Computer Science I CSCI-141
Fast Sorting Lecture

10/14/2017

1 Problem

When students and employees join the RIT community they are assigned an ID. This is the logical way in which the system stores their information. However, since ID's are not common knowledge it is much easier to locate an RIT member using their name. Given a list of names that are in ID order, we'd like to efficiently sort them into alphabetical order. To do this, we'll write a program that creates an alphabetical listing for names in a text document.

We already know one way to sort: insertion sort. However, that algorithm's worst case time complexity is $O(N^2)$, which is not very good for large N. Now we want to sort more *efficiently*, spending as little time sorting as possible since documents could be very large. We will examine two other sort algorithms with smaller complexities.

2 Solution Design and Analysis: Merge Sort

The algorithm known as *Merge Sort* splits the list in two parts, sorts the parts, and puts the sorted parts back together again.

2.1 Algorithm

If the list is empty or a *singleton list*, it is trivially sorted. Otherwise, we split the list into two lists of roughly equal size, *half1* and *half2*, and recursively sort each. After sorting the halves, we merge them together. Here is the pseudocode[1]:

```
define merge_sort( L )
  if L is empty
      return L
  otherwise if L has only 1 item
      return L
  otherwise
      split L into two halves half1 and half2
          so that their sizes differ by at most 1.
      return merge( merge_sort( half1 ), merge_sort( half2 ) )
```

Note: Merge Sort is a *divide-and-conquer* algorithm. Divide-and-conquer algorithms divide the input into several parts, solve the parts recursively, and then combine the results. In `merge_sort`, splitting into `half1` and `half2` is the dividing part, recursive calls conquer those parts, and the function `merge` combines the results.

---

1. Pseudocode is a language that looks a lot like a programming language, but does not have a rigorous, fixed syntax. It also contains English phrases that would be considered easy to translate into a real programming language.

Split

One way to divide is to split down the middle[2]. That approach has some nice properties; however, it is not the only way to do it, and it may be hard to determine the middle of the list. We will use another method that can be a little bit faster depending on the underlying implementation of lists: the **even-odd split**. The idea is to put every even-position element in one list, and every odd-position element in another.

```
define split( L )
   evens = []
   odds = []
   is_even = True
   for e in L
      if is_even
         evens.append( e )
      otherwise
         odds.append( e )
      is_even = not is_even
   return ( evens, odds )
```

Merge

Finally, we need an algorithm for merging. Since the sublists to merge are already sorted, we can compare the first elements to determine the smallest element of the merged list. We move past that smallest element (in whichever list it was in), and continue in this fashion until one of the lists is empty. The remainder of the merged list is the remainder of the non-empty list.

```
define merge( sorted1, sorted2 )
   result = []
   index1 = index2 = 0
   while index1 < len( sorted1 ) and index2 < len( sorted2 )
      if sorted1[index1] <= sorted2[index2]
         result.append( sorted1[index1] )
         index1 = index1 + 1
      otherwise
         result.append( sorted2[index2] )
         index2 = index2 + 1
   if index1 < len( sorted1 )
      result.extend( sorted1[index1:] )
   otherwise if index2 < len( sorted2 )
      result.extend( sorted2[index2:] )
   return result
```

2. If the size of the original list is odd, the sizes of the new lists differ by exactly 1.

Example

We illustrate the ideas of the merge sort algorithm by providing a substitution trace and an execution trace of the list of numbers [5,7,1,3,4,8,6,2].

$$
\begin{aligned}
&\texttt{merge\_sort}([5,7,1,3,4,8,6,2]) = \\
&\texttt{merge}(\texttt{merge\_sort}([5,1,4,6]),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}(\texttt{merge\_sort}([5,4]),\texttt{merge\_sort}([1,6])),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}(\texttt{merge}(\texttt{merge\_sort}([5]),\texttt{merge\_sort}([4])),\texttt{merge\_sort}([1,6])),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}(\texttt{merge}([5],[4]),\texttt{merge\_sort}([1,6])),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}([4,5],\texttt{merge\_sort}([1,6])),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}([4,5],\texttt{merge}(\texttt{merge\_sort}([1]),\texttt{merge\_sort}([6]))),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}([4,5],\texttt{merge}([1],[6])),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}(\texttt{merge}([4,5],[1,6]),\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge\_sort}([7,3,8,2])) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}(\texttt{merge\_sort}([7,8]),\texttt{merge\_sort}([3,2]))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}(\texttt{merge}(\texttt{merge\_sort}([7]),\texttt{merge\_sort}([8])),\texttt{merge\_sort}([3,2]))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}(\texttt{merge}([7],[8]),\texttt{merge\_sort}([3,2]))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}([7,8],\texttt{merge\_sort}([3,2]))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}([7,8],\texttt{merge}(\texttt{merge\_sort}([3]),\texttt{merge\_sort}([2])))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}([7,8],\texttt{merge}([3],[2]))) = \\
&\texttt{merge}([1,4,5,6],\texttt{merge}([7,8],[2,3])) = \\
&\texttt{merge}([1,4,5,6],[2,3,7,8]) = \\
&[1,2,3,4,5,6,7,8]
\end{aligned}
$$

