Sorting Magic

Yhaliff Said Barraza

May 27, 2019

1 Sorting

Have you ever needed to order/sort something *your life, time, fiends, numbers?* well that boring as fuck, fortunately for you computers can do that for you with the magic of **sorting algorithms!!!!**

1.1 Bubble sort

1.1.1 Problem definition

You what to sort stuff Numbers in this case

1.1.2 Input and output

Example :arr = 1|5|2|3|8|4 this is your **Input** after will sort it, you will receive **This** arr = 1|2|3|4|5|8 which is your **Output**

Now let's start simple width the most basic form a sorting know to **MAN KINDDD** BUBBLE SORT
The way bubble sort work is the following manner

1.1.3 Explanation

1. Start at the beginning of the array
2. check if the values at the current index is bigger that the one at the next index
3. if true than Swap the values
4. move to the next index

5. repeat until the current index is the last index

6. keep repeating all previous steps, until the number of representations is equal to the amount of element in the array OR the array is sorted.

If that hard to understand then lets look at some pseudo code

1.1.4 pseudo code

Note: "N" mean input which is an array in this case and x means index

Algorithm 1 Bubble sort

```
1: for each number in N do
2:     for each number in N do
3:         if $N[x] > N[x+1]$ then
4:             Swap(N[x],N[x + 1])
5:         end if
6:     end for
7:     comment if no Swap occurred then we finish sorting
8: end for
9: Return N// Sorted
```

simple words To put in simple words what bubble sort does, it move the largest elements (numbers) to the end of the array, in the process moving the smaller element to the beginning of the same array **Now Time to put this in code** Then do a benchmark

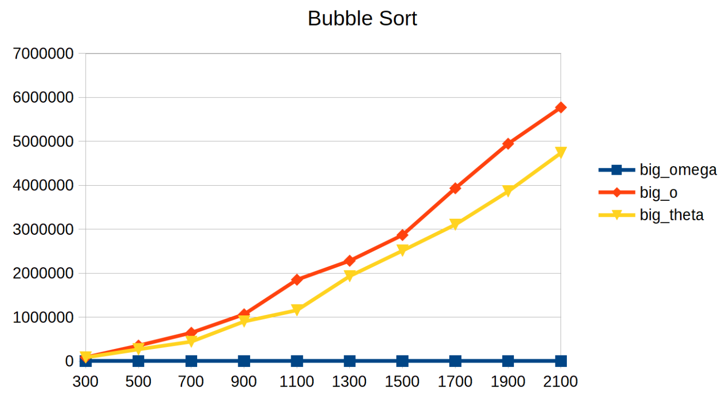
1.1.5 Code

Here is the C++ implementation

```cpp
void BubbleSort(std::vector<int> &Vec)
{
  // to not have to sort an necessarily
  bool isSorted = true;

    for (auto Elemento : Vec)
    {
      for (int j = 0; j < Vec.size() - 1; ++j)
      {
        if (Vec[j] > Vec[j + 1]) {
              Swap(Vec[j], Vec[j + 1]);
              isSorted = false;
          }
      }

      if (isSorted) { break; }

      isSorted = true;
    }
}
```

1.1.6 Benchmark

TIME FOR BENCHMAKING
Here is the resulting of bench-marking Bubble sort, should also mention that the y-axis represents *Microseconds* and the x-axis represent the amount of elements in the array



Sort.png

1.2 Insertion sort

Now it's time to talk about another algorithm, one that likes going **DEEP** the man that go his own way (backwards) muh boi **INSERTION SORT!!!!**

1.2.1 Problem definition

It's still the same problem that i mention in 1.1.1

1.2.2 Input and output

the same that i said in 1.1.2 sill applies here

1.2.3 Explanation

The way that Insertion sort work is very particular because it start's at (almost) the end of an array , and works it's way to the end it would be best explained with a step by step guide of insertion sort
something

1. Start at the pen-ultimate index of the array(will call this N[1])
2. Check if the item in the current index is bigger than it's neighbor if true then do step 3 else go to step 4
3. swap the values and move forward then repeat step 2 until your reach the last index in the array
4. Now start the entire process at N[1] - (the amount of times you reached this step) and repeat step 2 and 3 until you've go through the all the array

note now after explain the process of insertion sort, we can use this knowledge convert the previous steps into code.

