**Chapter 11 – Week 14 – Exercises**

Exercises #1 – page 396

1. **Write a tester program that counts and displays the number of iterations of the following loop:**

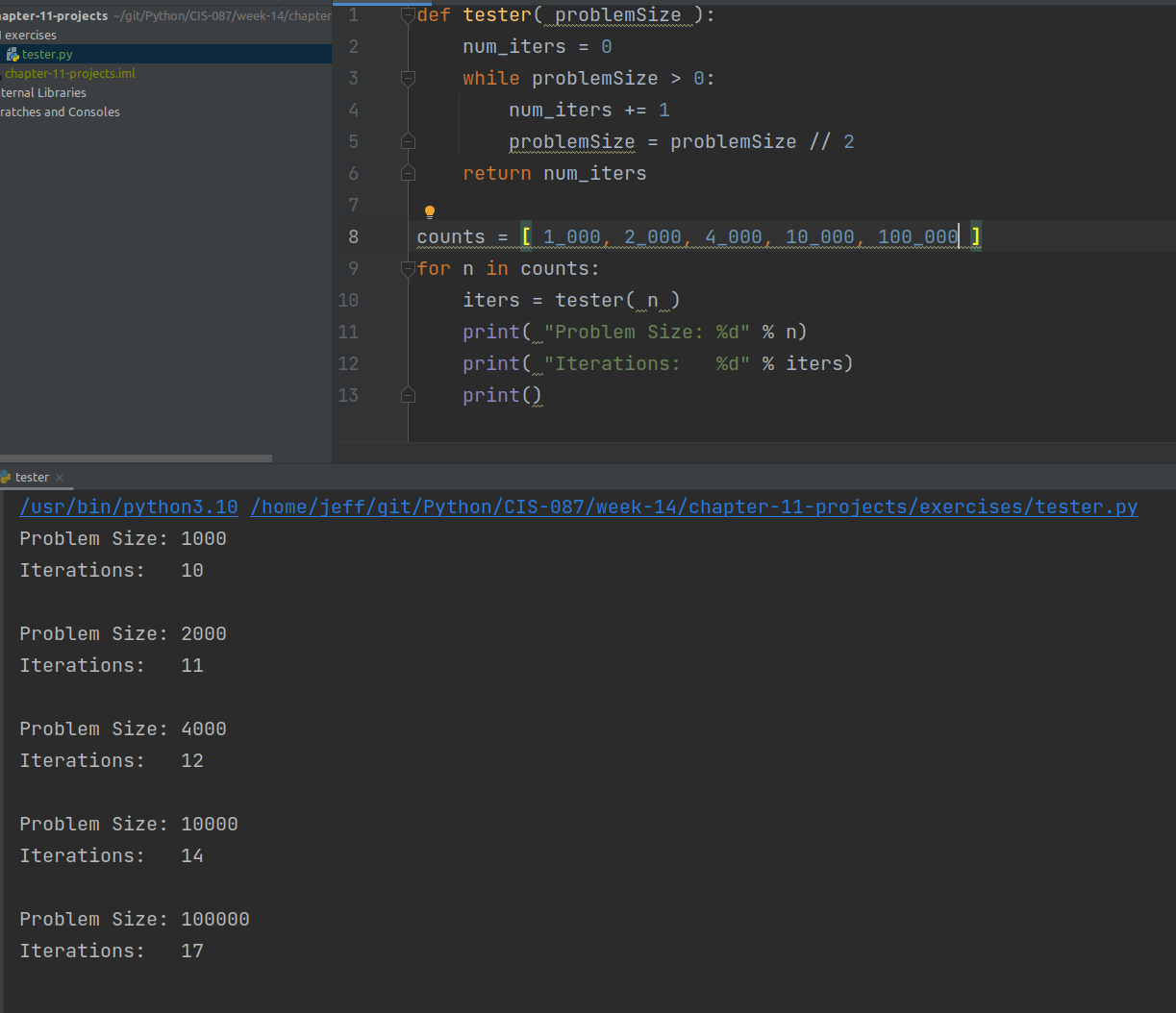
**while problemSize > 0:**

**problemSize = problemSize // 2**

def tester( problemSize ):  
 num\_iters = 0  
 while problemSize > 0:  
 num\_iters += 1  
 problemSize = problemSize // 2  
 return num\_iters  
  
counts = [ 1000, 2000, 4000, 8000 ]  
for n in counts:  
 iters = tester( n )  
 print( "Problem Size: %d" % n)  
 print( "Iterations: %d" % iters)  
 print()

