Anthony Gringeri
Algorithms: Project 2
Executive Summary Report

Introduction

This project involves using recursion and dynamic programming to calculate the probability of a baseball team winning a series of games, given *n* number of games needed to win the series and probability *p* of winning a game, which is the same for each game played. The first algorithm was written using just recursion, and the second using recursion as well – with the caveat of using dynamic programming strategies. Dynamic programming (DP) involves saving computed values in some type of auxiliary data structure. This trade-off is well worth it, making the DP algorithm much more efficient as you will see. My algorithm creates a two-dimensional array and saves already-computed values to its position, accessing the saved value when needed to avoid unnecessary recalculations.

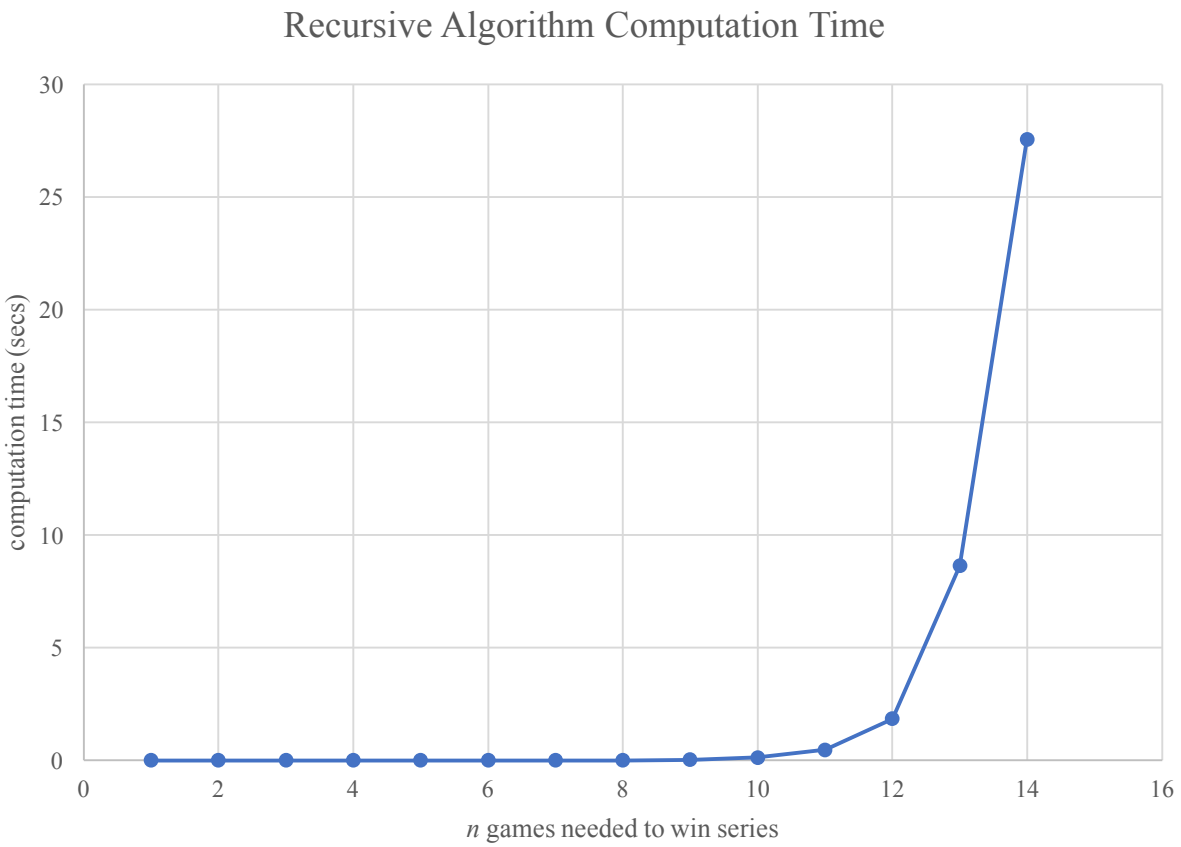
Summary of time and space efficiency analysis

