Christian Ayala – 28971284 September 17, 2013
CSE 7350 – Algorithm Engineering Homework 1

Problem 1 [Algorithm Complexity] (15 points)

Performing a comprehensive analysis of time and space complexity for an algorithm is, in many cases, a non-trivial exercise. It requires a deep understanding of the problem statement and the algorithm used to solve the problem.

In most cases, we are more concerned with the time complexity of an algorithm, because space (in terms of computer memory) has become decreasingly expensive. However, there are problems in which the space requirements for an algorithm are prohibitively large. In such cases, space becomes a premium resource, and knowing which algorithms use that space most efficiently is extremely valuable.

In regards to space analysis for an algorithm, there are several factors to consider. First, from a practical standpoint, the algorithm itself must live in the memory of the computer. In many cases, the algorithm implementation represents a very small percentage of the space required to solve a problem. When the input or output size becomes significantly large, the space required for the algorithm implementation remains constant. For this reason, the size of the implementation is typically not considered in the overall space complexity of the algorithm (though in cases where the input/output is small, it may have an impact).

Moving away from implementation specific factors, the second element of space complexity analysis is the amount of space needed for the input to the algorithm. In the case where the input is a set of integers, adding n integers to the input results in an input space increase of n, or growth of order n. In the case of a complete graph (a.k.a. a full mesh), an input of n vertices has a total of $(n*(n-1))/2$ edges, resulting in a growth order of n^2. Since memory is finite, such a large growth factor should be avoided. The third element of space complexity analysis is the amount of memory needed for the output of the algorithm. Some algorithms (sorting algorithms like bubble sort and quick sort) can be done in-place, requiring no additional output space. Others produce an answer of "yes" or "no" (such as problem 3iv in this homework), and therefore need a constant amount of output space. Still others may need an output space that is dependent on the size of the input.

The fourth element is the space required by the algorithm to compute the output (the intermediate space). As mentioned previously, bubble sort is an in-place algorithm, and does not require additional computation space. On the other hand, merge sort is typically not done in-place. Due to the divide-and-conquer strategy of dividing the input into subsets, it requires computation space that grows linearly with the input. The last element is a detail of the algorithm: the data structures used when computing the output. Different data structures have different space requirements, and can alter the space (and time) requirements of an algorithm. For example, arrays are sequential blocks of space in

memory, whereas lists (which are similar to arrays from an abstract point of view) are not. Lists require additional pieces of data internally to store information about where all of the elements are stored in memory, and may provide a notable increase in the space usage of an algorithm.

When analyzing algorithms, it can be useful to determine both absolute and relative complexities. As I will explain, however, having one without the other (in particular having an absolute complexity without a relative complexity) does little to verify the real-world effectiveness of an algorithm. Absolute complexities of space/time represent the requirements of an algorithm *on a particular computer*. For example, the run-time of bubble sort on an input size of n random integers, where n=100,000, may take X number of seconds and Y bytes of RAM on my personal computer. While this gives an *instance* of performance, it tells very little (if anything) about the actual complexity of the algorithm, because it is impossible to extrapolate this out to all possible computing scenarios. Instead, we typically look at the relative complexity of an algorithm. This focuses more on the abstract methodology used to solve the algorithm. Instead of runtime, it focuses on the number of operations the algorithm performs, typically as a function of the input. In addition, it is more concerned with the abstract "units" of space required as a function of the input, as opposed the physical number of bytes required for each element of the input (which is implementation specific).

With relative complexities in mind, we can then *predict* the absolute complexity of the algorithm once we take into consideration the processing power of the machine it will run on. The number of operations and the "units" required will stay the same independently of the machine (because it is a relative measurement), but based on the absolute performance of other machines, we can attempt to predict the runtime/space usage of this machine. Based on these metrics, it is possible to optimize a machine for a particular algorithm (indeed this is being done for the mining of Bitcoin currency, where miners can purchase specialized hardware that has been optimized for the algorithm used to mine Bitcoins).

