Report

Task Overview

Samsung and Apple are two tech giants and the biggest rivals of each other. Recently, Samsung has lost a lawsuit against Apple and must pay 1 Billion dollars to Apple. However, this amount will be payed in coins. There are unlimited coins in Samsung's vault and the values of coins are all primes, including 1 as well as a Gold Coin which is equivalent to the amount to be paid.

It is assumed that I am a principal software engineer at Samsung who is given a task to write a program that will calculate how many ways a given amount can be payed using a specific number of coins.

Algorithm Description

Input format

Input will come in as a file "input.txt", containing input in every line. Each line can represent three different variations of the input:

1. Single Integer - represents the money to be payed using **all possible combinations**
2. Two Integers - first integer represents money to be payed, second one the **number of coins used to pay the amount**
3. Three Integers – first integer represents money to be payed, and other two **the range of number of coins that can be used**

Output format

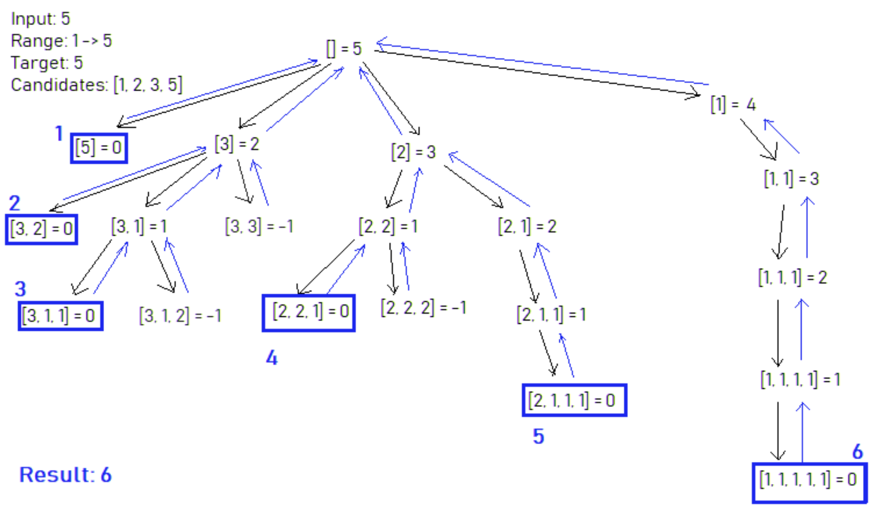
Output will come as a new file "output.txt" which will contain an integer in every line which will represent solution to the corresponding line from the input file.

Basic Idea

Algorithm used in this assignment will use depth first search to go through each combination of primes. Each node will have the values 'target' and an array 'current' which will represent current combination of primes. When target reaches 0, this means that the solution (combination) has been found and search will continue using backtracking. However, if the value of target is lower then 0, algorithm will only backtrack to the previous unexplored node and continue search from there.

Main function

Initially, code was written in Python to test the idea as the implementation is much easier. However, computational time took a very long time, so I decided to write the code in C to compute results in a more efficient manner. Implementation took slightly longer as C is low level language. Pseudo-Code for the C Language implementation is shown below.



Pseudo-Code

```
main():

    input_path ← path input by the user

    input_file ← open for reading using the input_path
    output_file ← open for writing

    while there are lines to be read from input_file (current_line ← line):
        parsedLine [] ← [0, 0, 0]
        1. parse the current_line into parsedLine []

        n, low_bound, high_bound ← parsedLine[0, 1, 2]

        if low_bound is 0:
            low_bound ← 1
            high_bound ← n

        if high_bound is 0:
            high_bound ← low_bound

        candidates[] ← empty array size n

        a ← 2. generateCandidates(n, array, candidates) number of primes up to n

        reverse order in candidates for optimization

        result ← 0
        3. combinations() call combinations function and pass result as pointer

        print result to output file

    close both files
    return
```

Clarification

Firstly, user will input path to the "input.txt" file. Algorithm then declares two file pointers, input_file (open using the user input path) and output_file which will be the file output is printed on. Now for every line in the file, declare parsedLine[] array as [0, 0, 0]. This way, when using parsedLine function, parsedLine[] array will, depending on if there is one integer in the line, parsed line will come out as:

case 1: [n, 0, 0]

case 2: [n, a, 0]

case 3: [n, a, b]

where first element is value n, second element is lower bound and third element high bound of the range of primes in the combination of sum.

