Searching

One of the most common tasks that is undertaken in Computer Science is the task of searching. We can search for many things: a value (like a phone number), a number (like seven), a position (like who finished the race in second place), or an object (like an image of a dog). Typically, searching will require doing a specific task over and over again (i.e., searching in one position, and then searching in another position), until we find what we are looking for. This requires repetition. We have already learned how to deal with and represent repetition. Our first task is to design an algorithm that can be used to search for a given value.

Activity 1

For this activity, you will participate in the demonstration of an algorithm to find some value (specifically, the maximum value) in a list of values. Approximately ten students will be asked to stand in front of the class. These will be known as *memory* students; that is, they represent a subset of a computer's memory. Each *memory* student will secretly write a number of their choosing on a post-it note. Collectively, this represents a list of numbers stored in the computer's memory.

One other student, known as the *computer* student, will represent the computer as it executes the algorithm. This student will follow instructions (like a computer would) and keep track of a single value representing the current maximum value in the list of numbers. This student will have a few post-it notes (or a small dry-erase board) to keep up with the current maximum value. The *computer* student understands several instructions:

START – instructs the *computer* student to go to the beginning of the *memory* student line;
MOVE – instructs the *computer* student to move to the next *memory* student in the line;
COMPARE – instructs the *computer* student to compare the number of the current *memory* student to the one currently recorded and respond with *greater*, if the current *memory* student's number is greater, or *less*, otherwise;
STORE – instructs the *computer* student to record the current *memory* student's number; and
DISPLAY – instructs the *computer* student to display the currently recorded value.

Can you design an algorithm that searches for the maximum value in the list?

Sequential search
The activity above was a demonstration of the sequential search. Sequential searching has a lot of applications in computer science.

Definition: ***Sequential search*** *(also known as* ***linear search****) is the process of locating a value in a list of values by checking each element, one at a time, until either the value is located or the end of the list has been reached.*

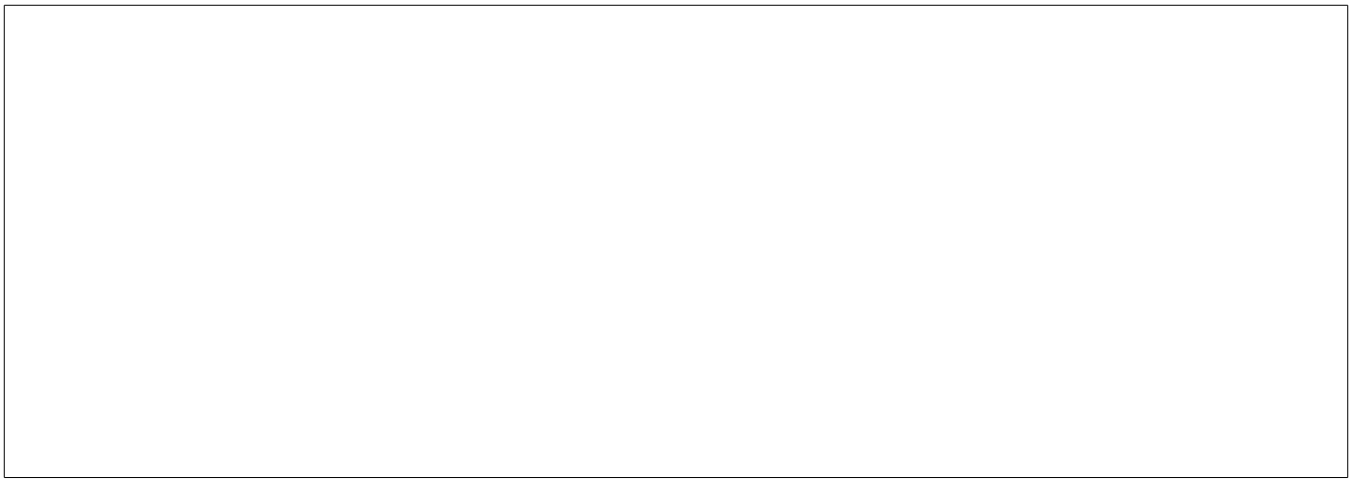
The algorithm described in the activity above can be represented more generally in pseudocode as follows:

```
 1: i ← 1
 2: max ← value of item i in the list
 3: repeat
 4:     increment i
 5:     if value of item i in the list > max
 6:     then
 7:         max ← value of item i in the list
 8:     end
 9: until i = the length of the list
10: display max
```

This algorithm searches for (and displays) the largest value in a list of numbers. How could it be modified to search for (and display) the smallest value in a list of numbers? Try to modify the algorithm above in the space below:

```
 1:
 2:
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
```

The sequential search can also be used to find a specific value (such as 100) instead of something more general as we did above (such as the maximum value). How could the algorithm that finds the maximum value be modified to search for a specified value and display whether it was found (or not)? Try to modify the algorithm above in the space below:



