Solutions to Homework 8

Help Center

Problem sparse_array_out

```
function success = sparse_array_out (A, filename)
    fid = fopen (filename,'w+');
    success = fid>=0;
    if ~success
        error ('Error opening file %s\n',filename)
    end
    [r,c,v] = find(A);                      % locations and values of non-zero element
s
    [nr,nc] = size(A);                      % dimensions of A
    nze = length(v);                        % number of non-zero elements
    fwrite (fid, [nr,nc,nze], 'uint32');    % global data
    for k = 1:nze                           % for each non-zero element ...
        fwrite (fid, [r(k),c(k)], 'uint32'); % row-column index
        fwrite (fid, v(k), 'double');       % value
    end
    fclose(fid);
end
```

Problem sparse_array_in

```
function A = sparse_array_in (filename)
    A = [];
    fid = fopen(filename,'r');
    if fid<0
        error ('Error opening file %s\n',filename)
    end
    x = fread(fid,3,'uint32');                  % global data (nr, nc, nze)
    A = zeros(x(1),x(2));                       % initialize new nr x nc matrix
    for k=1:x(3)                                % for each non-zero element ...
        z = fread(fid,2,'uint32');              % row, column
        A(z(1),z(2)) = fread(fid,1,'double');   % value
    end
    fclose(fid);
end
```

Probel letter_counter

```
function n = letter_counter(fname)
    fid = fopen(fname,'r');
```

```
    if fid < 0
        n = -1;
    else
        x = fread(fid,inf,'char');      % read entire file
        x = x(isletter(char(x)));       % pick the letters
        n = length(x);                  % count them
        fclose(fid);
    end
end
```

Probel letter_counter (alternative solution)

A shorter variant

```
function n = letter_counter(fname)
    n = -1;
    fid = fopen(fname,'r');
    if fid >= 0
        n = sum(isletter(fread(fid,inf,'char=>char')));
        fclose(fid);
    end
end
```

Problem saddle

```
function s = saddle(M)
    [r c] = size(M);
    s = [];
    if r > 1
        cols = min(M);          % find the min value in each column
    else
        cols = M;               % vector is a special case, min would give a single val
ue
    end
    if c > 1
        rows = max(M');         % find the max value in each row
    else
        rows = M;               % vector is a special case, max would give a single val
ue
    end
    for ii = 1:c                % visit each column
        for jj = 1:r            % and each row, that is, each element of M
            if M(jj,ii) == cols(ii) && M(jj,ii) == rows(jj) % if both conditions hold
                s = [s; jj ii];                             % saddle point! Let's add i
t!
            end
        end
    end
```

```
end
```

Problem prime_pairs

```
function p = prime_pairs(n)
    if isprime(2+n)             % many times the answer is 2
        p = 2;
    elseif rem(n,2)             % if not, and n is odd, no such prime exists
        p = -1;
    else
        for p = primes(1e5)     % check all primes smaller than 100,000
            if isprime(p+n)     % if p+n is prime
                return;         % found it! Return immediately
            end
        end
        p = -1;                 % none found (btw, we never get here)
    end
end
% It turns out that for n-s smaller than 100,000 that are even, there is
% always a pretty small such prime. In fact, the largest is 227.
% So we could use primes(300) instead of primes(1e5) to make this even
% faster. Also, the for-loop would be slow, if we did not check for even n-s,
% since it would need to go through all primes smaller than 100,000 to
% realize that no solution exists. So, handling the first two cases (p is 2
% and n is odd) separately makes the function very efficient.
```

Problem prime_pairs (alternative solution)

No loop at all. This illustrates yet again that there is always a MATLAB built-in function for almost anything reasonable...

```
function p = prime_pairs(n)

    allp = primes(1e5+n);       % Get all primes up to max value + n
    p = intersect(allp,allp+n); % Get which values are prime when n is added
    if isempty(p)               % Check to see if there are any such values
        p = -1;
    else
        p = p(1)-n;             % If so, subtract off the n to get the smaller value of
the prime pair
    end
end
% Elegant solution, but because it does not check for odd n and because it
% always handles the entire vector of primes even though the answer, if it
% exists, is small, it is about 4x slower than the for-loop version above
% even though the built-in function intersect is very fast.
```