As the combination() function takes in parameters lower bound and upper bound no matter what the case is, it is important to format these values such that function computes problem according to the task.

If second element (lower bound) is 0, then all combinations are to be found so range is between 1 and N primes inside combination. Therefore lower bound is 1 and high_bound is n.

Raw Input: 5 -> parsedLine = [5, 0, 0] -> n = 5, low_bound = 1, high_bound = 5

On the other side, if third element is 0, then input asks for combinations only in range of lower bound, meaning range is from lower bound to lower bound. Therefor high_bound takes the value of lower bound.

Raw Input: 8 3 -> parsedLine = [8, 3, 0] -> n = 8, low_bound = 3, high_bound = 3

Lastly, if none of the variables are 0, then n, lower bound and upper bound all have assigned values and nothing is to be formatted.

Raw Input: 6 2 5 -> parsedLine = [6, 2, 5] -> n = 6, low_bound = 2, high_bound = 5

Next, candidates() function will generate all primes up to the value n. When looking at the results, it was noticed that reversing candidates array optimized the running time of the algorithm. Afterwards, **result** variable is passed by address into the combinations() function. This function computes every solution using backtracking and DFS. For every solution found, actual value of **result** is incremented 1.

At the end of every line, print the result onto the "output.txt" file and move to the next line in the input file.

After every line is formatted, computed and result printed to the output file, it is important to close both input and output files to avoid any errors that might occur because of that.

Parsing the Input

This function takes in two arrays as a parameter, and formats them in a way described on a previous page. Inputs with one integer get formatted as [n, 0, 0], two integer inputs as [n, a, 0] and three integer inputs as [n, a, b].

Pseudo-Code

```
parseLine(line, parsedLine):

        index, temp, pIndex ← 0
        while True:

                if current character in line (line[index]) is '\n' or '\0' or ' ':

                        if its ' ':

                                parsedLine[pIndex] ← temp/10
                                temp ← 0 reset temporary value
                                pIndex++ go to next element in ParsedLine[]
                                index++ go to next character

                        else:

                                parsedLine[pIndex] ← temp/10
                                break from the loop

                temp += integer value of current character
                temp *= 10
                index++ go to next character
```

Clarification

This function loops through every character in a line, all until it reaches end of line character or end of string character, at which point it breaks the loop and returns function call. Assuming every input consists of integers and spaces, when reaching the space, we got the value of the number in the input.

For example, if input was 6 2 5. Program would compute this as such:
temp = 0 + int(6) = 6
temp *= 10 = 60
next character ' ' → parsedLine[0] = 60/10 = 6

Temp = 0, parsed line index++ → 1, index++ → next character

- ➔ Repeat for 2 and 5

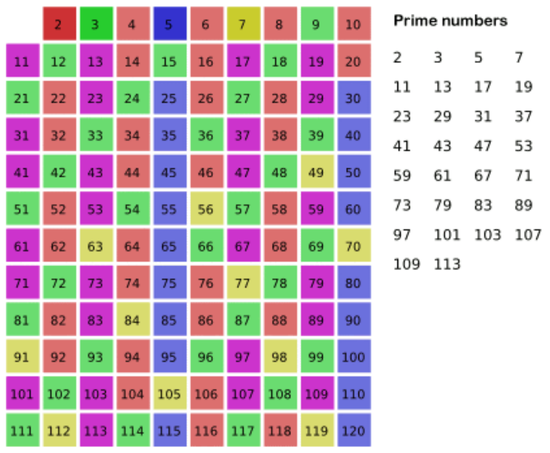
At the end of the call, parsedLine[] will have three elements [6, 2, 5].

Generate Candidates

Candidates list is a list of elements that will be used as candidates for a combination of sums. In this task, candidates are list of primes from 1 to n and n if n is not prime number. For example, if n was 6, candidates would be:

Candidates[6] = Primes[6] + [6] = [1, 2, 3, 5] + [6] = [1, 2, 3, 5, 6]

To find all primes up to N, I have used Sieve of Eratosthenes. What happens in this case is, that for every unmarked integer in a sequence (starting at 2), mark every leading multiple in the sequence. For every 2, mark 4, 6, 8, 10, 12 until n, next unmarked number is 3, so mark 3, 6, 9, 12, 15 until n, and follow these steps till there is no more multiples in the list of numbers from 1 to n. All the remaining (unmarked) numbers in a list are primes.