Is this algorithm efficient? What happens if the value searched for is not in the list? What happens if the value searched for is near the beginning of the list? We observe that, if the value searched for is near the beginning of the list, the algorithm sets the variable *found* to `true` but continues to look through the remainder of the list. This is not very efficient. Consider a list of 1 million values such that the value searched for happens to be found at the beginning of the list. The algorithm above would absolutely find the value at the beginning of the list; however, it would then continue to needlessly search through the remaining 999,999 values in the list. There is no stop condition in the case that the value searched for is found. We can fix this quite easily, however, by slightly modifying the algorithm:

```
 1: n ← value to search for
 2: i ← 1
 3: found ← false
 4: repeat
 5:     if value of item i in the list = n
 6:     then
 7:         found ← true
 8:     else
 9:         increment i
10:     end
11: until i > the length of the list or found = true
12: display found
```

By adding an additional **exit condition** (something that terminates the repetition), we can capture the case that the value is found at any point in the list and immediately terminate the search. When designing algorithms, we must carefully examine them to find inconsistencies, problems, and to ensure that they, in fact, solve the problem. In addition, we must not forget to consider efficiency, performance, and scalability. For a very short list, efficiency may not matter; however, we don't usually deal with small amounts of data in practice.

Here is an example of the sequential search for the value 80 applied to a list containing the following values: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100:

Search List	Comparison	Action
10, 20, 30, 40, 50, 60, 70, 80, 90, 100	$10 \neq 80$	Target not found; continue
10, **20**, 30, 40, 50, 60, 70, 80, 90, 100	$20 \neq 80$	Target not found; continue

Search List	Comparison	Action
10, 20, **30**, 40, 50, 60, 70, 80, 90, 100	30 ≠ 80	Target not found; continue
10, 20, 30, **40**, 50, 60, 70, 80, 90, 100	40 ≠ 80	Target not found; continue
10, 20, 30, 40, **50**, 60, 70, 80, 90, 100	50 ≠ 80	Target not found; continue
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	60 ≠ 80	Target not found; continue
10, 20, 30, 40, 50, 60, **70**, 80, 90, 100	70 ≠ 80	Target not found; continue
10, 20, 30, 40, 50, 60, 70, **80**, 90, 100	80 = 80	Target found; stop

The search begins by comparing the target value, 80, with the first value in the list, 10. Since 10 ≠ 80, then the search continues to the next value in the list. This is continued until the value is found in the eighth position. This sequential search requires a total of eight searches to find the target value. We can apply the sequential search to a number guessing problem: on average, how many tries would it take to correctly guess a number from one to 1,000? In the best case, one. In the worst case, 1,000. On average, 500.

Let's generalize this for a list of *n* values and call the tries *comparisons* (since that's what is actually being done). In the best case, it takes one comparison (if the target value is at the beginning of the list). In the worst case, it takes *n* comparisons (if the target value is at the end of the list). If many searches for various target values were performed, an *average* number of comparisons could be calculated – and we would find that approximately half of the list would need to be searched through. This should not be surprising, since the *good* cases (where the target value is at the front of the list), and the *bad* cases (where the target value is at the end of the list) tend to cancel each other out over many searches. On average, searching for a target value in a list containing *n* items requires *n* / 2 comparisons.

Activity 2

Can you come up with a better algorithm to correctly guess a randomly picked number from one to 1,000? Students will be paired up in groups of two. One student (called the *picker*) will secretly pick a number from one to 1,000; the other student (called the *guesser*) will attempt to guess the secret number under the following constraints:

The *guesser* submits a single number at a time;
The *picker* replies **HIGHER** if the secret number is higher than the *guesser*'s submission, or **LOWER** otherwise; and
The *picker* replies **CORRECT** if the *guesser*'s submission is correct.

Each group of students should design an algorithm to correctly guess the number picked by the *picker*. Record the number of guesses that it took to guess correctly. Then switch places: the *picker* becomes the *guesser* and vice versa.

Binary search

Some of you may have realized that the runtime of a sequential search is dependent on the size of the list that is being searched. Sequentially searching through a list of 1,000 items (as in the activity above) will typically take ten times as long as searching through a list of 100 items (i.e., 100 is one-tenth of 1,000). In the best case, the item will be found at the beginning of the list (i.e., it requires searching through only one item). In the worst case, the item will be found at the end of the list or not found at all (i.e., it

requires searching through the entire list). On average, the algorithm would have to search through about half of the list.

Consider the scenario of searching for a name that corresponds to a specific phone number in a phone book. This is the reverse of what is normally done (i.e., searching for a phone number that corresponds to a given name). How could this be done? The only way is to perform a sequential search, starting at the beginning of the phone book. This could take a long time, and it would be utterly depressing if the phone number was not found in the phone book. While the sequential search is effective, there are scenarios (like this one) where a more efficient way of searching is preferable.

Consider the normal approach to searching a phone book for a phone number that corresponds to a given name. A sequential search would require starting the search at the beginning of the phone book and continue until either the name is found or until the entire phone book is exhausted. However, a phone book has a specific quality that can be taken advantage of which makes searching significantly easier and faster: it is ordered in a meaningful way. The names in a phone book are arranged in alphabetical order. This makes searching easier in that only a small subset of the names have to actually be searched through. When searching through a phone book, the usual process is to thumb through it until the first letter of the given name is reached. From there, a sequential search (of sorts) is performed to search for the given name.

