Distribution of Scores

```
Range   Scores
29-32:  32
33-36:  33
37-40:
41-44:  42  44
45-48:  46  47  47  48
49-52:  52
53-56:  54  56
57-60:  57  57  60 60
61-64:  64  64
65-68:  66  68  68
69-72:  69  71  71  71  72  72  72
73-76:  73  73  74  76  76
77-80:  77  77  77  79  80  80 80
81-84:  83  83  83  84  84  84
85-88:  85  86  86  86  86  87  87  88
89-92:  89  89  89  90  90
93-96:  93  94  94
97-100: 97  97  98  98  98  99  99 100 100 100

Average: 75.93      Median: 80.0
```

Typo fixes in Yellow boxes.

Part I: Do both problems.

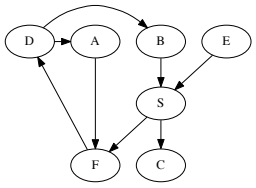
Problem Z: Graph Traversals ***(16 points)*** On the right is a directed graph. In what order are the vertices visited for:

(a) *(8 points)* a *depth-first search* that starts at S?

(b) *(8 points)* a *breadth-first search* that starts at S?

Note: There are multiple correct answers for parts a and b.

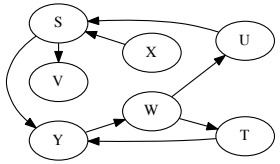
Exam 2a



Part (a): S C F D A B E

Part (b): S C F D A B E

Exam 2b



Part (a): S V Y W T U X

Part (b): S V Y W T U X

Problem T: Graph Structure ***(24 points)*** An undirected graph $G = (V,E)$ is an *unrooted tree* when G is connected and has no cycles. Give an $O(|V|+|E|)$-time algorithm that, given G (represented by adjacency lists), tests whether G is an unrooted tree. Explain why your algorithm is correct. You may assume that $V = \{1,\ldots,n\}$ for some positive integer n.

An answer for Question T. Do a depth-first search from any vertex s to determine:
(A) whether all vertices are reachable from s and
(B) whether there any cycles in the graph by checking for back edges.
If the answer is yes to (A) (i.e., the graph is connected) and no to (B) (i.e., the graph is acyclic), then the graph is a tree, otherwise, the graph fails to be a tree.

Here is one way to test for back edges: Add a new global variable *cyclic*, initialized to FALSE and change the loop in `dvsVisit` to:

for each v adjacent to u **do**
  if not($visited[v]$) **then** { $parent[v] \leftarrow u$; $dfsVisit(v)$ }
  else if $v \neq parent[u]$ **then** $cyclic \leftarrow$ TRUE

Since this is just a DFS on a adjacency-lists representation, the time is $O(|V|+|E|)$.

An alternative $O|V|$)-time answer for Question T. Recall: Trees satisfy $|V|-1 = |E|$. So:

First, start counting the number of edges in G. If this count ever goes above $|V|-1$, quit counting and return FALSE. If the count finishes with a sum $< |V|-1$, return FALSE; otherwise ($|E| = |V|-1$), do a dfsVisit from any vertex; if every vertex is reachable, return TRUE else return FALSE.

Time analysis: The count takes $\Theta(|V|)$ time and, if we do the DFS part, $|E| = |V|-1$ and thus $\Theta(|V|+|E|) = \Theta(|V|)$ in this case.

Part II: Do two of the three problems

IMPORTANT: If you try all three problems, mark which two you want graded; otherwise, I will grade all three and give you the *lowest* two marks.

Problem P: Paths in Graphs ***(30 points)*** Suppose $G = (\{1,\ldots,n\},E)$ is a directed graph that describes a communications network and, r is a function such that, for each $i,j \in \{1,\ldots,n\}$:

- if $(i,j) \in E$, then $r(i,j)$ = the probability that a message sent from i to j along edge (i,j) gets through; and
- if $(i,j) \notin E$, then $r(i,j) = 0$.

(So, $0 \le r(i,j) \le 1$ for all i and j.) For a path $P = (i_0,i_1),(i_1,i_2),\ldots,(i_{k-1},i_k)$, the *reliability* of P is: $r(i_0,i_1) \times \cdots \times r(i_{k-1},i_k)$. The *most-reliable* path from i to j is the path from i to j that has the *largest* reliability. *(Assume that G has at least one path between any two vertices.)*

YOUR PROBLEMS.

