Fabio Andre Camargo Ferraz

DATA 200

Prof. Yi Lu

May 9th, 2018

Gambling and Situation: Generalized Dice Game

Introduction:

Imagine a game where there are only two players and each player picks 5 numbers between 2 and 12. For example, one can pick the sequence "5, 6, 6, 9, 12". Now that both players have picked their combination of numbers, two dice are rolled and their results are summed. If the sum is 6, for instance, you can cross off one from your list, if it is in it. If the sum is a number you don't have on your list, you do nothing. Whoever crosses all numbers first wins.

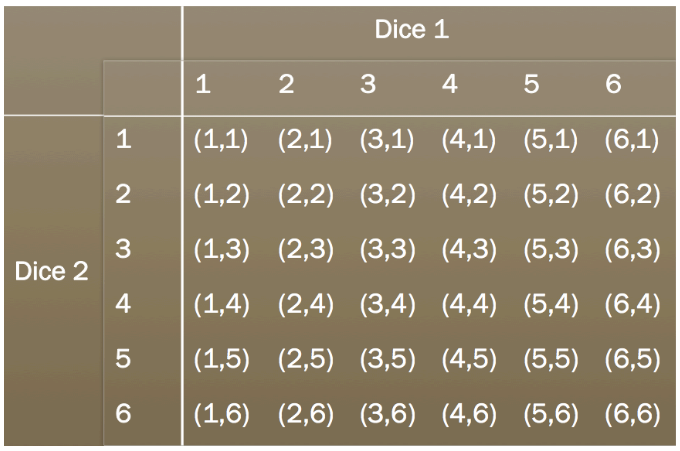
Now, the first thing I attempted to find was the best combination of 5 numbers I could pick to have the best chance at winning this game. With a very simple Python program that compared 2 combinations of numbers, I was easily able to determine the best combination. Therefore, because just 5 numbers were not enough for an interesting result, I decided to generalize the problem so that both players can pick any amount of numbers they want and instead of only using two dice, they can use as many as they'd like. Furthermore, I created a program on Python to find the optimal combination of numbers for this more generic version of the problem. However, I will still introduce the logic behind this program with the simpler version of the problem (of only 5 numbers and 2 dice).

The aspect of this problem that makes it interesting is the fact that we cannot just choose the 5 of the most likely values, because repetitions limit the possible combinations you can have between your numbers. That is, if you have five 7's, there is only one possible order you can get your numbers, while if you have 5 distinct numbers, you can have a lot more (5! = 120), which makes it a lot more likely to happen (at least in this case). So, the interesting part of the problem is figuring out when to use distinct numbers and when to repeat a more likely sum.

Another thing about the problem that is worth mentioning is the occurrence of ties. In both programs I wrote for this problem, I did not count ties as a win for neither of the players. This is because the way I compare combinations is by taking a ratio between them, and by adding a win for every tie the difference will always become more insignificant. Consequently, not counting ties as wins always gives a clearer difference between the two combinations of sums, which is very important for very big combinations of sums.

Main content:

To better understand the problem, it is very important to understand the probabilities that revolve around it. By this, I mean the probabilities of getting each of the sums. So, let us understand where they come from.



		Dice 1					
		1	2	3	4	5	6
Dice 2	1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
	2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
	3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
	4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
	5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
	6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

All the combinations you can possibly have with 2 6-sided dice

By looking at this table, we can get a better understanding of how likely each of the sums is to occur. This is because there are 36 possible combinations of different sums and each sum has a specific amount of possible combinations.

For example, if the sum of 2 only happens on (1,1), meaning there is only one combination that gets us our result out of the 36 possible combinations. Therefore, the probability of getting a 2 as a sum, which we can call P(2) is equal to $\frac{1}{36} = 2{,}78\%$.

Another example is 5, which has many other combinations:

(1,4), (2,3), (3,2) and (4,1). Therefore P(5) = $\frac{4}{36} = 11{,}11\%$.