A method that may be better suited for computing is to open the phone book in the middle. If the given name starts with a letter in the alphabet that comes before the one that is there, then the right half of the phone book can be ignored. With a single comparison, half of the phone book has been eliminated. This strategy can be continued by shifting to the part of the phone book that represents the halfway point in *the first half* of the phone book. Another comparison is performed to determine if the given name appears before or after this point. This continues until the given name is found. With each check, half of the remaining portion is eliminated.

This strategy can be used to guess a number from one to 1,000 much more efficiently than by doing a sequential search. So how is the halfway point (or middle value) calculated? Given a list of n items, we calculate the middle as follows:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1$$

Note that the brackets represent the *floor* function which means to round down to the largest previous integer. For example, the floor of 3.14 is 3, and the floor of 27.9 is 27.

Some of you may have naturally implemented this algorithm in the activity above. Let's formally define it now in pseudocode:

```
1: repeat
2:     n ← number of items in the current portion of the list
3:     mid ← floor(n / 2) + 1
4:     guess mid
5:     if response is HIGHER
6:     then
7:         discard the left half of the list
8:     else if response is LOWER
```

```
 9:      then
10:          discard the right half of the list
11:      end
12: until guess is correct
```

This algorithm is known as a binary search.

Definition: ***Binary search*** *is the process of locating a value in an* ***ordered*** *list of values by repeatedly comparing the value in the middle of the relevant portion of the list to the desired value and discarding the appropriate half of the list. Searching terminates when either the desired value is located or the list can no longer be divided in half.*

Here is an example of the binary search for the value 70 applied to a list containing the following values: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100:

Search List	Comparison	Action
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	60 < 70	Target not found; discard left half
70, 80, **90**, 100	90 > 70	Target not found; discard right half
70, **80**	80 > 70	Target not found; discard right half
70	70 = 70	Target found; stop

The list is initially split in the middle at 60 (since there are 10 values, and floor(10/2) + 1 is 6 – the sixth value in the list). Since 60 is less than 70, we can safely discard the left half of the list (including the split value 60) and continue with the right half of the list: 70, 80, 90, 100. This smaller sub-list is split in the middle at 90. Since 90 is greater than 70, we can safely discard the right half of the sub-list (including the split value 90) and continue with the left half of the sub-list: 70, 80. This smaller sub-list is split in the middle at 80. Since 80 is greater than 70, we can safely discard the right half of the sub-list (including the split value 80) and continue with the left half of the sub-list: 70. This sub-list has a single value (70) which, when compared to 70, is found to be the target value. In four comparisons, the target value was found in the list.

What would happen if we tried to search the list for a value that the list doesn't contain? Here is an example of the binary search for the value 45 applied to a list containing the same values as before:

Search List	Comparison	Action
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	60 > 45	Target not found; discard right half
10, 20, **30**, 40, 50	30 < 45	Target not found; discard left half
40, **50**	50 > 45	Target not found; discard right half
40	40 < 45	Target not found; discard left half
empty list	none	Target not found; stop

It should be evident that the binary search is significantly faster than a sequential search. It turns out that guessing a number from 1 to 10,000 using the binary search will take, at most, 14 guesses. Intuitively, this is because 10,000 can be divided in half roughly 14 times: 10,000 is reduced to 5,000,

then to 2,500, then to 1,250, then to 625, then to 313, then to 157, then to 79, then to 40, then to 20, then to 10, then to 5, then to 3, then to 2, and finally to 1 (14 total splits). We can actually calculate this precisely by solving the following equation: $2^n = 10000$.

We can visualize how the binary search works to find a value in a list of 10,000 values by illustrating each guess (the middle value):

Comparison	List size	Items on left	Middle value	Items on right	Remaining items
1	10,000	5,000	5,001	4,999	5,000
2	5,000	2,500	2,501	2,499	2,500
3	2,500	1,250	1,251	1,249	1,250
4	1,250	625	626	624	625
5	625	312	313	312	312
6	312	156	157	155	156
7	156	78	79	77	78
8	78	39	40	38	39
9	39	19	20	19	19
10	19	9	10	9	9
11	9	4	5	4	4
12	4	2	3	1	2
13	2	1	2	0	1
14	1	0	1	0	0

Consider a simpler problem of guessing a number from 1 to 1,000. How many guesses would that take? We know that $2^{10} = 1024$ and that $2^9 = 512$; therefore, 1,000 can be divided by two between 9 and 10 times. However, it's evident that it's closer to 10 than it is to 9. In fact, it is actually 9.97 times. Recall that, on average, it would take 500 guesses if the sequential search were used instead. The binary search is therefore 50 times faster than the sequential search for a list of 1,000 values (500 guesses / 10 guesses = 50). What about the comparison for a list of 1 billion values? The sequential search would take, on average, 500 million comparisons. The binary search would take, at worst, 30 comparisons. That's almost 17 million times faster!