(a) *(16 points)* Describe how to modify Dijkstra's Algorithm so that, given G and r as above, your algorithm returns a table $mostRel[2..n]$ such that $mostRel[i]$ = the reliability of the most-reliable path from 1 to i.

(b) *(9 points)* Modify your algorithm so that it produces an array $path[2..n]$ where, for each $i \in \{2,\ldots,n\}$, $path[i]$ is a list consisting of a most reliable path.

(c) *(5 points)* Explain why your algorithm is correct. You may use the fact that Dijkstra's Algorithm is correct. *(Hint: Use the definition of **entropy** on the reference page.)*

An answer for Question P.

(a) You **max** instead of min and **multiply** instead of adding. Here is the algorithm.

function Dijkstra′(G,r)
  array $rel[1..n]$; $ToDo \leftarrow \{2,\ldots,n\}$
  for $i \leftarrow 2$ **to** n **do** $rel[i] \leftarrow r(1,i)$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    choose $v \in ToDo$ with **maximal** $rel[v]$
    $ToDo \leftarrow ToDo - \{v\}$
    for each $w \in ToDo$ **do** $rel[w] \leftarrow \mathbf{max}(rel[w], rel[v]{*}rel(v,w))$
  return rel

(b) The above builds a tree of most-reliable paths, so first we modify the algorithm to keep track of each vertex's parent in this tree by: *(i)* Adding an array $pred[2..n]$ which is initialize to all *null*'s; *(ii)* Changing the innermost **for**-loop to:

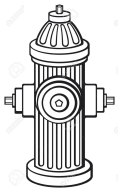
  for each $w \in ToDo$ **do**
    if $rel[w] < rel[v] * rel(v,w))$ **then** $\{\ rel[w] \leftarrow rel[v] * rel(v,w));\ \ pred[w] \leftarrow v\ \}$

Finally, we replace the "**return** *rel*" with:

for $i \leftarrow 2$ **to** n **do**
  $j \leftarrow i$; $path[i] \leftarrow$ the list consisting of just j
  while $j \neq 1$ **do** $\{\ j \leftarrow pred[j]$; Add j to the front of the list $path[i]\ \}$
return *path*

(c) Take $r(\cdot,\cdot)$, convert all the probabilities to their entropies, then run Dijkstra's original algorithm. It will construct the paths with the lowest entropies (i.e., paths with the highest reliabilities). Now convert the entropies in *dist* back to probabilities. By the rules for entropies, this version of *rel* will be identical to the one produced by the algorithm for part (a); so we are done.

Problem G: Greedy Algorithms ***(30 points)*** BACKGROUND. A town wants to place *fire hydrants* (see picture to the right) along a road so that each house is within 100 meters from some hydrant. So a hydrant placed at location x can cover all locations from $x-100$ to $x+100$, inclusive. *(Locations on the road are given in meters from the eastern end of the road.)* The locations of the houses along the road are: $x_1 < \cdots < x_n$. The town wants to use *as few hydrants as possible* to cover locations $x_1,\ldots,x_n$. You may assume the western end of the road continues to at least location x_n+100.



PROBLEMS.

(a) *(8 points)* Suppose $y_1,\ldots,y_k$ is an optimal placement of hydrants, where $y_1 < \cdots < y_k$. Let $y'_1 = x_1+100$. Explain why $y'_1, y_2, \ldots, y_k$ is also an optimal placement.

(b) *(16 points)* Present a $O(n)$ time greedy algorithm that, given $x_1,\ldots,x_n$ as above, finds an optimal placement of hydrants. Justify the run time.

(c) *(6 points)* Justify why your algorithm produces an optimal placement. *(Obvious hint: Use part a.)*

An answer for Question G. *(This is a reframing of Problem 5 from Homework 7.)*

(a) y'_1 is the highest location to place a hydrant to cover x_1. Hence, $y_1 \le y'_1$. Since x_1 is the lowest house location, the hydrant at y'_1 covers at least as many houses as one at y_1. Hence, since $y_1,\ldots,y_k$ covers all the houses, so does $y'_1,\ldots,y_k$ and since both placements have the same number of hydrants, the $y'_1,\ldots,y_k$ placements are optimal.