When considering relative complexities of an algorithm, we typically analyze two distinct metrics: upper and lower asymptotic bounds. To give a semi-formal definition, an upper asymptotic bound states that g(n) is an upper bound of f(n), if there exists an n_0 and c, such that for all $n > n_0$, f(n) < c*g(n), where c is some constant. Similarly, a lower bound is the same definition, except that f(n) < c*g(n) for all $n > n_0$. The upper bound signifies a theoretical worst-case complexity for the algorithm, in that the algorithm follows the upper bounds' complexity even if the input makes the algorithm perform least-efficiently. The lower bound represents a best-case complexity for the algorithm. Note that the two bounds need not be identical; in fact, for many algorithms, this is not the case. This adds an issue in measuring these values, in that we have to consider exactly what input sets cause a best- and worst-case complexity. As an example, a quicksort has a time

complexity of $O(n^2)$ in the worst case, and O(nlgn) in the best case. If the selection of pivots results in poor partitioning of the data set, then the worst-case complexity is achieved. Knowing that complexity (and the scenario that causes it) allows for the algorithm to be refined such that it rarely happens.

In relation to these asymptotic bounds, there is a third metric that can be more difficult to quantify: average-case complexity. This measures how the algorithm performs in *most* cases, which can be somewhat ambiguous. However, it is still a useful metric, and one that is perhaps more useful in practice as opposed to theory. In the case of the previously mentioned quicksort, in the vast majority of cases the worst-case complexity can be avoided, and therefore has an average complexity of O(nlgn). Meanwhile, bubble sort has a worst-case time complexity of $O(n^2)$, and if the data is already (or very nearly) sorted, a best-case time complexity of O(n). However, given that most real-world data is not already sorted (hence the need for a sorting algorithm), on average it runs with time complexity $O(n^2)$. Immediately it is apparent that, even though these two algorithms share the same upper-bound function, on average quicksort will outperform bubble sort by a significant margin.

When discussing algorithms, in many practical cases we are focusing on deterministic algorithms. These are algorithms that, for a given set of input, will always (a.k.a. deterministically) return the same output. Since computers are essentially state machines, it makes sense to logically map a deterministic algorithm (which is typically a set of finite steps) to a machine's instruction set. What is useful about deterministic algorithms is that they are simpler to analyze, and can be reliably replicated in machines. However, they are not always the most efficient way to solve a problem. In some cases, probabilistic algorithms may perform better, or have better time/space complexities. In probabilistic algorithms, randomness is typically employed as a step in the algorithm, and depending on the randomness of the input, the algorithm may or may not provide a correct response. For example, suppose we wish to find the location of a specific element m from a list of n integers. The deterministic approach would be to sequentially check each element in the list until we find the value m. This has time complexity O(n). However, suppose we take a probabilistic approach, and randomly choose an element n_k. If $n_k = m$, then we are done. Otherwise, find a new random number, and repeat. Though this is a contrived example, it shows that in some cases it will perform better than the deterministic approach, while in others it will perform worse, depending on the luck of the random numbers chosen.

Christian Ayala – 28971284 September 17, 2013
CSE 7350 – Algorithm Engineering Homework 1

Problem 2 [Measuring Growth Rates]

a) Text problem 1-1, where the algorithm to solve the problem takes f(n) nanoseconds.
We assume here that 1 month = 30 days.
1 second = 10^9 nanoseconds.

In the case that n would be a decimal number, I round up to the next integer value. For example, for the 2^n function to take exactly 1 second, the n value should be approximately 29.89735…. In this case, I round up to 30.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg *n*	2^{10^9}	$2^{6*10^{10}}$	$2^{3.6*10^{11}}$	$2^{8.62*10^{12}}$	$2^{2.59*10^{14}}$	$2^{3.11*10^{15}}$	$2^{3.11*10^{17}}$
$\sqrt{n}$	10^{18}	$3.6*10^{21}$	$1.296*10^{25}$	$7.465*10^{27}$	$6.719*10^{30}$	$9.675*10^{32}$	$9.675*10^{36}$
n	10^9	$6*10^{10}$	$3.6*10^{11}$	$8.64*10^{12}$	$2.59*10^{14}$	$3.11*10^{15}$	$3.11*10^{17}$
*n*lg*n*	$3.962*10^7$	$1.945*10^9$	$9.858*10^{10}$	$2.111*10^{12}$	$5.674*10^{13}$	$6.326*10^{14}$	$5.591*10^{16}$
n^2	31,623	244,949	$1.898*10^6$	$9.295*10^7$	$5.0912*10^7$	$1.764*10^8$	$1.764*10^9$
n^3	1,000	3,915	15,327	44,209	137,366	314,489	1,459,729
2^n	30	36	42	47	52	55	62
n!	13	14	16	17	18	19	21

