 4^2 = 5^2$.

(e) $F(n) = F(n-2) + n$

$F(n) = \Theta(n^2)$

10. (a) Use Huffman's algorithm to construct an optimal prefix code for the alphabet $\{A, B, C, D, E, F\}$ where the frequencies of the symbols are given by the following table.

A	2	1100
B	8	00
C	9	10
D	3	1101
E	7	01
F	5	111

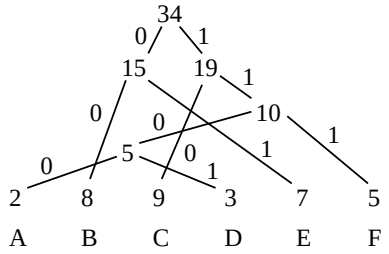


Figure 1: Huffman's algorithm for optimal prefix-free code

(b) Use the Hu-Tucker algorithm to find an optimal alphabetic prefix-free code on those same letters using the same distribution.

A	2	000
B	8	001
C	9	01
D	3	100
E	7	101
F	5	11

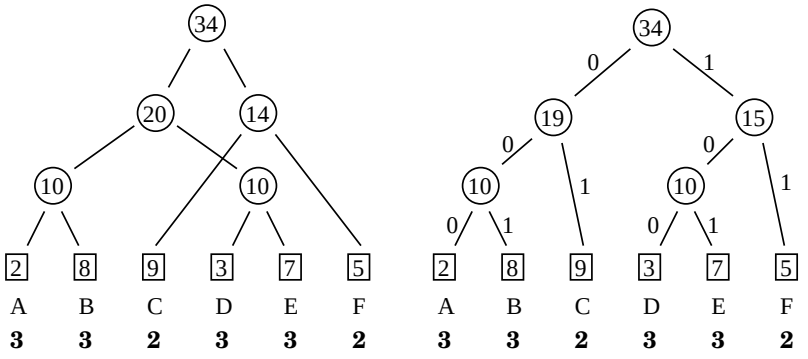


Figure 2: Hu-Tucker algorithm for optimal alphabetic prefix-free code

11. [10 points] Write pseudo-code for binary search.

This is one way you could answer this question. What I am looking for is whether your code uses divide-and-conquer. I am writing the code in C++, but you don't have to.

```
bool found;
int index;

// We assume that an integer n and an ordered array A[n], and an integer sought
// are given.  The value of found will become true if sought is an entry of A,
// false otherwise.  If found is true, A[index] = sought.

void binarysearch(int sought,int lo,int hi)
 // 0 <= lo < hi < n
 {
  if (lo > hi)
   found = false;
  else
   {
    int mid = (lo+hi)/2;
    if (A[mid] == sought)
     {
      found = true;
      index = mid;
     }
    else if (sought < A[mid])
     binarysearch(sought,lo,mid-1);
    else
     binarysearch(sought,mid+1,hi);
   }
 }

void main()
 {
  binarysearch(sought,0,n-1);
 }
```

12. Find the asymptotic complexity, in terms of n, for each of these fragments, expressing the answers using O, Θ, or Ω, whichever is most appropriate. [10 points each]

(a)
```
for(int i = 1; i*i < n; i++)
    cout << "Hi!" << endl;
```
Answer: $\Theta(\sqrt{n})$

(b)
```
for(int i = n; i > 1; i = sqrt(i));
```

```
 cout << "Hi!" << endl;
```

Answer: $\Theta(log \log n)$

Find the asymptotic time complexity, in terms of n, for each of these code fragments, expressing the answers using O, Θ, or Ω, whichever is most appropriate. [10 points each]

(a)
```
int f(int n)
 {
  if (n < 2) return 1;
  else return f(n-1)+f(n-1);
 }
```

Answer: $\Theta(2^n)$

(b)
```
void hello(int n)
  {
   if(n >= 1)
    {
     for(int i = 1; i < n; i++)
      cout << "Hello!" << endl;
     hello(n/2);
     hello(n/2);
    }
  }
```

Answer: $\Theta(n \log n)$

13. [20 points] Define the Collatz function as follows:

```
int collatz(int n)
  {
    assert(n > 0);
    if(n == 1) return 0;
    else if (n%2) return collatz(3*n+1); // n is odd, greater than 1
    else return collatz(n/2); // n is even
   }
```

Write pseudo-code for a memoization algorithm which prints collatz(n) for all n from 1 to 1000.

