Assignment 2 Solution

Knapsack problem

Recursive, DP and linear DP

```matlab
function value = knapRecursive(objectIndex, sizeList, valueList, roomInKnapsack)
% RECURSUVE, returns the value of the main function
    if (roomInKnapsack == 0)
       value=0;
       return;
   end
   if (roomInKnapsack<0)
       value = -1000000000000000;
       return;
   end
   if (objectIndex == 0)
       value=0;
       return;
   end
   if (roomInKnapsack - sizeList(objectIndex) < 0)
       value = knapRecursive(objectIndex-1, sizeList, valueList, roomInKnapsack);
   else
       value = max(knapRecursive(objectIndex-1, sizeList, valueList, roomInKnapsack), ...
                       knapRecursive(objectIndex-1, sizeList, valueList, roomInKnapsack - sizeList(objectIndex))+valueList(objectIndex));
   end
end
```

Simple recursive knapsack code

```
function [ cacheValues ] = knapDP(sizeList, valueList, roomInKnapsack)
%returns the solution values contained in a hash table
%key is numberOfItems knapsackSize, computed by knapKey
    numberOfObjects = size(sizeList,1);
    cacheValues = hash;
    for i=1:1:numberOfObjects
        for j=0:1:roomInKnapsack
            if (i==1)
                if (j-sizeList(i)<0)
                    cacheValues(knapKey(i, j)) = 0;
                else cacheValues(knapKey(i, j)) = max(0, valueList(i));
                end
            else
                    if (j-sizeList(i)<0)
                        cacheValues(knapKey(i, j)) = cacheValues(knapKey( i-1,j));
                    else cacheValues(knapKey(i, j)) = ...
                            max(cacheValues(knapKey( i-1, j)), ...
                                cacheValues(knapKey(i-1,j-sizeList(i)))+valueList(i));
                    end
            end
        end
    end
end
```

Basic quadratic space algorithm

```matlab
function [ useObject ] = knapSolutionDP(sizeList, valueList, roomInKnapsack)
% Trace back through the whole solution array
   cacheValues = knapDP(sizeList, valueList, roomInKnapsack);
   numberOfObjects = size(sizeList,1);
   useObject = zeros(numberOfObjects,1);
   solutionSize = roomInKnapsack;
   % work back through the objects
   for objectIndex = numberOfObjects:-1:1
      if (solutionSize-sizeList(objectIndex)>=0)
         backIndex = objectIndex-1;
         if (backIndex > 0)
            % ~= is not equals, we used this object, reduce solution size
            if (cacheValues(knapKey(objectIndex, solutionSize)) ~= cacheValues(knapKey(backIndex, solutionSize)))
               useObject(objectIndex) = 1;
               solutionSize = solutionSize - sizeList(objectIndex);
            end
         else useObject(objectIndex) = (cacheValues(knapKey(objectIndex, solutionSize)) ~= 0); % did we use object 1?
         end
      end
   end
end
```