Pseudo-Code

```
generateCandidates(n, array, candidates = []):

    array[] ← [1, 2, 3, 4.... N]
    for number in range 2 -> N:
        for every following number in a sequence:
            if following number is a multiple of number:
                mark that number as 0
    a = 0 number of candidates

    for number in array:
            if number is n:
                bool in ← true check if n is prime
            if number not 0: append primes to candidates and increment a
                candidates.append(number), a++

    if in is false: candidates.append(n) if n not prime append it to the list
    return a edit candidates list as a parameter and return a as a value
```

Combinations Function

This problem can be described as a variation of a Combination Sum problem. Given a set of candidate numbers (in our case primes and n), and target number N, find all subset combinations of these candidates that sum up to target N. This problem is restricted by the lower and upper bound which represent number of elements in a combination. As previously described, I will approach this problem using backtracking, as it seems like most efficient way of doing it.

Function will take few parameters in:

1. Candidates[] → list of candidates
2. Target → value of n initially, target decreases as we approach solution
3. Index → index at the current candidate
4. Current[] → list representing current combination
5. Current_len → integer representing length of the current combination
6. Candidates_len → integer representing length of the candidates list
7. Combos → integer passed as reference, directly counting number of solutions
8. Low_b, up_b → lower and upper bound of the problem

The bound will work such that backtracking loop will take combinations in length of 0 up to the upper bound. When checking if the solution is reached (target is 0), if statement will check if length of combination is greater or equal to the lower bound. This will be clearer if looking at the pseudo code of the algorithm.

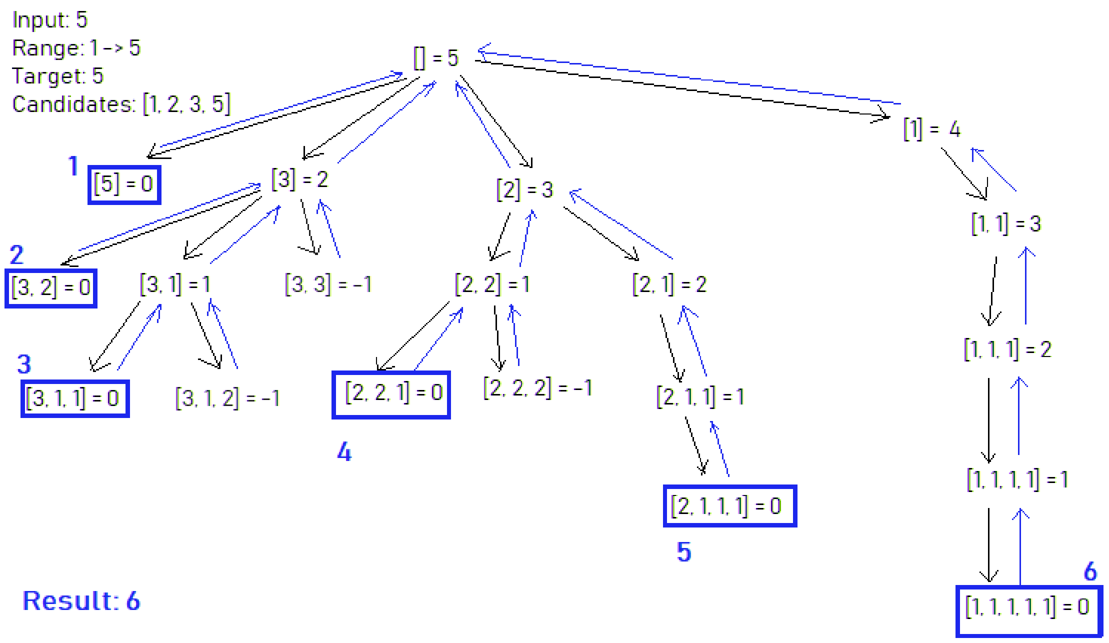
Pseudo-Code

```
combinations(parameters):
    if target <= 0:
        if target is 0 and lower bound <= current_len: length is valid
            combos++ increment solutions count

        return backtrack
    for i in range 0 -> upper bound:
        if current_len is i: check if current_len is in range 0 to upper bound

            if index < candidates_len:
                value ← next candidate
                current.append(value)

                combinations(parameters where target -= value)
                current.pop()
                combinations(parameters with index++ for next candidate)
    return
```