The following recurrence equation was given and describes the problem at hand:

$$P(i,j) = pP(i-1,j) + qP(i,j-1), for\ i,j > 0, where\ q = (1-p)$$

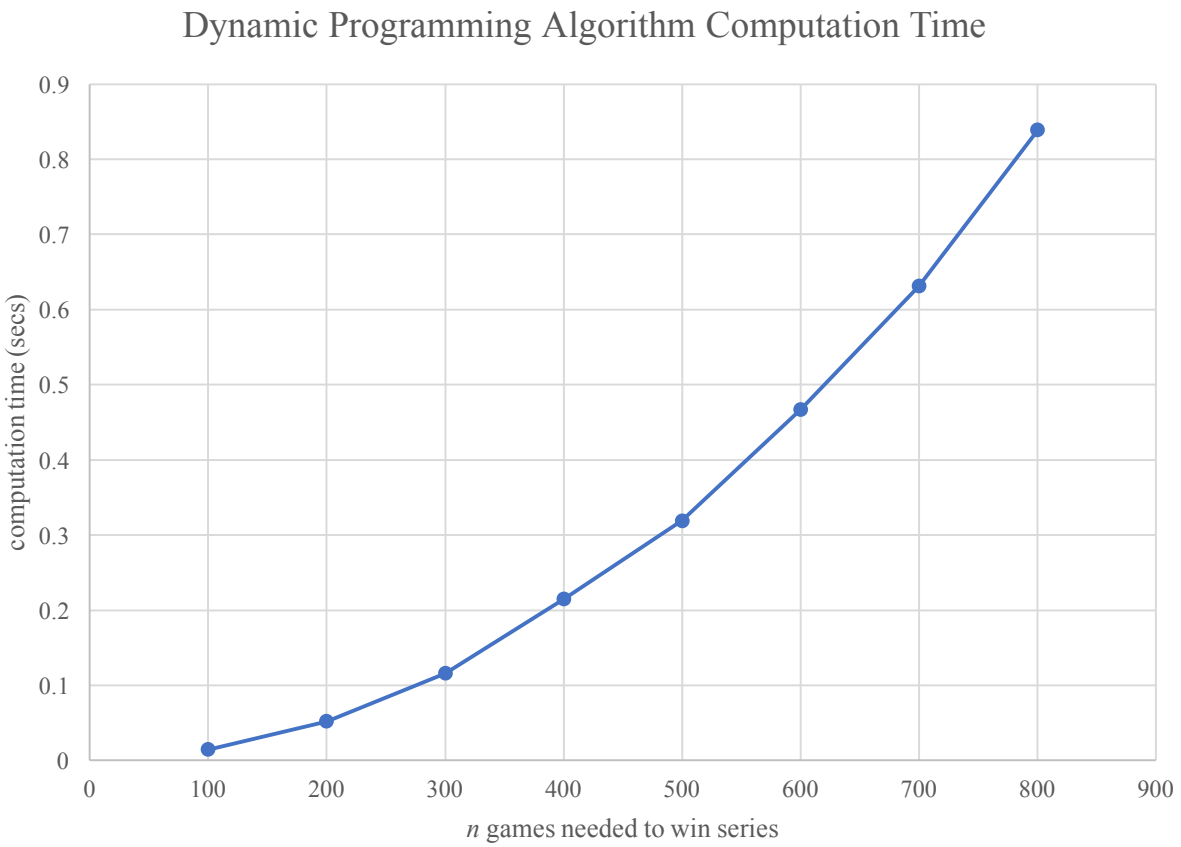
Analyzing both algorithms provided useful data, but were met with some obstacles. In testing the strictly recursive algorithm, any values of *n* above 14 could not be computed in a reasonable amount of time given my resources. The strictly recursive algorithm hands off its work to a helper function which keeps track of both i and j in the given recurrence equation, recurring until a base case is reached. This method proves extremely inefficient as soon as values of $n = 10$ or larger are tested. No values greater than $n = 14$ could be tested on my machine. From the data that was collected, the following graph was composed:

Anthony Gringeri
Algorithms: Project 2
Executive Summary Report



Because of the nature of the recursive function, I classified the strictly-recursive algorithm in the efficiency class of $O(2^n)$. Because this algorithm is in-place, using no auxiliary data structures, this algorithm has no absolute space efficiency. Given the large amount of data that needs to be stored on the stack of the machine running the code, it might be acceptable to consider the space efficiency up to the efficiency class of $O(n)$.

In testing the DP algorithm, I could test up to values of $n \sim 800$, until a still unknown testing anomaly would not allow the algorithm to finish. I propose that it might be related to the depth of the recursion, not the actual computations themselves, since tests up to $n \sim 800$ were completed in less than one second. Nonetheless, enough data was collected to conclude the efficiency class of the DP algorithm.