The execution trace diagram is in Figure 1 on the next page.

2.2 Implementation

See `merge_sort.py`.

2.3 Complexity: Running Time Analysis

To simplify our notation, let N denote the original number of elements in the original list. The time complexity, $T(N)$, of the merge sort has three phases: the split function, the recursive calls and the merge function.

2.3.1 The split function

We perform a constant number of operations in each iteration of the for-loop: the loop overhead, the *if* check, an *append* operation, and an update to `is_even`. The for-loop

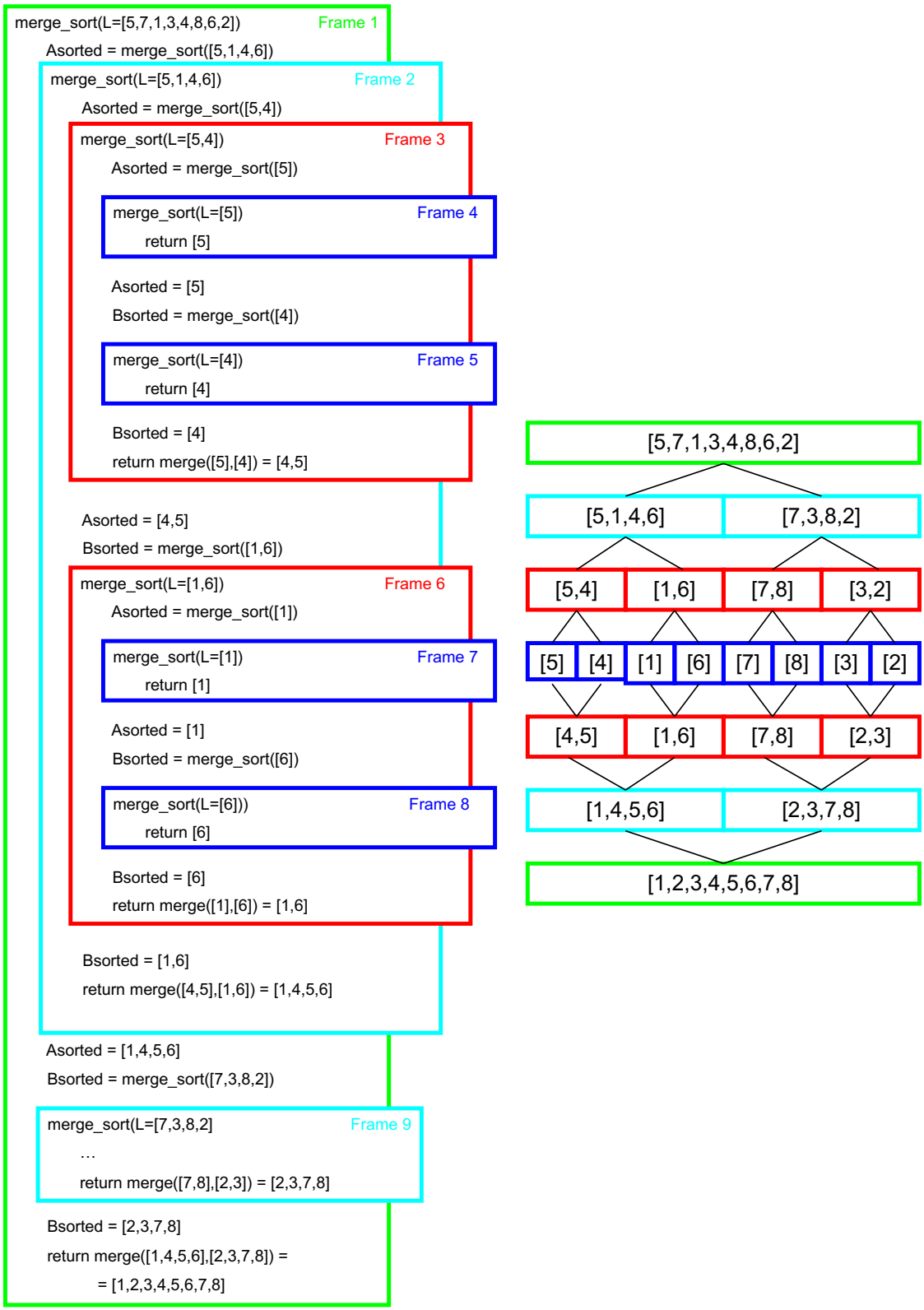

Figure 1: The figure on the left is the execution trace diagram, showing only the recursive calls to `merge_sort`. The figure on the right shows the operations schematically.

performs exactly N iterations. If we account for the creation of empty `evens` and `odds`, initializing `is_even`, and the return statement, the overall number of steps in the split function is $4N + 4$. Recall that whenever the number of operations is bounded by a linear function, we say that the running time is $O(N)$.

2.3.2 The merge function

The while-loop goes through a constant number of steps in every iteration. In every iteration, the index for either `sorted1` or `sorted2` is incremented; that means the total number of iterations cannot be more than the sum of the sizes of `sorted1` and `sorted2`. We say that the number of iterations has an *upper bound* of the sum of the sizes of `sorted1` and `sorted2`. Observe that the size of `sorted1` plus the size of `sorted2` is N because those lists were obtained by splitting the original list, L. Therefore the while-loop goes through at most N iterations. The merge performs a constant number of steps outside the while-loop. Thus, the total number of steps in the merge function is $O(N)$.

2.3.3 The sorting function

Whenever $N > 1$, the sorting function has these major steps:

1. the split function that takes $O(N)$ steps;
2. a recursive call on the first half, whose size is roughly $N/2$;
3. a recursive call on the second half, whose size is also roughly $N/2$;
4. and, finally, the $O(N)$ merge function.

Recall that mergesort is an example of a "divide-and-conquer" algorithm. When we split data, roughly in half, it will take about $O(\log_2 N)$ steps until we have exactly 1 element to examine. Then we have exactly N elements to merge together, which results in an overall time complexity of $T(N) = O(N \log_2 N)$.