From this, we get:

P(2) = 2.78%	P(6): 13.89%	P(10): 8.33%
P(3) = 5.55%	**P(7): 16.67%**	P(11): 5.55%
P(4) = 8.33%	P(8): 13.89%	P(12): 2.78%
P(5): 11.11%	P(9): 11.11%	

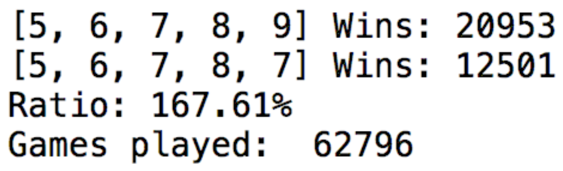
Now, with the probabilities of each of the sums in hand, we can come up with a first possible best option. For this, I suggest we pick the 5 different sums that simply have the highest probabilities of occurring.

For this problem (5 numbers, 2 dice), the obvious option is **[5, 6, 7, 8, 9]**, as they have the highest values.

Now, it obviously makes no sense to switch any of these numbers with a number that has a lower probability of happening (such as 10). Also, if we do want to switch a number, the logical one to remove is one with the lowest probabilities, that is, **5** and **9**. Also very clearly, the first best option to switch one of these numbers by is **7**, as it has the highest probability of occurring, which might be enough to overcome the problem of repeating values (and losing possibilities of combinations of results.

So, the first step is to compare the combinations **[5, 6, 7, 8, 9]** and **[5, 6, 7, 8, 7].**

Which gives a very favorable result in favor of **[5, 6, 7, 8, 9]:**



```
[5, 6, 7, 8, 9] Wins: 20953
[5, 6, 7, 8, 7] Wins: 12501
Ratio: 167.61%
Games played:  62796
```

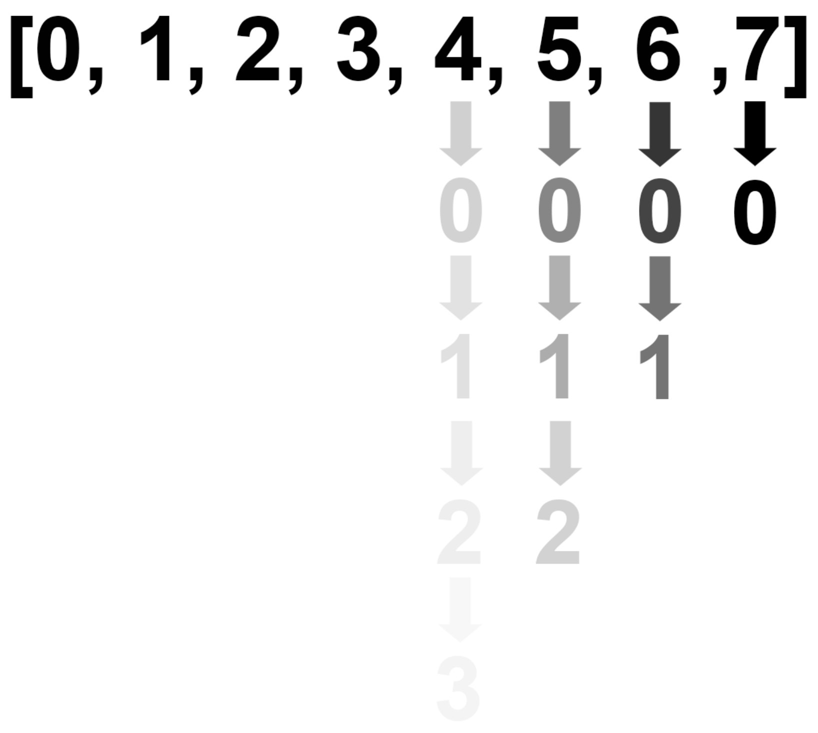
Output of results between [5, 6, 7, 8, 9] and [5, 6, 7, 8, 7].

Since the initial combination is better than the best other option, it is clearly the ideal combination for the problem. So, how can we apply this same logic for the more generic and involved problem?

For the more generic Python program, I made it so that the initial list of elements contained all the most individually probable sums in decreasing order of probability.
For example, **[5, 6, 7, 8, 9]** is written **[7, 8, 6, 9, 5].**

This way I can easily keep track of which element I am currently trying to substitute (simply the last element on the list which I have not yet found a better number).

