CS50 Linear Search

Overview

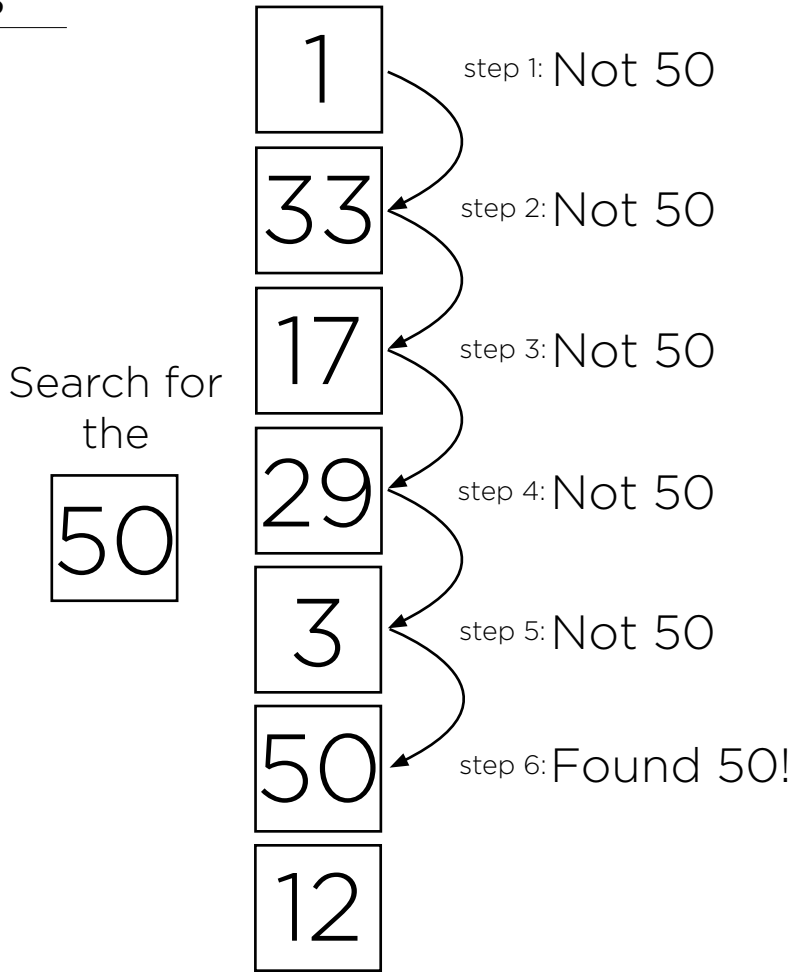
There are many different **algorithms** that can be used to search through a given list. One such algorithm is called **linear search**. This algorithm works by simply checking every element in the list in order. It will start at the beginning of the list and increment through the list until the desired element is found. In this way, linear search would require checking every **element** before reaching the conclusion that the element does not exist in the list.

Key Terms

- algorithm
- linear search
- element
- computational complexity
- constant

Efficiencies and Inefficiencies

While the linear search algorithm is correct, it is almost never the most efficient. The worst, or least efficient, case would be when the desired element is the last element in the list or not in the list at all (both have the same efficiency). We would have to look through every element in the list to find the one we're looking for. This would take *n* steps, where *n* is the length of the list. The **computational complexity** of linear search would be O(*n*). That may seem pretty daunting, especially if we have a list that has millions of elements. However, the best case scenario is much better; if the desired element is the first in the list, we would find it in one step. Although linear search is not usually the most efficient, linear search can be useful in certain situations. Take for instance a list that you know nothing about, like a stack of papers that are out of order. It is just as efficient to search this stack linearly as it would be to search for elements randomly. In this case, we cannot search for elements in a more efficient way since we don't know anything about the list. There is no information about the list's organization for us to leverage. Now you can see why it would seem rather convenient to sort a list before searching it. Granted, sorting also takes up time and space, but this additional step will save you some time if you plan on searching a list multiple times or if you have a very large list.