Clarification

Algorithm checks for every combination in range from 0 to upper bound to account for every value in the candidates list. This is one reason why it was important for candidates list to be inverted. If we started search from the first prime, which is 1, and upper bound was smaller then N, then it would be unnecessary to check for all combinations as even all 1s in that range wouldn't give you N. However, it is possible to have biggest candidate as a first element in the combination.

Now we will check if the 'index' value which points to the next candidate is smaller then length of the candidates. If it is, it means we do have candidate to append to the current combination list. After the value was appended, call the function again where target is now decreased by the value of the appended candidate.



This will loop recursively until every combination is checked. By comparing output on the next page with the diagram above, we can see clearly what is happening and how the algorithm is working.

Backtrack condition 'if target <= 0' will check if we reached combination that went over the target value or is solution to the problem. If target reached 0, and current length is greater or equal to the lower bound value, then we got the solution. In this case, combos (result) variable will be incremented, and stack returned.

Output on the left shows only two solutions, however process is the same when finding any solution.

```
Comb: []
Target: 5
Index: 0
Candidates: [5, 3, 2, 1]

 ||
 ||    add 5
\  /
 \/

Comb: [5]
Target: 0
Index: 0
Candidates: [5, 3, 2, 1]
Target is 0, Solution 1 found, backtrack.
```

← combination is [5] and target is 0, therefore we backtrack to where combination is []

```
 ||
 ||     pop 5
\  /
 \/

Comb: []
Target: 5
Index: 1
Candidates: [5, 3, 2, 1]
```

← increment index

```
 ||
 ||    add 3
\  /
 \/
```

← append the value at new index

```
Comb: [3]
Target: 2
Index: 1
Candidates: [5, 3, 2, 1]
```

← combination is [3] and target is 2, therefor algorithm evokes first call

```
 ||
 ||    add 3
\  /
 \/
```

← add the same value again

```
Comb: [3, 3]
Target: -1
Index: 1
Candidates: [5, 3, 2, 1]
No solution. Backtrack
```

← combination is [3, 3] and target is -1, no solution is found so we backtrack

```
 ||
 ||     pop 3
\  /
 \/

Comb: [3]
Target: 2
Index: 2
Candidates: [5, 3, 2, 1]
```

← remove last element in the combo and increment index

```
 ||
 ||    add 2
\  /
 \/
```

← now add the value at the next index [2]

```
Comb: [3, 2]
Target: 0
Index: 2
Candidates: [5, 3, 2, 1]
Target is 0, Solution 2 found, backtrack.
```

← combination is now [3, 2] and target is 0, therefor solution is found, and we backtrack again

Results

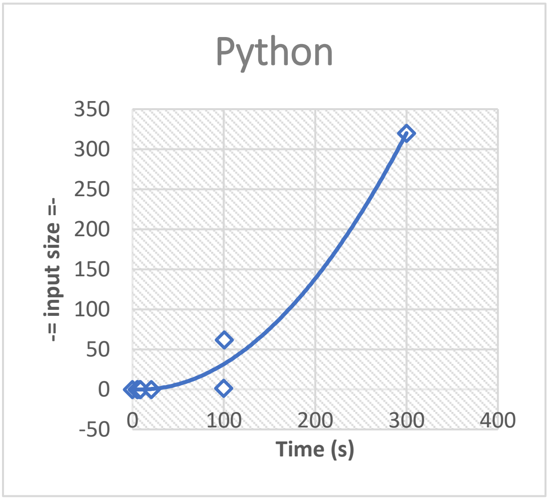
Algorithm was run on home laptop, LENOVO Ideapad 110. Algorithm was implemented in Python and C language and each algorithm was run 3 times. Table below shows time for each run and average for these results. Each run seemed to produce correct output.