b) We assume that $k \geq 1$, $\varepsilon > 0$, and $c > 1$, are all constants.

	A	*B*	*O*	θ	Ω
a.	$\lg^k n$	n^{ε}	Yes	No	No
b.	n^k	c^n	yes	No	No
c.	$\sqrt{n}$	$n^{\sin n}$	no	No	No
d.	2^n	$2^{n/2}$	Yes	Yes	Yes
e.	$n^{\lg c}$	$c\lg^n$	yes	No	no
f.	lg(*n*!)	$\lg(n^n)$	yes	No	No

c) Rank the following functions:

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

From smallest order of growth to largest:

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

Christian Ayala – 28971284 September 17, 2013
CSE 7350 – Algorithm Engineering Homework 1

Problem 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 is an integer, and
$M = \{m_0, m_1, \ldots m_n\}$

QUESTION: Is there a median number m_j in a set of n = 2k+1 integers, such that there are k integers less than or equal to m_j, and k integers greater than or equal to m_j?

Time complexity analysis: One possible way to solve this problem is similar to quicksort.

1) Pick a random number from set M, called m_r (this is like a quicksort pivot).
2) Divide all other numbers in set M into two distinct subsets
    a. A set which contains all values less than or equal to m_r (set L)
    b. A set which contains all values greater than m_r (set G)
3) If the number of elements in set L is equal to k, then this implies that the number of elements in set G is equal to k, and m_r is your median value
4) If the number of elements in set L is less than k, then this implies that the number of elements in set G is greater than k, and the median is an element in G. Repeat the process at step 1, except we pick a number .
5) If the number of elements in set L is greater than k, then this implies that the number of elements in G is less than k, and the median is an element in L. Repeat the process at step 1 using set L.

The best-case running time for this algorithm is O(n). This happens if we randomly select the median at the first iteration, and we only partition the n values once. This requires n comparisons, and is therefore O(n). The worst-case running time for this algorithm is $O(n^2)$. If the algorithm randomly selects the smallest element at each iteration, we cause k iterations of the algorithm, each of requires n-i comparisons (where i is the iteration number). Since k is approximately n/2, we have a worst-case growth rate of $O(n^2)$. Given the randomness of the pivot selection, however, it is unlikely that this algorithm will perform at the worst-case, and is much more likely to run at O(n).

Space complexity for this algorithm is again dependent on the number of iterations we perform. At each iteration of the algorithm, we need n-(i-1) additional space between the two subsets, where i is the iteration number. At best, we do only one iteration, and only create two subsets of data, which constitutes a growth of n elements of space (O(n)). At worst, we perform k iterations,

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

INSTANCE: A set M of n integers, where $n \geq 4$, and $M = \{m_0, m_1, \ldots, m_n\}$

QUESTION: Are there two subsets of M, named L (with cardinality 2), and G (with cardinality 2), such that the elements of L are less than or equal to all elements of M, and the elements of G are greater than or equal to all elements of M?

