**Sorting Algorithms**

**6.1 Bubble Sorting Algorithms**

**About**

* Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
* During each single iteration. Each two values are compared and the larger one will move to the left. Then the largest value in the whole array will move all the way to the left.

**Example1**

* Sort the following array -> A = [y,x,e,r,s,f]
* Then will go over each number and apply the following condition
* If y is larger than x swap them, so that x is first, and y is second
* The algorithm will keep doing that until it completes the full array. Then goes over the whole array again, until the whole array is sorted. Therefore, you may go over the while array multiple times
* During each iteration. Each two values are compared and the larger one will move to the left. Then the largest value in the whole array will move all the way to the left.

**Example2**

* In the below image each blue highlight represents an action that this algorithm is doing for the array [-2,45,0,11,-9] to sort it
* During each iteration. Each two values are compared and the larger one will move to the left. Then the largest value in the whole array will move all the way to the left.

Table

Description automatically generated with medium confidence

**The first Iteration**

. Table

Description automatically generated with medium confidence

**All iterations**

Graphical user interface, text, application, email

Description automatically generated

**1 pass**

* During the first iteration. You compare each element to the element next to it and move the larger element to the left. Therefore, nothing moved until we got to element 22 which was swapped with element 4. Then, in the same pass element 22 was also swapped with 3. Then we complete the first iteration when we complete the full row.

Graphical user interface, text, application, email

Description automatically generated

**Also 1 pass**

* During the first iteration. You compare each element to the element next to it and move the larger element to the left. Therefore, we start by swapping 10 by 5 and 10 by 7. Then we could not swap 10 by 14 because 14 is larger than 10. Then in the same pass we could not swap 14 by 25, because 25 is larger. Then we went to 25 and we were able to swap it by 6. Because its larger. Then we complete the first iteration when we complete the full row.
* During each iteration. Each two values are compared and the larger one will move to the left. Then the largest value in the whole array will move all the way to the left.

**Run Time**

* This algorithm is very slow. Because the worst-case scenario when the whole array is opposite to a sorted order. This algorithm will need to complete the elements n number of times and move the elements n number of times to complete the sorting and its sorting run time will be O(n)\*O(n) = O(n^2).
* Our best-case scenario we have is when the whole array is sorted. Even then we will need to cover the full array and our run time will be O(n) -> Ω(n)
* The worst-case scenario will also be O(n^2)
* The average run time of this algorithm is O(n^2).
* Note. When representing the best-case scenario do not type O(n) type sigma to the n-> Ω(n)

**Pros**

* This algorithm is very easy to implement

**Cons**

* This algorithm is very slow, and it is the most inefficient algorithm. The reason why it is has an O(n^2) worst case, is because its code involves a nested for loop with always produce n^2

**6.2 Selection Algorithms Introduction**

* Sorting is very important for arrays, because it can improve our run time from O(n) to O(1)
* There are multiple different sorting algorithms
* The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order).
* During each iteration. The algorithm covers the whole array. The algorithm begins by picking the first element it sees and compares it to every element in the array.
* This algorithm swaps the elements. From their locations. Then technically moves the small elements to a new sub-array. That sub-array is just the start of the main array

**Example**

* This algorithm

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

* The first pass you cover the entire array and move the smallest value to absolute left aka (sub-array). Therefore, you swap the 5 and the 10, because 5 is the smallest. We complete the first iteration when we are able to swap a single element to the absolute left aka (sub-array).
* The second pass you cover the entire array again and move the new smallest value to absolute left aka (sub-array). Therefore, you swap the 6 and the 10, because 6 is now the smallest. Again you complete the iteration when you complete swapping one element
* This algorithm swaps the elements. From their locations. Then technically moves the small elements to a new sub-array. That sub-array is just the start of the main array

**Run time**

* The algorithm begins with a run time of O(n) but the more it sorts the elements the more the run time becomes O(1), because less elements are involved, due to the moving of the elements to a new sub array.
* Our best-case scenario we have is when the whole array is sorted. Even then we will need to cover the full array and our run time will be O(n^2) -> Ω(n^2)
* The worst-case scenario will also be O(n^2)
* On an average the run time to sort an array using this algorithm is O(n^2). Although the magnitudes of this and bubble sort are the same, this one is technically faster because it reduces the size of the search area each time, unlike bubble sort which searches the entire array each time.