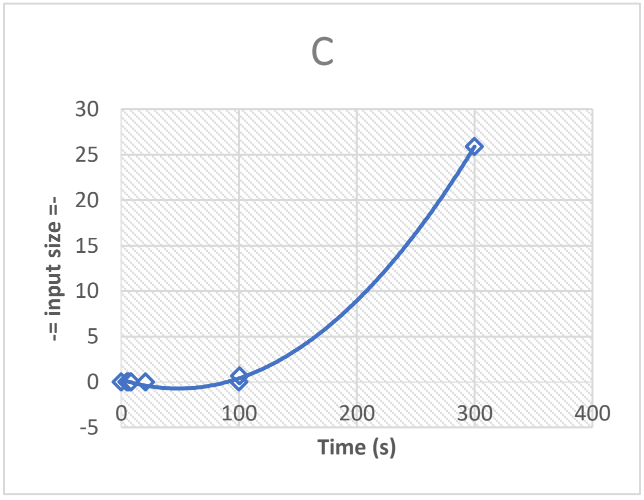
Input	C (runtime in s)				Python (runtime in s)				Output
	1st	2nd	3rd	Average	1st	2nd	3rd	Average	
5	0.000	0.000	0.000	0	0.000	0.000	0.000	0	6
6 2 5	0.000	0.000	0.000	0	0.000	0.000	0.000	0	7
6	0.000	0.000	0.000	0	0.000	0.000	0.000	0	9
8 3	0.000	0.000	0.000	0	0.000	0.000	0.000	0	2
8 2 5	0.000	0.000	0.000	0	0.000	0.000	0.000	0	10
20 10 15	0.000	0.000	0.000	0	0.007	0.017	0.012	0.012	57
100 5 10	0.022	0.023	0.019	0.021	1.901	1.622	1.301	1.608	14 839
100 8 25	0.675	0.653	0.712	0.68	56	62	68	62	278 083
300 12	25.83	26.32	25.52	25.89	312	298	351	320	4 307 252
300 10 15	123.3	119.5	120.7	121.2	2 981	2 501	3 010	2 830	32 100 326
Total	149.877	146.596	146.5	147.659	3 350	2 862	3 430	3 213	

From what is evident above, we can conclude that the runtime of C language is extremely faster than Python language. This is to be expected as C language is a lower level language. Additionally, all times seem to be consistent, however particular times may vary for longer computations (more evident for Python).

This is because some other programs were run while solution was computed. This is possible that during runtime in Python, 1 and 3, as computation took almost 10 minutes longer than the run 2.

Graphs below show how average input size affects runtime of the algorithm.





Performance Analysis

This performance analysis will be done on the C language implementation. Performance analysis of an algorithm depends on two factors, amount of memory used, and amount of compute time consumed on CPU. They are notified as complexities in terms of:

- Space complexity
- Time complexity

Space Complexity

Space Complexity is the amount of memory needed for algorithm to run to completion. From start of execution to its termination. There are two types of components in the algorithm, fixed components and variable components.

- Fixed components – components independent of inputs, instruction space, space for temporary variables, constants or constant size components
- Variable components – components that depend on size of the inputs or other instances, some examples can be recursion stack space, heaps, trees, graphs etc.

Therefor a total space of any algorithm with input n can be written as:

$$\text{Space}(n) = \text{Fixed}(n) + \text{Variable}(n)$$

Superwits Academy, P. (2019). Performance Analysis.
Retrieved from https://www.superwits.com/library/design-analysis-of-algorithm/course-content-daa/performanceanalysis

Table below will show all fixed and variable components in the algorithm. As we are computing upper bound of the space complexity Big-O, it is assumed that components will take X number of units of space.

Fixed Components					
Component	Space (units)		Component		Space(units)
Char Input_path[]	C1	100	Clock_t start	C8	1
FILE* input_file	C2	1	Int result	C9	1
FILE* output_file	C3	1	Clock_t end	C10	1
Int line	C4	1	double time_spent	C11	1
Char current_line	C5	20	Int n	C12	1
Int parsedLine[]	C6	3	Int low_bound	C13	1
int current[]	C7	100	Int high_bound	C14	1

Fixed components have constant cost:

$$Fixed(X) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} = C_{total}$$

$$\text{Fixed}(n) = O(1)$$

Next up is analysis of components that are affected by the size of the input n. These include arrays whose size depends on the input as well as a recursive stack of the combination function.

