|  |  |
| --- | --- |
| Programming Project 1: Analyzing the Performance of Quick Sorts with 3 methods | |
| 2012-12751, 41 Sim Gyumin | |
|  | |
| Abstract | In this paper, we analyze the three Quick Sort which differs in detail methods. (normal, with median 3, with median 3 and insertion sort) It is analyzed by theoretical time complexity analysis, and actual program considers the best case, worst case, average case and standard deviation of cases.  In the result of an analysis, we could found that Quick Sort with M3 or Quick Sort with M3 and Insertion Sort is better in performance than Normal Quick Sort. Also, we would check the advantage and disadvantage of both Quick Sorts with M3 and Quick Sort with M3 and Insertion Sort. |
| Contents | I. Introduction to Project 1  II. Analysis of Quick Sort Algorithm  1. Description  2. Theoretical Analysis of Time Complexity  3. Programming Approach and Result  4. Comparison of Result with Theoretical Values  III. Analysis of Quick Sort with Median of 3 Algorithm  1. Description  2. Theoretical Analysis of Time Complexity  3. Programming Approach and Result  4. Comparison of Result with Theoretical Values  IV. Analysis of Quick Sort with Median of 3 and Insertion added Algorithm  1. Description  2. Theoretical Analysis of Time Complexity  3. Programming Approach and Result  4. Comparison of Result with Theoretical Values  V. Conclusion |
|  | |

I. Introduction to Project 1

In this Programming Project, our first goal is to analyze Quick Sort Algorithm theoretically. And the second goal is to code Quick Sort Algorithm with 3 methods and compare it with the theoretical value.

I used Python 3.6 to code algorithms. Thus any pseudo-code and the actual code will be written in Python expression. And here are some major hypotheses with my programming works:

a. I iterated each algorithm programs for 1000 times, and find the best case, worst case, average case in 1000 times of results.

The way I iterate each program is to make another function which needs target function, array, starting number and last number to input. Named ‘Statistic\_iter’, this function iterates input function 1000 times, append every operation time in an array, and find maximum/minimum/average value of an array. Description of ‘Statistic\_iter’ function is:

**Problem)** Given any function, iteration number, and input values of given function, iterate input function for iteration-number times. If given function didn’t get the proper result(sorted array), then raise an error. If given function did, analyze maximum/minimum/average value in iterated results.

**Inputs)**

func: given function

iter\_num: iteration number

l, u: inputs of given function. We hypothesize that array is shuffled each time we iterate given function; thus we didn't give specific array for the input value. We would run another function ‘get\_unordered' each time we iterate given function and take the result array to the input.

**Outputs)** tuple of values, which includes:

min\_op: minimum operation number, which is B(n)

average\_op: average operation number, which is A(n)

max\_op: maximum operation number, which is W(n)

op\_list: number of operations in each iteration of given function. For checking # of comparison.

*\* I will not write pseudo-code of either ‘Statistic\_iter’ or ‘get\_unordered’. It is the matter of hypothesis, and I thought pseudo-code of these function is not necessary for given assignment.*

b. In each iteration, I use a different unordered array for sorting. Named ‘get\_unordered’, this function shuffles an unordered array and take this array for output.

c. I take the length of an array for 128. It is sufficiently large to get the result of a program, and 128 was easy to calculate log of 2.

d. My method of checking algorithm is to test it for a smaller size of an array, or to test it for an ordered array, and else. But this debugging method isn’t written in source code.

e. I could use the value n-1 for each Partition(named ‘part’) operation number, but rather I choose to count every comparison in Partition Subalgorithm and Quick Sort Algorithm. This is because it is useful for debugging, and it is semantically rational.

f. I used a tuple of (sorted array, operation number) for output instead of using an only sorted array. Because in that way I can calculate # of comparisons better way.

With this hypothesis, I analyze these Quick Sort Algorithms with 3 methods.