The underlying logic of the DP algorithm is essentially the same as the strictly-recursive algorithm, with the exception that already-computed values are stored in an auxiliary two-dimensional list structure. Instead of recalculating values many times, it searches a specific position in the array for a calculated value. This tradeoff is well worth it, though, as the small change now makes this algorithm able to be completed in polynomial time in the efficiency of $O(n^2)$. As a trade-off, this algorithm has larger space efficiency of $O(n^2)$ due to the values stored in the auxiliary two-dimensional array.

Anthony Gringeri
Algorithms: Project 2
Executive Summary Report

Project questions and proposed solutions

1) See included baseball.py file.

2) Find the probability of the Red Sox winning a seven-game series if the probability of it winning a game is 0.4. This means that n is 4. In your report, show your dynamic programming table $P\ (i, j)$ with its entries rounded-off to two decimal places.

$\boldsymbol{P(i,j)}$	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$
$j = 0$		0.00	0.00	0.00	0.00
$j = 1$	1.00	0.40	0.16	0.06	0.02
$j = 2$	1.00	0.64	0.35	0.18	0.08
$j = 3$	1.00	0.78	0.52	0.32	0.18
$j = 4$	1.00	0.87	0.66	0.46	0.29

3) See included baseball.py

4) Describe the pros and cons of both your algorithms in your report.

- Strictly-recursive algorithm: This algorithm has the obvious con of being extremely inefficient. Although it is very slow, it uses no auxiliary data storage (apart from quite a large amount of data being stored on the stack) and is easy to read in python.

- DP algorithm: This algorithm, though still quite inefficient, can at least be completed in polynomial time. If the function could be translated into iterative instead of recursive, this would prove to be even more efficient. The trade-off is the data being stored in the auxiliary array which can get large quickly with a large problem domain.

5) Use a similar infrastructure that you used with Project 1 to time and test your algorithms. Record your algorithms time and space efficiencies as well as your test runs in your report.

n	Recursive Algorithm Computation Time (sec)	DP Algorithm Computation Time (sec)
1	0.000043430	0.000030500
2	0.000054018	0.000035388
3	0.000072313	0.000046033
4	0.000175103	0.000063517
5	0.000354315	0.000082775
6	0.001446244	0.000176607
7	0.003528381	0.000115473
8	0.011888830	0.000167719
9	0.037477766	0.000122794
10	0.128250919	0.000131990
11	0.478887597	0.000175043
12	1.848472058	0.000180275
13	8.640271058	0.000250516
14	27.558760679	0.000251750
100		0.014700908
200		0.052266121
300		0.116138999
400		0.214798833
500		0.318916679
600		0.466670729
700		0.631126362
800		0.838939313