**Pros**

* Selection sort is the next logical step from bubble sort. It does what bubble sort does, but has one large difference. Instead of searching through the entire array each time, it creates and only searches an "unsorted portion".
* This although slightly faster than bubble sort, doesn’t improve the run time by any large magnitude. It still is on average a runtime of n^2. You can see the math for this under the diagram. It is still making n(n-1)/2 comparisons. This reduces to (n^2-n)/2 operations. Remember when we go to infinity we grab the largest magnitude. So in this case, the n^2 is the only thing that stays.

**Cons**

* This is the second most inefficient sorting algorithm after bubble sort. The reason why it is has an O(n^2) worst case, is because its code involves a nested for loop with always produce n^2

**6.3 Insertion Sorting Algorithms Introduction**

* This algorithm combines both the bubble sort and the selection sort. During each iteration this algorithm takes a value into the new array (similar to selection sort). Then in that new array it will swap the values until they are all sorted (similar to the bubble sort)

**Example**

A picture containing calendar

Description automatically generated

1. The algorithm begins by making 9 the first member of its sorted subarray
2. Then in the second iteration it looks for the first number to the right of the sorted subarray. That is number 7. Therefore, it picks the number 7 and inserts it to the subarray and compares it to the number 9. If 7 is larger than 9 it does not insert it yet and puts it to the right of the sub-array. If 7 is smaller than 9 it goes to the left of 9
3. During the third iteration, it picks the first number to the right of the subarray. In this case it was number 6. Then it compares number 6 to the largest number in the subarray at this point which is number 9. If it is larger than 9 it goes to the right (it does not insert into the subarray yet, because it does not need too yet. Then moves to the fourth iteration) if not, then it compares 6 to 7. If 6 is larger than 7 it goes to the right if not, then 6 goes to the left. Then, it moved to the fourth iteration

* The meaning of (it does not insert into the subarray yet, because it does not need too yet. Then moves to the fourth iteration). This means the algorithm does not insert a larger number (A) than the largest number (B) in its sub array it keeps that larger number(A) to the right until an even larger number (C) than that large number (A) comes along. In that case it inserts the larger number (A) ( (A) becomes the largest number of the sub-array, because it has been inserted to it, due to the presence of (C)). Then (C) is kept to the right of the array, until the same happens to it when even an even larger number (D) comes or the algorithm is fully sorted.
  1. Note: D > C > A> B. Each of these letters represented a number in the explanation above to make it simpler to understand
  2. Please note: when the algorithm keeps a **number to its right (Due its value)**. This means the algorithm will compare other numbers to the right of the number that it is keeping to it right. If those numbers are smaller than (The largest number in the sub-array). Then, those numbers will be compared/inserted in the sub-array. While that number that is larger than the largest number in the sub-array will still be kept to the right, until an even larger number comes along or the array is fully sorted
* The reason behind this is because once the algorithm inserts a number into its sub-array that number must be compared against every element in that sub array until it is placed. For example, if the algorithm inserts a small number smaller than the smallest number in that sub-array. Then the algorithm will compare that number against every number in the array, until it is placed all the way to the left (this would time a great number of operations). Therefore, to limit the number of operations we briefly ignore the largest numbers until they are not the largest numbers anymore

**Run time**

* The algorithm’s run time is like bubble sort’s run time. However, their key difference is that in the new array it makes. It does not complete the iteration until that array is full sorted
* Our best-case scenario we have is when the whole array is sorted. Even then we will need to cover the full array and our run time will be O(n) -> Ω(n). But we will a smaller number of operations than bubble sort
* The worst-case scenario will also be O(n^2). When its full reversed
* On an average the run time to sort an array using this algorithm is O(n^2).

