**Computational Thinking**

IS103

**Section**

G3

**Team Id**

21

**Members**

Pham Minh Khoa

Nguyen Nhat Quang

**Question 1**

# This code works under the following assumptions:

# 1/ Every column of pixel in the fill area must be continuous.

# That means if two to-be-filled vertically continuous set of pixels

# in the same column is separated by a vertically continuous set of

# not-to-be-filled pixels then that two set of pixels are treated as

# two columns of pixel in the fill area

# 2/ The area outside of the canvas/fill area (x>x\_max || x<0 || y>y\_max || y<0)

# always has different color compared to the color of the canvas

# 3/ Other given assumption from the question:

# a. original filling point is inside the canvas with height of y\_max and width of x\_max

# b. fill color is always different from original color

def fill\_area(x,y,fill\_color, ori\_color)

points = []

# nearest up point

points[0] = [x, y-1]

# nearest down point

points[1] = [x, y+1]

# nearest left point

points[2] = [x-1, y]

# nearest right point

points[3] = [x+1, y]

for i in 0..points.length-1

# get coordinate of the surrounding point

surround\_x = points[i][0]

surround\_y = points[i][1]

# get the color of the surrounding point with color() method

surround\_color = color(surround\_x, surround\_y)

# get the color of the current point

current\_color = color(x,y)

# compare if color of the surrounding point is different

# from that of current point

# if true, the surrounding point is a boundary point

# if false, call the recursive method

if surround\_color != current\_color

# check to fill current point

if current\_color == fill\_color

# do nothing

else

# fill the current's cell with the set\_color() method

set\_color(x,y,fill\_color)

end

else

fill\_area(surround\_x,surround\_y,fill\_color,ori\_color)

end

end

end

**Question 2**

(a)

Number of alphabetic letters (a-z, A-Z): 52

Number of numeric characters (0-9): 10

Number of special symbol: 28

Number of combinations for all passwords without arrangement: 52\*52\*10\*10\*28 = 7,571,200

Suppose that we pick the positions for 2 alphabetic characters 1st, there is 5C2 ways to choose. Hence, if we are to choose positions for 2 numeric characters next, there’re only 3 slots left and thus the number of ways is 3C2. Apply the same logic, if we are to pick position for the special character, there’s 1C1 or 1 way to do that.

Hence, the number of combinations for all passwords with arrangement will be:

7,571,200 \* 5C2 \* 3C2 \* 1 = 227,136,000

The result holds true regardless of the order in which we pick the character (e.g numeric 1st => special character 2nd => alphabetical 3rd)

Number of possible unique passwords: **227,136,000**

(bi)

Let R be the group of reviewers and S be the group of staffs. Since they must sit together, there’re only 2 ways to arrange the 2 groups: either RS or SR.

Within each group, the number of ways to arrange the 4 people is: 4!

Hence, the total number of ways to arrange 4 reviewers and 4 staffs so that they can sit together in a group is: 4!\*4!\*2 = **1,152**

(bii)

Let R represents a reviewer and \_ represent a staff. There’re only 2 ways to arrange reviewers and staffs so that they sit alternately: either R\_R\_R\_R\_ or \_R\_R\_R\_R

Within each group, the number of ways to arrange the 4 people is: 4!

Hence, the total number of ways to arrange 4 reviewers and 4 staffs so that they can sit alternately is: 4!\*4!\*2 = **1,152**

(ci)

We will apply the same logic as (bi) with the slight modification of arranging people within each group.

Since we need to pick 4 staffs from a pool of 10, the number of ways to pick them with arrangement is 10P4.

For reviewers, the number of ways to pick them with arrangement will thus be 20P4

Hence, the total number of ways to arrange 4 reviewers and 4 staffs so that they can sit together in a group is: **10P4\*20P4\*2**

(cii)

The numbers of ways to arrange reviewers and staffs are similar to (ci) and thus are 20P4 and 10P4 respectively. The number of ways to arrange them to sit alternately is similar to (bii) and thus is 2.

Hence, the total number of ways to arrange 4 reviewers and 4 staffs so that they can sit alternately is: **10P4\*20P4\*2**

**Question 3**

The binary search algorithm can be improved by

1/Using interpolation algorithm with the 1st two characters to determine the mid point.

If the first two characters of lower equal to first two character of mid of upper => break because it will either lead to constantly mid = lower or divided by 0 exception.