Did you know?

To solve for n in $2^n = 1000$, we can use the formula that expresses the inverse of raising two to a power: $\log_2 n = 1000$. Logarithms represent the power to which some number, called the base (in this case, 2), must be raised to produce a given number (in this case, 1,000). Most calculators do not have a $\log_2$ function; however, they typically do have a $\log_{10}$ function (note that most calculators denote this as *log* and omit the base). We can convert easily by using the following conversion (where b is the given base and d is the desired base):

$$\log_b x = \frac{\log_d x}{\log_d b}$$

In the example above where the base is 2, the desired base is 10, and *x* is 1000, we can convert as follows:

$$\log_2 1000 = \frac{\log_{10} 1000}{\log_{10} 2} = 9.97$$

For giggles, how many tries would it take to guess a number from one to 1 billion?

$$\log_2 1\,billion = \frac{\log_{10} 1\,billion}{\log_{10} 2} = 29.9$$

On average, searching for a target value in a list containing *n* items requires, *at maximum*, the following number of comparisons:

$$\lceil \log_2 (n+1) \rceil$$

Note that the brackets represent the *ceiling* function which means to round up to the smallest following integer. For example, the ceiling of 3.14 is 4, and the ceiling of 27.1 is 28.

The following table shows the number of comparisons required to search through a list of values ranging from 0 to 10,000 items using both the sequential and binary search. It also includes a performance measure that compares how much better the binary search is when compared to the sequential search. It is amazing to see the huge difference in the performance of the two algorithms:

Number of items	Sequential search	Binary search	Performance
0	0	0	0
1,000	500	10	50
2,000	1,000	11	91
3,000	1,500	12	125
4,000	2,000	12	167
5,000	2,500	13	192
6,000	3,000	13	231
7,000	3,500	13	269
8,000	4,000	13	308
9,000	4,500	14	321
10,000	5,000	14	357

For a list of 1,000 items, the binary search is roughly 50 times faster than the sequential search, and for a list of 10,000 items, the binary search is roughly 376 times faster. This, however, does not take into

consideration that the binary search takes extra calculations (i.e., calculating the middle of the current portion of the list, discarding half of the list, etc). However, suppose that each binary search comparison takes 1/10 of a second and each sequential search comparison takes 1/1000 of a second. We can still observe that the binary search ridiculously outperforms the sequential search as illustrated in Figure 1.

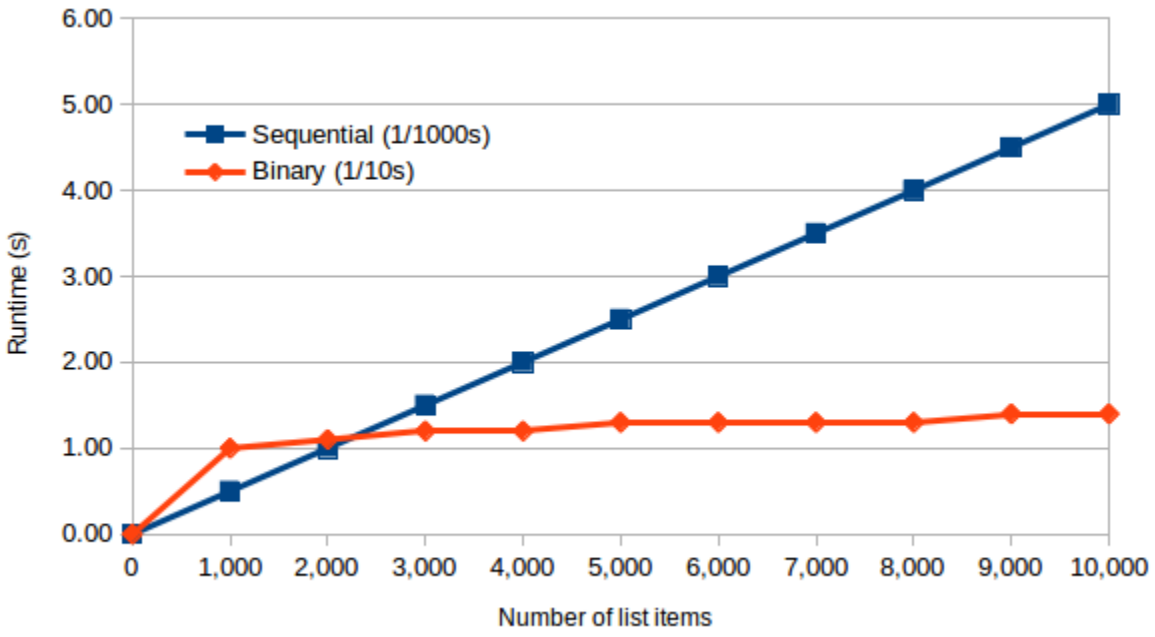
Also note that the time it takes to initially order the list of numbers is not considered. Since the binary search requires the list to be ordered, it is worthwhile to investigate various ordering methods and their performance.