Examples of Linear Search

Recall the phone book example. While looking for Mike Smith, linear search meant going through each page of the phone book one at a time. Even the algorithm where we flipped through two pages at a time can be considered linear. In fact, any algorithm in which the there is a **constant** being multiplied by the total number of elements in the list to determine the search time, is considered linear. For example, if you are turning three pages of the phone book at a time (remembering to make the correction if you overshoot), the search time would be *n/3*. Since 1/3 is being multiplied to *n*, this algorithm would be considered linear. Linear search should typically be avoided, as there is usually a better algorithm that can implemented to make the search more efficient. Sorting a list first, is one such way to search more quickly. Once a list is sorted we are able to leverage some of the concepts learned earlier to make a faster, efficient, and more elegant program.

CS50 Binary Search

Overview

There are many different **algorithms** that can used to search through a given array. One option is **linear search**, but it can be a rather lengthy process. Luckily, there is a faster searching algorithm: **binary search**. You might recall that binary search is similar to the process of finding a name in a phonebook. This algorithm's speed can be leaps and bounds better than linear search, but not without a cost: binary search can only be used on data that is already sorted.

Key Terms

- algorithms
- linear search
- binary search
- pseudocode

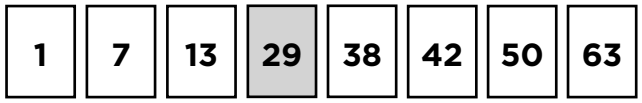
The Binary Search Algorithm

The basis of binary search relies on the fact that the data we're searching is already sorted. Let's take a look at what the binary search algorithm looks like in **pseudocode**. In this example, we'll be looking for an element *k* in a sorted array with *n* elements. Here, `min` and `max` have been defined to be the array indices that we are searching between, marking the upper and lower bounds of our search.

```
1  set min = 0 and max = n - 1
2  find middle of array
3  if k is less than array[middle]
4      set max to middle - 1
5      go to line 2
6  else if k is greater than array[middle]
7      set min to middle + 1
8      go to line 2
9  else if k is equal to array[middle]
10     you found k in the array!
11 else
12     k is not in the array
```

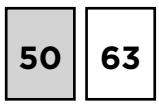
Using the algorithm described above, let's search for the number `50` in the array on the right. First we set the `min = 0` and `max = 7`. Next, calculate the `middle` of the array. Well, how do we decide whether to pick the array index `3` or `4` as the `middle`? It actually does not matter, as long as the algorithm is consistent. For our algorithm, let's choose `3`. We've now determined that the `middle` element is `array[3]`, or `29`. Since `50` is larger than `29` and our array is sorted, we know that `50` will not be on the left of `29`. Therefore, there is no need to check those elements. To continue the search, `min` is set to `4`, `max` remains at `7`, and we repeat the process!

Find the 50!

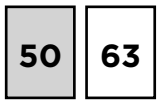


min = 0; max = 7; middle element = 29

min = 4; max = 7; middle element = 42



min = 6; max = 7; middle element = 50



50 is found!

Binary Search vs. Linear Search

In computer science, it is a common theme that whenever we make some improvement, it is at the cost of another factor. In this case, we trade the speed of a searching algorithm for the time it takes to sort the array. In some cases, it would be faster to just use linear search rather than to sort the data and then use binary search. Nevertheless, binary search is useful if you plan on searching the array multiple times. In case an element does not exist in the array, linear search would iterate through the entire array – however long it may be – to know that the given element does not exist in the array. In binary search, we can be more efficient. Using the algorithm described in our pseudocode, if the searched element is less than the initial value at `min` or larger than the initial value at `max` (the first and last elements of the array, respectively), the algorithm knows the element is not in the array.