2.4 Testing

Notice that our functions work with either numbers or strings because we can use the less-than comparison ($<$) to compare Python strings and numbers. To simplify the description of test cases, we will test using lists of numbers. We first test the non-recursive functions.

To test the `split` function, we should test with an input list of both even and odd size, and we should test lists that are both large and small. For example, we could test lists of size 0, 1, 2, 3, 4, 19, and 20. Since we are not using the value of the elements to decide how to spit the lists, we can use arbitrary values for our testing (e.g. $[1, 2, 3, 4, \ldots, 20]$) to be able to easily verify if the splitting function does what we designed it to do.

To test the `merge` function, we should test the following cases:

- Both lists are of size 0.
- Both lists are of size 1 and have these properties:
    1. First list's element is smaller than the second list's element;

    2. Second list's element is smaller than the first list's element; and
    3. Both elements are equal.
- One list is of size 0 and the other is of size 1.
- Both lists are of equal size and not very large, e.g. of size 4. We try the following sub-cases:
    1. Elements of the first list are all smaller than elements of the second list;
    2. Elements of the first list are all larger than elements of the second list;
    3. Elements interleave; e.g. the lists might be $[1, 3, 5, 7]$ and $[2, 4, 6, 8]$;
    4. Two lists have elements that are in random relation to each other — e.g. $[1, 2, 6, 7]$ and $[0, 3, 4, 9]$; and
    5. Some of the elements are identical. (There are many subcases here; we could test all elements being identical, then identical elements in the first list and different identical elements in the second list, then merging two copies of $[1, 2, 3, 4]$, etc.).
- Test two different variations in which the two lists are random, sorted lists of relatively larger size; e.g. sizes 17 and 18.

To test the main `merge_sort` function, we need to test the base case and the recursion. For the base case, we test input lists of size zero and one. For the list of size 1 we test several different values for the only element in the list.