Sorting

We have seen that the binary search is significantly faster than the sequential search. Why then do we even need to know about the sequential search? Why not use the binary search every time we need to search for something? What weaknesses does the binary search have? What scenarios can you imagine where a sequential search would be preferable to a binary search? The answer lies in the fact that a binary search can only be used when the data to be searched through is ordered in some meaningful way. This does not occur naturally (i.e., we must order the data first, prior to searching).

Humans seem to have a basic desire to structure and organize the world around them. One of the most basic ways of organizing a collection of objects is to arrange them according to some common characteristic such as size, weight, cost, or age. A group of objects is **sorted** when the objects are arranged according to some rule that involves the use of *orderable* **keys**. While the term *orderable* will not be rigorously defined, intuitively it means keys on which the concepts of *less than*, *greater than*, and *equal to* make sense, or on which *precedes* and *follows* have meaning. Orderable keys include numbers such as weight, height, age, income, and social security number. Strings of characters such as words, names, and addresses are also orderable. Numeric keys are generally arranged from smallest to largest, or, occasionally, largest to smallest.

Sorting is a very important problem. If information is not organized in some fashion, retrieval can be quite time consuming. To find a particular item in an unordered (or unsorted) list, we are forced to use



the sequential search rather than the more efficient binary search. While the sequential search approach may be acceptable for a small numbers of items, it becomes impractical when dealing with large numbers of objects. For this reason, computer scientists have focused a great deal of attention on the sorting problem and have devised a number of sorting algorithms. Over the years, these algorithms have been closely studied and the efficiency of each carefully analyzed.

This section presents three common approaches to the sorting problem: the **bubble sort**, the **selection sort**, and the **insertion sort**. As was the case with sequential and binary searches, each of the three sorts will be compared to determine their efficiency. In order to develop estimates of the runtimes of the three sort algorithms, we will use a computer capable of executing one million comparisons per second. While this may seem like a lot and to be very fast, today's computers are many times more powerful. In fact, they are capable of performing billions of basic low-level instructions per second. The number of higher-level operations they can perform each second depends on many factors other than CPU speed. In the case of sorting, the number of comparisons per second depends on factors such as whether the list being sorted is in main memory or on disk, and whether the computer is running other programs besides the sort procedure.

The take home message is not to place too much emphasis on the numbers that will be generated for predicted runtimes as they are highly dependent on the assumptions made concerning computing hardware. Instead, what is important is the *relative* performance of the algorithms, which is not affected by underlying hardware assumptions.

Bubble sort

The bubble sort processes a list of items from left-to-right. The sorted list is built, *in place*, from right-to-left; that is, the largest value in the list is placed in its final position first, and so on, in the same list. It works by repeatedly comparing two neighboring elements (i.e., in positions *i* and *i*+1). If the two neighboring elements are out of order, they are swapped. The algorithm advances the position *i* and repeats. If, at the end of one iteration of the entire list (also called a **pass**) any swaps were made, the process is repeated. If no swaps were made, it means the list is sorted, and the process terminates. The following is an example of the bubble sort on the list [7 9 3 5 1], with the underlined elements showing where the comparisons are made in each pass and red indicating the sorted portion of the list:

Pass 1		
List	**Comparison**	**Action**
7 9 3 5 1	7 < 9	no swap
7 9 3 5 1	9 > 3	swap
7 3 9 5 1	9 > 5	swap
7 3 5 9 1	9 > 1	swap
7 3 5 1 9	done with this pass	
Pass 2		
List	**Comparison**	**Action**
7 3 5 1 9	7 > 3	swap
3 7 5 1 9	7 > 5	swap
3 5 7 1 9	7 > 1	swap

3 5 1 7 9	done with this pass	
Pass 3		
List	**Comparison**	**Action**
3 5 1 7 9	3 < 5	no swap
3 5 1 7 9	5 > 1	swap
3 1 5 7 9	done with this pass	
Pass 4		
List	**Comparison**	**Action**
3 1 5 7 9	3 > 1	swap
1 3 5 7 9	done with the sort	

This sort is called the bubble sort because, during each pass, larger values *bubble* to the end of the list. Note that the bubble sort required four passes through the list of items. That is, it needed to go through some portion of the list starting at the beginning, compare values, and potentially swap values four times. In general for a list of *n* items, the bubble sort requires *n*-1 passes through the list. Did you notice that, as each pass was made, fewer and fewer elements in the list were compared. In the first pass, for example, four comparisons were made; in the second pass, three comparisons were made; in the third pass, two comparisons were made; and in the final pass, only one comparison was made. If we consider all passes in aggregate, on average about half of the values were compared; therefore, for a list of *n* items, an average of *n* /2 comparisons are made at each pass.