I will use C++, with some English statements which must be turned into code. I will assume that we use a sparse array to store memos.

```
sparsearray memo; // a structured type, a sparse array with default value -1

int collatz(int n)
 // input condition: n > 0
 {
```

```
  int temp = fetch(memo,n);
  if (temp == -1)
   {
    if (n == 1) temp = 0;
    else if (n%2) temp = 1+collatz(3*n+1) // n is odd, other than 1
    else temp = 1+collatz(n/2); // n is even
    store(memo,i,temp);
   }
  return temp;
 }

void main()
 {
  for(int n = 1; n <= 1000; n++)
   cout << "collatz[" << n << "] = " << collatz(n) << endl;
 }
```

14. [20 points] Give pseudocode for a recursive algorithm which computes the median of the union of two sorted lists in logarithmic time.

    I do not want to reveal this algorithm.

15. [20 points] Describe a randomized algorithm which finds the k^{th} smallest element of an unsorted list of n distinct numbers, for a given $k \leq n$, in $O(n)$ expected time. (By "distinct," I mean that no two numbers in the list are equal.)

    We assume we have a random number generator:

```
int random(int m)
 {
  return a uniformly distributed random integer in the range [0,m-1]
 }
```

    The code runs in expected time $O(\log n)$, even if the adversary can see our code in advance, but cannot predict the outputs of the random number generator.

```
// We assume integers k and n where 0 < k <= n,
// and an unordered array of integers A[n] are given.
// Recall that the indices of A are 0 .. n-1.
// The goal is to find the k-th smallest entry of A;
// what the value A[k-1] would be if A were sorted.

int find(X,int m,int ell)
 {
  // ell <= m
```

```
 // X is an array
 // returns the ell-th smallest entry of X
 if (m == 1) return X[0]; // k must be 1 in this case
 else
  {
   int Left[m];
   int Right[m];
   int leftsize = 0;
   int rightsize = 0;
   int pivot = X[random(m)];
   for (i = 0; i < m; i++)
    if (X[i] < pivot) Left[leftsize++] = X[i];
    else Right[rightsize++] = X[i];
   if (k <= leftsize) return find(Left,leftsize,k);
   else return find(Right,rightsize,k-leftsize);
  }
}

void main()
 {
  cout << find(A,n,k) << endl;
 }
```

Warning: this code does not work if the entries of A are not distinct. Do you see why?

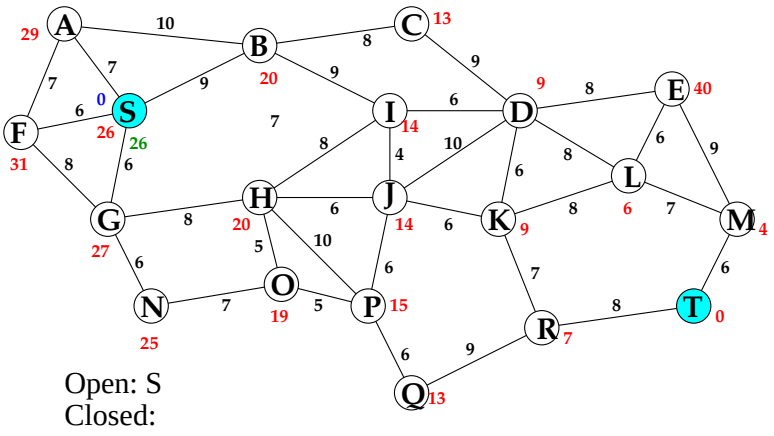
There is another implementation of find which uses the basic idea of quicksort, and works if the entries are not distinct. Unlike the above implementation, this implementation changes the entries of A, but does not require extra arrays.

```
int find(int first,int last,int ell)
 // input condition 0 < ell <= last-first+1
 // returns the ell-th smallest entry among A[first] ... A[last]
 {
  if (first == last) return A[first]; // search narrowed to one entry
  else
   {
    swap(A[first],A[first+random(last-first+1)]; //random entry in search space
    int pivot = A[first];
    int lo = first;
    int hi = last;
    while (first < last)
     {
      if (A[lo+1] <= pivot) lo++;
      if (A[hi] >= pivot) hi--;
      if (A[lo+1] > pivot and A[hi] < pivot) swap(A[lo+1],A[hi]);
```