II. Analysis of Quick Sort Algorithm

1. Description

**Problem)** Given an unordered array of S with n elements, rearrange the elements of S so that S[1]<= S[2]<= S[3]<= … <= S[n] holds.

**Inputs)**

S: given unordered array

l: least index of an array

u: upper index of an array

**Outputs)**

S: rearranged array

**Pseudo-code)**

QuickS\_1(S, l, u)

if l<u:

p= Partition(S, l, u)

QuickS\_1(S, l, p-1)

QuickS\_1(S, p+1, u)

return S

For understand this algorithm, we should also describe and analyze Partition Subalgorithm. Here is a description of Partition:

**Problem)** Given an array S with lth to uth index, partition S so that for one element x, every element that is smaller than x be rearranged in S[1: p-1], and the elements that are bigger than x be rearranged in S[p+1:u].

**Inputs)**

S: given unordered array

l: least index of an array that we intend to rearrange

u: upper index of an array that we intend to rearrange

(Thus l, u could be different from actual length of S. Also we could rearrange a smaller part of S.)

**Outputs)**

p: pivot index, that is, the index of x

(We don’t take Partitioned array S as output since we didn’t need it as a return value.)

**Pseudo-code)**

def part(S, l, u): # for convenience, I named it ‘part’

pivot\_val= S[l]

i= l+1

j= u

while i<=j:

while i<=u and S[i] <= pivot\_val:

i+= 1

while i<=j and S[j] > pivot\_val:

j-= 1

if i<j:

S[i], S[j]= S[j], S[i] #Possible in Python!

i+= 1

j-= 1

S[l], S[j]= S[j], S[l]

return j

2. Theoretical Analysis of Time Complexity

**Best Case)** In Quick Sort, the best case happens when pivot element of partition subalgorithm divides the array into two subarrays of the same length for every partition. Then one Quick Sort Algorithm of the nth array makes two n/2th subarrays, with the number of comparison operations for the partition.

In Partition Subalgorithm, we take the first element for pivot element and make a comparison of pivot element with all of other elements. Thus # of comparison operations are same with (length of array – 1). For the nth array, Partition Subalgorithm take n-1 comparison operations.

Therefore, time complexity of Quick Sort Algorithm comes with this recurrence relation:

Take k s.t holds, then . Since

(we didn’t consider floor function or ceiling function, in here and any other recurrence relations.)

**Worst Case)** In Quick Sort, the worst case happens when pivot element of partition subalgorithm couldn't divide given array for two subarrays. That is, if a value of pivot element is smallest or biggest at every partition algorithm running, it gives worst case. Then one Quick Sort Algorithm of the nth array only makes one n-1th subarray, with the number of comparison operations for partition. And Partition Subalgorithm of the nth array needs n-1 comparison operations, as same with the best case.

Therefore, time complexity of Quick Sort Algorithm comes with this recurrence relation:

Take k s.t holds, then and .

3. Programming Approach and Result

**Problem and Approach)** Problem is same with ‘Description of Quick Sort Algorithm’. I tried to code this algorithm similar with pseudo-code as possible as I can. The difference of actual code with pseudo-code is that I added operation counter variable ‘op’, and ‘sigma\_operation'. ‘op' works as a counter for each partition, and I added these op value at global sigma\_operation variable to counter all of the comparison operations.

(And also, I used ‘Statistic\_iter' function and ‘get\_unordered' function to iterate it. I would not write this hypothesis anymore in this paper.)