The logic behind this process is simplified in the figure:



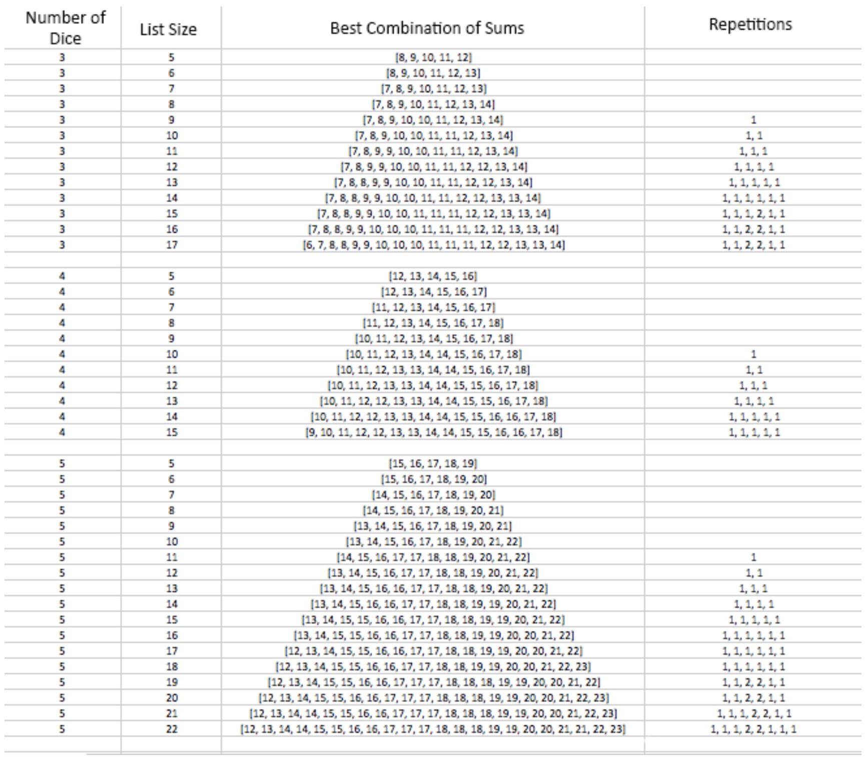
Where 0 will be the index of the sum with the highest probability and the lower the value, the higher the probability. So, we will first switch 7, the last (and worst) element, with 0, the best one. If 0 “loses”, that is, if the combination with 7 is better than the one with a second 0, then we can already finish and say **[0, 1, 2, 3, 4, 5, 6, 7]** is the best combination. This is because there is no other logical option to substitute 7. Since the element 0 is the most probable sum and there isn’t more 0’s than other elements in the list, so no other option would be better than either 7 or 0 for this case.

Now, if 0 “wins”, we will make the last element of the list equal to 0 and start testing the second to last, 6. For this, since we already have two 0’s in our list, we must start also test 1, as getting a “second 1” might be more beneficial than a “third 0” because of more repetitions, and also better than 6. So, we check to see if 0 is better than 6. If yes, switch 6 by 0 and check if 1 is better than that. This order follows the greyscale in the figure above from darker to lighter, until we get an element which wins against all others.

For example, if 4 wins against 0, 1, 2 and 3 above, that would give us the final best result with 4 in and whatever results were best for the last 3 elements, leaving the first ones unchanged.

This algorithm is definitely enough to find the best list to solve the problem. However, it would test too many unnecessary games. For example, if a "third 0" loses, we do not need to check for the possibility of a "third 1" winning, as 1 is less likely to happen than 0, so three 1's are less likely to happen than three 0's. Therefore, I implemented many methods to skip unnecessary checks, which drastically reduced the computational speed, especially for lists that are very large.

After ensuring the program was working perfectly and was consistently giving the right results, I recorded the ideal combinations for many variations of the problem so I could see how nicely the repetitions appeared.