**Pros**

* This algorithm, although smarter than the other two, is still about the same magnitude of run time. The best-case scenario is better than that of selection sort, as if the number to the right of the sorted subarray is larger than the entire sorted array, then it just moves the sorted array over to fill this number. It then just moves on. If the entire array comes in sorted, then it just does 1 pass through the entire array and returns that it is sorted.

**Cons**

* This algorithm, although smarter than the other two, is still about the same magnitude of run time.

**6.4 Recursion**

* In computer science, recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.
* **Example 1**
* Let us consider a problem that a programmer have to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply add the numbers starting from 1 to n. So the function simply looks like:

approach(1) – Simply adding one by one

f(n) = 1 + 2 + 3 +……..+ n

* but there is another mathematical approach of representing this. Approach(2) – Recursive adding

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

* There is a simple difference between the approach (1) and approach(2) and that is in approach(2) the function “ f( ) ” itself is being called inside the function, so this phenomenon is named as recursion and the function containing recursion is called recursive function, at the end this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.
* **Example 2**
* Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.
  + For example, we can define the operation "find your way home" as:
  + If you are at home, stop moving.
  + Take one step toward home.
  + "find your way home".
* Another example of recursion would be finding the maximum value in a list of numbers. The maximum value in a list is either the first number or the biggest of the remaining numbers. Here is how we would write the pseudocode of the algorithm:

Function find\_max( list )

possible\_max\_1 = first value in list

possible\_max\_2 = find\_max ( rest of the list );

if ( possible\_max\_1 > possible\_max\_2 )

answer is possible\_max\_1

else

answer is possible\_max\_2

end

end

* Example 3
* What kinds of problems are well solved with recursion? In general, problems that are defined in terms of themselves are good candidates for recursive techniques. The standard example used by many computer science textbooks is the factorial function.
* The factorial function, often denoted as n!, describes the operation of multiplying a number by all the positive integers smaller than it. For example, 5! = 5\*4\*3\*2\*1. And 9! = 9\*8\*7\*6\*5\*4\*3\*2\*1.
* Take a good close look at the above, and you may notice something interesting. 5! can be written much more concisely as 5! = 5\*4!.

Graphical user interface, application

Description automatically generated

* And 4! is actually 4\*3!.

Text

Description automatically generated with medium confidence

* We now see why factorial is often the introductory example for recursion: the factorial function is recursive, it is defined in terms of itself. Taking the factorial of n, n! = n\*(n - 1)! where n > 0.
* factorial(3) = 3 \* factorial(2) = 3 \* 2 \* factorial(1) = 3 \* 2 \* 1 \* factorial(0) By our function definition, the factorial(0) should be 0! = 0 \* factorial(-1). Wrong. This is a good time to talk about how one should write a recursive function, and what two cases must be considered when using recursive techniques.
* There are four important criteria to think about when writing a recursive function. What is the base case, and can it be solved? What is the general case? Does the recursive call make the problem smaller and approach the base case?
  + Base Case
    - The base case, or halting case, of a function is the problem that we know the answer to, that can be solved without any more recursive calls. The base case is what stops the recursion from continuing on forever. Every recursive function must have at least one base case (many functions have more than one). If it doesn't, your function will not work correctly most of the time, and will most likely cause your program to crash in many situations, definitely not a desired effect.
    - Let's return to our factorial example from above. Remember the problem was that we never stopped the recursion process; we didn't have a base case. Luckily, the factorial function in math defines a base case for us. n! = n\*(n - 1)! as long as n > 1. If n = = 1 or n = = 0, then n! = 1. The factorial function is undefined for values less than 0, so in our implementation, we'll return some error value. Using this updated definition.