Variable Components		
Component	**Space (units)**	
Int array[]	n	First integer from input
Int candidates[]	n	First integer from input
Int new_candidates[]	a	Number of primes up to n upper bound -> n/(ln(n)-1)

For variable components above, we can conclude that:

$$Variable_1(n) = n + n + n/(\ln(n)-1)$$
$$= O(n) + O(n) + O(n/(\ln(n)-1))$$
$$= O(n)$$

Combinations function is called recursively which will result in a stack of some size. As backtracking and Depth First Search 'reach' for the lowest possible node, it is reasonable to assume that the biggest stack required for the computation is the upper bound of the input. To determine if this claim is true, test was run, and it shows relation between the biggest stack call and upper bound for every input.

Combinations Function			
input	Upper bound	Size of a stack	Upper bound = = stack size
5	5	5	True
6 2 5	5	5	True
6	6	6	True
8 3	3	3	True
8 2 5	5	5	True
20 10 15	15	15	True
100 5 10	10	10	True
100 8 25	25	25	True
300 12	12	12	True
300 10 15	15	15	True

As evident from the table above, claim is true, and we can conclude that the size of the stack is equal to the upper bound of the input. In the stack, space for variables is created on function beginning and destroyed upon termination.

Its space complexity is equal to the number of calls times the memory used per one call:

$$Variable_2(n) = upper\_bound(n) * space/call$$
$$space/call = O(\text{all the parameters})$$
$$= candidates[] + target + index + current[] + current\_len + candidates\_len + combos + low\_b + up\_b$$
$$= candidates[] + c + c + current[] + c + c + c + c + c + c$$
$$= O(n) + O(n) + O(1)$$
$$= O(n)$$

Therefor we can conclude that the space complexity of the algorithm is equal to the fixed values space complexity plus the space complexity for the variables that depend on the input size.

$$Space(n) = Variable_1(n) + Variable_2(n)$$
$$= O(n) + O(up\_bound(n)) = O(n) + O(up\_bound(n) * n)$$

worst case scenario can be where upper bound is n so:

$$up\_bound(n) = n$$

$$\Rightarrow Space\ Complexity = S(n) + S(up\_bound(n)*n) = O(n^2)$$

Where 'n' is amount to be payed and 'up_bound(n)' upper bound of the input

Time Complexity

Time complexity of the algorithm is the computational time of the algorithm. In other words, it is time computer needs to run the completion. The compile time is independent of the instance characteristics. Following factors affect time complexity:

- Characteristics of the compiler
- Computer machine

As the time is affected by the factors that are dependent on the machine, more theoretical approach is taken. Complexity can be calculated as the sum of all complexities in the algorithm, and biggest term is taken as a Big-O notation to represent upper bound of the time function. To find time that depends on the input, we can ignore steps with O(1) as that time is the same for every input.

Algorithm Breakdown

Algorithm can be broken down into steps:
Some steps were missed out as they do not depend on input size

1. Loop for every line:
    a. *Parse the information from the line*
    b. *Format input*
    c. *Generate candidates*
    d. *Reverse candidates*
    e. *Call combinations function*

What is input size?

As discussed previously in this report, input varies from 1 to 3 integers. As algorithm works in a way where it loops through all possible combinations till it reaches upper bound, or till sum of all elements in the combination is equal to N.

The worst possible scenario for input would be the input as one integer. In this case, boundaries would be 1 and N, meaning that all possible combinations of sums would have to be found.

Time complexity of this algorithm will be difficult to be analyzed. However, knowing that the algorithm uses DFS and backtracking, estimation for the worst-case scenario can be defined.

Firstly, let's assume that the array candidates has 'k' elements that are smaller or equal to the target 'n'.

1. Initial state
    a. We must call the function initially
2. First level
    a. Assume 'm' is value of the node
    b. At the first level, there are 'k' possible moves of 'm' values
        i. *for n = 5, where k = 4, there are k values:* ***m = [5, 3, 2, 1]***
3. Next levels
    a. Node will only branch out **if nodes target > leaf node's target value**
        i. *For n = 5, let m = 3, possible leaf nodes are 2 and 1*
        ii. *For next level, let m = 2, there are no possible leaf nodes, so algorithm backtracks*
        iii. *Branch until target is reached or overtaken*
        iv. *Branching factor* ***b*** *for further nodes is:*
            1. *b = 1 (self) + number of remaining candidates (r)*

If we could not repeat elements in the combination sum, we would have complexity of

$$ {}_nC_0 + {}_nC_1 + {}_nC_2 + \dots {}_nC_n = 2^n $$

where n is number of candidates (primes).

