Measuring Algorithm Efficiency

1. **[Algorithm Complexity]**

Measuring the space and time complexity of an algorithm, regardless of the implementation, is no small feat. Space and time complexity are affected by a plethora of different elements and an analysis does not result in a simple metric which indicates the overall performance of an algorithm. Instead, the analysis yields many indicators and metrics which are useful for determining a better understanding of how an algorithm would perform across a variety of circumstances and situations.

One of the most important considerations of any algorithm analysis is space complexity. While time complexity is also a major component of the decision to use an algorithm for a specific problem, space complexity presents an estimation of usage of tangible resources, specifically that of both secondary memory and primary memory. Space complexity can have a significant impact upon the evaluation of an algorithm, especially as the size of the problem set, represented as n, increases.

In the context of algorithm analysis, space complexity typically has five primary elements, the first of which is *the amount of memory required to store an algorithm's source code*. The space in which the source code is stored, which is located in secondary memory prior to execution and gradually moved in and out of primary memory during execution, is of a constant size and it is typically not a primary concern for most algorithms given the abundance of inexpensive secondary storage present on most machines.

Before an algorithm begins execution and after it has finished, two more elements of space complexity must be taken into consideration: *the amount of memory required to store input data* and *the amount of memory required to store output data*. The former is especially a concern with algorithms that perform calculations on a large amount of data. Fortunately, in the context of input storage, the complexity grows linearly as n grows. This is not necessarily the case for the amount of memory required to store output data. Depending on the problem the algorithm addresses and the desired output, there may be minimal output data, an equivalent amount of output data as input data, or an increased amount of output data over that of input data. Finally, for both input and output data, the space complexity of the necessary data storage becomes even more of a concern in the context of time complexity; depending on the the access and write speeds of the storage medium, increases in n have serious implications.

Next, space complexity must consider *the amount of working memory needed by the algorithm during execution*. This may be the most important component of space complexity as it is the most impacted by the size of the problem set, and as *n* increases exponentially, the space needed may rapidly approach the total space available and may even exceed it, prohibiting the algorithm from executing to completion. Furthermore, if the initial data must be preserved, redundant data may need to be present in primary memory at the same time. For instance, consider a problem with input data of 4GB. Assuming the input data is read into RAM, an algorithm with space complexity $O(n^2)$ would require a total of 20GB, which exceeds the ram the majority of common laptops making the algorithm impractical for those machines. If, however, an algorithm with a space complexity of $O(n \log n)$ is used, the total RAM needed is just under 7.5GB. The above example is primarily concerned with the storage of the data during execution, however, the quantity of environment variables and the use of stack space must also be considered, especially with an algorithm that utilizes recursion. It is also important to distinguish between two different types of space complexity and the impact they have upon required working memory: *polynomial and exponential growth*. Polynomial growth consists of space complexities of *n* to the power of a constant, such as n^2 or n^3. Just as polynomial space complexities present a significant increase over linear space complexity, exponential growth complexity algorithms require substantially more space than those of polynomial complexity. Exponential growth complexities consist of a base constant raised to the power of *n*, such as 2^n, and while they initially grow at a rate less than that of polynomial, they quickly surpass polynomial and become exceedingly large. For example, given 16GB of input data, an algorithm of n^2 growth, n^3 growth, and 2^n growth would require 256GB, 4096GB, and 65,536GB of memory respectively.

The final element of space complexity is not a consideration purely in the context of available space, rather it is a consideration of *the use of data structures and the total amount of main memory needed by an algorithm during execution*. As technology has advanced over the years, the cost of nearly all forms of memory has significantly decreased while performance has continued to increase. While this is true for most memory present in modern machines, the quantity of each type present in a given machine has followed the traditional memory hierarchy. Depending on the total space an algorithm needs, which can be significantly impacted by the choice of utilized data structures, the machine may not be able to effectively utilize high performance memory and may rely mostly on slower mediums of memory.

Yet another important component of algorithm space and time complexity analysis is *the concept of relative and absolute complexities*. Both relative and absolute complexities have purpose, however, the complexity

measurement used for a given analysis is dependent upon the perspective from which an algorithm is analyzed. *Absolute Complexity* refers to the complexity of an algorithm's performance, both in quantifiable time and in space, within a specific computational environment while R*elative Complexity* refers to the expected complexity without consideration of the environment, measured in generic units of time and space. While both measurements are invaluable, they do not yield the same caliber of information. While relative complexity can be very useful for evaluating an algorithm in the context of one machine, it does not yield any useful information for other dissimilar machines. For this reason, relative complexity is considered to be the de facto representation of algorithm complexity for the majority of analyses.