Problem bowl

```
function score = bowl(balls)
    index = 0;                              % index into balls
    first = 1;                              % multiply next ball
    second = 1;                             % multiply ball after next
    score = 0;                              % cummulative sum
    if sum(balls > 10 | balls < 0) > 0      % single hit must be between 0 and 10 i
nclusive
        score = -1;                         % error!
        return;
    end
    for ii = 1:10                           % first ten frames
        index = index + 1;                  % take next ball
        if index > length(balls)            % not enough balls
            score = -1;                     % error!
            return;
        end
        score = score + first * balls(index);   % count score including extra from prev
ious strike or spare
        first = second;                     % move multiplier value from second to
first
        second = 1;                         % reset multiplier for the ball after n
ext to 1
        if balls(index) == 10               % strike
            first = first + 1;              % so next counts extra
            second = 2;                     % and so is the one after next
            continue;                       % go to next frame, there is no second
ball in this one
        end
        index = index + 1;                  % take next ball
        if index > length(balls)            % not enough balls
            score = -1;                     % error
            return;
        end
        score = score + first * balls(index);           % count score including extra f
rom previous strike
        first = second;                                 % move multiplier value from se
cond to first
        second = 1;                                     % reset multpilier for the ball
after next to 1
        if balls(index) + balls(index -1) == 10         % spare
            first = first + 1;                          % so next counts extra
        elseif  balls(index) + balls(index -1) > 10     % cannot score higher than 10 i
n a frame
            score = -1;                                 % error!
            return;
        end
    end
```

```
    for ii = [first second]                         % max 2 extra balls if needed
        if ii < 2                                   % no extra ball here
            break;                                  % we are done
        end
        index = index + 1;                          % take next ball
        if index > length(balls)                    % not enough balls
            score = -1;                             % error!
            return;
        end
        score = score + (ii-1) * balls(index);  % extra balls: count them one less than
a normal ball
    end
    if index < length(balls)                        % additional ball in the input
        score = -1;                                 % error!
    end
end
```

Problem maxsubsum

traditional brute-force solution with four nested loops

```
function [x y rr cc s] = maxsubsum(A)
    [row col] = size(A);
    % initialize result to the 1-by-1 subarray at the top left corner of A
    x = 1;                                          % top left corner of subarray
    y = 1;                                          % top left corner of subarray
    rr = 1;                                         % height of subarray
    cc = 1;                                         % width of subarray
    s = A(1,1);                                     % sum

    for r = 1:row                                   % height of subarray
        for c = 1:col                               % width of subarray
            for ii = 1:row-r+1                      % start position row
                for jj = 1:col-c+1                  % start position col
                    tmp = sum(sum(A(ii:ii+r-1,jj:jj+c-1))); % sum up candidate
                    if tmp > s                      % if larger than current max
                        s = tmp;                    % set the new values
                        x = ii;
                        y = jj;
                        cc = c;
                        rr = r;
                    end
                end
            end
        end
    end
end
```

Problem maxsubsum (alternative solution)

Using Kadane's algorithm. Kadane's algorithm finds the contiguous subvector with the max sum within a vector using a single loop. For a detailed explanation, google "Kadane's algorithm maximum subarray problem." It is much faster than the previous solution because it needs only three nested loops. Try both with a 100x100 matrix and you'll see the difference :)

```
function [fx1 fy1 rr cc mx] = maxsubsum(A)
    [row col] = size(A);
    mx = A(1,1)-1;
    for ii = 1:row
        tmp = zeros(1,col);
        for jj = ii:row
            tmp = tmp + A(jj,:);
            [y1 y2 cur] = kadane(tmp);
            if cur > mx
                mx = cur;
                fx1 = ii;
                rr  = jj-ii+1;
                fy1 = y1;
                cc  = y2-y1+1;
            end
        end
    end
end

function [x1, x2, mx] = kadane(v)
    mx = v(1);
    x1 = 1; x2 = 1;
    cx1 = 1;
    cur = 0;
    for ii = 1:length(v)
        cur = cur+v(ii);
        if(cur > mx)
            mx = cur;
            x2 = ii;
            x1 = cx1;
        end
        if cur < 0
            cur = 0;
            cx1 = ii + 1;
        end
    end
end
```

Problem queen check

It uses the fact that a diagonal either starts in the first column or ends in the last column (or both). Only sum and max built-in functions are used.

```
function ok = queen_check(board)
    n = 8;
    ok = true;
    v = board(:);                           % create a vector in col major order
    w = v(end:-1:1);                        % reverse order, so last col becomes first
col
    for ii = 1:n
        tests = [
                sum(board(:,ii))            % row #ii
                sum(board(ii,:))            % col #ii
                sum(v(ii:n+1:(n-ii+1)*n))   % diagonal starting in the first column goi
ng down
                sum(v(ii:n-1:ii*n-1))       % diagonal starting in the first column goi
ng up
                sum(w(ii:n+1:(n-ii+1)*n))   % diagonal starting in the last  column goi
ng up
                sum(w(ii:n-1:ii*n-1))       % diagonal starting in the last  column goi
ng down
                ];
        if max(tests) > 1                   % these should be all 0 or 1
            ok = false;                     % otherwise return false
            return;
        end
    end
end
```

Problem queen check (alternative solution)

Surprise, surprise: MATLAB has a built-in function called diag and flip

```
function safe = queen_check (B)
    inC = sum(B);                       % sum of queens in each column
    inR = sum(B,2)';                    % sum of queens in each row
    F = flip(B);                        % flipped board for antidiagonals
    for k=-6:6
        inD(k+8) = sum(diag(B,k));      % sum of queens in each diagonal
        inE(k+8) = sum(diag(F,k));      % sum of queens in each antidiagonal
    end
    safe = max([inR inC inD inE])<=1;   % queen counts at most one
end
```