CS50 Computational Complexity

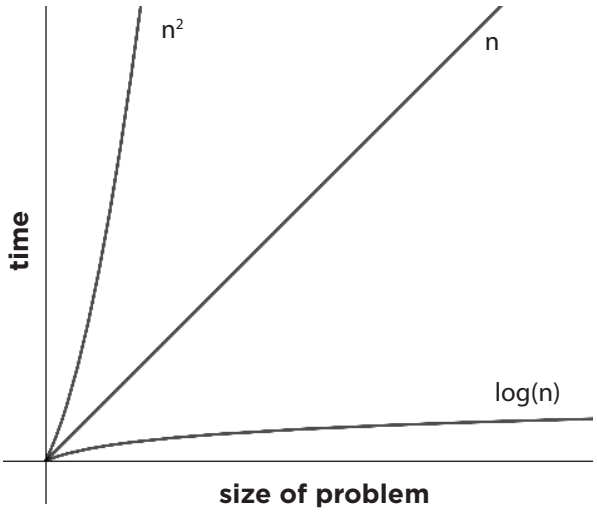
Overview

Key Terms

- computational complexity
- Big O
- Big Ω

The subject of computational complexity (also known as time complexity and/or space complexity) is one of the most math-heavy topics, but also perhaps one of the most fundamentally important in the real-world. As we begin to write programs that process larger and larger sets of data, analyzing those data sets systematically, it becomes increasingly important to understand exactly what effect those algorithms have in terms of taxing our computers. How much time do they take to process? How much RAM do they consume? One aspect of **computational complexity** is the amount of time an algorithm takes to run, in particular considering the theoretical worst case and best case scenarios when running programs.

Computational Complexity Notation



Big O notation, shorthand for "on the order of", is used to denote the worst case efficiency of algorithms. Big O notation takes the leading term of an algorithm's expression for a worst case scenario (in terms of *n*) without the coefficient. For example, for linear search of an array of size *n*, the worst case is that the desired element is at the end of the list, taking *n* steps to get there. Using Big O notation, we'd say linear search runs in *O(n)* time. We can calculate the computational complexity of bubble sort in the same way, albeit with a little more math. Remember that bubble sort involved comparing things by pairs. In a list of length *n*, *n - 1* pairs were compared. For example, if we have an array of size 6, we would have to compare `array[0]` and `array[1]`, then `array[1]` and `array[2]`, and so on until `array[4]` and `array[5]`. That's 5 pairs for an array of size 6. Bubble sort ensures that after *k* passthroughs of the array, the last *k* elements will be in the correct location. So in the first passthrough there are *n-1* pairs to compare, then on the next passthrough only *n-2* comparisons and so forth until there is only 1 pair to be compared. In math $(n-1) + (n-2) + \ldots + 1$ can be simplified to $n(n-1)/2$ which can be simplified even further to $n^2/2 - n/2$. Looking at the expression $n^2/2 - n/2$, the leading term would be $n^2/2$, which is the same as $(1/2)\ n^2$. Getting rid of the coefficient, we are left with n^2. Therefore, in the worst case scenario, bubble sort is on the order of n^2, which can be expressed as $O(n^2)$. Similar to big O, we have **big** Ω (omega) notation. Big Ω refers to the best case scenario. In linear search, the best case would be that the desired element is the first in the array. Because the time needed to find the element does not depend on the size of the array, we can say the operation happens in constant time. In other words, linear search is **Ω(1)**. In bubble sort, the best case scenario is an already sorted array. Since bubble sort only knows that a list is sorted if no swaps are made, this would still require *n-1* comparisons. Again, since we only use the leading term without the coefficients, we would say bubble sort is *Ω(n)*.

Comparing Algorithms

Big O and big Ω can be thought of as upper and lower bounds, respectively, on the run time of any given algorithm. It is now clear to see which algorithms might be better to use given a certain situation. For instance, if a list is sorted or nearly sorted, it would not make sense to implement a selection sort algorithm since in the best case, it is still on the order of n^2, which is same as it's worst case run time. Binary search may seem to be the fastest search but it is clear to see that searching a list once with linear search is more efficient for a one time search, since binary search run requires a sort algorithm first, so it could take $O(log(n)) + O(n^2)$ to search a list using binary if the list is not already sorted.