Number of Dice	List Size	Best Combination of Sums	Repetitions
3	5	[8, 9, 10, 11, 12]	
3	6	[8, 9, 10, 11, 12, 13]	
3	7	[7, 8, 9, 10, 11, 12, 13]	
3	8	[7, 8, 9, 10, 11, 12, 13, 14]	
3	9	[7, 8, 9, 10, 10, 11, 12, 13, 14]	1
3	10	[7, 8, 9, 10, 10, 11, 11, 12, 13, 14]	1, 1
3	11	[7, 8, 9, 9, 10, 10, 11, 11, 12, 13, 14]	1, 1, 1
3	12	[7, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 14]	1, 1, 1, 1
3	13	[7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 14]	1, 1, 1, 1, 1
3	14	[7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13, 14]	1, 1, 1, 1, 1, 1
3	15	[7, 8, 8, 9, 9, 10, 10, 11, 11, 11, 12, 12, 13, 13, 14]	1, 1, 1, 2, 1, 1
3	16	[7, 8, 8, 9, 9, 10, 10, 10, 11, 11, 11, 12, 12, 13, 13, 14]	1, 1, 2, 2, 1, 1
3	17	[6, 7, 8, 8, 9, 9, 10, 10, 10, 11, 11, 11, 12, 12, 13, 13, 14]	1, 1, 2, 2, 1, 1
4	5	[12, 13, 14, 15, 16]	
4	6	[12, 13, 14, 15, 16, 17]	
4	7	[11, 12, 13, 14, 15, 16, 17]	
4	8	[11, 12, 13, 14, 15, 16, 17, 18]	
4	9	[10, 11, 12, 13, 14, 15, 16, 17, 18]	
4	10	[10, 11, 12, 13, 14, 14, 15, 16, 17, 18]	1
4	11	[10, 11, 12, 13, 13, 14, 14, 15, 16, 17, 18]	1, 1
4	12	[10, 11, 12, 13, 13, 14, 14, 15, 15, 16, 17, 18]	1, 1, 1
4	13	[10, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16, 17, 18]	1, 1, 1, 1
4	14	[10, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 18]	1, 1, 1, 1, 1
4	15	[9, 10, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 18]	1, 1, 1, 1, 1
5	5	[15, 16, 17, 18, 19]	
5	6	[15, 16, 17, 18, 19, 20]	
5	7	[14, 15, 16, 17, 18, 19, 20]	
5	8	[14, 15, 16, 17, 18, 19, 20, 21]	
5	9	[13, 14, 15, 16, 17, 18, 19, 20, 21]	
5	10	[13, 14, 15, 16, 17, 18, 19, 20, 21, 22]	
5	11	[14, 15, 16, 17, 17, 18, 18, 19, 20, 21, 22]	1
5	12	[13, 14, 15, 16, 17, 17, 18, 18, 19, 20, 21, 22]	1, 1
5	13	[13, 14, 15, 16, 16, 17, 17, 18, 18, 19, 20, 21, 22]	1, 1, 1
5	14	[13, 14, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 21, 22]	1, 1, 1, 1
5	15	[13, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 21, 22]	1, 1, 1, 1, 1
5	16	[13, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 22]	1, 1, 1, 1, 1, 1
5	17	[12, 13, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 22]	1, 1, 1, 1, 1, 1
5	18	[12, 13, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 22, 23]	1, 1, 1, 1, 1, 1
5	19	[12, 13, 14, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 22]	1, 1, 2, 2, 1, 1
5	20	[12, 13, 14, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 22, 23]	1, 1, 2, 2, 1, 1
5	21	[12, 13, 14, 14, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 22, 23]	1, 1, 1, 2, 2, 1, 1
5	22	[12, 13, 14, 14, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 21, 22, 23]	1, 1, 1, 2, 2, 1, 1, 1

It was very important to pay a close attention to the code to fix or improve any mistake, even though some versions of the code can take less than 5 second, as the longest completed search for an ideal combination of sums took 1.5h (see fig. 1). Furthermore, depending on your parameters, the game can be quite fun to pay attention to, seeing the numbers substitute themselves and playing the game a million times in so little time (see fig. 2).

Concluding remarks and future discussions

I won't explain the entire algorithm inside the python code for the generalized version of the problem, as it is very big, convoluted and the overall logic was more easily presented this way. However, I do have some things to mention about my code, which will be available and commented at the end of this paper.

