1 Hello World!

Note 12

Determine the computability of the following tasks. If it's not computable, write a reduction or self-reference proof. If it is, write the program.

(a) You want to determine whether a program P on input x prints "Hello World!". Is there a computer program that can perform this task? Justify your answer.

(b) You want to determine whether a program P prints "Hello World!" before running the kth line in the program. Is there a computer program that can perform this task? Justify your answer.

(c) You want to determine whether a program P prints "Hello World!" in the first k steps of its execution. Is there a computer program that can perform this task? Justify your answer.

Solution:

(a) Uncomputable. We will reduce `TestHalt` to `PrintsHW`(P,x).

```
TestHalt(P, x):
  P'(x):
    run P(x) while suppressing print statements
    print("Hello World!")
  if PrintsHW(P', x):
    return true
  else:
    return false
```

If `PrintsHW` exists, `TestHalt` must also exist by this reduction. Since `TestHalt` cannot exist, `PrintsHW` cannot exist.

(b) Uncomputable. Reduce `PrintsHW`(P,x) from part (a) to this program `PrintsHWByK`(P,x,k).

```
PrintsHW(P, x):
  for i in range(len(P)):
    if PrintsHWByK(P, x, i):
      return true
  return false
```

(c) Computable. You can simply run the program until k steps are executed. If P has printed "Hello World!" by then, return true. Else, return false.

The reason that part (b) is uncomputable while part (c) is computable is that it's not possible to determine if we ever execute a specific line because this depends on the logic of the program, but the number of computer instructions can be counted.

2 Code Reachability

Note 12

Consider triplets (M, x, L) where

- `M` is a Java program
- `x` is some input
- `L` is an integer

and the question of: if we execute $M(x)$, do we ever hit line L?

Prove this problem is undecidable.

Solution: Suppose we had a procedure that could decide the above; call it `Reachable(M, x, L)`. Consider the following example of a program deciding whether $P(x)$ halts:

```
def Halt(P, x):
    def M(t):
        run P(x)   # line 1 of M
        return     # line 2 of M
    return Reachable(M, 0, 2)
```

Program M reaches line 2 if and only if $P(x)$ halted. Thus, we have implemented a solution to the halting problem — contradiction.

3 Strings

Note 10

What is the number of strings consisting of:

(a) n ones, and m zeroes?

(b) n_1 A's, n_2 B's and n_3 C's?

(c) $n_1, n_2, \ldots, n_k$ respectively of k different letters?

Solution:

(a) This is an $n+m$ length string. We choose n of those positions to be 1, and the rest will automatically be 0. Thus, the count is $\binom{n+m}{n}$. Another way of thinking about this is that there are $n+m$ positions, so we can consider $(n+m)!$ permutations. In this permutation, there are n ones, and the order of these ones doesn't actually matter. Every $n!$ way to order the ones is actually the exact same string, thus we divide by $n!$. Similarly, we divide by $m!$ to account for the zeros. Thus, we retrieve $\frac{(n+m)!}{n!m!}$.

(b) For this question, it is easier to consider the second method from the previous solution. There are $n_1+n_2+n_3$ positions, so we can consider $(n_1+n_2+n_3)!$ permutations. In this permutation, there are n_1 A's, and the order of these A's doesn't actually matter. Every $n_1!$ way to order the ones is actually the exact same string, thus we divide by $n_1!$. Similarly, we divide by $n_2!$ to account for the B's and also by $n_3!$ to account for the C's.

Alternatively, we could've used the counting positions strategy to approach this problem, though it is harder to generalize. We could consider an $n_1+n_2+n_3$ length string. First, we'll choose n_1 of those positions to be an A. Then, out of the n_2+n_3 positions left, we'll choose n_2 to be a B. Thus, the count becomes $\binom{n_1+n_2+n_3}{n_1}\binom{n_2+n_3}{n_2}$ which does evaluate to the same quantity.

(c) Using the same logic from the previous part, we generalize for a size k alphabet.

$$(n_1+n_2+\cdots+n_k)!/(n_1!\cdot n_2!\cdots n_k!)$$

.

4 You'll Never Count Alone

Note 10

(a) An anagram of LIVERPOOL is any re-ordering of the letters of LIVERPOOL, i.e., any string made up of the letters L, I, V, E, R, P, O, O, L in any order. For example, IVLERPOOL and POLIVOLRE are anagrams of LIVERPOOL but PIVEOLR and CHELSEA are not. The anagram does not have to be an English word.

How many different anagrams of LIVERPOOL are there?

(b) How many solutions does $y_0 + y_1 + \cdots + y_k = n$ have, if each y must be a non-negative integer?

(c) How many solutions does $y_0 + y_1 + \cdots + y_k = n$ have, if each y must be a positive integer?

Solution:

(a) In this 9 letter word, the letters L and O are each repeated 2 times while the other letters appear once. Hence, the number 9! overcounts the number of different anagrams by a factor of $2! \times 2!$ (one factor of 2! for the number of ways of permuting the 2 L's among themselves and another factor of 2! for the number of ways of permuting the 2 O's among themselves). Hence, there are $9!/(2!)^2$ different anagrams.

(b) $\binom{n+k}{k}$. We can imagine this as a sequence of n ones and k plus signs: y_0 is the number of ones before the first plus, y_1 is the number of ones between the first and second plus, etc. We can now count the number of sequences using the "balls and bins" method (also known as "stars and bars").

(c) $\binom{(n-(k+1))+k}{k} = \binom{n-1}{k}$. By subtracting 1 from all $k+1$ variables, and $k+1$ from the total required, we reduce it to problem with the same form as the previous problem. Once we have a solution to that we reverse the process, and adding 1 to all the non-negative variables gives us positive variables.

Alternatively, we can derive a method similar to stars and bars/balls and bins; here, the restriction to positive integers means that we cannot have any empty groups. In particular, instead of arranging all of the objects (i.e. all the stars and all the bars), we can instead choose where to place the bars.

Looking at the "gaps" between the stars (i.e. the 1's), we have a total of $n-1$ places to put the bars in between the n stars. Selecting k of these positions (we can't have two bars occupy the same gap, otherwise we'd have an empty group), we have a total of $\binom{n-1}{k}$ ways to group the 1's.