Algorithm	Big O	Big Ω
linear search	O(n)	Ω(1)
binary search	O(log(n))	Ω(1)
bubble sort	$O(n^2)$	Ω(n)
insertion sort	$O(n^2)$	Ω(n)
selection sort	$O(n^2)$	$\Omega(n^2)$

CS50 Bubble Sort

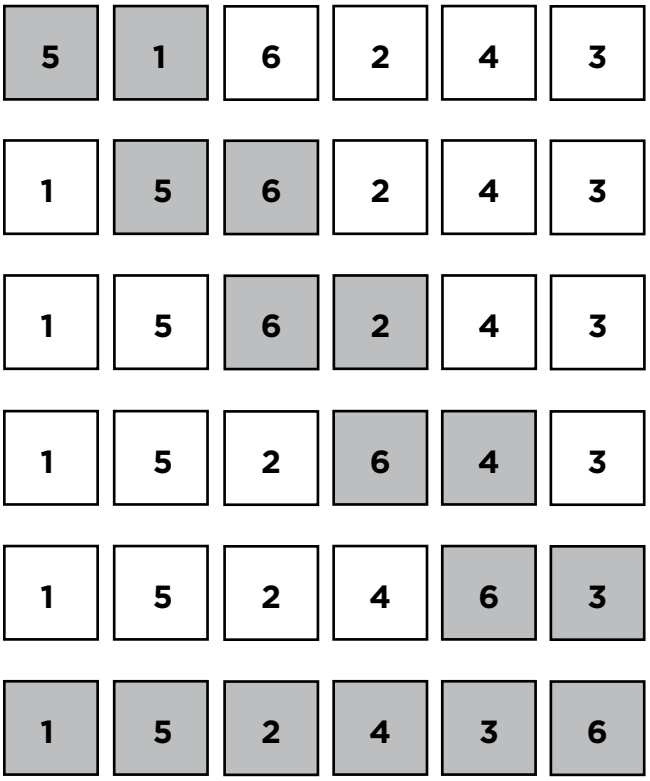
Overview

There are limited ways to search a list that is unsorted. It is often more efficient to sort a list and *then* search it. One of the most basic sorting algorithms is called **bubble sort**. This algorithm gets its name from the way values eventually "bubble" up to their proper position in the sorted array. This basic approach to sorting narrows the scope of our problem to focusing on ordering just two elements at a time, instead of an entire **array** at a time. This approach is very straightforward, but possibly at the expense of making an inordinate number of swaps just to put one single element into position.

Key Terms

- bubble sort
- array
- pseudocode

Step-by-step process of 1 pass through in bubble sort



Implementation

Bubble sort works by comparing two adjacent numbers in a list, and swapping them if they are out of order. Looking at the example on the left, if we are given an array of the numbers 5, 1, 6, 2, 4, and 3 and we wanted to sort it using bubble sort our **pseudocode** for one single pass might look something like this:

```
for every element in the array
        check if element to the right is smaller
                if so swap the two elements
        else move on to the next element in the list
```

When this is implemented on the example array, the program would start at 5 and compare it with 1. Since 1 is smaller than 5 it would swap them. It would then move on to compare 5 and 6 since those are in the correct order, we just move on to the next element. Next 6 and 2 are compared, and so on.

Finally after doing this for all the elements in the array we are left with the array 1, 5, 2, 4, 3, and 6. It's not completely sorted, but notice that after the first passthrough, the 6 is already in its correct location. After n passthroughs the last n elements are in their correct position. This fact can be used to optimize this algorithm since it is not necessary to look at those correctly sorted elements. It is this effect of the larger elements "bubbling" to the right side that gives this algorithm its name!

Sorted Arrays

If bubble sort was implemented only as above, we would only go through one passthrough, but as the example shows, it is not guaranteed that the array will be sorted after one pass. So how many times should this algorithm be run? Well in the worst case scenario, a reverse sorted list (6, 5, 4, 3, 2, 1), it might need to run 5 times. Indeed the same would hold true for *n* elements, the algorithm might need to run *n-1* times. That seems wasteful though, since it would only need to run that maximum number of times if the array is a "worst case scenario" (more on that in the time complexity module).

