KEY

CS5050 FIRST MIDTERM (all questions worth 12 points each)
NAME: ANUMBER:

1) A colleague has developed the following DP algorithm. The problem is that it requires too much space with large **n** and she needs your help in redesigning it. Give your solution as annotations on the algorithm below:

```
                           6
Bool solutionCache[n, Size] //allocate the cache
for i = 0 to n-1 // loop through the objects
     for j = 0 to Size  // loop over the sizes                    (i-k)% 6
          solution = false;
          for k=1 to 5 //check back the last 5 objects
               if (i-k)>= 0
                    solution = solution && solutionCache[i-k, j-size[i-k]]
               end if
          end for                   i%6
          solutionCache[i, j] = solution;
     end for
end for
return solutionCache[n, Size]           n%6
```

2) Consider the *linear space divide and conquer* DP algorithm for the knapsack problem where there are **n** items and the space remaining is **S**. Consider applying this algorithm to a different problem where we want to identify a subset of objects that exactly fit into the knapsack. Let **Bool leftColumn** be a linear array containing the solutions for objects **1…n/2-1** and **Bool rightColumn** be a linear array containing the solutions for objects **n/2…n**. So location **[j]** in the array is the solution for a knapsack of size **j**.

   a. Write the pseudo code to determine the **bestSize** split value.

```
bestSize = -1
for j = 0 to S
    if leftColumn[j] && rightColumn[S-j]
        bestSize = j
        break
    end
end
```

   b. Is there a way to terminate early for this Boolean knapsack problem? If so explain.

if (bestSize == -1) return false

there exists no solution so return without having to check further sub solutions

KEY

Here is the "cookbook" solution for D&C recurrence relations.

$f(n) = a\ f(n/b) + c\ n^k$	
if $a > b^k$	**$f(x) \sim= n^{(\log_b a)}$**
if $a = b^k$	**$f(x) \sim= n^k \log n$**
if $a < b^k$	**$f(x) \sim= n^k$**

3) Use the cookbook to help fill in the following table:

	f(n)=3f(n/2)+n	**f(n)=3f(n/3)+n**	**f(n)=2f(n/3)+n^2**
How many recursive calls?	$3 = a$	$a = 3$	$a = 2$
Reduction in problem size?	$2 = b$	$b = 3$	$b = 3$
Work done each call?	$n^{1 = k}$	$n^{1 = k}$	$n^{2 = k}$
Closed form solution	$n^{\log_2 3}$	$n \log n$	n^2

4) We have seen three variations of dynamic programming: a) simple where we use the full cache array, b) when we do a linear scan keeping only a fixed number of columns, and c) when we combine linear scan with divide and conquer. Explain the circumstances where each algorithm (a, b, c) is preferable to the others.

a) when space is not a problem and need the objects or DP needs all the previous columns

b) when space needs to be minimized and only the solution - not the objects are needed

c) when space needs to be minimized and both the solution and objects are needed

5) Write efficient pseudo code to solve the following specific problem: Given two linear functions **a0+a1x** and **b0+b1x**, compute the three coefficients **c0, c1, c2** where **c0+c1x+c2x²** is the product of multiplying the two input linear functions:

$C_0 = a_0 * b_0;$

$C_2 = a_1 * b_1;$

$C_1 = (a_0 + a_1) * (b_0 + b_1) - C_0 - C_2;$

$(a_0 + a_1 x)(b_0 + b_1 x) =$

$a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2$

$(a_0 + a_1)(b_0 + b_1) = (a_0 b_1 + a_1 b_0) + a_1 b_1 + a_0 b_0$

KEY

6) You are working for a game company and they are developing a new game that is played on a 256 by 256 board. The user controls a frog that can jump up 9 squares, or down 5 squares or left 11 squares, or right 4 squares. The goal is to reach a fly that is placed on the board at the 64, 64 coordinate on the board. They need you to write an algorithm to determine if the user can catch the fly if they start at coordinate **i, j**. Write the recursive algorithm **bool canCatch(int i, int j)** (not the DP or other optimizations).

```
if (i > 256 || j > 256) return false
if (i ≤ 0 || j ≤ 0) return false;
if (i == 64 && j == 64) return true;
return canCatch(i-11, j) ||
       canCatch(i+4, j) ||
       canCatch(i, j+9) ||
       canCatch(i, j-5)
```

7) The most efficient way to evaluate a single polynomial at a value **x** is to use the add-then-multiply technique. Here are example for polynomials with increasing coefficients:

$P(x) = p_0$

$P(x) = p_0 + xp_1$

$P(x) = p_0 + x(p_1 + xp_2)$

$P(x) = p_0 + x(p_1 + x(p_2 + xp_3))$

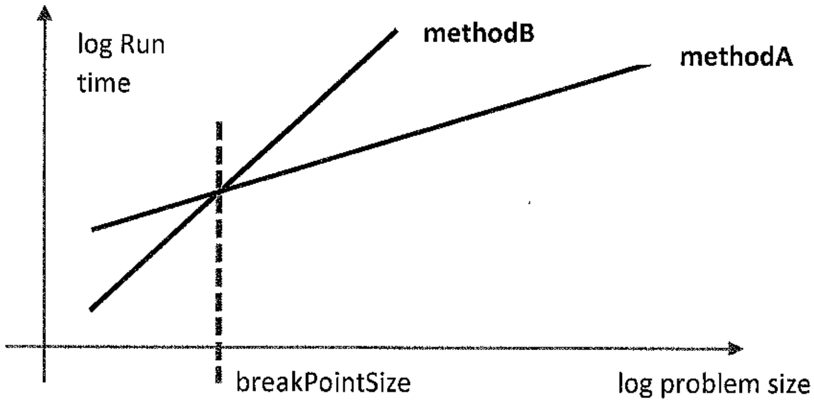
$P(x) = p_0 + x(p_1 + x(p_2 + x(p_3 + p_4x)))$

Write a function **double evalPoly(double[] P, double x, int n)** that uses this technique to efficiently evaluate the polynomial **P** (an array of coefficients, where **P[i]** is **P_i** **x** is the value and **n** is the number of coefficients. You can write the code using recursion or iteration.

```
sol = P[n-1]
for i = n-2 down to 0
    sol = P[i] + x * sol;
end
return sol
```

KEY

8) Given the following performance log-log graph comparing **methodA** and **methodB** the run time problem size:



a. What is the significance of the problem size where the two lines cross?

here is where the most efficient algorithm switches over. when size < breakPoint size method B is faster, otherwise method A is faster

b. Write a function that with run fastest for all problem sizes. This code will call **methodA** and/or **methodB**.

if size < breakpoint size
return method B( )
else return method A( )

c. What can we determine about the functions knowing that they appear linear on the log/log graph?

the function mapping problem size to run time must be of the form $time = c n^a$ since

$$\underbrace{\log time}_{} = \underbrace{a}_{\text{slope}} \underbrace{\log n}_{} + \underbrace{\log c}_{\text{offset}}$$