Diagram

Description automatically generated

* The example above: once we get to the end by taking the factorial each time. The answer is 1 because 1! = 1. Then we can go back multiple the results of that by the one prior to it and the one prior to that, which ultimately will give us 3\*2 which is equal to 3!
  + The General Case
    - The general case is what happens most of the time, and is where the recursive call takes place. In the case of factorial, the general case occurs when n > 1, meaning we use the equation and recursive definition n! = n\*(n - 1)!.
  + Diminishing Size of Problem
    - Our third requirement for a recursive function is that the on each recursive call the problem must be approaching the base case. If the problem isn't approaching the base case, we'll never reach it and the recursion will never end. Imagine the following incorrect implementation of factorial
  + Avoiding Circularity
    - Another problem to avoid when writing recursive functions is circularity. Circularity occurs when you reach a point in your recursion where the arguments to the function are the same as with a previous function call in the stack. If this happens you will never reach your base case, and the recursion will continue forever, or until your computer crashes, whichever comes first. This is stack overflow
* The Call Stack
* Recursion might not be the most efficient way to implement an algorithm. Each time a function is called, there is a certain amount of "overhead" that takes up memory and system resources. When a function is called from another function, all the information about the first function must be stored so that the computer can return to it after executing the new function.
* When a function is called, a certain amount of memory is set aside for that function to use for purposes such as storing local variables. This memory, called a frame, is also used by the computer to store information about the function such as the function's address in memory; this allows the program to return to the proper place after a function call (for example, if you write a function that calls printf(), you would like control to return to your function after printf() completes; this is made possible by the frame).
* Every function has its own frame that is created when the function is called. Since functions can call other functions, often more than one function is in existence at any given time, and therefore there are multiple frames to keep track of. These frames are stored on the call stack, an area of memory devoted to holding information about currently running functions.
* A stack is a LIFO data-type, meaning that the last item to enter the stack is the first item to leave, hence LIFO, Last In First Out. Compare this to a queue, or the line for the teller window at a bank, which is a FIFO data structure. The first people to enter the queue are the first people to leave it, hence FIFO, First In First Out. A useful example in understanding how a stack works is the pile of trays in your school's dining hall. The trays are stacked one on top of the other, and the last tray to be put on the stack is the first one to be taken off.

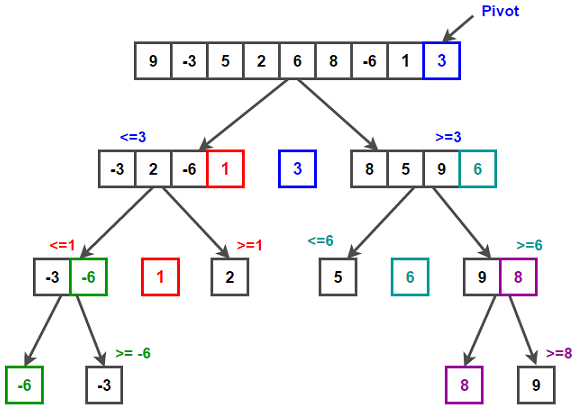
In the call stack, the frames are put on top of each other in the stack. Adhering to the LIFO principle, the last function to be called (the most recent one) is at the top of the stack while the first function to be called (which should be the main() function) resides at the bottom of the stack. When a new function is called (meaning that the function at the top of the stack calls another function), that new function's frame is pushed onto the stack and becomes the active frame. When a function finishes, its frame is destroyed and removed from the stack, returning control to the frame just below it on the stack (the new top frame).

**6.5 Quick Sort Algorithms**

This algorithm works on the idea of pivoting. This is the idea of breaking arrays into sub-arrays and further down into sub-sub arrays.