How can you ensure you only run this algorithm the necessary amount of times, maybe saving a few steps? Well, if this algorithm is run and no swaps are made, it must be true that the array is sorted (think about it)! Maybe then it would make sense to amend our implementation to include a counter for the amount of swaps made. If `counter == 0`, then the array is sorted, however if `counter > 0`, then more passthroughs are needed to sort the array. Now we only decide at the end of every passthrough whether more passthroughs are necessary!

CS50 Selection Sort

Overview

Sorted arrays are typically easier to search than unsorted arrays. One algorithm to sort is bubble sort. Intuitively, it seemed that there were lots of swaps involved; but perhaps there is another way? **Selection sort** is another sorting algorithm that minimizes the amount of swaps made (at least compared to bubble sort). Like any optimization, everything comes at a cost. While this algorithm may not have to make as many swaps, it does increase the amount of comparing required to sort a single element.

Key Terms

- selection sort
- array
- pseudocode

Implementation

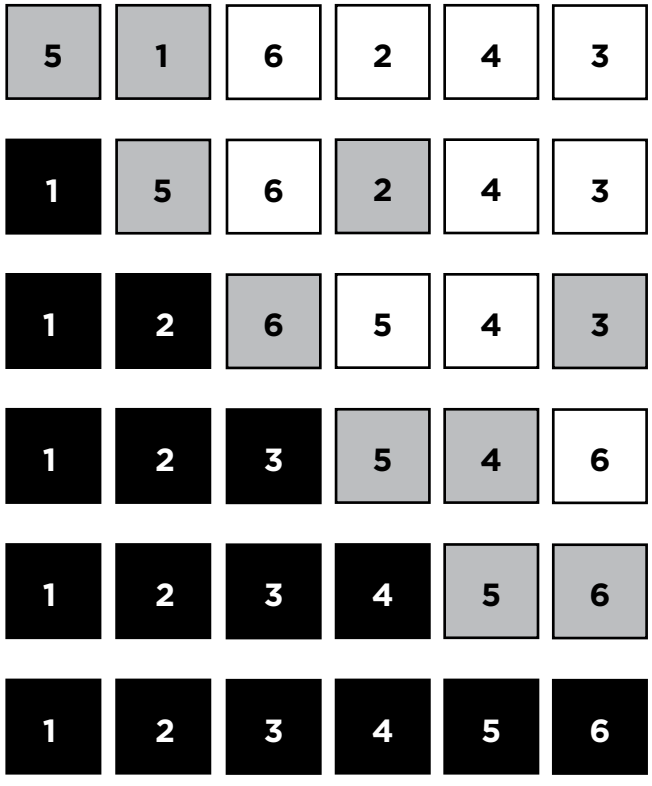
Selection sort works by splitting the array into two parts: a sorted **array** and an unsorted array. If we are given an array of the numbers 5, 1, 6, 2, 4, and 3 and we wanted to sort it using selection sort, our **pseudocode** might look something like this:

```
repeat for the amount of elements in the array
       find the smallest unsorted value
       swap that value with the first unsorted value
```

When this is implemented on the example array, the program would start at `array[0]` (which is 5). We would then compare every number to its right (1, 6, 2, 4, and 3), to find the smallest element. Finding that 1 is the smallest, it gets swapped with the element at the current position. Now 1 is in the sorted part of the array and 5, 6, 2, 4, and 3 are still unsorted. Next is `array[1]`, 5 is our current element and we need to check all elements to the right to check for the smallest (note that by only checking to the right we are only looking at the unsored array). Finding that 2 is the smallest element of the unsorted array we swap it with 5, and so on.

Notice that in the second-to-last iteration, we can clearly see that the array is now sorted, but a computer cannot look at the larger picture like we can. It can only process the information directly in front of it, and so therefore, we continue the process. Once 5 is recognized as the smallest element of the unsorted array, only then can the algorithm be stopped, since the "unsorted" array is of size one and any list of size 1 is necessarily sorted.