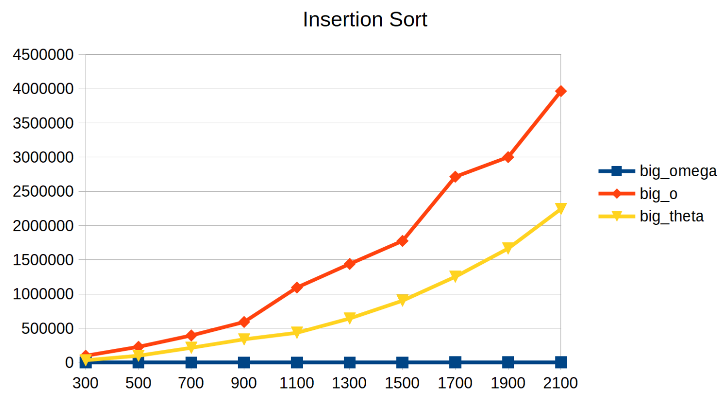
1.2.4 Code

```cpp
void InsertionSort(std::vector<int> &Vec)
{
  for (int i = Vec.size() - 1; i > 0; --i) {

    if (Vec[i] < Vec[i - 1])
    {
      // this is so we don't go out of bounds
      int CurrentPos = i;
      // making sure we don't go out of bonds
      while (CurrentPos <= (Vec.size() - 1) && Vec[CurrentPos] < Vec[CurrentPos
      {
        Swap(Vec[CurrentPos - 1], Vec[CurrentPos]);
        CurrentPos++;
      }
    }
  }
}
```

and teh amunt of fast now it's time to answer another question you probably have in your head , ? how fast is this thing ? will the simple answer is **BENCH MARKING** and it depends

1.2.5 BENCH MARKING

Here's the result's of bench making and what i said in 1.1.6 is still true here.



Sort.png

1.3 Quick-Sort

Now you are probably thinking to yourself right now *I what some goddamn SPEED* to that i say it's time to introduce my man of fast, the recursive devil, His name is literally Synonyms for rapid , swift, instantaneous and **SPEED** , **QUICK-SORT!!!!!** ~~under the right circumstances but that's for later~~

1.3.1 Problem definition

again what i said at subsubsection 1.1.1 still applies here .

1.3.2 Input and output

nothing has changed so far so what I said in subsubsection 1.1.2 is still true.

1.3.3 Explanation

Now your probably thinking *HOW THE FUCK DOES IT GO FAST ???* which i will enplane in a series of steps.

NOTE : When i say *left* I'm referring to element closer to the beginning of the array, when i say *right* I'm referring to element's closer to the end of the array.

NOTE 2 : The variable used to keep track of the amount of elements will be called *"Count"*

NOTE 3 : you need to keep track of the *end* and *beginning* of the array

1. First chose a pivot (it can be in the center,beginning or end of the array) I'm choosing the end in this case.

2. then move all element that are smaller than the pivot to the left (you **Need** to keep track of how many elements are bigger than the pivot.
3. then when you reach the end of the array use the variable Count to place the value you chose as pivot to it's appropriate location .
4. Now return the variable Count + 1.
5. Now divide the array in 2 half's using the pivot to determine the center.(unless the left side is greater then the right side)
6. Now repeat Step 1 until you can no longer do Step 5 after that you done.

Partition if your wondering how to do this in just 1 function well you don't(unless you want to), quick-sort (at lest the recursive version) requires a second function to find the partition.(i will show the function in different blocks)

1.4 code

Now Here is the code for **Partition**

```
/*! this is to help find a pivot for quick-sort*/
int Partition(std::vector<int> &Vec, int LowerLimit, int UpperLimit)
{
  int GreaterThanPivotCount = LowerLimit - 1;
  int PivotPos = UpperLimit;

  for (int i = LowerLimit; i < UpperLimit; ++i)
  {

    if (Vec[PivotPos] > Vec[i])
    {
      ++GreaterThanPivotCount;
      Swap(Vec[i], Vec[GreaterThanPivotCount]);
    }

  }
  Swap(Vec[UpperLimit], Vec[GreaterThanPivotCount + 1]);

  return GreaterThanPivotCount + 1;
}
```