**Inputs)** Same with a description of the algorithm.

**Outputs)**

S: rearranged array

(sigma\_operation: number of comparisons to run the algorithm)

**Actual Code)**

QuickS\_1(S, l, u)

global sigma\_operation

#to add every operation number of recursive algorithms.

if l<u:

(p, op) = part(S, l, u) #op is number of operation done in partition.

sigma\_operation += op

QuickS\_1(S, l, p-1)

QuickS\_1(S, p+1, u)

return (S, sigma\_operation)

And here is a description of Partition:

**Problem & Inputs)** Same with a description of the algorithm.

**Outputs)**

p: pivot index, that is, the index of x

(op: number of comparisons to run the subalgorithm)

**Actual Code of Partition Subalgorithm)**

def part(S, l, u):

pivot\_val= S[l]

i= l+1

j= u

op= 0

while i<=j:

while i<=u and S[i] <= pivot\_val:

i+= 1

op+= 1

if i<=u: op+= 1

# if while loop ended because S[i]<= pivot\_val, then it also means that computer had a comparison, thus we add 1 operation number.

while i<=j and S[j] > pivot\_val:

j-= 1

op+= 1

if i<=j: op+= 1

if i<j:

S[i], S[j]= S[j], S[i]

i+= 1

j-= 1

S[l], S[j]= S[j], S[l]

return (j, op)

**Results)** For this code, I tested this algorithm for 3 ways: first is to input ordered array for worst case, second is to input an array [7,0,2,1,5,4,6,3,11,8,10,9,13,12,14,15], which will give us the best case. And third is to iterate this algorithm for 1000 times with random unordered arrays. The result of this test is:

Worst case: QuickS\_1([0,1,2, … ,127], 0, 127): # of comparisons is 8255 (Python gives index starts with 0)

Best case: QuickS\_1([7,0,2,1,5,4,6,3,11,8,10,9,13,12,14,15], 0, 15): # of comparisons is 46

Iteration of algorithm: Statistic\_iter(QuickS\_1, 1000, 0, 127): minimum # of comparisons is 775, average # of comparisons is 942.19, maximum # of comparisons is 1312, standard deviation of # of comparisons is 5558.3419

4. Comparison of Result with Theoretical Values

For Theoretical Analysis, W(128)= 127\*128/2= 8128, B(128)= 128\*7-128+1= 769 and B(16)= 16\*4-16+1= 49.

The reason that the error occurred is:

a. In actual code, each partition does one more operation. For example, consider the case part([0,1,2, … ,127], 0, 127). Computer takes 0 the pivot element and does the first comparison pivot\_val < 1. Since 1 is bigger than pivot value, computer operates the next loop, then do the comparisons pivot\_val < 127, pivot\_val < 126, … to pivot\_val < 1.

In this comparisons, it is impossible to avoid this duplication of comparison between pivot\_val and 1. Thus for every Partition Subalgorithm part([0,1,2, … ,127], 0, 127), part([1,2,3, …,127], 0, 126), … part([126, 127], 0, 1). Thus actual operation of worst case gives # of comparison 8128(theoretical # of comparisons) + 127 = 8255.

b. Theoretical analysis often gives bigger value than actual # of comparisons because we omitted the existence of floor function.

Take the example of [7,0,2,1,5,4,6,3,11,8,10,9,13,12,14,15];

[**7**,0,2,1,5,4,6,3,11,8,10,9,13,12,14,15] # 16 of comparisons

[**3**,0,2,1,5,4,6,7,**11**,8,10,9,13,12,14,15] # 7+8 of comparisons

[1,0,2,3,5,4,6,7,9,8,10,11,13,12,14,15] # 3+3+3+4 of comparisons

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] # 1 of comparisons

Thus it might take the best type of partition for 3 time. Therefore, it would be rational if , the number of recursion in theoretical analysis is also 3. But in theoretical analysis, k= 4, thus it gives us the bigger value 49. (also the error above in a also occurred)

Reason that theoretical anlaysis misunderstood such recursion number is that, we omitted the floor function by to . Actual length of array the algorithm partitioned is 16, to 7 and 8, to 3, 3, 3 and 4, to 2.

But in this omitted recurrence relation, an anticipation of the length of an array is 16, to 8 and 8, to 4, 4, 4, 4, to eights of 2.

Although these two errors occurred, these errors are independent with the order of functions. We would ignore such errors next time, about .

With regard to these errors, the minimum, maximum and average value of comparisons seems rational within B(128) and W(128). It is shown that average value of comparisons is quite biased in B(128), but it also is rational since the average number of operations are known to follow an exponential decay curve.