Next, we test the cases close to the boundary base case: we test lists of size two and three. For both cases we test different values of elements in the lists, we should test all possible orderings: completely sorted, sorted from the largest to the smallest, and a random order. In particular, if we test with elements $10, 20, 30$, we test the following inputs: $[10, 20, 30]$, $[10, 30, 20]$, $[20, 10, 30]$, $[20, 30, 10]$, $[30, 10, 20]$, and $[30, 20, 10]$. Next, we test longer lists with elements in sorted order, "backwards" sorted order, and a couple of random orders. We should test inputs of both even and odd sizes since the algorithm behaves slightly differently for inputs of different parity. Testing with an input size equal to a power of two, a power of two plus one and minus one, and "far" from a power of two. For example, we might test inputs of the following sizes (longer than three): 9, 10, 31, 32, 33, and 100.

Finally, we need to test if the implementation behaves correctly for inputs with repeating elements. For example, for lists of size 3 we could test inputs: $[10, 10, 10]$, $[10, 10, 20]$, $[10, 20, 10]$, and $[20, 10, 10]$. Similarly, for lists of longer sizes, we want to include test cases when all the elements are identical, when there are groups of identical elements (e.g. half of the elements are equal to 10 and half are equal to 20), and when there are many different elements, some repeating and some not.

It is a good idea to try extremely large inputs (e.g., 10,000 numbers – we would need a testing program to generate such a large input) and test that the implementation runs much faster than a $O(N^2)$ sort such as insertion sort.

After testing with numbers, we test with similar lists of strings.

3 *Another* Solution Design and Analysis: Quick Sort

3.1 Algorithm

In merge sort, splitting the list in two was easy; the tricky part was putting the results back together. We might wonder if there's a way to sort such that the splitting phase is more complicated, but the joining together is easy. Note that, if all the elements of one part are less than all the elements of the other, then only concatenation is needed to put the parts back together. *Quick sort* is the resulting algorithm.

With quick sort, we divide, or *partition*, the list into three parts. Pick an element from the list and call it the *pivot*; for the sake of simplicity, we will pick the first element. The resulting three lists contain all the elements: the list of elements less than the pivot, the list of elements equal to the pivot, and the list of elements greater than the pivot.

For example, if the input list is `[5, 9, 2, 6, 4, 3, 5, 1, 3, 8, 7]`, then the three parts are `[2, 4, 3, 1, 3]`, `[5, 5]`, and `[9, 6, 8, 7]`.

It is not necessary to sort the list of elements equal to the pivot. When sorting the other two sublists, we get `[1, 2, 3, 3, 4]`, `[5, 5]`, and `[6, 7, 8, 9]`. Putting them together yields `[1, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9]`.

Here is the pseudocode:

```
define quick_sort(L)
  if L == []
      return []
  otherwise
      pivot = L[0]
      ( less, same, more ) = partition( pivot, L )
      return quick_sort( less ) + same + quick_sort( more )
```

To partition, we can loop through the elements of L, creating the lists as we go.
Here is the pseudocode:

```
define partition( pivot, L )
   ( less, same, more ) = ( [], [], [] )
   for e in L
      if e < pivot
         less.append( e )
      otherwise if e > pivot
         more.append( e )
      otherwise
         same.append( e )
   return ( less, same, more )
```

Example

We illustrate the ideas of the quick sort algorithm with a substitution trace of the list of numbers $[5, 7, 1, 3, 4, 8, 6, 2]$.