The Steps:

1. Choose a pivot ( a point or an element location within the array)
   1. It is preferred that the pivot is chosen as the left most element of the array, or the middle or the right most element of the array
2. Divide the array based on the value of the pivot
   1. Anything that is larger than the element inside that element location will go into a new sub-array located to the right of the original array in an unsorted fashion
   2. Anything that is less than the element inside that element location will go into a new sub-array located to the left of the original array in an unsorted fashion
3. Then perform a second divide on each of the divided sub array to create further subarrays. Then keep dividing and conquering until everything is sorted



*Example of the pivot being chosen as the right most element*

|  |  |
| --- | --- |
| 1. Choose the pivot | Quicksort – Divide and Conquer Algorithm and Time Management – MZL |
| 1. Place the pivot in the middle location with respect to the whole array.    1. Then anything larger than the pivot goes to the right and anything smaller goes to the left | Quicksort – Divide and Conquer Algorithm and Time Management – MZL |
| 4.1.0 While on the left, select another pivot, within your new sub-array.  4.1.1 Anything larger than it goes to the right and anything smaller goes to the left  4.1.2 Keep doing 1.0 - 4.2.2 **until you do not have any more sub-arrays to divide and only have a single element in each location** | Quicksort – Divide and Conquer Algorithm and Time Management – MZL |
| 4.2.0 While on the right, select another pivot, within your new sub-array.  4.2.1 Anything larger than it goes to the right and anything smaller goes to the left  4.2.2. stop once you do not have more sub-arrays to divide  4.2.3 Keep doing 1.0 - 4.2.2 **until you do not have any more sub-arrays to divide and only have a single element in each location** | Quicksort – Divide and Conquer Algorithm and Time Management – MZL |
| 5. 0. The result will have a fully sorted array. Basically, any element that is located in a single location is sorted | **3 total levels** |

* If we look at the example above, we can read it from left to right and have a sorted array. We have {-6,-3,1,2,3,5,6,8,9} Each **level** of this algorithm takes n amount of time.
* In the example above we had 3 **levels**
* Because of the way we split up the information however, there will only be log(n) levels. This means the general run time of the algorithm is going to be the number of levels multiplied by n, or nlogn, a strong improvement on the n^2 we have been seeing.

**Run time**

* Our best-case scenario is O(nlogn) -> Ω(nlogn). The best case in this scenario is if we pick a perfect pivot point to split the data. It **doesn't matter if the array comes in sorted or not**. The pivot point is what matters the most. If we pick a pivot point that perfectly splits the data each time, then we will have log(n) **levels**, and therefore the nlogn run time.
* On an average the run time to sort an array using this algorithm is O(nlogn). The average time happens when we choose a decent pivot point. If we can split up the data and let the program divide and conquer, we can get a nlogn time.
* The worst-case scenario will also be O(n^2). With this algorithm however, the pivot point is important. If we keep picking pivot points that don’t split up the data, then we will not divide the data correctly. Instead of log(n) **levels**, we have the possibility of getting n **levels**. This means it will come out to n \* n again, getting us the n^2 run time.

**Pros**

* The average and best-case scenarios are desirable and better than the other algorithms

**Cons**

* The main con that prevents people from using this algorithm is its worst case of is O(n^2) which can easily cause the program to crash

**6.6 Merge Sort Algorithms**

* This algorithm begins by splitting an array to sub-arrays and sub-arrays of sub-arrays until all the elements are individual single cell arrays
* Then the algorithm will recombine the sub-arrays that it split up. However, during each combination process, it sorts the array