III. Analysis of Quick Sort with Median of 3 Algorithm

1. Description

**Problem, Inputs, Outputs)** Problem, Inputs, and Outputs of Quick Sort with Median of 3 Algorithm is same with Quick Sort above. I’ll skip these in here.

**Pseudo-code)**

def QuickS\_M3(S, l, u):

m= int((l+u)/2)

if l<u:

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

if S[m]>S[u]:

S[m], S[u]= S[u], S[m]

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

S[l+1], S[m]= S[m], S[l+1]

p = Partition(S,l+1,u-1)

QuickS\_M3(S,l,p-1)

QuickS\_M3(S,p+1,u)

return S

It also needs Partition Subalgorithm, but it is same with Partition Subalgorithm written above. I’ll skip it in here.

2. Theoretical Analysis of Time complexity

**Best Case)** In Quick Sort with Median of 3, Best case happens when the pivot element of partition subalgorithm divides the array into two subarrays of the same length for every partition. Then one Quick Sort M3 Algorithm of an nth array makes two n/2th subarrays.

In Partition Subalgorithm, we find the median value for taking pivot element in the least index element, upper index element, and m= ┌((l+u)/2)┐th element. It takes at least 2 operations, and at most 3 operations. Then this algorithm makes a comparison of pivot element with all of other elements except it already compared in taking pivot element. In this work, # of comparison operations is (length of array – 3). Then for the nth array, Partition Subalgorithm take at least n-1 comparison operations, and at most n comparison operations.

Therefore, time complexity of Quick Sort Algorithm comes with this recurrence relation:

Take k s.t holds, then . Since

**Worst Case)** In Quick Sort M3, the worst case happens when pivot element of partition subalgorithm couldn’t divide given array for two subarrays. That is, if the value of pivot element is smallest or biggest at every partition algorithm running, it gives worst case. When we partition nth array, partition algorithm states pivot element at S[1], and state one element in S[0], one element in S[n]. Suppose the worst case then this pivot element couldn't divide the elements to the right of pivot element the one partition only makes one n-2th subarray.

And as we mentioned in the best case, the worst case of Partition Subalgorithm of the nth array needs n comparison operations.

Therefore, time complexity of Quick Sort M3 Algorithm comes with this recurrence relation:

Take k s.t holds, then and .

3. Programming Approach and Result

**Problem and Approach, Inputs and Outputs)** Problem and Inputs are same with Quick Sort M3 description. And as same with Quick Sort Programming Approach, I added operation counter variable ‘op’ and ‘sigma\_operation’. I also tried to code this algorithm similar with pseudo-code as possible as I can.

**Actual Code)**

def QuickS\_M3(S, l, u):

global sigma\_operation

m= int((l+u)/2)

if l<u:

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

sigma\_operation+= 1

if S[m]>S[u]:

S[m], S[u]= S[u], S[m]

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

sigma\_operation+= 1

sigma\_operation+= 1

S[l+1], S[m]= S[m], S[l+1]

(p, op)= part(S,l+1,u-1)

sigma\_operation+= op

QuickS\_M3(S,l,p-1)

QuickS\_M3(S,p+1,u)

return (S, sigma\_operation)

**Results)** For this code, I tested this algorithm for 3 ways: first is to input an array [2,12,4,10,6,14,8,1,3,5,7,9,11,13,15,0] for worst case, second is to input an ordered array [0,1,2, … ,15] for best case. And third is to iterate this algorithm for 1000 times with random unordered arrays. The result of this test is:

Worst case: QuickS\_M3([2,12,4,10,6,14,8,1,3,5,7,9,11,13,15,0], 0, 15): # of comparisons is 80

Best case: QuickS\_M3([0,1,2, … ,15], 0, 15): # of comparisons is 43

Iteration of algorithm: Statistic\_iter(QuickS\_M3, 1000, 0, 127): minimum # of comparisons is 760, average # of comparisons is 842.202, maximum # of comparisons is 1001, standard deviation of # of comparisons is 1495.6012

As we can see, all representive value of algorithm got better with the method Median of 3. Minimum result becomes smaller from 775 to 760, average result also does from 942.19 from 842.202, maximum result also does from 1312 to 1001. Interesting part of it is the great decrease of standard deviation; it becomes smaller by 5558.3 to 1495.6.