Time complexity analysis: This can be solved in linear time. In a few words, this can be called the linear scan for min and max variables. We initialize four integer values as follows: min1 and min2 variables that are initialized to infinity, and max1 and max2 variables that are initialized to negative infinity. For each number m_i in set M, perform the following analysis:

1) If m_i is less than min1, save m_i into min1 and proceed to step 4.
2) If m_i is greater than min1 but less than min2, save m_i into min2 and proceed to step 4.
3) If m_i is greater than both min1 and min2, proceed to step 4.
4) If m_i is greater than max1, save m_i into max1 and begin step 1 again with m_{i+1}
5) If m_i is less than max1, but greater than max2, save m_i into max2 and begin step 1 again with m_{i+1}
6) If m_i is less than both max1 and max2, begin step 1 again with m_{i+1}

After repeating this process n times, min1 and min2 will represent the subset L, and max1 and max2 will represent the subset G. Each of the 6 steps runs in constant O(1) time. Since this occurs n times, overall time complexity is O(n).

Space complexity analysis: Since we only need an additional 4 variables of space regardless of the input size, space complexity is constant O(1).

(iii) Determining that a graph is bipartite.

INSTANCE: A graph G containing a set of vertices and edges, denoted by (V, E).

QUESTION: Can G be divided into two disjoint sets of vertices A and B, such that every edge connects exactly one vertex from set A and exactly one vertex from set B? Restated, does G contain any odd-numbered cycles between its vertices?

Time complexity analysis: This can be determined in linear time. In a few words, we can call this the depth-first coloring traversal. First, we assume that all vertices have a color, namely white. Start with a given vertex in G, namely v_1. Apply the color red to v_1. We then assign colors to each of the vertices in the tree by selecting an opposite color from that of its parent vertex (so if a parent vertex is red, the vertex is assigned blue, and vice-versa). If at any point in the process, we discover two connected nodes with the same color (that is not white), then the graph can be determined to be non-bipartite, and we

return a value false. Otherwise, if all vertices have been visited and no two connected vertices were assigned the same colors, then the graph is considered to be bipartite, and return a value true.

Space complexity analysis: the algorithm must assign colors to each vertex in the graph. Since the vertices might not be readily colored, additional space per vertex may be necessary to perform the analysis. Therefore, the space complexity is theoretically O(V), where V is the number of vertices in G. However, assigning one of three colors to a vertex can be done very efficiently, and can therefore require a very small amount of space.

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

INSTANCE: A list M of n integers.

QUESTION: Does there exist a subset of M, named V, such that V = $\{v_1, v_2, \ldots, v_k\}$, where V = $\{v_i \mid v_i \neq v_j$, for all j = 1…k, j ≠ i$\}$,
and k = n (a.k.a. cardinality(M) = cardinality(V))?

Time complexity analysis: In a few words, this can be called the sort and scan method for finding duplicates. Since there can be no assumption about the contents of list M, the most efficient method that comes to mind is to first sort the list. We can use an efficient sort like quicksort, which has an average complexity of O(n lg n) (the worst case for this algorithm is $O(n^2)$, but supposing we have a reliable way of choosing a good pivot, we can say O(n lg n)). With the elements sorted, do a linear scan of the array checking for duplicates with the following methodology. Starting with i = 1 to i = n-1, if $m_i == m_{i+1}$, we return false. Else, we increment i and examine the next pair of elements. If we reach i = n, then we have not found a duplicate, and we return true. Since the sorting has complexity O(n lg n) and the scan has a complexity of O(n) (since we are doing n-1 comparisons), the overall complexity is O(n lg n).

Space complexity analysis: The sort that I have mentioned above can be done in-place. Therefore, it does not require any additional space. The process of linearly scanning the list of sorted elements for duplicates also does not require any additional space. Therefore, space complexity can be considered constant, or O(1).

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*.

Suppose we have a solution set of P paths. The verification algorithm is a simple path-marking algorithm. For each path P_i, we mark each edge E_j as visited. If the edge was

already previously marked as visited, then the solution is incorrect. After marking all edges for all paths, then we have verified that P is a correct set of edge-disjoint paths. Finally, if the cardinality of set P is less than *k*, then the solution has been verified.

This has complexity O(E), where E is the total number of edges in the graph. It may be the case that every edge in the graph is part of an edge-disjoint path. In this case, we must traverse and mark every edge in the graph.