Anthony Gringeri
Algorithms: Project 2
Executive Summary Report

Algorithm pseudocode

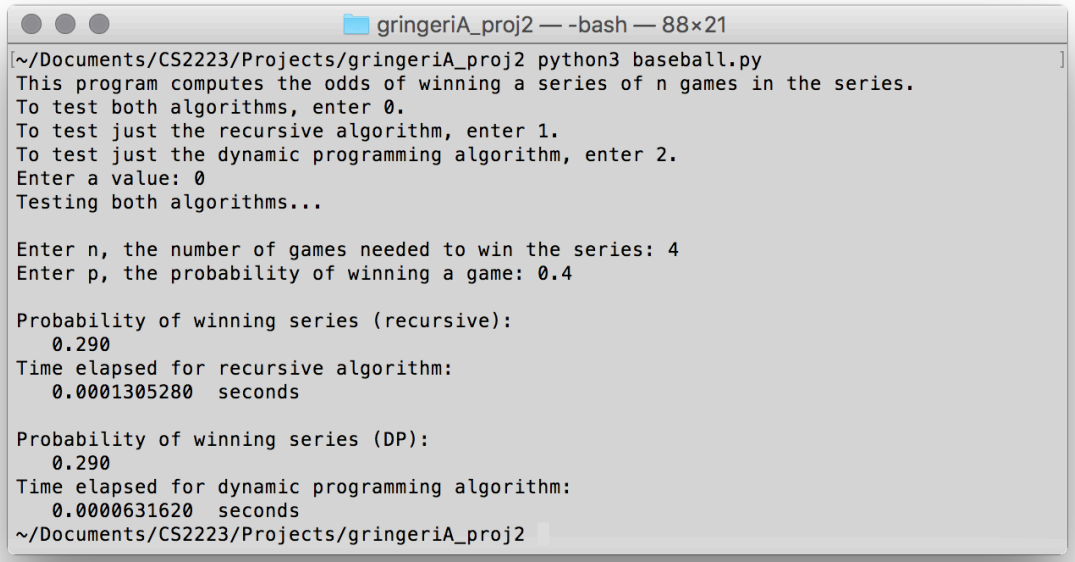
Strictly-recursive algorithm

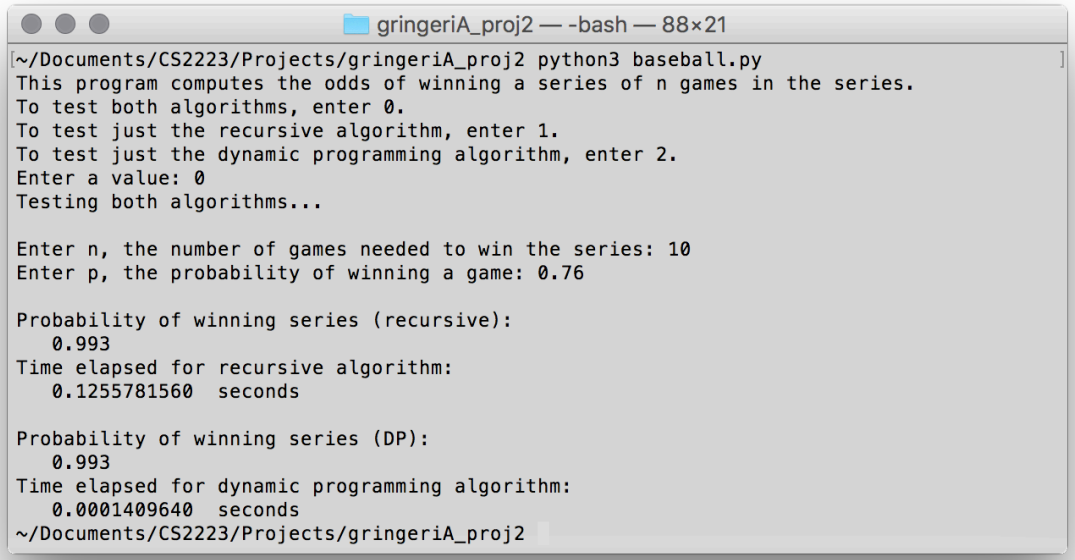
```
recursiveProb(n, p):
    return recursiveProbHelper(n, n, p)

recursiveProbHelper(j, i, p):
    if ((i < 0) or (j < 0)):
        return false // error
    else if (i == 0):
        return 1 // base case
    else if (j == 0):
        return 0 // base case
    else:
        return (((p)*(recursiveProbabilityHelper((i-1), j, p)))
            + ((1-p)*(recursiveProbabilityHelper(i, (j-1), p))))
```