Problem roman2

Nice and short solution

```
function A = roman2 (R)
% This function initially assumes the supplied input is valid. If it is not valid,
% the result, when converted back to Roman, will differ from the original input.
```

```
    Roman = 'IVXLC';
    Arabic = {1 5 10 50 100};
    LastValue = 0;                  % V is value, LastValue is last V
    A = uint16(0);
    for k = length(R):-1:1          % scan backward from last character
        P = strfind(Roman,R(k));    % search list of valid Roman characters
        if isempty(P)               % if invalid
            V = 0;                  % value is zero
        else                        % else
            V = Arabic{P};          % value is Arabic equivalent
        end
        if V<LastValue              % if subtractive situation
            A = A-V;                % subtract this value
        else                        % else
            A = A+V;                % add this value
        end                         % (in either case, V=0 did nothing)
        LastValue = V;              % update last value used
    end
    if A>=400 || ~strcmp(R,A2R(A))  % if out of range or result does
        A = uint16(0);              % not generate original string
    end                             % send back zero
end

% convert Arabic to Roman
function R = A2R (A)
% Remove subtraction by including secondary moduli.
    Roman = {'I' 'IV' 'V' 'IX' 'X' 'XL' 'L' 'XC' 'C'};
    Arabic = {1 4 5 9 10 40 50 90 100};
    R = ''; k = 9;
    while k>0                       % remove larger moduli first
        if A>=Arabic{k}             % if value is at least current modulus
            A = A-Arabic{k};        % remove modulus from value
            R = [R Roman{k}];       % append Roman character
        else                        % else
            k = k-1;                % consider next smaller modulus
        end
    end
end
```

Problem roman2 (alternative implementation)

Uses a Finite State Machine (FSM). For a detailed description, download this PDF document.

```
function num = roman2(rom)
% State machine-based implementation
        % the variable states contains the value of each state
        % the index into this vector is the ID of the given state
    states = [0 1 1 1 3 8 5 1 1 1 10 10 10 30 80 50 10 10 10 100 100 100];
        % each row of trans contains one state transition
```

```
        % 1st col: current state; 2nd col: input char; 3rd col: next state
    trans = [
        1  'I'  2; 1  'X' 11;  1 'C' 20; 1 'L' 16; 1  'V'  7;
        2  'I'  3; 2  'V'  5;  2 'X'  6;
        3  'I'  4;
        7  'I'  8;
        8  'I'  9;
        9  'I' 10;
        11 'X' 12; 11 'V'  7; 11 'I'  2; 11 'L' 14; 11 'C' 15;
        12 'X' 13; 12 'V'  7; 12 'I'  2;
        13 'V'  7; 13 'I'  2;
        14 'V'  7; 14 'I'  2;
        15 'V'  7; 15 'I'  2;
        16 'V'  7; 16 'I'  2; 16 'X' 17;
        17 'V'  7; 17 'I'  2; 17 'X' 18;
        18 'V'  7; 18 'I'  2; 18 'X' 19;
        19 'V'  7; 19 'I'  2;
        20 'V'  7; 20 'I'  2; 20 'C' 21; 20 'X' 11; 20 'L' 16;
        21 'V'  7; 21 'I'  2; 21 'C' 22; 21 'X' 11; 21 'L' 16;
        22 'V'  7; 22 'I'  2; 22 'X' 11; 22 'L' 16;
    ];

    state = 1;                                          % initial state: 1
    num = 0;                                            % initial value: 0
    for ii = 1:length(rom)                              % take input from left
        state = next_state(state, rom(ii), trans);      % find next state
        if state == -1                                  % no such transition
            num = 0;                                    % illegal roman number
            break;                                      % get out
        end
        num = num + states(state);                      % otherwise, increase value
    end
    num = uint16(num);
end

function state = next_state(state,ch,trans)
    for ii = 1:size(trans,1)                            % check each legal transition
        if trans(ii,1) == state && trans(ii,2) == ch    % for current state and input c
har
            state = trans(ii,3);                        % return next state
            return;
        end
    end
    state = -1;                                         % no transition found
end
```

Problem bell

```
function x = bell(n)
```

```
    % Check input (integer >= 1)
    if (n ~= floor(n)) || (n < 1)
        x = [];
    elseif (n == 1)
        % Special case of n = 1
        x = 1;
    else
        % Make matrix of zeros
        x = zeros(n);
        % Fill in top-left corner for 2-by-2
        x(1:2,1:2) = [1 2;1 0];
        % Loop over remaining "lines"
        for k = 3:n
            % 1st element of the line k is the last element of line k-1
            x(k,1) = x(1,k-1);
            % Loop over the remaining elements
            for j = 2:k
                % jth element is sum of j-1 element of current line plus
                % j-1 element of previous line
                x(k-j+1,j) = x(k-j+1,j-1) + x(k-j+2,j-1);
            end
        end
    end
end
```

Published with MATLAB® R2014a

Created Mon 15 Jun 2015 1:29 PM PDT

Last Modified Tue 16 Jun 2015 8:32 AM PDT