(b) We assume there is at least one house.

  $j \leftarrow 1$; $y_1 \leftarrow x_1+100$
  for $i \leftarrow 2$ **to** n **do** $\{$ **if** $(x_i > y_j + 100)$ **then** $\{\ j \leftarrow j+1;\ \ y_j \leftarrow x_i+100\ \}\ \}$
  return $[y_1,\ldots,y_j]$

The **for**-loop has $(n-1)$ iterations and a constant-time the body; hence, it is $\Theta(n)$ time.

(c) Suppose $\hat{y}_1,\ldots,\hat{y}_\ell$ is an optimal placement of hydrants.
CLAIM: For $j = 1,\ldots,k$, $y_1,\ldots,y_j$ cover all the houses that $\hat{y}_1,\ldots,\hat{y}_j$ do.
BASE CASE: $j=1$. This is just part (a).
INDUCTION STEP: Suppose $0 < j < k$. **IH:** The claim holds for $y_1,\ldots,y_{j-1}$. Let $\hat{x}$ be the lowest location not covered by $\hat{y}_1,\ldots,\hat{y}_{j-1}$. Then $\hat{y}_j \le \hat{x}+100$. By the algorithm, $\hat{x}+100 \le y_j$. Hence, it follows that the claim holds for j.
Finally, since $y_1,\ldots,y_{k-1}$ fail to cover all the houses, by the claim $\hat{y}_1,\ldots,\hat{y}_{k-1}$ also fail to cover all the houses. Hence, $k \le \ell$. Therefore, $y_1,\ldots,y_k$ is optimal.

Problem D: Dynamic Programming ***(30 points)*** Two teams, the *Ants* and the *Bees*, are to play a tournament. The first team to win k games is the tournament winner. Assume that, in any given game, the *Ants* have probability p of winning and the *Bees* have probability $1-p$ of winnning. (So there are no ties.) Let $P(i,j)$ be the probability that the *Ants* will win the tournament, given that they need to win i more games and that the *Bees* need to win j more games.

(a) *(3 points)* Explain why $P(k,k) =$ the probability that the *Ants* win the tournament.
(b) *(3 points)* Explain why $P(0,j) = 1$ when $j > 0$.
(c) *(3 points)* Explain why $P(i,0) = 0$ when $i > 0$.
(d) *(6 points)* Explain why: $P(i,j) \ = \ p \cdot P(i-1,j) + (1-p) \cdot P(i,j-1)$ when $i,j \in \{1,\ldots,k\}$
(e) *(10 points)* Use the formulas of parts b, c, and d to give a dynamic programming algorithm for computing $P(k,k)$.
(f) *(5 points)* Give a tight $\Theta(\cdot)$ bound on the running time of your algorithm.

An answer for Question D.

(a) At the start of the tournament, both teams have k games to win. Hence, by the definition of P, $P(k,k) =$ the probability that the *Ants* win the tournament.
(b) $P(0,j) = 1$, since the *Ants* have won their k-many games.
(c) $P(i,0) = 0$, since the *Bees* have won their k-many games.
(d) If the *Ants* need i more wins & the *Bees* need j more wins, then in the next game, either
  (i) the *Ants* win, in which case the *Ants*'s probability of winning the tournament is $P(i-1,j)$, or
  (ii) the *Bees* win, in which case the *Ants*'s probability of winning the tournament is $P(i,j-i)$.
As **(i)** happens with probability p and **(ii)** happens with probability $1-p$, we have

$$P(i,j) \ = \ p \cdot P(i-1,j) + (1-p) \cdot P(i,j-1).$$

(e) Here is the algorithm:

for $m \leftarrow 1$ **to** k **do** { $p[0,m] \leftarrow 1$; $p[m,0] \leftarrow 0$ }
for $i \leftarrow 1$ **to** k **do**
  for $j \leftarrow 1$ **to** k **do** $p[i,j] \leftarrow q \cdot p[i-1,j] + (1-q) \cdot p[i,j-1]$
return $p[k,k]$

(f) The **for**-loop on m is $\Theta(k)$. The nested **for**-loops both go from 1 to k and the $p[i,ij] \leftarrow \ldots$ statement is constant time. So their cost is $\Theta(k^2)$. Hence, the entire algorithm is $\Theta(k^2)$.