In the context of relative space and time complexity, there are additional metrics that can provide insight into the estimated performance of a given algorithm. Primarily, these are the *upper and lower asymptotic* bounds which represent bounds for the theoretical *worst-case performance* and *best-cast performance* respectively. These bounds are dependent on the size of the problem set and essentially establish either a single function of n or two distinct functions of n that will always bound the algorithm, at any problem set size. Both bounds are incredibly informative and help indicate whether a given algorithm will outperform others at specific ranges of n or for all values of n.

The knowledge of worst-case performance and best-case performance are not always the best indicator of how an algorithm will perform at known values of n. For instance, if an algorithm focused on determining the median value of a randomized set of integers has an upper bound of $n \log n$ and a lower bound of n, the problem is theoretically guaranteed to complete somewhere within n units of time and $n \log n$ units of time. While this is an accurate range, it is substantially more likely that the run time will typically fall on or near $n \log n$ units of time. For many algorithms, such as this one, the vast majority of set sizes and distributions of data result in a time complexity which is very similar to one of the bounds. For this reason, often times the most influential metric of an algorithm's performance is its *average-case performance*. Average-case performance is useful in indicating the typical performance of an algorithm within an upper and a lower bound and is one of the most commonly used performance indicators, especially in the context of comparing two or more algorithms directed at solving the same problem.

In regards to the approach an algorithm might take at solving a specific problem, there are two primary methodologies: the *deterministic approach* and the *probabilistic approach*. A deterministic algorithm is one that employs an implementation which will always produce an output and does so by performing the same set of steps throughout execution. As a result, the output of a deterministic algorithm is independent of the ordering of

the input. Alternatively, a probabilistic algorithm, otherwise known as a randomized algorithm, is one that employs randomness as a component of its logic. While this may initially seem like a disorderly approach to solving a given problem, that is not necessarily the case. In the context of data searching, for instance, data is not typically sorted, hence the need for a search as opposed to directly referencing the location where the item is expected. In this situation, a deterministic algorithm that iterates through randomized data sequentially must iterate through an average of $n \div 2$ items and is bound by $O(n)$, however, if the item being searched for item is located near the n^{th} index, the algorithm must iterate approximately n times. The use of a probabilistic algorithm, which would still be bound by $O(n)$, may result in far fewer comparisons before the item is found.

A third category of algorithms exist which are somewhat similar to probabilistic algorithms, however, unlike probabilistic algorithms, they may generate different results even given the same input data. These algorithms are known as *Nondeterministic Algorithms*, and they are commonly used to generate approximate results. While not useful for many situations, nondeterministic algorithms can be very valuable in the context of problems with which obtaining a deterministic solution is too costly.

Yet another category of algorithms, *Interactive Algorithms*, take a more dynamic approach to a solving a given problem. Interactive algorithms have two primary "participants", a "prover" and a "verifier". They initially begin with both participants being provided with the same information, both participants exchange information, and both perform some form of computation which utilizes randomness. The end result of the algorithm is a response from the "verifier" regarding whether or not the claims of the "prover" are accurate[1]. As opposed to more traditional "static" algorithms, interactive algorithms are a probabilistic algorithm utilizing a multi-stage process.

2. **[Measuring Growth Rates]**

1-1 Comparison of running times

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ ~~microseconds~~ nanoseconds.

For the following calculations, the following assumptions are made:

- There are 1.0e9 nanoseconds in one second
- There are 30.42 days in a month

[1] http://www.cs.tau.ac.il/~canetti/f09-materials/f09-scribe3.pdf

- If the resulting value of *n* is not an integer, the corresponding value in the chart will be a truncated (rounded down) value to ensure the operation can complete within the given time