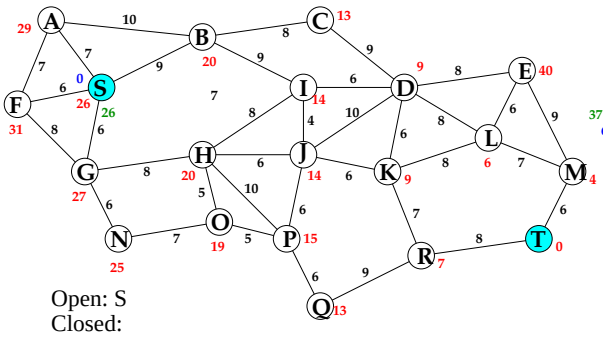
```
    } // now lo = hi
   if (ell <= hi-low+1) return find(first,lo,ell);
   else return find(lo+1,last,ell-lo);
  }
 }

void main()
 {
  return find(k,0,n-1);
 }
```

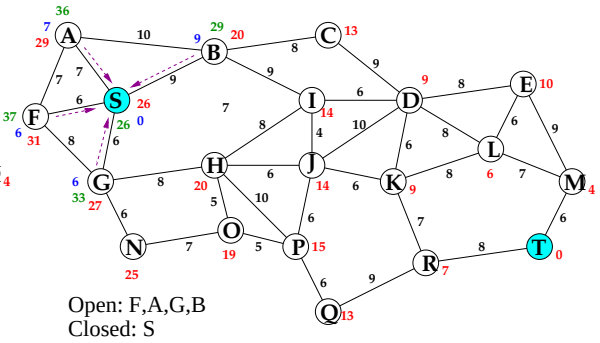
16. [20 points] Walk through the A^* algorithm for the following weighted graph to find the shortest path from S to T. Edge weights are shown in black, and the values of the heuristic are shown in red.



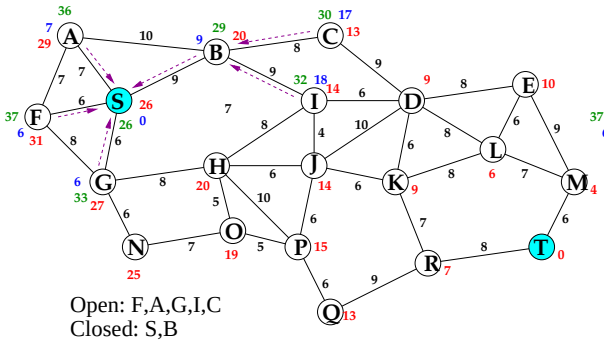
Here are 14 steps of my implementation, I got 39. With so much calculation, I find it hard to believe that I didn't make some error. If I give this problem on the exam, it will be a much smaller example.