(vi) Clique of size *j* (See page 47 of Gary and Johnson)

Suppose we have a solution V', where |V'| = J. To verify that V' is indeed a clique, at each vertex v_i in V', we count all distinct edges. For each v_i, there should be J-1 edges coming out of v_i. For each of the J vertices we check for J-1 edges, which is a total complexity of $O(J^2)$, where J is the size of the clique.

(vii) Set Packing (Page 221)

Suppose we have a solution set of C', where C' is a subset of the finite sets in C, and |C'| = K. To verify that C' has mutually distinct sets, we must compare each element in C_i to all elements in C_j, where $j = 1 \ldots k$, $j \neq i$. If there exists an element in C_i that also exists in C_j, then the solution is invalid. If there are a total number of *n* elements over all sets in C', then we have a total complexity of $O(n^2)$.

4. [Estimation]

For this problem I am using the following definition for a random geometric graph: Randomly choose a sequence of n independent and uniformly distributed points on $[0,1]^2$, and given a fixed r > 0, connect two points if their Euclidean distance is at most r.

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

a) at least distance r from the boundary,
In this case, these are internal points. From a given point, r refers to a radius of a circle, wherein other points in the circle are connected to the point. In the case where r = 0.5, the circle is enclosed by the square. In this case, the ratio of the area of the circle (and the number of points likely to be connected to the center point) is pi/4. Therefore, at a given r, the degree of a vertex is approximately n*r*(pi/4), where n is the number of points in the graph.

b) on the boundary (convex hull),
In this case, it has a maximum Euclidian distance away from the other points. The circle emanating from a point will at most be half as large as the internal points, so I estimate a degree of at most 0.5*n*4*(pi/4).

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

c) all vertex pairs testing,
All vertices must be compared against all other vertices for their Euclidean distances. Since this is n points comparing against n-1 other points, we have a complexity of $O(n^2)$.

d) the line sweep method,
Since there is only one sweep through the graph, and each node is visited once and compared to those that are near them on the line, I estimate a complexity of O(n).

e) the cell method.
Since we are dividing the graph into cells recursively, I estimate a complexity of O(nlgn)

5 [Verification by Digit Sums]

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

a) [1+4+2-3]+[2-2+1+4] = 9 = 1 (mod 4)

[3+3-2+1] = 5 = 1 (mod 4)

[-1+4-2-3]+[-2-2-1+4] = -3 = 3 (mod 6)

[-3+3+2+1] = 3 (mod 6)

Therefore, this expression is correct

b) [2+3]*[4-3] = 5 = 5 (mod 9)

[1+0-4+1] = -2 = 7 (mod 9)

Therefore, this statement is faulty.

d) [8+1+5+6]*[3+7+4+1] = 3 (mod 9)

[2+6+4+3+3+5+9+5] = 1 (mod 9)

Therefore, this statement is faulty.

e) [3+1+2] * [8+1+1+0] = 6 (mod 9)

[2+6+2+0+3+2+0] = 6 (mod 9)

[3-1+2] * [-8+1-1+0] = 1 (mod 11)

[2+6+2+0+3+2+0] = 1 (mod 11)

The digit sum check modulo method did not find the statement to be faulty, even though 312*8110=2,530,320.

f) [4+5+2]-[3+2+6] = 0 (mod 9)

[1+2+5] = 8 (mod 9)

Therefore, this statement is faulty.

g) [2+5+9]+[1+3+6] = 8 (mod 9)

[3+9+5] = 8 (mod 9)

[2-5+9]+[1-3+6] = 10 (mod 11)

[3-9+5] = 10 (mod 11)

The digit sum check modulo method has verified this statement as correct

h) No answer

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

a) [-2+1+3+4] = 6 = 6 (mod 9)

[1+0+6+8] = 15 = 6 (mod 9)

Therefore this statement is valid.

b) $101001101010101 = 51525_8$

[5-1+5-2+5] = 12 = 3 (mod 9)

[2+1+4+6+1] = 15 = 6 (mod 9)

Therefore, this statement is invalid

6. [Implementation Testing] (15 points)

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.