time/ function	1 second (1.0e9 ns)	1 minute (6.0e10 ns)	1 hour (3.6e12 ns)	1 day (8.64e13 ns)	1 month (2.63e15 ns)	1 year (3.156e16 ns)	1 century (3.156e18 ns)
lg *n*	$2^{1.0e9}$	$2^{6.0e10}$	$2^{3.6e12}$	$2^{8.64e13}$	$2^{2.63e15}$	$2^{3.156e16}$	$2^{3.156e18}$
sqrt *n*	$(1.0e9)^2$	$(6.0e10)^2$	$(3.6e12)^2$	$(8.64e13)^2$	$(2.63e15)^2$	$(3.156e16)^2$	$(3.156e18)^2$
n	1.0e9	6.0e10	3.6e12	8.64e13	2.63e15	3.156e16	3.156e18
n* lg *n	$e^{w(1.0e9\ log(2))}$	$e^{w(6.0e10\ log(2))}$	$e^{w(3.6e12\ log(2))}$	$e^{w(8.64e13\ log(2))}$	$e^{w(2.63e15\ log(2))}$	$e^{w(3.156e16\ log(2))}$	$e^{w(3.156e18\ log(2))}$
n^2	sqrt(1.0e9)	sqrt(6.0e10)	sqrt(3.6e12)	sqrt(8.64e13)	sqrt(2.63e15)	sqrt(3.156e16)	sqrt(3.156e18)
n^3	3rt(1.0e9)	3rt(6.0e10)	3rt(3.6e12)	3rt(8.64e13)	3rt(2.63e15)	3rt(3.156e16)	3rt(3.156e18)
2^n	29	35	41	46	51	54	61
***n*!**	12	13	15	16	17	18	20

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, ~~o~~, Ω, ~~ω~~, or Θ, of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	Ω	Θ
$\lg^k n$	n^ϵ	Yes	No	No
n^k	c^n	Yes	No	No
sqrt(n)	$n^{\sin(n)}$	No	No	No
2^n	$2^{n/2}$	Yes	Yes	Yes
$n^{\lg c}$	$c^{\lg n}$	Yes	No	No
lg(n!)	$\lg(n^n)$	Yes	No	No

3-3 Ordering by asymptotic growth rates

Rank the following functions by order of growth; that is, find an arrangement. $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, …, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions f(n) and g(n) are in the same class if and only if $f(n) = \Theta(g(n))$.

$(5/2)^n$	n^2	(lg n)!	$\lg^2 n$	2(constant)
$n2^n$	2^n	$2^{\lg n}$	n!	n / lg n

Equivalence classes are indicated below by comma-separated functions on the same line, however, they are still ordered according to the specifications above.

2(constant)
n/lg n
$lg^2 n$
n^2
$2^{lg\ n}$
2^n, $(5/2)^n$
$n2^n$
(lg n)!, n!

3. **[Problem Statements]**

Express the following five loosely described problems carefully in { Instance, Question } form as utilized in "Computers and Intractability". For each problem discuss the best time and space complexity you are aware of for solving the problem (from scratch) along with a few words naming or describing the method.

(i) Finding the median of n = 2 k + 1 integers.

INSTANCE: A set M of $n = 2k + 1$ integers where $k \geq 0$.
QUESTION: Is there a integer n of set M, where there exist k integers of less than or equal value and k integers of greater than or equal value?

One approach to this problem would be to implement an algorithm similar to *quicksort*. This implementation would consist of the following steps:

1. Select a random integer j from set M.
2. Iterate through the set, sorting values into two distinct sets X and Y based on their value in comparison to j; one set for values less than or equal to j and one for values greater than j.
3. If both sets have an equal length (equal to k)… the median has been found (j).
4. Else… repeat from step 1 using the larger set of sets X and Y

Time Complexity: The time complexity of this solution is very similar to that of quicksort. Best case scenario, the random integer picked is the median value and after the first splitting, both sets X and Y have the same length, resulting in a time complexity of $O(n)$. For the average case and worst case, the time complexity will mirror that of quicksort, with $O(n \log n)$.

Space Complexity: The utilization of space for this algorithm would mimic that of quicksort. Worst case this algorithm would use O(log n) space through the utilization of recursion.

(ii) Finding the 2 largest and 2 smallest of n integers.

INSTANCE: A set M of n integers where $n \geq 4$.
QUESTION: Are there integers m_w, m_x, m_y, m_z of set M such that m_w and m_x are smaller than n -2 integers and such that m_y and m_z are greater than n - 2 integers?