|  |  |
| --- | --- |
| 1.0 Begin by breaking down the array into sub-arrays. Then break these sub-arrays down. Until you have every element of the initial as a (single element array) |  |
| 2.0 Then, merge all the individual elements that we created in step 1.0 with the individual elements that are right next to them.  However, while merging them we will also **sort** them. Making them into sorted sub-arrays |  |
| 3.0 This step is like step 2.0.  Where, the sorted sub-arrays we made in the previous step are merged & sorted with the other sorted sub-arrays whom we also made in the previous step.  To make larger sorted sub-arrays |  |
| 4.0 The final is sorting & merging the last sorted sub-arrays to make one major sorted array. |  |
| 5. 0. The result will have a fully sorted array. Basically, any element that is in a single location is sorted | * The size of the **is 8 elements** * The number of **operations is 3** * There is a constant relationship between the number of elements and the number of operations * This makes the run O(nlogn) * This occurs, because the rate of increase of operations is not linear with respect to the size of the array. At **8 elements** you have 3 operations. At **16 elements** you will have **4 operations**. At **32 elements** you will have **5 operations** |

**Run time**

* Our best-case scenario is O(nlogn) -> Ω(nlogn). The same algorithm is applied no matter what type of array comes in. This means it will always be nlogn
* On an average the run time O(nlogn). The same algorithm is applied no matter what type of array comes in. This means it will always be nlogn
* The worst-case scenario will also be O(nlogn). This occurs because, the array breaks down the data until it is in single unit subarrays. It then recombines them using cursors that only have touch each element once for each combination. Thus, each level is going to be at most n amount of “touches”. Because of the way it breaks up the algorithm, it has logn number of levels. This means that it will always run at nlogn timing, as there is no way to break it up worse than logn levels. So it will be n amount of touches per level multiple by logn amount of levels, giving us an even nlogn no matter what array comes in.

**Pros**

* This algorithm has the best run time.

**Cons**

* There is no cons in this algorithm when compared to Bubble sort or selection sort, or insertion sort, or to quick sort

**6.7 The stability vs instability of algorithms**

**Stability vs instability while sorting**

* Stability in sorting algorithms means that the position of each element within the list will be preserved
* Sometimes the order of data within data structures is important. Certain sorting algorithms adhere to this importance, while others don’t. If a data structure takes order in to account, we call it stable, if it doesn’t, we call it not stable or non-stable.
* Exp: This an example of a stable vs. unstable array.
  + Sort the following array [50,3,43,3]
    - There are two 3s in the above array
    - These 3s have a relationship. In which one of the 3s must always come first
      * We can view this unsorted array as [50,3a,43,3b]
        + Therefore, for this unsorted array to be a sorted “stable” array the order of the 3s must be upheld
    - The sorted stable array will be: [3a,3b,43,50]
    - The sorted unstable array will be: [3b,3a,43,50]

**Algorithms Analyzation Based on Stability**

* Bubble sort: **Stable** – This algorithm is stable because it just swaps the largest value up the structure to the top. If two objects are the same, no swap takes place. This means equal values that were to the left will stay to the left.
  + (2a 1 2b) -> (1 2a 2b) (The 2a will never swap with the 2b because swaps don’t take place when two values are equal)
* Selection sort: **Nonstable** – By default selection sort isn’t stable. This is because it takes a number and swaps it to the left “sorted” side. This gives the possibility that a number can be swapped behind another equal number.
  + (2a 2b 1) -> (1 2b 2a) (The 2a was swapped with the lowest number 1 which was on the right side of 2b. The swap takes place, and the order is not preserved)
* Insertion sort: **Stable** – Insertion sort uses a similar swapping mechanism as Bubble sort. It starts from the right while swapping, but never swaps equal values, meaning there is never a chance for elements positions to be flipped.
* Quick sort: **Nonstable** – Quick sort uses a splitting mechanism to order to sort. If this “pivot” is a number which has a duplicate, there is a good chance the order will be broken.
  + (2a 2b 1) -> (1 2b 2a) (In this scenario, 2b was chosen as the pivot, anything less than it went to the left, and anything greater than or equal went to the right. 2a is equal so it went to the right, breaking the stability of the algorithm)
* Merge sort: **Stable** – Merge Sort splits up the data and recombines it in a way that grabs the smallest element and sticks it back in to the array. It however adheres to the order of values, giving preference to values that are in the left subarray to maintain order.