The reason to use interpolation is to estimate the position of the key more accurately for "fairly uniform data set". Execution of interpolation is as below:

# base 37 is used because we have 26 alphabetic letters,

# 1 special character (white space),

# and 10 numeric characters

temp1 = a[upper][0]\*37 - a[lower][0]\*37 + a[upper][1] - a[lower][1]

temp2 = k[0]\*37 + k[1] - a[lower][0]\*37 - a[lower][1]

# mid = 0

if (temp2!=0 && temp1!=0)

# Use interpolation search for the first two characters if the "lower" two characters

# isn't coincident with the "upper" first two characters or the "mid" first two characters

mid = lower + temp2 \* (upper - lower) / temp1

else

mid = (lower+upper)/2

end

2/ Bringing the equality check inside the while loop out:

While it’s true that the performance of the best case scenario is better if we place the equality check (i.e return mid if k == a[mid]) inside the while loop, the possibility of getting the mid point value exactly identical to the key being searched in the early loops is pretty low.

Hence, a way to improve the average scenario and worst case is to reduce the number of equality check that the while loop performs. We do that by modifying the exit condition of the while loop and moving the equality check outside of the loop as below:

while **lower + 1 < upper**

#Interpolation codes here

if k < a[mid]

upper = mid

else

lower = mid

end

end

**if k != a[lower]**

**return nil**

**else**

**return lower**

**end**

end

Full implementation:

def bsearch(a, k)

lower = 0

upper = a.length-1

while lower + 1 < upper

# base 37 is used because we have 26 alphabetic letters,

# 1 special character (white space),

# and 10 numeric characters

temp1 = a[upper][0]\*37 - a[lower][0]\*37 + a[upper][1] - a[lower][1]

temp2 = k[0]\*37 + k[1] - a[lower][0]\*37 - a[lower][1]

# mid = 0

if (temp2!=0 && temp1!=0)

# Use interpolation search for the first two characters if the "lower" two characters

# isn't coincident with the "upper" first two characters or the "mid" first two characters

mid = lower + temp2 \* (upper - lower) / temp1

else

mid = (lower+upper)/2

end

if k < a[mid]

upper = mid

else

lower = mid

end

end

if k != a[lower]

return nil

else

return lower

end

end

\*Special note: The q3.rb version submitted on server doesn’t contain interpolation. The implementation above runs well locally but generates a “execution time out” error on the server. The code on server is as follow:

def bsearch(a, k)

lower = -1

upper = a.length

while lower + 1 < upper

mid = (lower + upper)/2

if k < a[mid]

upper = mid

else

lower = mid

end

end

if k != a[lower]

return nil

else

return lower

end

end

**Question 4**

The binary search algorithm can be improved by:

1/ Cut off to insection sort in the subarrays of <= 10 elements: For small subarrays, calling recursions repeatedly will create a comparatively substantial overhead cost.Hence, cutting off to insection sort for small subarrays will drastically reduce the space needed, which lead to performance enhancement. Empirical method shown that CUT\_OFF=10 is an optimal cutoff threshold

2/ Shuffling the original array: This will increase stability of the algorithm. The original algorithm has complexity O(n^2) for worst case scenario. Shuffling the given array have to reduce the probability of getting the “bad” scenarios, which undoubtedly help to increase the stability of the timing.

3/ Improve the order of recursion: For each recursion in the original method qsort(a,lower,upper) it will called two of its “child” ( qsort(a,lower,mid) and qsort(a,mid+1,upper)). If the mid value always “skew” to the right side the number of elements in the stack which is used by the system to store the order of recursion will be drastically overstocked.

Imagine it as a “binary tree” of recursion order in which the left-side nodes will always be called before the right-side nodes, it makes sense to visit the smallest sub-tree first and so forth in order to minimize the number of elements stored inside the stack at one times.

Our implementation use the “order” array to store that stack. We also implement auto-expansion for that array (double the size when the number of elements is about to exceed) in order not have to expand the array size in every steps, hence reduce the number of steps.

Full implementation:

def qsort(a, lower, upper)

order = Array.new(2)

order[0] = lower

order[1] = upper

count = 1

while (order.any?)

r = order[count]

l = order[count-1]

if (r <= l+9)

if (r>l) then isort(a,l,r) end

order[count] = nil

order[count-1] = nil

count-=2

next

end

mid = partition(a, l, r)

if (count+3>order.size)

temp = Array.new(order.size\*2)

for i in 0..(count-2)

temp[i] = order[i]

end

order = temp

end

if (mid-l > r-mid)

order[count-1]=l

order[count]=mid-1

order[count+1]=mid+1

order[count+2]=r

else

order[count-1]=mid+1

order[count]=r

order[count+1]=l

order[count+2]=mid-1

end

count += 2

end

return a

end

def partition(a, p, r)

x = a[p]

i = p

j = r + 1

while true

loop { j = j - 1; break if a[j] <= x || j == p }

loop { i = i + 1; break if a[i] > x || i == r }

if i < j

a[i], a[j] = a[j], a[i] # perform swap

else

a[p], a[j] = a[j], a[p] # perform swap with partitioning item

return j

end

end

end

def isort(a,lower,upper)

for i in lower+1..upper

key = a[i]

j = i

while j > 0 and a[j - 1] > key

a[j] = a[j - 1]

j = j - 1

end

a[j] = key

end

end