Through the use of 4 integer variables, the two largest and 2 smallest integers can be found in a single pass through the set. This solution assumes that the single highest or single lowest number may also be the second highest or second lowest number if the list contains duplicates of that number. This solution consists of the following steps:

1. Create variables *minLow*, *minHigh*, *maxLow*, *maxHigh* and initialize them all to the value of the first item of the list, create an iterator variable, *i*, and instantiate it to the index of the second integer in the list
2. Iterate through the list starting at the integer at index *i* and perform the following operations against the current value at that index (denoted by *currentVal*):

```
if (currentVal > maxHigh) {
        maxLow = maxHigh
        maxHigh = currentVal
}
else if (currentVal > maxLow) {
        maxLow = currentVal
}
else if (currentVal < minLow) {
        minHigh = minLow
        minLow = currentVal
}
else if (currentVal < minHigh) {
        minHigh = currentVal
}
```

Time Complexity: While this problem may appear complex, the solution is rather simple. Through the use of 5 variables and one scan of the the set, the two largest and two smallest integers can be found in linear time, or O(n).

Space Complexity: Throughout this process, the data is not being sorted or copied; the only process taking place is comparisons and storing the index of the four values being searched for. As a result, the space complexity of this solution is constant, or O(1).

(iii) Determining that a graph is bipartite.

INSTANCE: A graph $G = (V, E)$.
QUESTION: Can the vertices V of graph G be divided into two disjoint sets M, N such that every edge E of graph G connects a vertex of set M to a vertex in set N.

A simple solution for determining whether a graph is bipartite involves assigning an attribute with only 2 possible values to each edge while iterating through all the edges in the graph. For this implementation, these values will be ‘A’ and ‘Z’. This solution would be executed in the following steps:

1. Start with the first vertex of the graph and assign it a value of ‘A’ indicating it’s a part of disjoint set ‘A’

2. Follow each of that vertex's edges and assign each of those vertices a the opposite value, which would be 'Z' if the parent vertex was 'A'
3. Move to the child vertex's children and assign the opposite value ('A')
4. Traverse the tree and continue the above steps with every remaining unvisited vertex and its children
5. At any point during this process, if a vertex's value is overwritten and the value changes from 'A' to 'Z' or vice versa, the graph is not bipartite
6. Once all vertices have been visited, if no vertex had it's value overwritten to a different value, the graph is bipartite.

Time Complexity: Because this solution requires that each vertex be visited once and it's edges examined once, the overall time complexity of this solution is O($|V| + |E|$).

Space Complexity: Throughout the execution of this solution, each vertex is assigned an attribute, which must be stored somewhere. As a result, the only space required would be an array with one index allocated for each vertex which would be used to store the attribute for each vertex. Assuming each vertex could be identified by a unique ID or index value, this solution could be executed with a space complexity of O($|V|$).

(iv) Determining that a list of n numbers has no duplicates.

INSTANCE: A list M of n integers where $n > 0$.
QUESTION: Can sorted list M contain n integers such that every integer k_i, where $1 \leq i$, is greater than k_{i-1}?

This solution assumes list M is sorted, however, the implications of an unsorted list are discussed in the complexity sections below. Before discussing the actual solution, first we must consider a few edge cases: In the event that list M is empty ($n = 0$) or that list M contains only one integer, the list cannot contains duplicates. For all sorted lists where $n > 1$, it is possible to detect duplicates with a single pass through the list. This can be accomplished by the following:

1. Start at the index (i) of the second integer (k) in the list
2. Compare the value of k_i against the value of k_{i-1}
3. If… the value of k_i is greater than the value of k_{i-1}, advance i and continue
4. Else… the list contains duplicates

Time Complexity: Under the assumption the data in the list has been sorted, this solution can be performed in linear time, O(n), with a single iteration through the list. If the data is not sorted, however, the time complexity is entirely dependent upon that of the sorting algorithm chosen. If quicksort were to be used, for example, the complexity of sorting (average case O($n \log n$)) would dictate the time complexity of the overall solution.

Space Complexity: The implementation of this solution would only require the storage of a single integer variable to keep track of the current index. However, similar to the time complexity discussed above, if the list is not sorted the space complexity is dependent upon that of the sorting algorithm used.

For the following just describe a verification algorithm and the method and efficiency of checking the answer using the verification algorithm. Problem numbers and pages refer to Gary and Johnson.

(v) Determining that the maximum number of edge disjoint paths between vertices v and w in a graph is less than k.