(a)

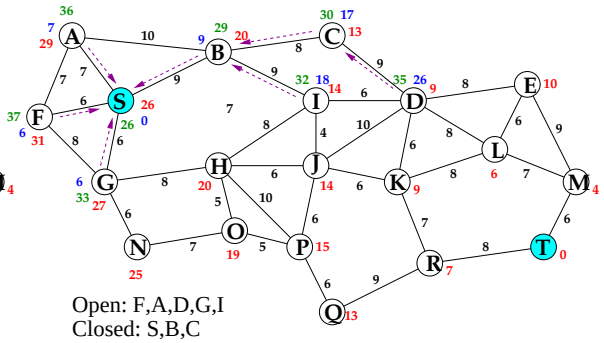


(b)



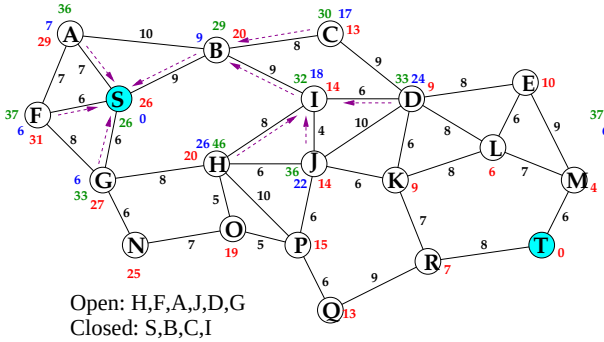
Open: F,A,G,I,C
Closed: S,B

(c)



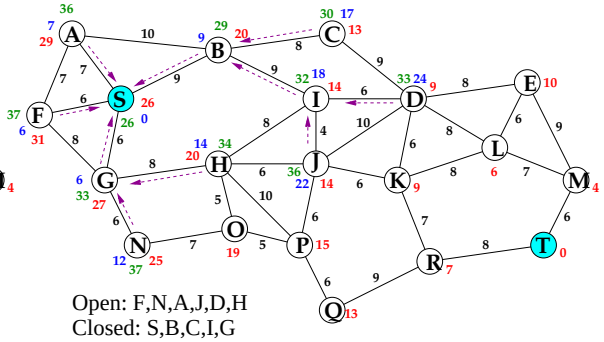
Open: F,A,D,G,I
Closed: S,B,C

(d)



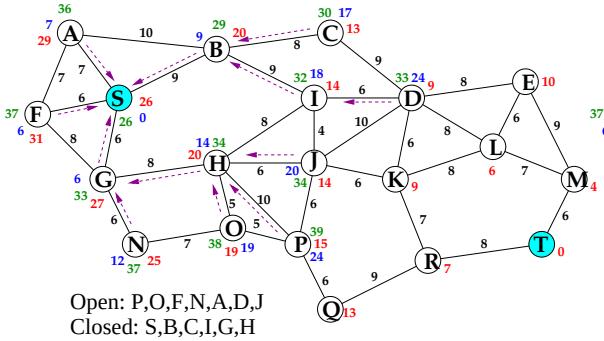
Open: H,F,A,J,D,G
Closed: S,B,C,I

(e)



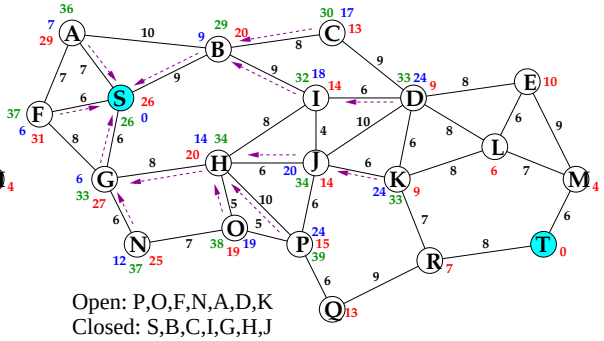
Open: F,N,A,J,D,H
Closed: S,B,C,I,G

(f)



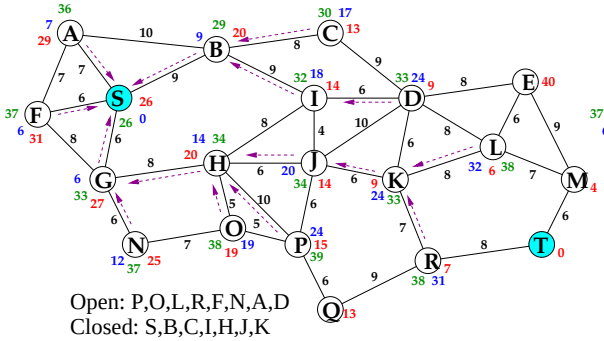
Open: P,O,F,N,A,D,J
Closed: S,B,C,I,G,H

(g)