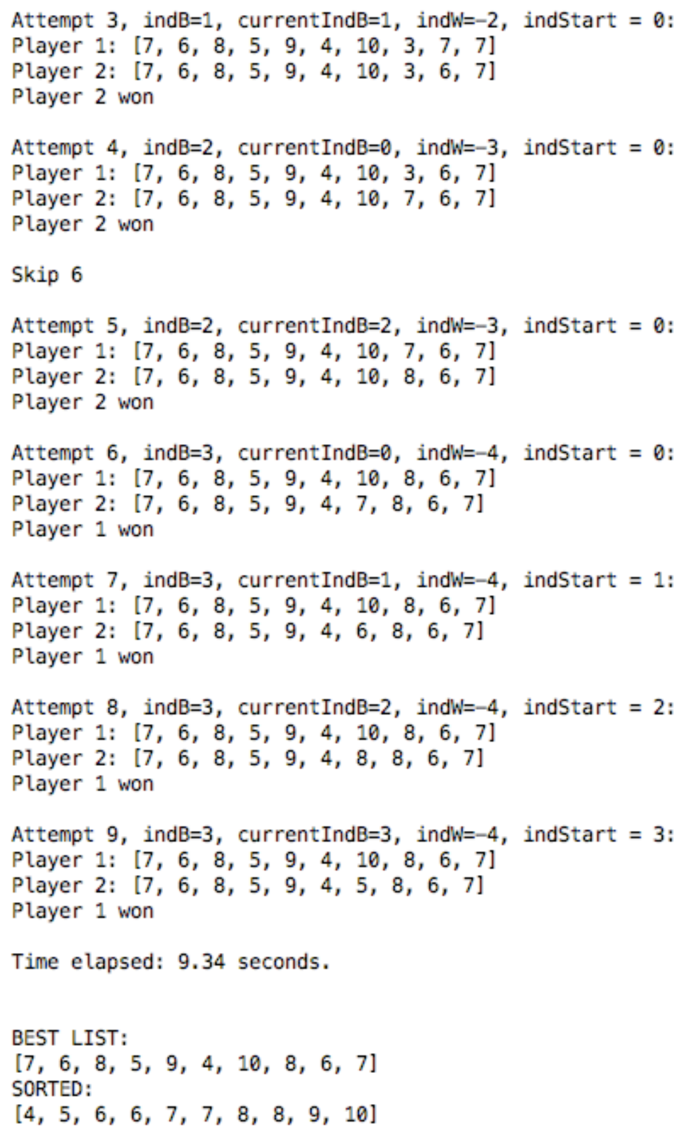
Firstly, my program only takes up to 5 dice. This is because even though I considered generalizing it completely, I feared Python would struggle with the syntax of the formulas and I felt it would just make it slower. Therefore, I "hard coded" the distribution of all the probabilities, which were available in a very convenient paper called *The Probability Distribution of the Sum of Several Dice*, where one can find the probability distribution of the sum of *k* 6-sieded dice. Furthermore, it would not be impossible to also create a function for the probability distribution of the sum of k **n**-sided dice, completely generalizing the initial problem. However, I still fear it would overcomplicate and slow down the program with not much usefulness. Furthermore, adding n-sided dice would not make a big difference, as the only things that would need to be changed in the code are the way to find probabilities (making it more general) and consequently where the middle of the list is (for the best element). Finally, I also feared if I went in this direction, my paper would be more about probability than data science.