Some Reference Facts

a. $a^m \cdot a^n = a^{m+n}$.
b. $a^{m \cdot n} = (a^m)^n = (a^n)^m$.
c. $a^n \cdot b^n = (a \cdot b)^n$.
d. $\log_a a^n = n$.

e. $a^{\log_a n} = n$.
f. $c \cdot \log_2 a = \log_2 a^c$.
g. $\log_2(a \cdot b) = (\log_2 a) + (\log_2 b)$.

h. $(\log_a x)/(\log_a b) = \log_b x$.
i. $a^{\log_b c} = c^{\log_b a}$.
j. $\sum_{k=1}^{n} k = \frac{1}{2} \cdot n \cdot (n+1)$.

k. $\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1}$.
l. $\sum_{k=1}^{\infty} 2^{-k} = 1$.
m. $\sum_{k=1}^{n} k^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$.

n. $\sum_{k=1}^{\infty} k \cdot 2^{-k} = 2$.
o. For $a > 1$ and $b, c > 0$: $\lim_{n \to \infty} \frac{(\log_a n)^b}{n^c} = 0$.

p. For $a > 1$ and $b, c > 0$: $\lim_{n \to \infty} \frac{n^b}{a^{c \cdot n}} = 0$.

The Limit Rule. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, then:

	$c = 0$	$0 < c < \infty$	$c = \infty$
$f(n) \in O(g(n))$	True	True	False
$f(n) \in \Theta(g(n))$	False	True	False
$f(n) \in \Omega(g(n))$	False	True	True

A Version of Depth-First Search. $G = (V,E)$ is assumed to be represented by adjacency lists and $V = \{1,\ldots,n\}$. Also, for each vertex u: *parent*[u] ends up with u's parent in the DFS tree and d[u] ends up with u's discovery time.

```
Global vars:
visited[1..n], parent[1..n], d[1..n], time

procedure dfsInit(G)
  time ← 0
  for u ← 1,...,n do
    visited[u] ← false
    parent[u] ← nil
    d[u] ← (−1)

function dfs(G)
  dfsInit(G)
  for u ← 1,...,n do
    if not (visited[u]) then dfsVisit(u)
  return parent[1..n]

procedure dfsVisit(u)
  visited[u] ← true
  d[u] ← time
  time ← time + 1
  for each v adjacent to u do
    if not(visited[v]) then
      parent[v] ← u;  dfsVisit(v)
```

A Version of Breadth-First Search. $G = (V,E)$ is assumed to be represented by adjacency lists and $V = \{1,\ldots,n\}$. Vertex $s \in V$ is the start of the breadth-first search. For each vertex u: *parent*[u] ends up with u's parent in the BFS tree.

```
function bfs(G,s)
  for u ← 1,...,n do parent[u] ← nil
  Q ← an empty queue
  enqueue(Q,s);
  while Q is not empty do
    v ← dequeue(Q)
    for each w adjacent to v do
      if parent[w] = nil then
        parent[w] ← v;  enqueue(Q,w)
  return parent[1..n]
```

Dijkstra's Algorithm. GIVEN: $G = (\{1,\ldots,n\}, E)$ and a function $len: V \times V \to \mathbb{R} \cup \{\infty\}$ that:

(i) if $(u,v) \in E$, then $len(u,v) > 0$, and
(ii) if $(u,v) \notin E$, then $len(u,v) = \infty$.

GOAL: For each $i \in \{2,\ldots,n\}$, compute $dist[i]$ = the length of a *shortest* path from 1 to i.

```
function Dijkstra(G, ℓ)
  array dist[1..n]
  ToDo ← {2,...,n}
  for i ← 2 to n do dist[i] ← len(1,i)
  for i ← 1 to n − 1 do
    choose v ∈ ToDo with minimal dist[v]
    ToDo ← ToDo − {v}
    for each w adjacent to v do
      d ← dist[v] + len(v,w)
      if (d < dist[w]) then dist[w] ← d
  return dist
```

Entropy. Define $H(p) = -log_2 p$ for probability p. $H(p)$ is called the *entropy* of p. Note:

1. Since $0 \le p \le 1$, we have $0 \le H(p) \le \infty$.
2. $p < q \iff H(q) < H(p)$.
3. $H(p \times q) = H(p) + H(q)$.