It seems to be the effect of choosing pivot element in 3 elements. By this method, the possibility of getting a bad pivot element goes down radically because the probability that 3 elements all have a bad value for the pivot is much smaller than the probability that one element has a bad value to be a pivot element.

4. Comparison of Result with Theoretical Values

For Theoretical Analysis, ,, and .

In this experiments, there are some errors in the best case and the worst case, but since the error is within we ignore the error in here. It might be the error came up with the reason written above; we would not repeat it again.

IV. Analysis of Quick Sort with Median of 3 and Insertion added Algorithm

1. Description

**Problem, Inputs, Outputs)** Problem, Inputs, and Outputs of Quick Sort with Median of 3 and Insertion added Algorithm is same with Quick Sort above. I’ll also skip these here.

**Pseudo-code)**

def QuickS\_Insertion\_added(S, l, u):

m= int((l+u)/2)

if u-l+1>= 9:

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

if S[m]>S[u]:

S[m], S[u]= S[u], S[m]

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

S[l+1], S[m]= S[m], S[l+1]

p = part(S,l+1,u-1)

QuickS\_Insertion\_added(S,l,p-1)

QuickS\_Insertion\_added(S,p+1,u)

else:

S= InsertionS(S, l, u)

return S

For understand this algorithm, we should also describe Insertion Algorithm and Insert Subalgorithm. Here is a description of InsertionS:

**Problem, Inputs, Outputs)** Problem, Inputs and Outputs of InsertionS Sort Algorithm is same with any other sorting algorithms above.

**Pseudo-code)**

def InsertionS(S, l, u):

if l<u:

return Insert(InsertionS(S,l,u-1), l, u)

else:

return S

**Problem of Subalgorithm Insert)** Given an unordered array S and the index l to u, insert the last element of subarray S[l:u] in the right index i in S[1:u-1] so that all elements of S[l:i-1] is smaller than S[i], and all elements of S[i+1:u-1] is bigger than S[i].

**Inputs of Subalgorithm Insert)**

S: given unordered array

l: least index of array such that we want to insert the last element in

u: upper index of array such that we want to insert the last element in

**Outputs of Subalgorithm Insert)**

S: array such that the last element is inserted in

**Pseudo-code)**

def insert(S, l, u):

for i in range(u-1, l-1, -1):

if S[i]> S[i+1]:

S[i], S[i+1]= S[i+1], S[i]

else:

return S

return S

2. Theoretical Analysis of Time Complexity

The basic concept of analysis is similar to Quick Sort M3; the best case would happen when pivot element of partition subalgorithm divides the array into two subarrays of the same length for every partition. And the worst case would happen when pivot element of partition subalgorithm couldn't divide given array for two subarrays.

The difference between Quick Sort M3 and Quick Sort M3 Insertion added is the base case. In Quick Sort M3 Insertion added, the base case is when n<=8, and then does insertion sort for this small subarrays. For this knowledge as a basis, let's find the time complexity in either best case and worst case.

**Best Case)**

Take k s.t holds, then . Since

**Worst Case)**

Take k s.t holds, then and .

3. Programming Approach and Result

**Problem and Approach, Inputs and Outputs)** Problem and Inputs are same with a description of 'Quick Sort M3 with Insertion added'. We return the ordered array and the operation count value as Output.