From a very simple perspective, the maximum number of edge disjoin paths can be limited by evaluating the number of edges of both *v* and *w*. Because edge-disjoint paths cannot share a single edge, the maximum number of edge-disjoint paths from *v* to *w* can be no larger than the number of edges of the vertex with the fewest edges.

From a more complex perspective, we can evaluate the maximum number of edge-disjoint paths by utilizing a methodology in which after a given path uses an edge, the edge is marked as visited. Given a set which claims to be the maximum combination of edge-disjoint paths, while evaluating each path and marking each visited edge, the solution will correct if and only if no path included an edge that had previously marked as visited. Performing this analysis has a time complexity of O(|*E*|) where *E* represents the set of edges in the proposed solution.

(vi) Clique of size J (see page 47 of Gary and Johnson).

Given a potential solution set of vertices *V* where *J* = |*V*|, we can determine whether the set of vertices is a clique by examining each node and its edges. For any given vertex of *J* vertices of set *V*, each vertex will have a connection to every other vertex, which is *J - 1* edges. To determine whether *V* is a clique of size *J*, every vertex of *V* must have *J - 1* edges, which connect to every other vertex.

This solution requires counting only the number of edges belonging to all |*V*| vertices of set *V*, resulting in a time complexity of O(|*V*|).

(vii) Set Packing: SP3 (p. 221).

Given collection *C* of finite sets and a potential solution of C' where C' contains two pairwise disjoint subsets: C_i and C_j. In order to determine whether C_i and C_j are mutually distinct, every vertex within C_i must be searched for within C_j. If every search for every vertex of C_i returns no matches, the solution is correct and the proposed solution contains two mutually distinct subsets.

In order to perform this evaluation, each vertex of C_i results in a search through all of C_j's nodes; overall the time complexity of this evaluation is O(n^2).

4. **[Estimation]**

Within a Random Geometric Graph (RGG), a specified number of nodes (indicated by **N**) are randomly placed within a topological space. Any two nodes are then connected by a link if they fall within a specific radius (indicated by **r**) of one another. The following discussions assume a square RGG space.

For a random geometric graph, G(N, r), estimate the average degree of a vertex:

(a) at least distance r from the boundary:
Given that a circle with *r* radius is as least *r* distance from any boundary, the entire area of the circle must exist within the RGG space (i.e., no portion of the circle exists outside the RGG). As a result, the number of nodes within any given circle of this type with radius *r* is equal to the percent of the area of that circle in respect to the area of the entire RGG (i.e., $\pi r^2 \div$ Total RGG Area) multiplied by the number of total number of nodes within the RGG (*N*). All nodes within this circle would be within the *r* value of the vertex at the center of the circle, and therefore, would be linked with that node. Finally, in order to calculate the degree of the specified vertex, we must subtract one from the total number of nodes contained within the circle. As a result, the average degree of a vertex at least *r* distance from the boundary is represented by the following: **$(\pi r^2 \div$ Total RGG Area) * N - 1**.

(b) on the boundary (convex hull)
Continuing the discussion above, a circle of radius *r* drawn around a vertex that fits entirely within the RGG space contains an average number of nodes represented by the same equation ($\pi r^2 \div$ Total RGG Area) * *N*. If a vertex exists on the vertex, however, the encompassing circle does not fit entirely within the RGG space. At most, half of the circle will exist within the RGG space and at minimum, only a quarter. Given this logic along with the equation discussed above, the average number of vertices within a circle of this character has an upper bound of ($.5\pi r^2$ / Total RGG Area) * *N* and a lower bound of ($.25\pi r^2 \div$ Total RGG Area) * *N*. Therefore, the average degree (*d*) of the vertex at the center of that circle would be represented by the following: **($.25\pi r^2 \div$ Total RGG Area) * *N - 1* ≤ *d* ≤ ($.5\pi r^2 \div$ Total RGGArea) * *N - 1***.

and estimate the time (big Oh) of determining all edges employing:

(c) all vertex pairs testing:
In order to test all vertex pairs, a test must be performed for every node of *N* total nodes which must be made against every other node in the RGG. As a result, this operation would result in an estimated complexity of **$O(n^2)$**.

(d) the line sweep method:
The Sweep Line method makes one pass across the RGG, stopping at each node only once. At each node, a test is performed to identify paired nodes. The estimated overall complexity of this approach is **$O(n \log n)$**.