1. **Run the program you created in Exercise 1 using problem sizes of 1000, 2000, 4000, 10\_000, and 100\_000. As the problem size doubles or increases by a factor of 10, what happens to the number of iterations.**

Program execution is shown below (see next page). As the problem size increases, so does the iteration count. The iteration count goes up by the log2 of the problem size. It is O(log2).



1. **The difference between the results of two calls of the time function time() is an elapsed time. Because the operating system might use the CPU for part of this time, the elapsed time might not reflect the actual time that a Python code segment uses the CPU. Browse the Python documentation for an alternative way of recording the processing time and describe how this would be done.**

Python has a module timeit that runs a snippet of code a large number of times and returns the average of the runs. This utility can be run from the command line or using a function call from within a Python script.

Exercises #2 – page 401

1. **Assume that each of the following expressions indicates the number of operations performed by an algorithm for a problem size of n. Point out the dominant term of each algorithm, and use big-O notation to classify it.**

*a) 2n – 4n2 + 5n*

*dominant term: 2n*

*big O (2n)*

*b) 3n2 + 6*

*dominant term: 3n2*

*big O (n2)*

*c) n3 + n2 - n*

*dominant term: n3*

*big O (n3)*

1. **For problem size n, algorithms A and B perform n2 and ½n2 + ½n instructions respectively. Which algorithms does more work? Are there particular problem sizes for which one algorithm performs significantly better than the other? Are there particular problem sizes for which both algorithms perform approximately the same amount of work?**

These algorithms are both big O(n2). So, for extremely large work loads, the amount of work is similar. Of course, the second algorithm is likely to take about half the time of the second one in these circumstances.

1. **At what point does an n4 algorithm begin to perform better than a 2n algorithm?**

An O(n4) algorithm will begin to perform better than a O(2n) almost immediately (To be precise, when n > 16)

Exercises #3 – page 405

1. **Suppose that a list contains the values**

**20 44 48 55 62 66 74 88 93 99**

**at index positions 0 through 9. Trace the values of the variables left, right, and midpoint in a binary search of this list for the 5target value 90. Repeat for the target value 44.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 20 | 44 | 48 | 55 | 62 | 66 | 74 | 88 | 93 | 99 |

Target: **90**

Start: left = 0 right = 9 midpt = 4 **90** > 62

left = 5 right = 9 midpt = 7 **90** > 88

left = 8 right = 9 midpt = 8 **90** < 93

left = 8 right = 7 END WITH NO MATCH

Target: **44**

Start: left = 0 right = 9 midpt = 4 **44** < 62

left = 0 right = 3 midpt = 1 **44** = 44

MATCH FOUND AT INDEX 1

1. **The method we usually use to look up an entry in a phone book is not exactly the same as a binary search because, when using a phone book, we don’t always go to the midpoint of the sub-list being searched. Instead, we estimate the position of the target based on the alphabetical position of the first letter of the person’s last name. For example, when we are looking up a number for “Smith”, we first look toward the middle of the second half of the phone book, instead of in the middle of the entire book. Suggest a modification of the binary search algorithm that emulates this strategy for a list of names. Is its computational complexity any better than that of the standard binary search?**

To implement this algorithm, one could determine what percentage of names start with each letter of the alphabet. Using these values dictionary of starting percentages could be generated for each alphabet letter. A would always start at 0. B would start at 0 plus the percentage of words that begin with the letter a. C would start at the percentage of words that started with A plus the percentage that began with B. Continue until Z.

When a name is given, look at the first letter. Use the letter as an index into the dictionary to determine the “percentage location” in the list to begin. Multiply the percentage location to begin by the list length. This gives the index at which the search should begin.