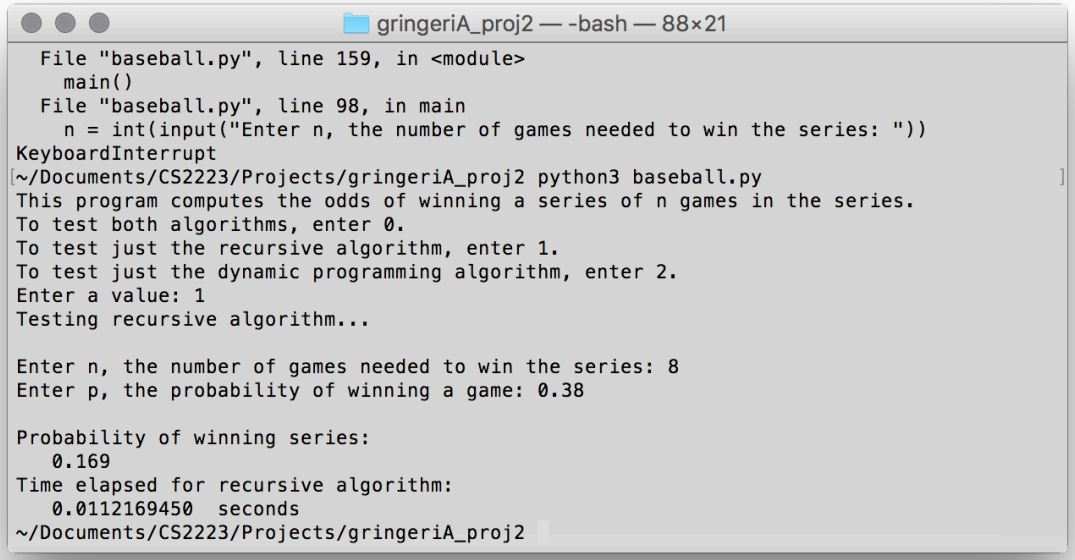
Dynamic programming algorithm

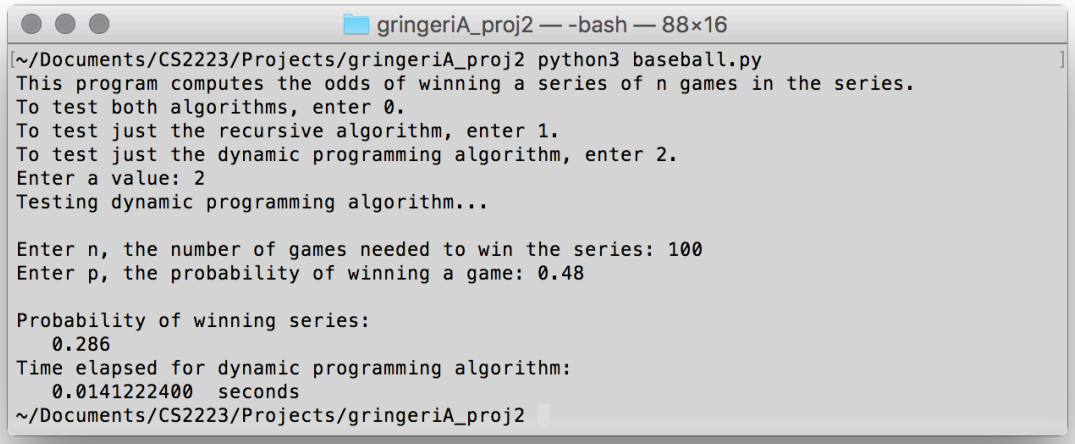
```
DPProb(n, p):
    matrix[n+1][n+1] = {0 ... 0} // all elements initialized to zero
    return DPHelper(n, n, p, matrix)

DPHelper(j, i, p, matrix):
    if ((i < 0) or (j < 0)):
        return false // error
    else if (i == 0):
        return 1 // base case
    else if (j == 0):
        return 0 // base case
    else if (matrix[i][j] != 0):
        return matrix[i][j]
    else:
        matrix[i][j] =  (((p)*(DPHelper((i-1), j, p, matrix)))
            + ((1-p)*(DPHelper(i, (j-1), p, matrix))))
        return matrix[i][j]
```

Screenshots of program testin






gringeriA_proj2 — -bash — 88×21
File "baseball.py", line 159, in <module>
main()
File "baseball.py", line 98, in main
n = int(input("Enter n, the number of games needed to win the series: "))
KeyboardInterrupt
~/Documents/CS2223/Projects/gringeriA_proj2 python3 baseball.py
This program computes the odds of winning a series of n games in the series.
To test both algorithms, enter 0.
To test just the recursive algorithm, enter 1.
To test just the dynamic programming algorithm, enter 2.
Enter a value: 1
Testing recursive algorithm...
Enter n, the number of games needed to win the series: 8
Enter p, the probability of winning a game: 0.38
Probability of winning series:
0.169
Time elapsed for recursive algorithm:
0.0112169450 seconds
~/Documents/CS2223/Projects/gringeriA_proj2


gringeriA_proj2 — -bash — 88×16
~/Documents/CS2223/Projects/gringeriA_proj2 python3 baseball.py
This program computes the odds of winning a series of n games in the series.
To test both algorithms, enter 0.
To test just the recursive algorithm, enter 1.
To test just the dynamic programming algorithm, enter 2.
Enter a value: 2
Testing dynamic programming algorithm...
Enter n, the number of games needed to win the series: 100
Enter p, the probability of winning a game: 0.48
Probability of winning series:
0.286
Time elapsed for dynamic programming algorithm:
0.0141222400 seconds
~/Documents/CS2223/Projects/gringeriA_proj2