(e) the cell method:
The Cell Method, which is also known as Smallest Last Ordering, has an overall estimated complexity of **$O(|V| + |E|)$** to achieve pair identification and linkage.

5. **[Verification by digit sums]**

1.4.3 - Which of the following expressions can be shown to be faulty by the digit sum check modulo |β+1| or |β-1|:

a) $1423_5 + 2214_5 = 3321_5$

[1 + 4 + 23 = 4] + [22 + 1 +4 = 5] % 4 = **1** = [3 + 32 + 1 = **5**] % 4

[-1 + 4 - 2 - 3 = 2] + [-2 - 2 - 1 + 4 = 1] % 6 = **3** = [3 + 3 + 2 + 1 = 3] % 6

Check Sum: **Verified**

b) 23_{10} x 43_{10} = 1041_{10}

[2 + 3 = 5] x [4 - 3 = 1] % 9 = **5 ≠ 2** = [1 + 0 - 4 + 1 = -2] % 9

Check Sum: **Faulty**

d) 8156_{10} x 3741_{10} - 26433595_{10}

[8 + 1 + 5 + 6 = 20] x [3 + 7 + 4 + 1 = 15] % 9 = **3 ≠ 1** = [2 + 6 + 4 + 3 + 3 + 5 + 9 + 5 = 37] % 9

Check Sum: **Faulty**

e) 312_{10} x 8110_{10} =2620320_{10}

[3 + 1 + 2 = 6] x [8 + 1 + 1 + 0 = 10] % 9 = **6** = [2 + 6 + 2 + 0 + 3 + 2 + = 15] % 9

[-3 + 1 - 2 = 4] x [-8 + 1 - 1 + 0 = 8] % 11 = **10 ≠ 1** = [-2 + 6 + 2 + 0 - 3 + 2 - 0 = 1] % 11

Check Sum: **Faulty**

f) 452_{10} - 326_{10} = 125_{10}

[4 + 5 + 2 = 11] - [3 + 2 + 6 = 11] % 9 = **0 ≠ 8** = [1 + 2 + 5 = 8] % 9

Check Sum: **Faulty**

g) 259_{10} + 136_{10} = 395_{10}

[2 + 5 + 9 = 16] + [1 + 3 + 6 = 10] % 9 = **8** = [3 + 9 + 5 =17] % 9

[-2 + 5 - 9 = 6] + [-1 + 3 - 6 = 4] % 11 = **10 ≠ 1** = [-3 + 9 - 5 = 1] % 11

Check Sum: **Faulty**

h) 1386_{10} div 125_{10} - 10_{10}, remainder 11_{10}

[1 + 2 + 5 = 8] x [1 + 0 = 1] + [1 + 1 = 2] % 9 = **1 ≠ 0** = [1 + 3 + 8 + 6 = 18] % 9

Check Sum: **Faulty**

1.4.4 - Check the following octal-decimal and binary-decimal identities by appropriate digit sums modulo 9:

a) $2134_8 = 1068_{10}$

[2 + 1 - 3 + 4 = 4] % 9 = **4 ≠ 6** = [1 + 0 + 6 + 8 = 15] % 9
Mod 9 Check: **Faulty**

b) $1010011010101012 = 21461_{10}$

[1 x 8 = 8] % 9 = **8 ≠ 5** = [2 + 1 + 4 + 6 + 1] = 14 % 9

Mod 9 Checksum: **Faulty**

6. **[Implementation Testing]**

You are to implement and test a program for summing 1/x as x runs over all approximately eight million (23 fraction bits) single precision floating point numbers in the interval [1, 2). You are to do this on a server, PC (or Mac) of your choice. You are first asked to predetermine estimates of your implementation's computation time (architecture and compiler dependent), result accuracy (algorithm and round-off dependent), and result value (real valued sum estimate). Specifically:

a) Discuss the computational environment for your tests, including the compiler, operating system, machine MHz and cycle times for appropriate instructions and whether pipelining affects your execution time.