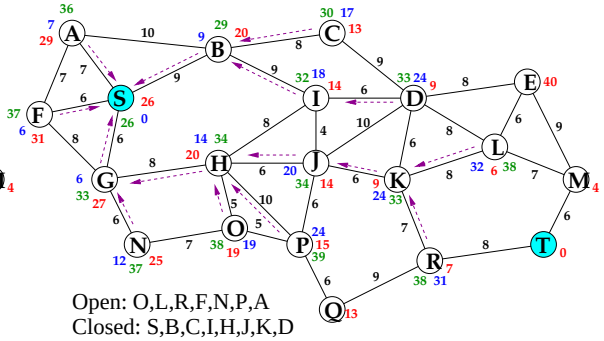
Open: P,O,F,N,A,D,K
Closed: S,B,C,I,G,H,J

(h)



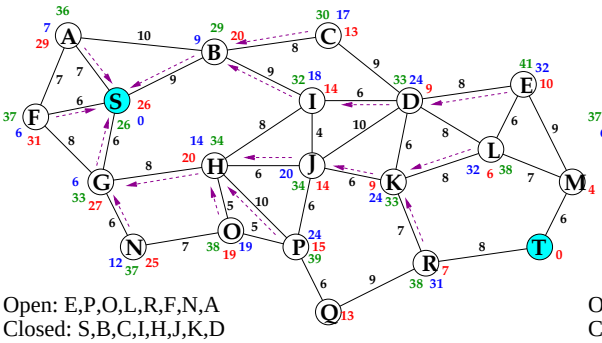
Open: P,O,L,R,F,N,A,D
Closed: S,B,C,I,H,J,K

(i)



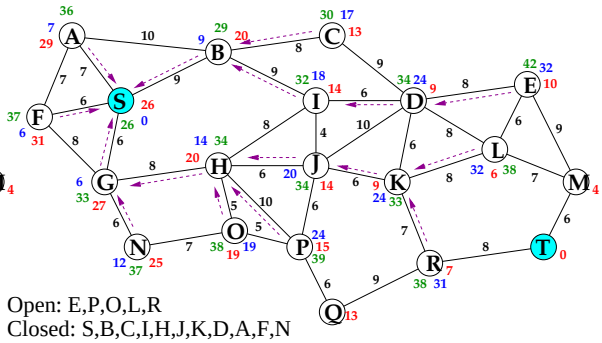
Open: O,L,R,F,N,P,A
Closed: S,B,C,I,H,J,K,D

(j)



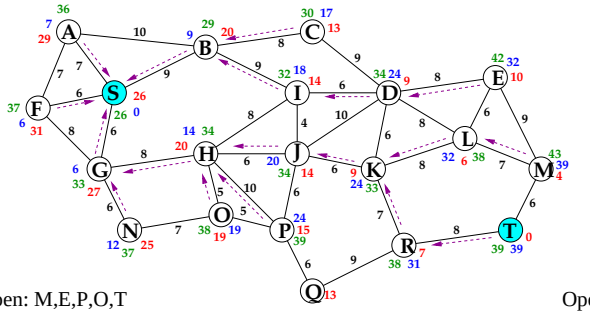
Open: E,P,O,L,R,F,N,A
Closed: S,B,C,I,H,J,K,D

(k)



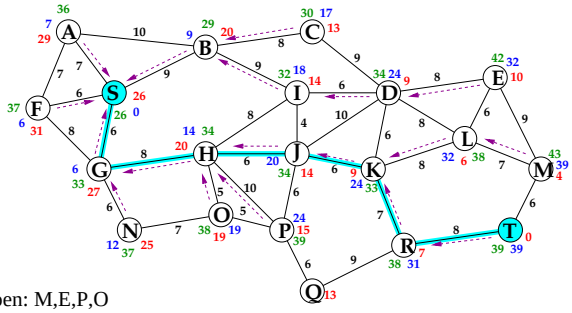
Open: E,P,O,L,R
Closed: S,B,C,I,H,J,K,D,A,F,N