We can now calculate the number of comparisons made using the bubble sort as follows:

*comparisons = number of passes * average number of comparisons per pass*

As stated, for a list of *n* items, *n*-1 passes are required, each of which *on average* makes *n*/2 comparisons. In total, the number of comparisons made using the bubble sort is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For the list above of five items, the number of comparisons is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

In each pass of the bubble sort on the list in the example above, swaps were made. Let's take a look at an example where the list of items is almost already sorted:

Pass 1		
List	**Comparison**	**Action**
1 2 3 5 4	1 < 2	no swap
1 2 3 5 4	2 < 3	no swap

1 2 3 5 4	3 < 5	no swap
1 2 3 5 4	5 > 4	swap
1 2 3 4 5	done with this pass	
Pass 2		
List	**Comparison**	**Action**
1 2 3 4 5	1 < 2	no swap
1 2 3 4 5	2 < 3	no swap
1 2 3 4 5	3 < 4	no swap
1 2 3 4 5	done with the sort	

Since no swaps were made in pass 2, then the sort is immediately terminated. Terminating the bubble sort during a pass if no swaps are made is a *tweak* on the original bubble sort called the optimized bubble sort. Can you design an algorithm in pseudocode for the optimized bubble sort? Try it in the space below:

The method illustrated on the sort the list [7 9 3 5 1] in the table above is great for showing each comparison and illustrating the bubble sort in detail; however, it is overkill when simply showing how a list is sorted. We can slightly deviate from the method used previously and instead summarize each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	List
original list	7 9 3 5 1
1	7 3 5 1 9
2	3 5 1 7 9
3	3 1 5 7 9

4	1 3 5 7 9

Show how the bubble sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7
1	
2	
3	
4	
5	

Did you know?

Note the **for-next** construct in the pseudocode for the bubble sort above. It is used to repeat a *fixed* or *known* number of times. You have already seen the **repeat-until** construct that repeats until some condition is true. A *for-next* construct can be rewritten into a *repeat-until* construct as follows:

The *for-next* version:

```
for i ← 1..n
        ...
next
```

The *repeat-until* version:

```
i ← 1
repeat
        ...
        i ← i + 1
until i = n
```

It is not always possible to rewrite a repeat-until construct into a for-next construct; for example (assuming that the length of the list is unknown):

```
repeat
        remove item from list
until the list is empty
```

In this case, there is no way of knowing when the list will be empty if we don't originally know the length of the list (as assumed).

Selection sort

The selection sort is probably the most natural approach to sorting. Most people would likely come up with this algorithm when asked to design one that, for example, reorders a line of people based on their height. The selection sort processes a list of items from left-to-right. The sorted list is also built, in place, from left-to-right; that is, the smallest value in the list is placed in its final position first, and so on. Since the list is sorted in place, we keep track of both an unsorted portion and a sorted portion. And

since the list is built from left-to-right, then the sorted portion is on the left. The selection sort works by repeatedly finding the smallest value in the unsorted portion of the list and swapping it with the first value in the unsorted portion. This action increases the sorted portion of the list and shrinks the unsorted portion of the list by one value. This is repeated until there are no values in the unsorted portion of the list.

The following is an example of the selection sort on the list [7 9 3 5 1] with the underlined elements showing where the comparisons are made in each pass and red indicating the sorted portion of the list:

Pass 1			
List	**Comparison**	**Smallest so far**	**Action**
7 9 3 5 1	7 < 9	7	
7 9 3 5 1	7 > 3	3	
7 9 3 5 1	3 < 5	3	
7 9 3 5 1	3 > 1	1	
1 9 3 5 7	done with this pass		swap 7 and 1
Pass 2			
List	**Comparison**	**Smallest so far**	**Action**
1 9 3 5 7	9 > 3	3	
1 9 3 5 7	3 < 5	3	
1 9 3 5 7	3 < 7	3	
1 3 9 5 7	done with this pass		swap 9 and 3
Pass 3			
List	**Comparison**	**Smallest so far**	**Action**
1 3 9 5 7	9 > 5	5	
1 3 9 5 7	5 < 7	5	
1 3 5 9 7	done with this pass		swap 9 and 5
Pass 4			
List	**Comparison**	**Smallest so far**	**Action**
1 3 5 9 7	9 > 7	7	
1 3 5 7 9	done with the sort		swap 9 and 7

Note that the selection sort also required four passes through the list of items. For a list of *n* items, the selection sort requires *n*-1 passes through the list. Again, note that as each pass was made, fewer and fewer elements in the list were compared. If we consider all passes in aggregate, on average about half of the values were compared; therefore, for a list of *n* items, an average of *n* /2 comparisons are made at each pass.

We can now calculate the number of comparisons made by using the formula defined above:

*comparisons = number of passes * average number of comparisons per pass*

For a list of *n* items, *n*-1 passes are required, each of which *on average* makes *n*/2 comparisons. In total, the runtime performance of the selection sort is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For the list above of five items, the number of comparisons is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

Again, we can show how the selection sort works by summarizing each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	List
original list	7 9 3 5 1
1	1 9 3 5 7
2	1 3 9 5 7
3	1 3 5 9 7
4	1 3 5 7 9

This is sufficient to show how the selection sort works. Can you design an algorithm in pseudocode for the selection sort? Try it in the space below:

Show how the selection sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7
1	
2	
3	
4	
5	

Insertion sort

The insertion sort is the procedure that most people use to arrange a hand of cards. To begin to understand the algorithm, think of the list as having both a sorted portion and an unsorted portion (as we did in the previous sorts). The first item of the list is considered to be a sorted list one item long, with the rest of the list (items 2 through *n*) forming an unsorted portion.