Step-by-step process for selection sort



Sorted Arrays

Unlike bubble sort, it is not necessary to keep a counter of how many swaps were made. To optimize this algorithm, it might seem like a good idea to check if the entire array is sorted after every successful swap to avoid what happened in the last two steps of the pseudocode above. This process too, comes at a cost, that is even more comparisons that we have to make. We are guaranteed though, that no matter the order of the original array, a sorted array can be formed after *n-1* swaps, which is significantly fewer than that of bubble sort. In selection sort, in the worst case scenario, n^2 comparions and *n-1* swaps are made. Unfortunately, that's also the case in the best-case scenario!

CS50 Insertion Sort

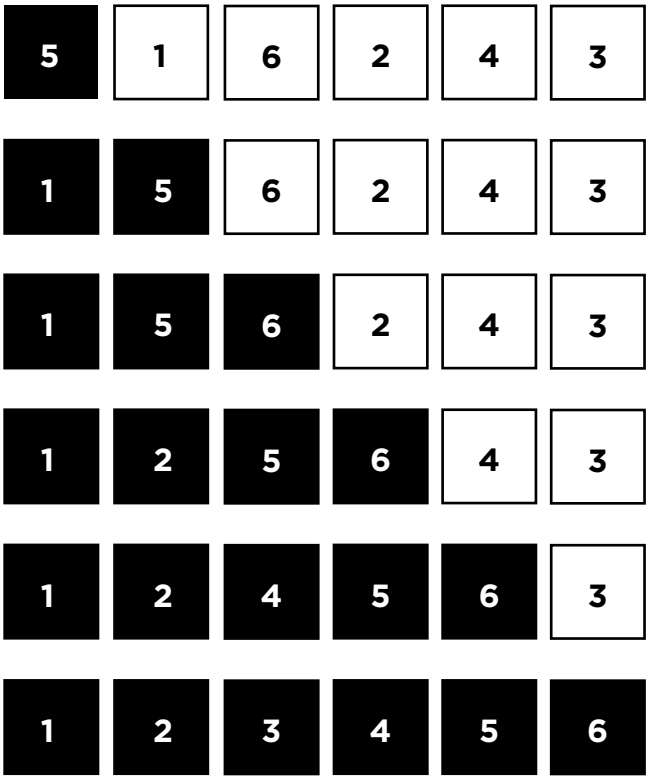
Overview

Insertion sort is yet another algorithm to sort arrays, but this time it does not require multiple iterations over the **array**. Like usual, optimizations usually force the programmer to sacrifice something else. However these sacrifices are nearly negligible, in the case of insertion sort, when the array is small or the array is nearly sorted. Similar to selection sort, the array of elements will be split into two parts: a sorted portion and an unsorted portion.

Key Terms

- insertion sort
- array
- pseudocode

Step-by-step process for insertion sort



Implementation

In the case of having two elements in the array, the implementation is relatively simple. Consider the first element to be automatically in the sorted portion of the list. Look at the next element in the array, and determine where it fits in the sorted list. This can be applied to a larger array with the following **pseudocode**:

```
for each unsorted element, n, in the array

    determine where in the sorted portion of the
    array to insert n

    shift sorted elements rightwards as necessary
    to make room for n

    insert n into sorted portion of the list
```

When this is implemented on the example array, the program would start at `array[1]`, which is 1, since an array of size one (`array[0]`) is already sorted. 5 would get shifted over to the right, and 1 would be moved to `array[0]`. Next, the program looks at 6. 6 is greater than 5 so no elements need to be shifted to make room for it. And so on and so forth. Eventually with this procedure, the entire array will be sorted.

Sorted Arrays

There is no guarantee that after any step in the implementation, any elements are in the correct location in the array. Even if an element did happen to be in its correct location in the initial array, it is likely that it would be moved to a different location within the sorted array before it would be shuffled back into its final location. Also note that in insertion sort, all the elements to the right of the newest element in the sorted list have to be shifted over one space. So while it may seem like insertion sort involves n steps, we are increasing the amount of times an element is being moved because elements are being shifted over to accomodate new elements rather than just being swapped. This is, however, dependent on the order of the initial array. It is also worth considering that sorting algorithms need to address cases in which two elements are equal. When dealing with few unique elements the key is just to be consistent. If, in the implementation of insertion sort, there was a line that checked if the current element was equal to an element in the sorted array, it should always have the same outcome, whether it is to the right or the left of that element is irrelevant.