(l)



Open: M,E,P,O,T
Closed: S,B,C,I,H,J,K,D,A,F,N,R,M

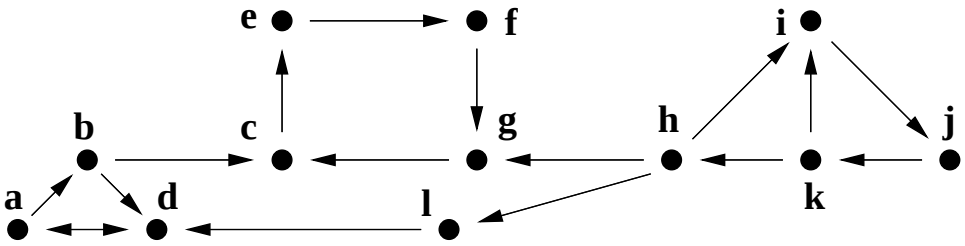
(m)



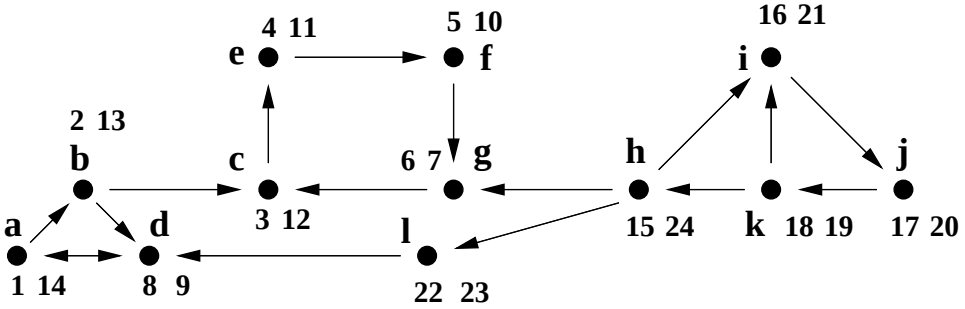
Open: M,E,P,O
Closed: S,B,C,I,H,J,K,D,A,F,N,R,M,T

(n)

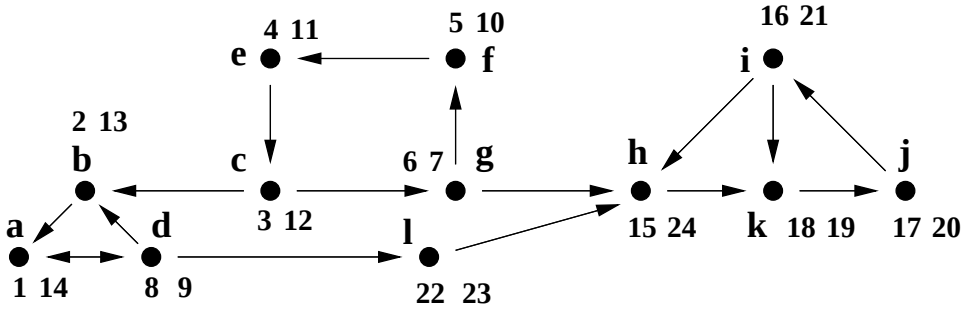
17. [20 points] Walk through Kosaraju's algorithm to find the strong components of the directed graph.