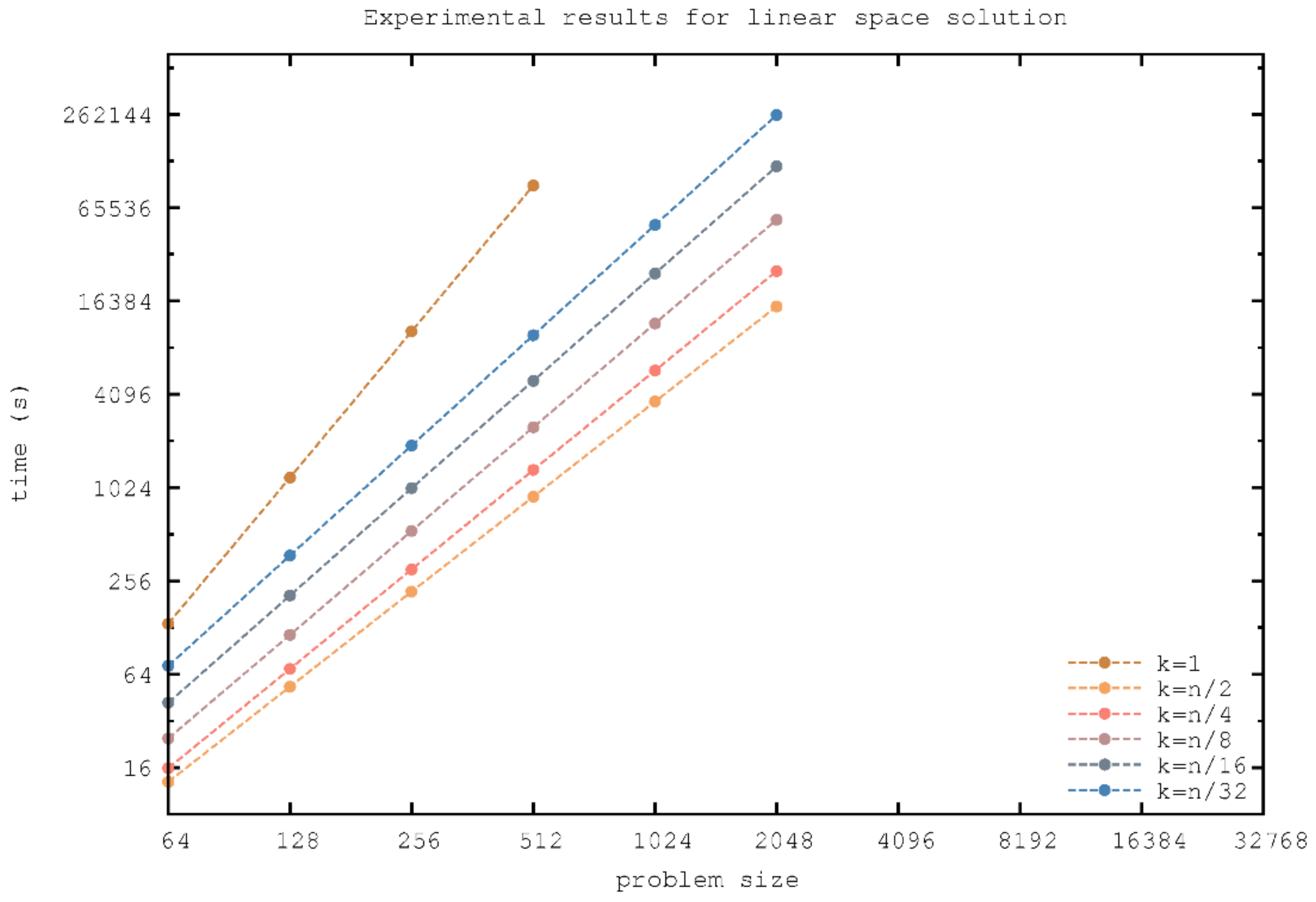
Traceback routine implemented iteratively. Returns a linear array where useObject[i] is true if i was used.

```
function [ lastColumn ] = knapDPLinear(sizeList, valueList, roomInKnapsack)
%returns the solution values in a linear array 1 column roomInKnapsack rows
    numberOfObjects = size(sizeList,1);
    cacheValues = hash;
    for i=1:1:numberOfObjects
        for j=0:1:roomInKnapsack
            if (i==1) %first column
                if (j-sizeList(i)<0) %out of range
                    cacheValues(knapKey(mod(i,2), j)) = 0;
                else cacheValues(knapKey(mod(i,2), j)) = max(0, valueList(i));
                end
            else
                if (j-sizeList(i)<0) %out of range
                    cacheValues(knapKey(mod(i,2), j)) = cacheValues(knapKey(mod(i-1,2), j));
                else cacheValues(knapKey(mod(i,2), j)) = ...
                        max(cacheValues(knapKey( mod(i-1,2), j)), ...
                            cacheValues(knapKey(mod(i-1,2), j-sizeList(i)))+valueList(i));
                end
            end
        end
    end
    lastColumn = zeros(1, roomInKnapsack);
    for j = 0:1:roomInKnapsack
        lastColumn(1, j+1) = cacheValues(knapKey(mod(numberOfObjects,2), j));
    end;
end
```

Linear space DP algorithm, note the use of mod to reuse memory

```matlab
function [ sol ] = knapDPsolutionDC(sizeList, valueList, roomInKnapsack)
%DQ recursive algorithm, linear space
  numberOfObjects = size(sizeList,1);
  if (numberOfObjects == 1) % done, return answer
     sol = (sizeList(1) <= roomInKnapsack);
   else midObject = numberOfObjects/2;
     %columns are +1 to avoid the start from 1 problem
     %solve both sides using linear scan
     leftColumn = knapDPLinear(sizeList(1:midObject), valueList(1:midObject),  roomInKnapsack);
     rightColumn = knapDPLinear(fliplr(sizeList(midObject+1:numberOfObjects)), fliplr(valueList(midObject+1:numberOfObjects)), roomInKnapsack);
     %find max and maxArg
     maxValue = 0;
     bestSize = 0;
     for knapSize = 0:1:roomInKnapsack
        if (maxValue < leftColumn(knapSize+1) + rightColumn(roomInKnapsack - knapSize+1))
           maxValue = leftColumn(knapSize+1) + rightColumn(roomInKnapsack - knapSize+1);
           bestSize = knapSize;
        end
     end
     % solve remaining two sides recursively
     leftSolutionIndexes = knapDPsolutionDC(sizeList(1:midObject), valueList(1:midObject), bestSize);
     rightSolutionIndexes = knapDPsolutionDC(sizeList(midObject+1:numberOfObjects), valueList(midObject+1:numberOfObjects), roomInKnapsack-bestSize);
     % append the two solutions together
     sol = horzcat(leftSolutionIndexes, rightSolutionIndexes);
  end
end
```

Divide and conquer linear space algorithm. Returns a linear array of objects used same as trace back



Run time results for the D&C algorithm with different problem splits. Note now when the sub-problems are split to 1 and n-1 we get n^2 time, otherwise we get nlogn time. The multiplier increases are we go away from ½, which corresponds to an offset on the log graph