CS50 Recursion

Overview

Recursive solutions to problems are typically contrasted with iterative ones. In a **recursive solution**, a function (or a set of functions) repeatedly invokes slightly modified instances of itself, with each subsequent instance tending closer and closer to a base case. In the meantime, the intermediate calls are all left waiting, having "passed the buck" to a downstream call to give it the answer it needs. Recursive procedures, when contrasted with iterative ones, can sometimes lead to incredibly efficient, elegant, and, some might even say, beautiful solutions.

Key Terms
- recursive solution
- iterative solution
- base case
- recursive case
- call stack
- active frame

Recursion versus Iteration

Recursive solutions can often replace clunkier iterative ones. One great example of this is with programs that calculate the factorial of a number. Remember that "n factorial," or `n!`, simply represents the product of all integers less than and including `n`. So `3!` would be six (`3 * 2 * 1`). Consider the two approaches on the right for implementing a function to find the factorial of an integer. The first implementation looks familiar. This is known as an **iterative solution** as we are iterating through a loop, substituting in different values for `i`. In this solution, we have declared two variables (`product` and `i`), while in the recursive solution, we have not declared any. Also, note that `recurse()` is fewer lines shorter than `iterate()`.

```
int iterate(input)
{
    int product = 1;
    for(int i = input; i > 0; i--)
    {
        product *= i;
    }
    return product;
}
```

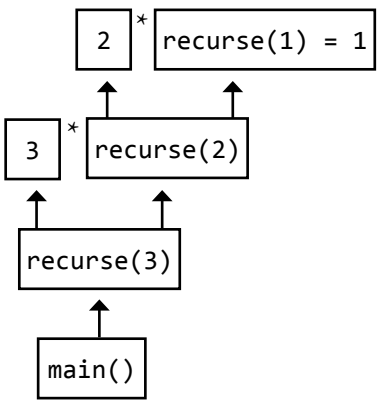
```
int recurse(input)
{
    if(input == 1)
    {
        return 1;
    }
    return input * recurse(input - 1);
}
```

Implementation

Recursive solutions to problems are made up of two parts: the base case and the recursive case. The **base case** is what allows us to break out of an infinite loop. Without a base case our program would continue to run until it no longer had the space to do so and resulted in a segmentation fault. In our example, the base case is when input == 1. The **recursive case** is where the function invokes itself. This appears in the last line of `recurse()`, where recurse is called again. In this way, recurse repeatedly calls itself until `1` is the value being passed into the function.

Call Stack Representation

When a recursive function (or any function for this matter) is called, it creates a new frame on the stack. Every subsequent function called within `main()` is created on top of the previous frame. This stack, where all of our function calls exist, is called the **call stack**. This means that the function at the top of the stack is the most recently called function. We call this the **active frame**. Say we pass in 3 as the input for `recurse()`, then `main()` will call `recurse(3)`, which will call `recurse(2)`, and so on. This process will continue to occur until the base case is met. Once this happens, that return value trickles down and is plugged back into the function calls left open in the call stack. In our example, 1 would be plugged into `recurse(1)`, destroying this frame in the call stack and leaving `recurse(2)` as the active frame. The number 2 would be passed into `recurse(2)` in the same way and so on until `recurse(3)` returned 6 and passed that back to `main()`. At this point, `main()` would be the only function left in the call stack, since all the other calls to `recurse()` would have been destroyed after returning a value.



Note that in our sample iterative solution above, there would only be one function called, `iterate()`. So in some ways iterative solutions are simpler than recursive ones. And since iterative solutions can usually solve the same types of problems as recursive ones, there will almost never be a real world problem that requires us to use recursion as a means to solve it. Rather, recursion can be used to make our code more elegant and efficient.