Python Code for General Dice Game

```
import random
import operator
import time
def percent(a, b):# Simple function to take the 100*ratio of a & b
    if a > b:
        return round((100 * a/b), 2)
    if a < b:
        return round((100 * b/a), 2)
    if a == b:
        return 100
# Defining the function game() to determine the winner between two "players" with
the specified game conditions/parameters
def game(pl1, pl2, rollTimes, dieNum, dieVal, tieCount):
    player1 = pl1[:]
    player2 = pl2[:]
    p1 = player1[:]
    p2 = player2[:]
    p1Win = p2Win = 0

    p=0
    while p < rollTimes: # Loop until p reaches rollTimes
        sumDie = 0
        for n in range(dieNum):
            sumDie += random.randint(1, dieVal) # add a random value from 1 to
dieVal, dieNum times

        if (sumDie) in player1:
            player1.remove(sumDie)

        if (sumDie) in player2:
            player2.remove(sumDie)

        if not player1 and not player2:  # Tie
            player1 = p1[:]  # "Reset the board"
            player2 = p2[:]
            if tieCount:  # Only in case tieCount is on, we would add 1 win to each
player in case there is a tie
                p2Win += 1
                p1Win += 1

        elif not player1:  # If it was not a tie and player1 is empty, Player 1
won!
            player1 = p1[:]
            player2 = p2[:]  # "Reset the board"
            p1Win += 1       # Add 1 to Player 1 Wins


        elif not player2:  # Same thing for Player 2
            player1 = p1[:]
            player2 = p2[:]
            p2Win += 1
        p += 1   # +=1 to keep loop going
        if p1Win < 5 or p2Win < 5: # This is very useful for very big lists
            if p == rollTimes-1:   # Since they sometimes don't get cleared
                p=0                # so we can just keep the loop going for a while
            if p1Win > 200 or p2Win > 200:
                p=rollTimes

    if p1Win == 0:
        p1Win = 1
    elif p2Win == 0:
        p2Win = 1
    # Returning the result of the winner
    if percent(p1Win, p2Win)-100 <= 5:
        if p1Win > p2Win:
            return "too close Player 1"
        if p2Win > p1Win:
            return "too close Player 2"
    elif p1Win > p2Win:
        return "Player 1"
    elif p2Win > p1Win:
        return "Player 2"
    else:
        return str(percent(p1Win, p2Win))
# Start of program!

print("\nOnly using dice from 1-6")

dieVal = 6  # Number of sides on the dice
dieNum = int(input("Number of dice: "))
numNum = int(input("Number of numbers to pick: "))  #Number of picks
rollTimes = int(input("Roll each dice how many times? (Recommended = 200000, more
for bigger games): "))


maxSum = dieNum * dieVal   #The largest sum
numSums = maxSum - dieNum  #Number of possible sums
numComb = dieVal ** dieNum #Number of possible combination of numbers
print("\n")

dictProb = {}  # Create a dictionary to store numNum numbers and their respective
probabilites of occurring

nH = 0  #This number stands for the sum with the highest probability for each
different dieNum
if dieNum == 1:     #  I did not, however, divide them by numComb, as it is
unnecessary to do so now
    probList = [1, 1, 1, 1, 1, 1]  # List with the distribution of probabilities
before being devided by numComb
    nH = 3
elif dieNum == 2:
    probList = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]
    nH = 7
elif dieNum == 3:
    probList = [1, 3, 6, 10, 15, 21, 25, 27, 27, 25, 21, 15, 10, 6, 5, 3, 1]
    nH = 10
elif dieNum == 4:
    probList = [1, 4, 10, 20, 35, 56, 80, 104, 125, 140, 146, 140, 125, 104, 80,
56, 35, 20, 10, 4, 1]
    nH = 14
elif dieNum == 5:
    probList = [1, 5, 15, 35, 70, 126, 205, 305, 420, 540, 651, 735, 780, 780, 735,
651, 540, 420, 305, 205, 126, 70,
                35, 15, 5, 1]
    nH = 17
else:
    print("Error")

    # In case there are more numNum than numSum
elementsNeeded = 0  # we will add elementsNeeded number of the least likely element
to happen
for k in range(numNum):
    if nH not in list(range(dieNum, maxSum+1)):# Means we'e used all of nH on the
list, so we need to stop this loop
        elementsNeeded += 1 # So we can later add more elements in case numNum >
numSum
        break
    dictProb[nH] = probList[nH - dieNum] / numComb      # Adding the current nH to
the dictionary with its value
    nH += ((-1) ** k) * (k + 1)    # Makes nH equal to the next most likely element
(e.g. 7->8->6->9->5->10...)


sorted_dict = sorted(dictProb.items(), key=operator.itemgetter(1), reverse=True)
for i in sorted_dict:      #sorts the dictionary by probability and prints them in
order with their probabilities
    print(i)

currentList = []    # List with sums in decreasing order of probability that we
will use through the program to check values

for i in sorted_dict:   # Add just the Numbers in sorted_dict to currentList, so
they will be in decreasing order of probability. VERY important!!
    currentList.append(i[0])
for i in range(elementsNeeded): # In case there are elementsNeeded, add the lowest
probability number elementsNeeded times to the list
    currentList.append(currentList[-1])


print("\nCurrent List: ", end='')
print(currentList)
print("Sorted List: ")
print(sorted(currentList))
print()



otherOption = []
bestList = currentList[:]   # Create a new list to play around so we can keep
currentList with the values intact for future use
attempt = 0
indW = -1  # index of Worst: How far into the "worst" elements we'll switch by
better ones
indB = 0  # index of Best: How far into the "better" elements we'll test (The first
elements on the list)

indStart = 0  # Variable in charge of avoiding some unnecessary checks to speed the
program

best_found = False
startTime = time.time()  # Time keeping function. Slightly useless since rollTimes
is not constant, but fun to know it took 1.5h to find a list!

while not best_found:
    rollTimes2 = rollTimes
    currentIndB = 0+indStart       # This will allows us to skip unnecessary games!
    p1wins = 0
    while currentIndB < (indB+1):
        if currentIndB == ((len(bestList)+1)//2)-1:   # If for some reason we reach
the middle of the list, we better stop right there
            best_found = True
            print("Time to stop!!")
```