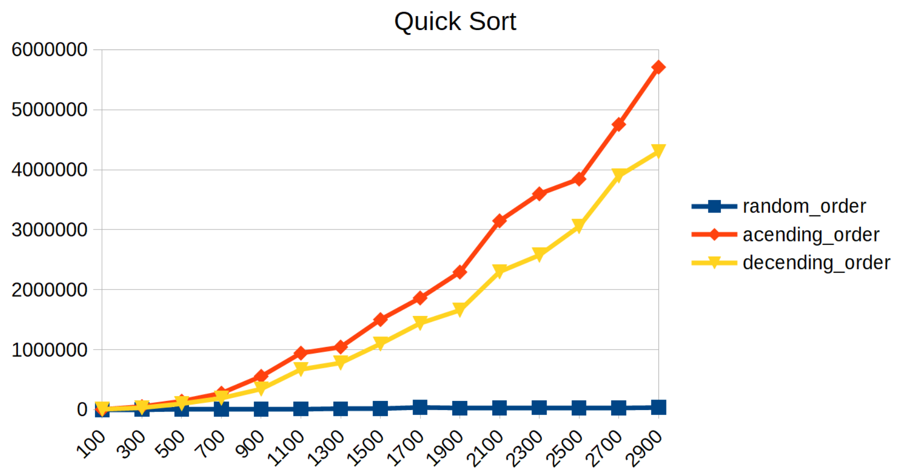
Here is the code for **Quick sort**

```
/*! this is going to used in the merge-sort*/
void QuickSort(std::vector<int> &Vec, int LowerLimit, int UpperLimit)
{
  if (LowerLimit < UpperLimit)
  {
    int Pivot = Partition(Vec, LowerLimit, UpperLimit);
    //PrintVector(Vec);
    QuickSort(Vec, LowerLimit, Pivot - 1);
    QuickSort(Vec, Pivot + 1, UpperLimit);
  }
}
```

How fast Time to find out how fast is quick sort how you know how **BENCH MAKING**

1.4.1 BENCH MAKING quick-sort

Here's the result's of bench making and what i said in 1.1.6 is still true here.



As you can see quick-sort is very fast in the case where the array is completely random in the other cases however it's not as efficient, Plus this wlll be the last bench mark becase i using another method(and frankliy i don't have the time to bench mark everything writing this already take a lot of time)

1.4.2 Master Method

Here is a way to calculate the efficiency of a recursive algorithms . it looks like this

$$T(x) = aT(\frac{n}{b}) + N \tag{1}$$

The 'a' represents the amount of time the function call itself, $\frac{n}{b}$ represents how many time the input is divided and the N is something else the function has to do now lets apply this to Quick sort

$$\begin{aligned} a &= 2 \\ b &= 2 \\ T(QuickSort) &= 2T(\frac{n}{2}) + partition \end{aligned} \tag{2}$$

A is 2 because the function only calls itself 2 time (per recursion) , B is 2 because when the function calls itself it also divides the array in 2 half, partition is something the function has to do for it to work so it takes the place of N

1.5 Merge Sort

Now lets talk about a sorting algorithms that separates and then creates Unity **Merge Sort**

For future reference by the way what i said in subsubsection 1.1.1 and subsubsection 1.1.2 still applies here so don't ask why those other sections **Problem definition** and **Input and Output** are not here

1.5.1 Explanation

I'll explain what merge sort does with steps
NOTE : Pivot means the middle , 'R' means the right side of a array and 'L' means the left side of a array

1. First get the Pivot of the array
2. check that L does is *Not* bigger than R
3. divide the array in 2 half, one half that starts at L up to Pivot and the other half starts at Pivot + 1 up to R .
4. keep doing this until you get arrays of size 1.
5. then merge 2 array (ordering them in the process)
6. you will get bigger arrays now just repeat step 5 until you've done that for the whole array .

Merge just like Quick-sort in section 1.3.3 within the paragraph Partition, you need a second function that is crucial to the entire process which in this case that function is merge.