Example:

input 5,
n = 4 *[1, 2, 3, 5]*

${}_nC_0 = 1 \rightarrow$ []

${}_nC_1 = 4 \rightarrow$ [1], [2], [3], [5]

${}_nC_2 = 10 \rightarrow$ [1, 2], [1,3], [1,5], [2, 3], [2, 5], [3, 5]

${}_nC_3 = 4 \rightarrow$ [1, 2, 3], [1, 2, 5], [1, 3, 5], [2, 3, 5]

${}_nC_4 = 1 \rightarrow$ [1, 2, 3, 4]

Total = 1 + 4 + 10 + 4 + 1 = $16 = 2^4$

But now, if we assume that elements can be repeated, we can imagine candidate array for n = 5 being [1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 5] where each element is repeated as many times until it reaches value of n.

Let n be k. Now the new value of k will be sum of all primes repeated as many times as it take to get to k or just above k. This can be described in a sigma notation as:

$$\sum_{i=1}^{k} \mathbf{celi}(\mathbf{k}/\mathbf{i})$$

Where **celi**(x) returns x as rounded up number and i takes in value of primes only

Example:

Input = 6

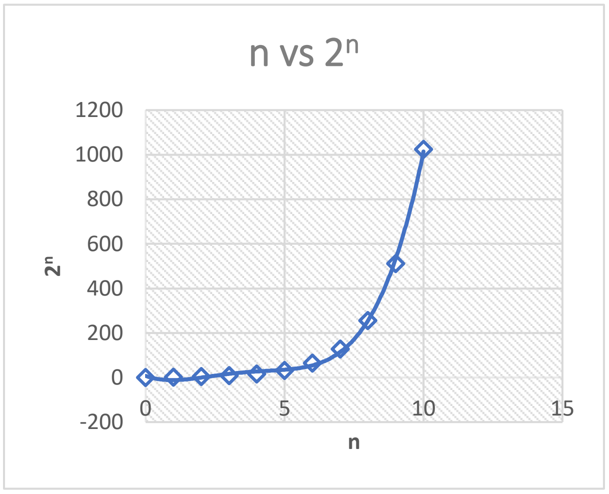
Candidates = [1, 2, 3, 5, 6]

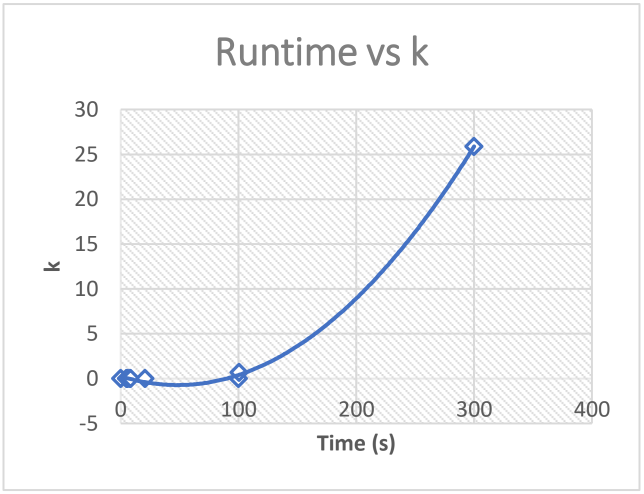
n = celi(6/1) + celi(6/2) + celi(6/3) + celi(6/5) + celi(6/6) = 6 + 3 + 2 + 2 + 1 = 14

Therefore, we can conclude that the time complexity in this algorithm is a combination sum problem of complexity 2^n where elements are repeated as many times as it takes them to reach target value of n or overtake it.

Time Complexity = O(2^n)

where n = $\sum_{i=1}^{k} \mathbf{celi}(\mathbf{k}/\mathbf{i})$





Evidently, two graphs do look similar in shape and it seems that the complexity O(2^n) acts like upper bound for the algorithm. Also, it is important to mention that other steps in the algorithm take Big-O value of O(1) or O(n). However, Big-O notation only requires the largest possible factor to be called as the complexity and unarguably 2^n will increase much quicker than the other ones.