The insertion sort removes the *first* item from the unsorted portion of the list and marks it as the item to be inserted. It then works its way from the *back* to the *front* of the sorted portion of the list, at each step comparing the item to be inserted with the current item. As long as the current item is larger than the item to be inserted, the algorithm continues moving *backward* through the sorted portion of the list. Eventually it will either reach the beginning of the sorted portion of the list or encounter an item that is less than or equal to the item to be inserted. When that happens the algorithm inserts the item at the current insertion point.

The entire process of selecting the first item from the unsorted portion of the list and scanning backwards through the sorted portion of the list for the insertion point is then repeated. Eventually, the unsorted portion of the list will be empty since all of the items will have been inserted into the sorted portion of the list. When this occurs, the sort is complete.

The following is an example of the selection sort on the list [7 9 3 5 1] with the underlined elements showing where the comparisons are made in each pass, the bold value indicating the current *first* item in the unsorted portion of the list, and red indicating the sorted portion of the list:

Pass 1			
List	**First item**	**Comparison**	**Action**
7 3 5 1	9	7 < 9	
7 9 3 5 1		done with this pass	insert 9
Pass 2			
List	**First item**	**Comparison**	**Action**
7 9 5 1	3	9 > 3	slide 9 over
7 9 5 1	3	7 > 3	slide 7 over
3 7 9 5 1		done with this pass	insert 3

Pass 3			
List	**First item**	**Comparison**	**Action**
3 7 9 1	5	9 > 5	slide 9 over
3 7 9 1	5	7 > 5	slide 7 over
3 7 9 1	5	3 < 5	
3 5 7 9 1		done with this pass	insert 5
Pass 4			
List	**First item**	**Comparison**	**Action**
3 5 7 9	1	9 > 1	slide 9 over
3 5 7 9	1	7 > 1	slide 7 over
3 5 7 9	1	5 > 1	slide 5 over
3 5 7 9	1	3 > 1	slide 3 over
1 3 5 7 9		done with the sort	insert 1

Again, we can show how the insertion sort works by summarizing each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	List
original list	7 9 3 5 1
1	7 9 3 5 1
2	3 7 9 5 1
3	3 5 7 9 1
4	1 3 5 7 9

Show how the insertion sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7
1	
2	
3	
4	
5	

Note that the insertion sort required four passes through the list of items. For a list of *n* items, the insertion sort requires *n*-1 passes through the list, which is the same as both the bubble sort and the selection sort. However, the number of comparisons made at each pass is not so simple to calculate because it depends on the original state of the list. Sometimes, the *first item* in the unsorted portion of

the list will be greater than everything in the sorted portion of the list, and therefore only require one comparison (to the last value in the sorted portion of the list). At other times, it may actually be the smallest value in the entire list and thus be compared to every value in the sorted portion of the list.

Since the number of comparisons made in the insertion sort is not so straightforward, we will need to determine formulas that represent the number of comparisons for the worst case, best case, and average case.

In the best case, only a single comparison will be made at each pass (i.e., the first item in the unsorted portion of the list is in its proper place in the list). This will happen, for example, if the list is already sorted. For a list of n items, the insertion sort requires n-1 passes, and for each pass, one comparison is made in the best case. The total number of comparisons in the best case is then:

$$(n-1)*1 \;=\; n-1$$

For the list above of five items, the number of comparisons in the best case is then four (5 – 1 = 4). Here is an example on the already sorted list [1 2 3 4 5]:

Pass 1			
List	**First item**	**Comparison**	**Action**
1 3 4 5	2	1 < 2	
1 2 3 4 5		done with this pass	insert 2
Pass 2			
List	**First item**	**Comparison**	**Action**
1 2 4 5	3	2 < 3	
1 2 3 4 5		done with this pass	insert 3
Pass 3			
List	**First item**	**Comparison**	**Action**
1 2 3 5	4	3 < 4	
1 2 3 4 5		done with this pass	insert 4
Pass 4			
List	**First item**	**Comparison**	**Action**
1 2 3 4	5	4 < 5	
1 2 3 4 5		done with the sort	insert 5

In the worst case, the first item belongs at the beginning of the sorted portion of the list and therefore is compared to every value in the sorted portion of the list. In the first pass, only one comparison is made; in the second pass, two comparisons are made; and so on. On average, then, $n/2$ comparisons are made per pass. For a list of n items, the insertion sort requires n-1 passes, and for each pass, an average of $n/2$ comparisons are made in the worst case. The total number of comparisons in the worst case is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For a list of five items, the number of comparisons in the worst case is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

In the first example shown with the list [7 9 3 5 1], the worst case indeed occurred (and 10 comparisons were made). Note that the list is nearly reverse sorted!