From this point on, a standard binary search could ensue with a maximum change in the midpoint being capped at 20 percentage points of the list size since we hope to be close to the searched value.

The computation complexity of this algorithm is still O(log2 n) making it no different than the standard. This may reduce the comparisons by 2 or 3 while increasing the complexity of the algorithm making it harder to support and often not reducing the computation time at all.

Exercises #4 – page 410

1. **Which configuration of data in a list causes the smallest number of exchanges in a selection sort? Which configuration of data causes the largest number of exchanges?**

Selection makes the smallest number of swaps (0) when the list is already sorted and it makes the largest number of swaps (nearly n) when the list is in reverse order.

1. **Explain the role that the number of data exchanges plays in the analysis of selection sort and bubble sort. What role, if any, does the size of the data objects play?**

The number of data exchanges a sort makes can add up. Nothing near the dominant value for large data sets, but it takes time nonetheless.

1. **Explain why the modified bubble sort still exhibits O(n2) behavior on the average.**

The modified bubble sort takes O(n2) on average because the average case will not include a completely sorted list (and thus short-circuit to the end) until nearly the entire list has been sorted. This means the dominant term will still be in effect.

1. **Explain why insertion sort works well on partially sorted lists.**

Insertion sort works well on sorted lists because the sort does not have to do any work (swaps) as it adds a member to the sorted portion of the list.

Exercises #5 – page 418

* 1. **Describe the strategy of quicksort and explain why it can reduce the time complexity of sorting from O(n2) to O(n log n).**

Quicksort finds the correct place in the list for a given value (called the pivot in the text) in the list and then moves all values less than the pivot prior to the pivot’s location and all values higher than the pivot to after the pivot’s location. Next, the values before and after the pivot are recursively sorted according to the same algorithm.

When conditions are perfect, the list will be divided log2 n times and sorted in O(n) time resulting in a O(n log2 n).

* 1. **Why is quicksort not O(n log n) in all cases? Describe the worst-case situation for quicksort and give a list of 10 integers, 1-10, that would produce this behavior.**

When conditions are not perfect, quicksort must split the list more and more often leading to a worst case situation of splitting the list by the largest or smallest value in the list. In this case, time complexity becomes O(n2)

If we are selecting the pivot as the middle item in list, the list <2,4,6,8,10,5,3,7,1,9> will result in the worst case value of O(n2) as seen in the table below.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 6 | 8 | **10** | 5 | 3 | 7 | 1 | 9 |
| 2 | 4 | 6 | 8 | **9** | 5 | 3 | 7 | 1 | 10 |
| 2 | 4 | 6 | **8** | 1 | 5 | 3 | 7 | 9 | 10 |
| 2 | 4 | 6 | **7** | 1 | 5 | 3 | 8 | 9 | 10 |
| 2 | 4 | **6** | 3 | 1 | 5 | 7 | 8 | 9 | 10 |
| 2 | 4 | **5** | 3 | 1 | 6 | 7 | 8 | 9 | 10 |
| 2 | **4** | 1 | 3 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | **3** | 1 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **2** | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

* 1. **The partition operation in quicksort chooses the item at the midpoint as the pivot. Describe two other strategies for selecting a pivot value.**

The text mentions two other algorithms for choosing the pivot value when using quicksort. One is to take an average of the first, last and middle value in the list. Another is to choose a random element.

* 1. **Jill has a bright idea: When the length of a sublist is quicksort is less than a certain number – say, 30 elements – run an insertion sort to process that sublist. Explain why this is a bright idea.**

Since the memory usage of insertion sort is constant while quicksort uses O(log2 n) and the advantages of the improved performance in the best case situations are minimized with a small list to sort, Jill will reduce memory consumption without reducing performance.

* 1. **Why is merge sort an O(n log n) algorithm in the worst case?**

Merge sort divides a list into half recursively until it can be divided no more. This operation is O(log2 n). The lists are then merged with each merge being O(n). This means the resulting complexity of n merges at O(log2 n) makes merge sort a O(n log2 n) algorithm.