The Operating System of the computational environment is Mac OS X Yosemite (10.10.5) which is running on a Retina MacBook Pro. The hardware utilizes a "2.0GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.2GHz) with 6MB shared L3 cache"[2]. Cycle times for appropriate instructions for the "Haswell" series of processors[3]:

Instruction	Instruction Code	Estimated Clock Cycles	Est. Pipelined Clock Cycles
Floating Point Division	FPDIV	10 - 24	13
Floating Point Addition	FPADD	1 - 3	1.5
Floating Point Comparison	FCOMP	1	1

Note: This data reflects the entire series of 'Haswell' processors, which indicates they may be a good starting point for estimating the runtime of the program but they are not necessarily accurate for the specific processor model used.

The program was initially developed in Python, but due to the complicated compilation and execution environment (including the use of a virtual machine) the final program was written in C++ and compiled using g++.

---

[2] https://support.apple.com/kb/SP690

[3] http://www.agner.org/optimize/instruction_tables.pdf

Finally, the program is impacted by pipelining in terms of comparisons and additions, however, because of the large number of clock cycles needed for division and the stalls they must introduce, a lot of the benefits of pipelining are diminished.

b) Predetermine an estimate of the time utilizing single precision for the variables for all computations from any documentation you can find from the hardware manufacturer and/or compiler and system provider.

In order to estimate the execution time of the of the program, first the number of clock cycles needed for each instruction type and for 2^{23} calculations must be calculated, which will be indicated on the chart below using the information discussed in the section above.

Instruction Code	Number of executions per iteration	Est. Pipelined Clock Cycles	Total Cycles for 2^{23} Calculations
FPDIV	1	13	$1 \times 13 \times 2^{23}$
FPADD	2	1.5	$2 \times 1.5 \times 2^{23}$
FCOMP	1	1	$1 \times 1 \times 2^{23}$

$(1 \times 13 \times 2^{23}) + (2 \times 1.5 \times 2^{23}) + (1 \times 1 \times 2^{23}) = 17 \times 2^{23} = 142{,}606{,}336$
Total Clock Cycles Needed: ~1.4e8

System Clock Cycles per second: **2e9 ≤ Cycles/sec ≤ 3.2e9**

Execution time with full processor capabilities (turbo boost):
1.4e81 ÷ 3.2e9 = .04456448 = ~44ms

Execution time with standard processor:
1.4e81 ÷ 2e9 = .071303168 = ~71ms

Execution Time Estimate: 44ms ≤ Runtime ≤ 71ms

c) Predetermine a rough estimate of the exact sum (hint: how many terms are being added and how large is an "average term").

During the calculation, the value of x is increased 2^{23} times from a starting value of 1 to an ending value of 2. This means that throughout this calculation, the average value of x is the average of 1 and 2, which is 1.5. As a result, each individual value of 1/x has an average of ⅔. Finally, summing that value 2^{23} times should result in a sum of **approximately 5,590,000**.

d) Predetermine an estimate of the accuracy. Single precision computation should be done in round to nearest mode as provided by standard C implementations. By accuracy of the sum we mean the difference between the rounded sum of rounded values compared to the exact sum of exact values.

With the use floating point values within C++, there is an measurable difference between the infinitely precise value and the actual value calculated, which is referred to as 'Rounding Error'. For the typical use of floating

point, an estimated error of 2.0% ± 1.0% is reasonable. Given the estimated sum above, a result in the following range is likely: 5,645,900 ≤ Actual ≤ 5,757,700.

e) Give the measured running time and the computed result for your implementation. Compare the results with your estimates of running time and approximate sum. Compute the sum in double precision and single precision and compare to give a reasonable value for the accuracy of the single precision sum. Compare your result with another student's results that might have performed the sum in a different order. Can you explain the size of the approximation error?

RUNTIME:
Estimated Execution Time: 44ms ≤ Runtime ≤ 71ms
Actual Average Execution Time: 50ms
Estimated Error (Assuming 2GHz): 13.6%

SINGLE PRECISION SUMMATION:
Estimated Summation Value: 5.59e6
Actual Summation Value: 5.76588e6
Estimated Error: 2.3%

DOUBLE PRECISION SUMMATION:
Actual SP Summation Value: 5.76588e6
Actual DP Summation Value: 5.81454e6
Estimated Difference in Value: 0.84%

After comparing results with another student, the results were very similar. Both computation environments consisted of Mac hardware/software with a similar processor and both programs were written in C++, compiled with g++. There were slight differences in the execution time, which may be caused by variations in the processor (while both had the same series of processor, the other student's was a more recent model), or they may have been caused by differences in the implementation of the program.