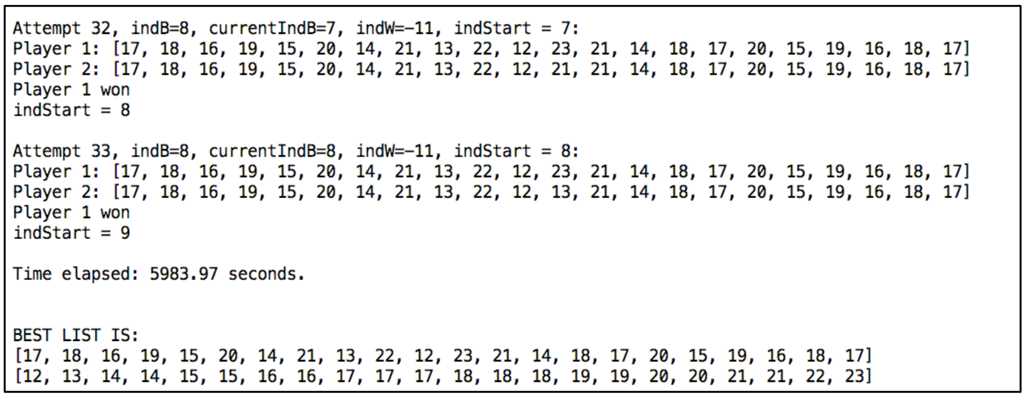
```
                break

            print()

            newList = bestList[:]
            newList[indW] = currentList[currentIndB]      # Switch current worse
element for next best possibility

            a = currentList[currentIndB]
            b = currentList[indStart]
            if newList.count(a) >= bestList.count(b)+indStart and a != b:   # This
skips some unnecessary games like testing a worse element when a better one just
failed
                currentIndB+=1
                print("Skip " + str(a))
                continue

            winner = game(bestList, newList, rollTimes2, dieNum, dieVal, False)  #Play
the game, defined above with our defined parameters
            attempt += 1

            print("Attempt " + str(attempt) + ", indB=" + str(indB) + ", currentIndB="
+ str(currentIndB) + ", indW=" + str(indW) + ", indStart = " + str(indStart) + ":
")
            print("Player 1: ", end='')
            print(bestList)
            print("Player 2: ", end='')
            print(newList)


            if (winner == "too close Player 1") or (winner == "too close Player 2"):
# If the game is too close, keep track of a possible second option
                otherOption = newList[:]
                print("Other option")
                print(sorted(otherOption))
                if rollTimes2 <= rollTimes * 16:     # Increase rollTimes to maybe get
a more accurate result
                    currentIndB -= 1    #Run the same game again in case it isn't as
close this time
                    rollTimes2 *= 4

                else:
                    rollTimes2 = rollTimes
                    if winner == "too close Player 1": # In case it really is close
three times in a row, whoever wins the third game, wins!
                        winner = "Player 1"
                    if winner == "too close Player 2":
                        winner = "Player 2"


            if winner == "Player 1":
                if sorted(bestList) == sorted(currentList):   # In case the first list
is already the ideal
                    best_found = True
                print("Player 1 won")
                if newList[indW] == currentList[indStart]:    # This basically makes it
so that we won't quit the loop in case the best element (first) does not work
                    indStart += 1                              # So that we can try the
next options, as the best options fail
                    p1wins += 1
                if p1wins == 2:             # If Player 1 wins 2 games in a row, it is
the best possible option, since we skip unnecessary checks
                    best_found = True

            elif winner == "Player 2":
                print("Player 2 won")
            bestList = newList[:]       # If Player 2 wins, make bestList the one
from Player 2.
            best_found = False
            if currentIndB == indB:     # If the index of the
                indB += 1
                break

        currentIndB += 1

    indW -= 1  # So that now, when we switch the "worst" element in the loop to see
if repeating any is better

print("\nTime elapsed: " + str(round(abs(time.time() - startTime),2)) + "
seconds.")

print()
print()
print("BEST LIST:")         #Print stuff
print(bestList)
print("SORTED: ")
print(sorted(bestList))

if len(otherOption) > 0:
    winner = game(bestList, otherOption, rollTimes*10, dieNum, dieVal, False)
    if winner != "Player 1": #So that if Player2 wins or if it is close, we will
print the close option
        print("\nOTHER OPTION:")
        print(sorted(otherOption))
```