I am using a Macbook Air running Mac OSX 10.8.4 Mountain Lion. It has a dual-core, Intel Core i7-4650U processor (code-name Haswell), clocked at a base clock speed of 1.7GHz. At the discretion of the Operating System, it can be dynamically adjust its clock speed up to 3.3GHz for a single core (this number is different if we are using multiple cores, but this program will be single threaded). I am using the clang++ version 4.2 compiler, which is a front-end for the LLVM version 3.2 compiler.

The processor currently in this laptop was released in late June, and I could not find cycle times for the operations for this processor. Instead, I am looking at the most recent processor throughput information from this document dated February 2012: http://gmplib.org/~tege/x86-timing.pdf. It comes from the main website of the GNU Multiple Precision Arithmetic Library.

From this document, I learned that the most recent Intel processor at the time has the following cycle counts for the following instructions:
`div`: 26 cycles
`add`: 1 cycle
`mov`: 1 cycle (for storing the results)
`cmp`: 3 cycles (for comparing if x < 2)

I will assume that these numbers are similar to those found in this processor. Division is not a pipelined operation, though add and mov are pipelined.

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.

The computations for this problem are shown in the following code block:

```
float sum = 0.0, x = 1.0;

while (x < 2.0) {
    sum += (1/x);
    x = nextafterf(x, 10.0); //Go to next floating point value
}
//Sum now has the total summation for 1/x, x is in [1,2)
```

First, we focus on the control statement in the while loop, `(x < maxValue)`. In this case, the processor applies branch prediction to this statement. In other words,

since x will be less than maxValue 2^{23} times, the processor assumes that this is true at each iteration of the loop, and therefore does not spend a large amount of time on this instruction. Therefore, we approximate it with the typical 3 clock cycles.

Next, the statement `sum += (1/x);` Here we have a division, followed by an add, followed an assignment (or in assembly terms, mov). We can approximate this with 28 clock cycles. Then, we have the nextafterf function, which we approximate with an additional cycle (it is difficult to estimate the cycle time for this function, since this may be as simple as a single add instruction inserted inline into to the code). In all, each loop is estimated to take 32 clock cycles. With 2^{23} iterations of the loop, it is estimated to take 268,434,456 clock cycles. With a max clock rate of 3.3 GHz, this is estimated to take 81,344 microseconds to complete.

This estimate is with pipelining not taken into consideration, as well as ignoring the fact that the operating system may perform context switches to other applications during runtime. In an effort to mitigate that problem, I closed all user applications, so that the operating system is less likely to need to context switch.

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

An average term for x is 1.5. There are an equal number of values greater than and less than 1.5, and they are symmetric distances from 1.5. Therefore, we can estimate the total sum as being $(2^{23})*(1/1.5) = 5{,}592{,}405.333333\ldots$

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

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?

The computed result using only float values was exactly 5,765,884. To restate, the expected result was 5,592,405.333333… Using the equation of:

```
deviation=(actualValue - expectedValue) / expectedValue;
```

We can calculate that we have a deviation of:

(5765884-5582405.3333) / 5582405.3333 = 0.03102

or 3.102% deviation from the expected sum value. This deviation may be explained by the loss of precision in the division algorithm. When dividing by such a precise number (such as the first floating point value after 1.0), the exact answer may have more decimal precision than what is available. The program was run 10 times, and the runtime of all runs were averaged. The average runtime was 47688 microseconds. Compared to the estimated runtime of 81344 microseconds, we have a deviation of approximately 41.4%. The significantly faster runtime than expected may have to do with pipeline optimizations made by the compiler.

Using double precision float variables, the computed sum was 5814540.234. Using the same algorithm for deviation, we have a deviation of 4.16%. The runtime was, on average, approximately 52465 microseconds. This is an increase of approximately 5000 microseconds. The increase in time, as well as the increase in deviation, is likely explained by the fact that we continuously have to convert from a single precision floating point (as calculated by 1/x) to a double precision floating point number. Some accuracy may be lost in this conversion, and this takes a small amount of time.