For the average case, we simply need to calculate the average **of the average** number of comparisons made in the best and worst cases. In the best case, only one comparison is made per pass; in the worst case, *n*/2 comparisons are made. The average is then *n*/4 comparisons. For a list of *n* items, the insertion sort requires *n*-1 passes, and for each pass, an average of *n*/4 comparisons are made in the average case. The total number of comparisons in the average case is then:

$$(n-1)*\frac{n}{4} = \frac{1}{4}n(n-1) = \frac{1}{4}(n^2-n)$$

For a list of five items, the number of comparisons in the average case is then:

$$\frac{1}{4}(5^2-5) = \frac{1}{4}(20) = 5$$

To summarize, an insertion sort of *n* items always requires exactly *n*-1 passes through the sorted portion of the list. What varies is the number of comparisons that must be performed per pass. The best case requires only one comparison per pass and occurs when attempting to sort a list that is already sorted. The worst case requires an average of *n*/2 comparisons per pass and occurs when the list is already sorted in *reverse* order. In the average case, only one half of the items in the sorted portion of the list will be examined during each pass before the insertion point is found – giving *n*/4 comparisons per pass.

Comparing sorts

When comparing the insertion sort to other sorts, generally the average case formula is used since this represents the expected performance of the algorithm. Occasionally, knowledge of the worst case behavior of the algorithm is also important. Understanding this behavior is useful when attempting to determine or limit the maximum amount of time a computing system will take to reach an answer, even in the worst case. Such behavior is important in real time applications such as airplane flight control systems.

The bubble sort and the selection sort always require exactly $1/2(n^2-n)$ comparisons to sort *n* items. In the worst case, the insertion sort also requires $1/2(n^2-n)$ comparisons. In the average (expected) case, however, the insertion sort requires $1/4(n^2-n)$ comparisons, and therefore requires about one half of the comparisons needed by the bubble and selection sorts. Figure 2 shows the runtime comparison of the three sorts, considering a machine capable of performing 1 million comparisons per second. The smaller number of comparisons needed by the insertion sort means that it is generally a faster algorithm than the bubble or selection sorts, assuming a comparison takes the same amount of time in both algorithms (a reasonable assumption). The insertion sort will be expected to process a 10,000 item list

in about 25 seconds (precisely 24.9975 seconds). The bubble and selection sorts are both expected to take about 50 seconds on the same problem (precisely 49.995 seconds) – or about twice as long.

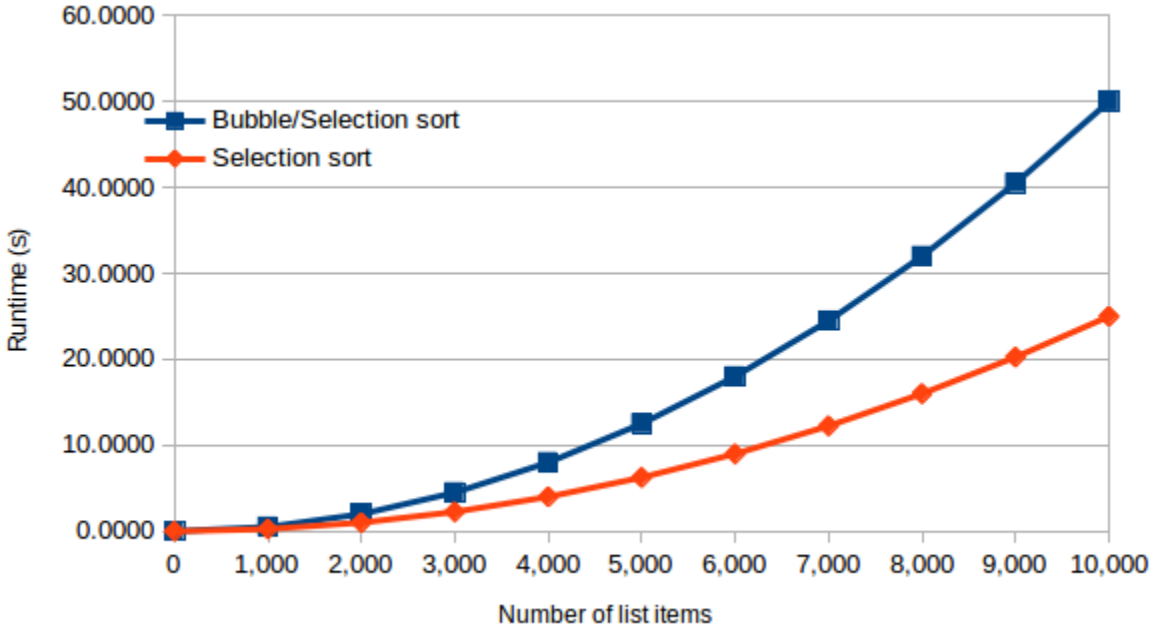


Figure 2: Runtime comparison of bubble, selection, and insertion sorts