**Actual Code)**

def QuickS\_Insertion\_added(S, l, u):

global sigma\_operation

m= int((l+u)/2)

if u-l+1>= 9:

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

sigma\_operation+= 1

if S[m]>S[u]:

S[m], S[u]= S[u], S[m]

if S[l]>S[m]:

S[l], S[m]= S[m], S[l]

sigma\_operation+= 1

sigma\_operation+= 1

S[l+1], S[m]= S[m], S[l+1]

(p, op) = part(S,l+1,u-1)

sigma\_operation+= op

QuickS\_Insertion\_added(S,l,p-1)

QuickS\_Insertion\_added(S,p+1,u)

else:

S= InsertionS(S, l, u)

return (S, sigma\_operation)

**Problem, Inputs, Outputs)** Problem and Inputs are same with Insertion Sort description. We return the ordered array and the operation count value as Output.

**Actual Code of Insertion Sort)**

def InsertionS(S, l, u):

if l<u:

return insert(InsertionS(S,l,u-1), l, u)

elif l==u:

return S

else: raise Exception('Something went wrong.') #for check the code error

**Problem, Inputs, Outputs)** Problem, Inputs and Outputs are same with insert subalgorithm description.

**Actual Code of insert subalgorithm)**

def insert(S, l, u):

global sigma\_operation

for i in range(u-1, l-1, -1):

sigma\_operation+= 1

if S[i]> S[i+1]:

S[i], S[i+1]= S[i+1], S[i]

else:

return S

return S

**Results)** For this code, I tested this algorithm for 3 ways: first is to input an array [2,13,4,15,6,14,8,1,3,5,7,12,11,10,9,0] for worst case, second is to input an ordered array [0,1,2, … ,15] for best case. And third is to iterate this algorithm for 1000 times with random unordered arrays. The result of this test is:

Worst case: QuickS\_Insertion\_added([2,13,4,15,6,14,8,1,3,5,7,12,11,10,9,0], 0, 15):

# of comparisons is 84

Best case: QuickS\_Insertion\_added([0,1,2, … ,15], 0, 15): # of comparisons is 29

Iteration of algorithm: Statistic\_iter(QuickS\_Insertion\_added, 1000, 0, 127): minimum # of comparisons is 738, average # of comparisons is 827.704, maximum # of comparisons is 1020, standard deviation of # of comparisons is 1861.5624

All representive value of algorithm is smaller than QuickS\_1 algorithm. Minimum result is smaller for 738 than 775, average result also is for 827.704 then 942.19, maximum result also is for 1020 to 1312.

But it is not clear than Quick Sort with Median 3 and Insertion Sort is better than Quick Sort only with Median 3. Quick Sort M3 Insertion added is better at minimum case(738<760) and average(827.704<842.202), but the maximum case and standard deviation are worse(respectively 1020>1001, 1861.6>1495.6). This result shows that changing the base case for insertion sort works better in speed, but since the performance of insertion sort is highly depend on remain subarray, the standard deviation goes higher than Quick Sort with only M3.

4. Comparison of Result with Theoretical Values

For Theoretical Analysis, ,, and .

In this experiments, there are some errors in the best case(even if it is correct it might have an error value that is canceled each other) and worst case, but since the error is within we ignore the error in here. It might be the error came up with the reason written above; we would not repeat it again.

V. Conclusion

Written below is the result of analyses of three methods in Quick Sort:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Min result | Average | Max result | Standard deviation |
| Normal Quick Sort | 775 | 942.19 | 1312 | 5558.34 |
| Quick Sort M3 | 760 | 842.202 | 1001 | 1495.60 |
| Quick Sort M3, Insertion | 738 | 827.704 | 1020 | 1861.56 |

With this result, we could analyze that the performance of Quick Sort is ‘Normal Quick Sort < Quick Sort M3 ≈ Quick Sort M3 with Insertion’. Normal Quick Sort shows worst performance as a poor representative values. We could conclude that Quick Sort M3 with Insertion is best for minimizing the average comparison operation number, but it is hard to conclude that Quick Sort M3 with Insertion is best since Quick Sort with only M3 also has an advantage of least standard deviation and has nice speed either.