NOTE : this is not the normal implementation of merge what your suppose to do is copy the array(from merge-Sort) into 2 temporary arrays then copy those back to the original array(from merge-Sort) in such a way that the resulting array becomes sorted.

1.5.2 Code

Here is my implementation of Merge sort.

```
 void Merge(std::vector<int> &Vec, int LowerLimit, int middle, int UpperLimit)
{
  // for when the array is bigger than 1
  int LowerArea = middle - LowerLimit;
  int UpperArea = UpperLimit - middle;

  if (LowerArea > 1 || UpperArea > 1)
  {
    QuickSort(Vec, LowerLimit, UpperLimit);
  }
  // for the case that there are only 2 elements to be sorted
  else if (Vec[LowerLimit] > Vec[UpperLimit])
  {
    Swap(Vec[LowerLimit], Vec[UpperLimit]);
  }

}
```

This is merge sort

```
void MergeSort(std::vector<int> &Vec, int LowerLimit, int UpperLimit)
{
  int Pivot = (UpperLimit + LowerLimit) / 2;

  if (LowerLimit < UpperLimit)
  {
    MergeSort(Vec, LowerLimit, Pivot);

    MergeSort(Vec, Pivot + 1, UpperLimit);

    Merge(Vec, LowerLimit, Pivot, UpperLimit);
  }


}
```

2 Searching

Now lets say you what something from your sorted array but are 2 lazy to look for it yourself **Searching algorithms to the rescue**

2.1 Problem Definition

You what to look for things (*numbers* , *people* , *hentai folder* you name it) and sometimes what your looking for can be in a array, data structure etc...

2.2 Input and Output

there are 3 option for the Output

1. what your looking for .
2. a true or false massage .
3. Jack shit(aka nothing).

the input can be an array, data structure etc...

2.3 Linear search

this is the most basic algorithms of all it's literately just a for loop here just look

2.4 Explanation

it's literally just a for loop that look at each element one by one until it reaches the end and fond *nothing* or it does find the element you where looking for .

2.5 code

```
template<class T>
inline bool LinearSearch(std::vector<T> &Vec, T &ItemToFind)
{
  for (auto PossibleMatch : Vec)
  {
    if (PossibleMatch == ItemToFind)
    {
      return true;
    }
  }

  return false;
}
```

and the complexity of this algorithm is N because it has to go through every part of the array to make sure that what your looking for is in the array or not.

2.6 Binary search

Now lets say you what something more efficient that linear search, well then let me show you **Binary search** and I should mention that what i said in subsection 2.2 and subsection 2.1 still applies here.

2.7 Explanation

this one is harder to understand than linear-search but I will do what I've bin doing for all of this document give a bunch of steps.

NOTE : The Value your looking for in the array is called "S" , the left half will be called "l" and the right half will be called "R"

1. look at the element that between l and R (in the center of the array)
2. if that element is S,then your done else continue.
3. check if the element is bigger or smaller that S then do 1 of 2 thing
   (a) the element is bigger than S so now l is equal to half of the array
   (b) the element is smaller than S so now R is equal to half of the array
4. Repeat step 1

NOTE : This algorithm requires that the array already be sorted.

In Simple Words What this algorithm does is divide the section of the array it will look in until it can't or it's found the S

2.7.1 Code

```
inline bool BinarySearch(std::vector<int>& Vec, int Number, std::size_t LeftHalf
{
  std::size_t SearchPoint = (LeftHalf + RightHalf) / 2;

  if (Number == Vec[SearchPoint])
  {
    return true;
  }

  /*This is so we don't have stack-overflow*/
  if (LeftHalf < RightHalf - 1)
  {/*Move to the right */

    if (Number > Vec[SearchPoint])
    {
      LeftHalf = (RightHalf + LeftHalf) / 2;
      return  BinarySearch(Vec, Number, LeftHalf, RightHalf);
    }
    else/*Move to the left*/
    {
      RightHalf = (RightHalf + LeftHalf) / 2;
      return  BinarySearch(Vec, Number, LeftHalf, RightHalf);
    }

  }
  else// this is for when there are only options 2 left
  {
    if (Vec[LeftHalf] == Number)
    {
      return true;
    }
    else if (Vec[RightHalf] == Number)
    {
      return true;
    }

  }

  return false;
}
```