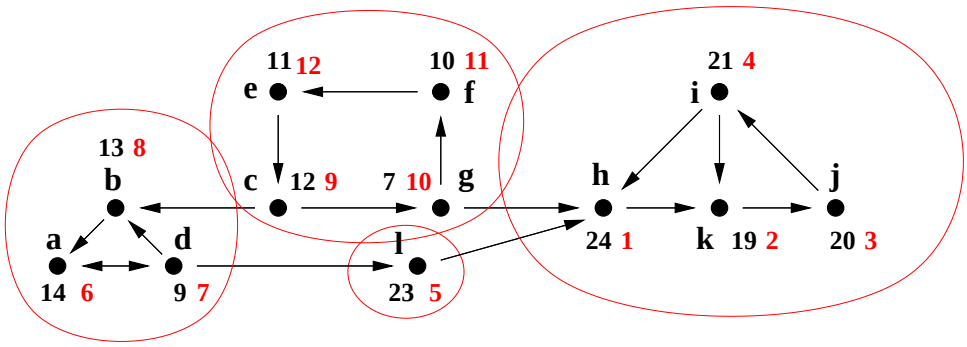
We first traverse the directed graph in depth first order. We mark each vertex v with two numbers: the first time we visit v and the finishing time, the time we have visited all descendants of v, that is, all vertices that can be reached from v. Each time the search stack is empty, we simply start at some unvisited vertex.



We now compute the transpose graph, i.e., the vertices are the same but the arcs are reversed.



We visit the vertices of the transpose graph in depth first order, starting from the vertex with the greatest finishing time, namely h.



Each time the search stack is empty, we call that a "break time," and we restart the search with the remaining vertex of highest finishing time. The nodes visited between any two break times make one strong component.

18. [20 points] Give pseudocode for the Bellman-Ford algorithm.

We have a weighted directed graph G. The vertices are integers $1 \ldots n-1$. The edges are ordered pairs of integers, $e[k]$ for $0 \leq k < m$. Each entry $e[k]$ is an arc from $e[k].s$ to $e[k].t$ of weight $e[k].w$. We assume there are no negative cycles. The goal is to compute the shortest path from 0 to i for each i. Our output will consist of two one-dimensional arrays, V and **back**. $V[i]$ will be the length (total weight) of the shortest path from 0 to i found so far, and **back**$[i]$ will be the second-to-the-last vertex on that path. Thus, **back**[0] is undefined. In the code, we assume all weights are integers.

```
V[0] = 0;
for(int i = 1; i < n; i++) V[i] = infinity;
bool finished = false;
while (not finished)
 {
  finished = true;
  for(int k = 0; k < m; j++)
   {
    int i = e[k].s;
```

```
    int j = e[k].t;
    int temp = V[i] + e[k].w;
    if(temp < V[j])
     {
      V[j] = temp;
      back[j] = i;
      finished = false;
     }
   }
 }
```

19. [20 points] Give pseudocode for the Floyd-Warshall algorithm.

We have a weighted directed graph. The vertices are the numbers from 0 to n-1. I will assume that the arc weights are given to us in a two-dimensional array W; $W[i, j]$ is the weight of the edge from i to j. We assume $W[i, j] = \infty$ if there is no arc from i to j. The output of our algorithm consists of two two-dimensional arrays, V and **back**. $V]i, j]$ will be the length of the shortest path from i j, and **back**$[i, j]$ will be the next-to-the-last vertex on that path. Thus **back**$[i, i]$ is undefined.

```
// initializing the values of V
for(int i = 0; i < n; i++)
 for(int j = 0; j < n ; j++)
  V[i,j] = infinity;
for(int i = 0; i < n; i++)
 V[i,i] = 0;
// Initializing paths consisting of one edge
for(int i = 0; i < n; i++)
 for(int j = 0; j < n ; j++)
  if(i != j)
   {
    V[i,j] = W[i,j];
    back[i,j] = i;
   }
// The main triple loop of Floyd-Warshall
for(int j = 0; j < n; j++) // the middle index must be
                           // the index of the outermost loop
 for(int i = 0; i < n; i++)
  for(int k = 0; k < n; k++)
   {
    temp = V[i,j]+V[j,k];
    if(temp < V[i,k])
     {
      V[i,k] = temp;
      back[i,k] = back[j,k]
     }
```

```
}
```