$$
\begin{aligned}
&\texttt{quick\_sort}([5,7,1,3,4,8,6,2]) = \\
&(\texttt{quick\_sort}([1,3,4,2]) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&((\texttt{quick\_sort}([]) + [1] + \texttt{quick\_sort}([3,4,2])) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([] + [1] + \texttt{quick\_sort}([3,4,2])) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + \texttt{quick\_sort}([3,4,2])) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + (\texttt{quick\_sort}([2]) + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ((\texttt{quick\_sort}([]) + [2] + \texttt{quick\_sort}([])) + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + (([] + [2] + \texttt{quick\_sort}([])) + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + (([2] + \texttt{quick\_sort}([])) + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + (([2] + []) + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2] + [3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + \texttt{quick\_sort}([4]))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + (\texttt{quick\_sort}([]) + [4] + \texttt{quick\_sort}([])))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + ([] + [4] + \texttt{quick\_sort}([])))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + ([4] + \texttt{quick\_sort}([])))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + ([4] + []))) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + ([2,3] + [4])) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&(([1] + [2,3,4]) + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&([1,2,3,4] + [5] + \texttt{quick\_sort}([7,8,6])) = \\
&([1,2,3,4,5] + \texttt{quick\_sort}([7,8,6])) = \\
&([1,2,3,4,5] + (\texttt{quick\_sort}([6]) + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + ((\texttt{quick\_sort}([]) + [6] + \texttt{quick\_sort}([])) + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + (([] + [6] + \texttt{quick\_sort}([])) + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + (([6] + \texttt{quick\_sort}([])) + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + (([6] + []) + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + ([6] + [7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + ([6,7] + \texttt{quick\_sort}([8]))) = \\
&([1,2,3,4,5] + ([6,7] + (\texttt{quick\_sort}([]) + [8] + \texttt{quick\_sort}([])))) = \\
&([1,2,3,4,5] + ([6,7] + ([] + [8] + \texttt{quick\_sort}([])))) = \\
&([1,2,3,4,5] + ([6,7] + ([8] + \texttt{quick\_sort}([])))) = \\
&([1,2,3,4,5] + ([6,7] + ([8] + []))) = \\
&([1,2,3,4,5] + ([6,7] + [8])) = \\
&([1,2,3,4,5] + [6,7,8]) = \\
&[1,2,3,4,5,6,7,8]
\end{aligned}
$$

3.2 Implementation

See `quick_sort.py`.

3.3 Complexity: Running Time Analysis

As before, let N denote the original number of elements in L.

3.3.1 The partition function

The pseudocode uses one loop that processes all the elements of L and the number of operations within the loop is constant; therefore the splitting part takes $O(N)$ steps.

3.3.2 The sorting function – the best case

Suppose $N > 1$; otherwise the sorting function takes only a constant number of steps. For $N > 1$, we have $O(N)$ steps for partitioning, the time needed for the execution of the two recursive calls, and the time for concatenation which is also $O(N)$. Ideally, the list of elements less than the pivot and the list of elements greater than the pivot, returned by the call to partition, would be the same length. If so, like with merge sort, the recursive calls on each half of the data results in total steps of $O(\log_2 N)$.

Notice that in this ideal case, $T(N)$ is the same as merge sort's time complexity function, and the result is $T(N) = O(N \log_2 N)$.

3.3.3 The sorting function – the worst case

However, the world is not always ideal. Imagine we give quick sort a list that is already in order. In that case, there are no elements less than the pivot, and all except values equal to the pivot are greater than the pivot. If the first element is unique, then the recursive call is on a list of length $N - 1$. Here, the recursive calls approach linear time, resulting in a time complexity of $T(N) = O(N^2)$, the same as insertion sort's worst case time complexity.

3.3.4 The sorting function – the typical case

If we assume that every list of a particular size is equally likely as an input, then we are much less likely to get a list that is in order, and more likely to get a list that is in some random order. Therefore, the *expected* performance of quick sort is $O(N \log_2 N)$.

3.4 Additional Improvements

We can implement quick sort to sort *in-place*, which means that we do not use extra lists as part of the computation. We partition the collection by rearranging L to contain the elements smaller than the pivot, followed by all the elements equal to the pivot, followed by elements larger than the pivot. This rearrangement does not use any other lists or other data structures. Further, when partitioning is *in-place*, nothing needs to be done to get the parts back together — they are already together!

An "adversary" can make our implementation of quick sort perform poorly by giving it lists that are already in order. Instead of assuming the likelihood of lists that are already in order, we can make the behavior of quick sort independent of any particular input. Choosing the pivot at random ensures that quick sort will perform well on any input with high probability, although that cannot be guaranteed for every possible input.

3.5 Testing

Testing is very similar to the merge sort algorithm.

We need to test the partition code, we should test with lists of size one, then size two with two possible pivot values: the smaller and the larger element from the list. Then we could test several lists of larger sizes, for example size four, ten, and twenty, and a variety of values for the pivot.

After we are sure that the partition code works, we test the sorting function with lists of sizes zero, one, two, and, for example, four, ten, and twenty. We should test a fully sorted input, as well as input in the reversed sorted order, and a completely unsorted input. Finally, we should test a very large list (e.g. one with a million elements). This last test checks if our code runs fast. We should see a very noticeable difference compared to insertion sort.