580012-F: Alex Taniguchi: 19 Dec 2016

Week 11: Strings in Ruby and Algorithmic Exercises; Monte Carlo Problem

**Exercise 1:** What is the computational complexity of the function “match”?

OK, so this time I’m basing this off the example of the slides from the homework in week 9. Not planning on getting this wrong twice, after all.

So first, we must consider how many iterations of the initial variable there is. In the example, this is the variable “i”. In this program we have the variable, w, which represents the # of iterations of array positions that we might use. As the exercise calls this variable, we will call “n” because it will be the length of the longer string “s”. It helps eliminate some fringe exceptions to know that p is a subset of s. This means that m<n or m<=n-1 (in the worst case m = n-1). The while string is counting through m using function submatch. “Submatch” is a ruby method defined by the variable j, which acts as an ascending count variable of array positions delta’d from position 0. P itself is referenced as a comparison variable. This moves one position at a time, in a linear line. Finally, we come to the variable p itself, where we must compare the submatch found to the number of letters matched.

Hence, the complexity is similar to the example in week 9. We see that it is made from the #(iterations of array positions in s) [=n] x #(iteration of submatch function) [=(n+1)-m] x #(operations for the while) [=O(m)] 🡪 simply a linear comparison until a success occurs, as method match is written, it doesn’t even have a failure-safe error message]

So, in short this becomes n \* (n+1)-m \* O (m). We then remember that constants do not count in computational complexity. So, in the worst case, we know that m = n-1. This means that I can replace the m in “(n+1) - m” as “(n+1)-(n-1)”. This becomes the constant 2 and is always a constant in any other case anyways, but as constants are irrelevant in the first place, we will remove this part from the final calculation.

This leaves me with #(iterations of array positions in s) \* #(operations for the while statement).

This is exactly n \* O (m) = O (mn).

Hence, the complexity will be O (mn).

**Exercise 2**: The reverse of a string without using the reverse function

def reverse(s)

result = "" # empty string(length 0)

i = s.length() – 1 #define variable I as one less than the length of the string

while i >= 0 do #while you have another string position

print s[i] #print the string variable there

i = i – 1 #take one step to the left, and as this in a while block, repeat

end #formally ‘ends’ the ‘while’ block

result # return the reversed string

end #ends all code ‘reverse’

**No. 11 Monte Carlo Exercise 1**: Make a Ruby function average (t, n) that computes averages when performing the function montecarlo (n) for t times.

#This code is written by Alex T.W.

def montecarlo(n)

m=0

for i in 1..n

x=rand() # random number in [0,1)

y=rand()

if x\*x + y\*y < 1.0

m = m + 1

end

end

return 4\*m\*1.0/n

end

def average(t,n)

sum = 0

for b in 0..t

sum += montecarlo(n)

end

x = sum / t

return x

end

#An alternate method I am ambandoning

# s = array.new(t)

# for b in 0..t

# s[b]=montecarlo(n)

# end

# v = 0

# for i in s[1]..s[t]

# v = v + i

# end

# v

# c = v / t

# c

#end

**No. 11 Monte Carlo Exercise 2**: Make a Ruby function average (t, n) that computes averages when performing the function montecarlo (n) for t times.

#This code is written by Alex T.W.

load("./montecarlo.rb")

def spherecarlo(n,r)

m=0

for i in 1..n

x=rand() # random number in [0,1)

y=rand()

if x\*x + y\*y < 1.0

m = m + 1

end

end

return (4.0/3.0)\*(4\*m\*1.0/n)\*(r\*\*3)

end

def savage(t,n,r)

sum = 0

for b in 0..t

sum += spherecarlo(n,r)

end

x = sum / t

return x

end