CS50 Merge Sort

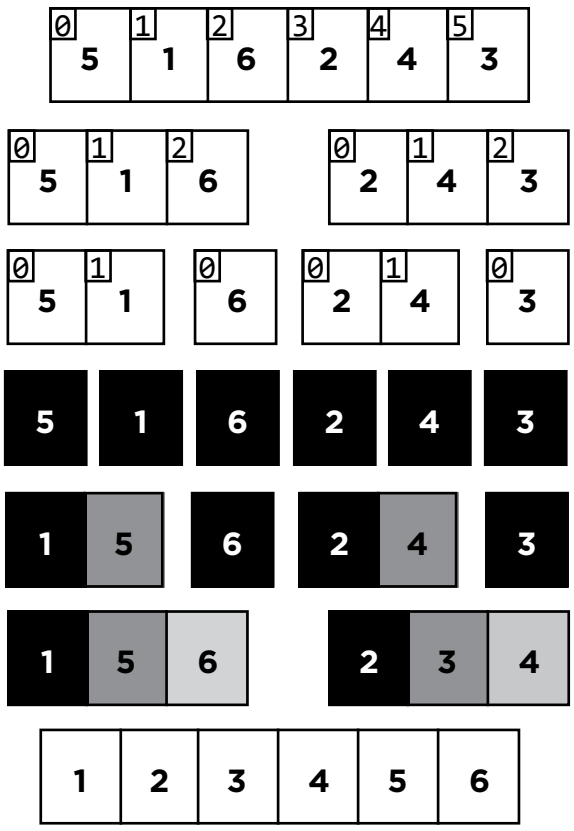
Overview

Sorting algorithms like selection sort, insertion sort, and bubble sort all suffer from the same general limitations and thus have the same worst-case runtime of $O(n^2)$. **Merge sort**, on the other hand, is fundamentally different, leveraging recursion to "pass the buck" of sorting, accomplishing a drastically superior runtime: *O(n log n)*!

Key Terms

- merge sort
- array
- recursive
- pseudocode

Step-by-step process for merge sort



Implementation

Merge sort works by breaking an **array** into sub arrays and merging the subarrays back in a **recursive** way. To understand how this works, let's take a look at the following **pseudocode**:

```
merge sort:
     if number of elements < 2
          return
     else
          sort the right half
          sort the left half
          merge sorted halves
```

Using the lines above and the array on the left (containing these numbers: 5 1 6 2 4 3), we are going to sort the left and right halves of the elements and merge them together. Note that when running merge sort, we only need enough space to store two copies of the array, despite the fact that the diagram on the left appears to require more space. At this point, we have no way of sorting the right or left halves, so we are going to recursively call the merge sort function. Similarly, we are going to continue to do this until we are left with all arrays of size 1. We'll need to handle running into an odd number of elements in a consistent way. Here, we implemented our program such that the left side of the split will have one more element than the right if the array has an odd number of elements.

After the elements are broken down into arrays of size 1, we are able to merge the sorted halves, since any array of size 1 is considered sorted. When we merge the two halves, we are removing the smallest numbers from the subarrays and appending them to the merged array, repeating until all elements of both subarrays are used up. (Note: The smallest elements will always be at the beginning of the subarrays, so we only need to check the first elements in the respective subarrays.) Since 6 was a single element array in the previous iteration, it does not need to be merged. We continue to do this until all the right and left halves are sorted from the previous iteration. Upon the next iteration, when we merge the arrays back into arrays of size 3, we need only look at the 0th index of each subarray to find the smallest element of the newly-merged array. In this case, this would be 1 and 6 for the left half and 2 and 3 for the right. Since 1 and 2 are the lowest numbers of their respective sides, they go into the 0th indices of the newly-merged array. And we'll continue to merge arrays in this way until the array is fully sorted.

Sorted Arrays

Like selection sort, merge sort has the same runtime in the best and worst case scenarios. Consider running merge sort on an already sorted array: since our program would have no way of knowing that it had already been sorted, it would have carry out the entire process the same way that it would with an unsorted array.