Appendix:



```
Attempt 3, indB=1, currentIndB=1, indW=-2, indStart = 0:
Player 1: [7, 6, 8, 5, 9, 4, 10, 3, 7, 7]
Player 2: [7, 6, 8, 5, 9, 4, 10, 3, 6, 7]
Player 2 won

Attempt 4, indB=2, currentIndB=0, indW=-3, indStart = 0:
Player 1: [7, 6, 8, 5, 9, 4, 10, 3, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 10, 7, 6, 7]
Player 2 won

Skip 6

Attempt 5, indB=2, currentIndB=2, indW=-3, indStart = 0:
Player 1: [7, 6, 8, 5, 9, 4, 10, 7, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
Player 2 won

Attempt 6, indB=3, currentIndB=0, indW=-4, indStart = 0:
Player 1: [7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 7, 8, 6, 7]
Player 1 won

Attempt 7, indB=3, currentIndB=1, indW=-4, indStart = 1:
Player 1: [7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 6, 8, 6, 7]
Player 1 won

Attempt 8, indB=3, currentIndB=2, indW=-4, indStart = 2:
Player 1: [7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 8, 8, 6, 7]
Player 1 won

Attempt 9, indB=3, currentIndB=3, indW=-4, indStart = 3:
Player 1: [7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
Player 2: [7, 6, 8, 5, 9, 4, 5, 8, 6, 7]
Player 1 won

Time elapsed: 9.34 seconds.


BEST LIST:
[7, 6, 8, 5, 9, 4, 10, 8, 6, 7]
SORTED:
[4, 5, 6, 6, 7, 7, 8, 8, 9, 10]
```

Fig. 1. An output of the general version of the Python code. Here, you can clearly see how each element is being picked and better understand the algorithm.



```
Attempt 32, indB=8, currentIndB=7, indW=-11, indStart = 7:
Player 1: [17, 18, 16, 19, 15, 20, 14, 21, 13, 22, 12, 23, 21, 14, 18, 17, 20, 15, 19, 16, 18, 17]
Player 2: [17, 18, 16, 19, 15, 20, 14, 21, 13, 22, 12, 21, 21, 14, 18, 17, 20, 15, 19, 16, 18, 17]
Player 1 won
indStart = 8

Attempt 33, indB=8, currentIndB=8, indW=-11, indStart = 8:
Player 1: [17, 18, 16, 19, 15, 20, 14, 21, 13, 22, 12, 23, 21, 14, 18, 17, 20, 15, 19, 16, 18, 17]
Player 2: [17, 18, 16, 19, 15, 20, 14, 21, 13, 22, 12, 13, 21, 14, 18, 17, 20, 15, 19, 16, 18, 17]
Player 1 won
indStart = 9

Time elapsed: 5983.97 seconds.


BEST LIST IS:
[17, 18, 16, 19, 15, 20, 14, 21, 13, 22, 12, 23, 21, 14, 18, 17, 20, 15, 19, 16, 18, 17]
[12, 13, 14, 14, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 21, 22, 23]
```

Fig. 2. Another output, this one being from a relatively long list, resulting in 1.5h of computation time

Bibliography:

Singh, Ashok K., et al. “The Probability Distribution of the Sum of Several Dice: Slot Applications.” College of Hotel Administration, University of Nevada, Las Vegas, digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=1025&context=grrj.

“Data Structures.” 4. More Control Flow Tools - Python 3.6.5 Documentation, docs.python.org/3/tutorial/datastructures.html.