BOLTS & NUTS ALGORITHM

MAY 14, 2019

AIN SHAMS UNIVERSITY – FACULTY OF ENGINEERING
1 Al-Sarayyat St, Abbassiya, Cairo 11517, Egypt

FACULTY OF ENGINEERING

AIN SHAMS UNIVERSITY

CSE224: Design and Analysis of Algorithms

BOLTS & NUTS ALGORITHM

Using Divide & Conquer

Submitted By:

Abdelrahman Ibrahim ELGhamry
Ghamry98@hotmail.com
16P3043@eng.asu.edu.eg

Hossam ELDin Khaled Mohmed
hossampen97@gmail.com
16P3025@eng.asu.edu.eg

Abdelrahman Amr Issawi
aid-issawi@hotmail.com
16P6001@eng.asu.edu.eg

Submitted to:

Prof. Dr. Gamal Abdelshafy

MAY 14, 2019

A REPORT FOR DESIGN AND ANALYSIS OF ALGORITHMS COURSE CODDED CSE224 WITH THE REQUIREMENTS OF AIN SHAMS UNIVERSITY

Table of Contents

1.0 INTRODUCTION 3

2.0 ANALYSIS 3

2.1 Brute Force 3

2.2 Divide and Conquer 4

3.0 IMPLEMENTATION 5

3.1 Quick Sort 5

3.2 Merge Sort 7

4.0 TEST CASES 9

4.1 Quick Sort 9

4.2 Merge Sort 10

5.0 BETTER ALTERNATIVE 11

5.1 Hashmap 11

6.1.1 Analysis 11

6.1.2 Time Complexity 11

6.1.3 Implementation 12

1.0 INTRODUCTION

This project is about implementing the bolts and nuts algorithm, this algorithm solves the problem of mapping one to one a set of bolts to a set of nuts where a nut can't be compared to nut and a bolt can't be compared to a bolt.

The algorithm aims to solve the stated problem efficiently, and for this matter there are different implementation that will be discussed in this report.

Divide and conquer approach is studied in more details and in comparison, with other approaches.

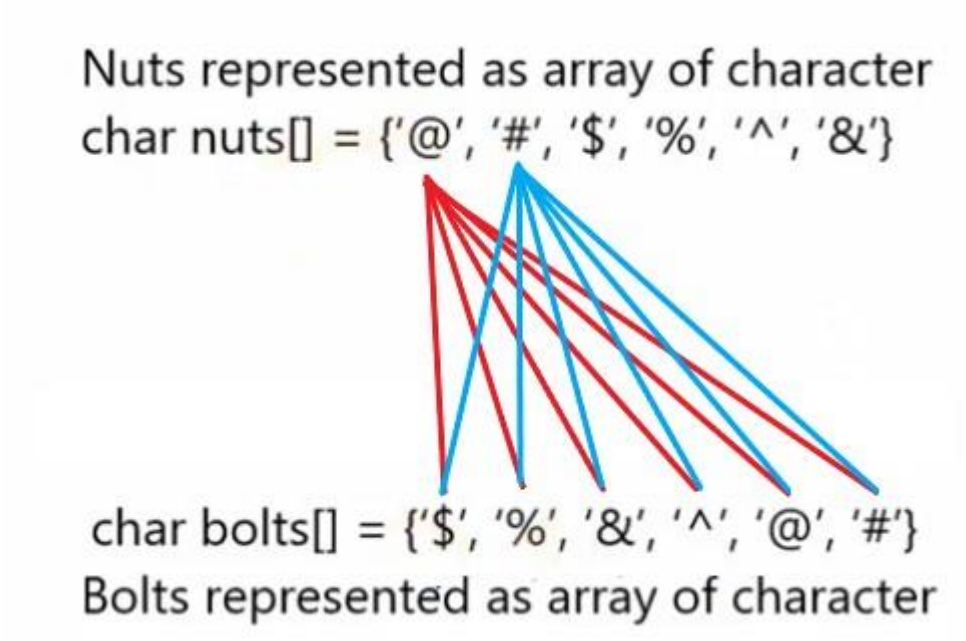
2.0 ANALYSIS

2.1 Brute Force

1. Iterate through the array of bolts.
2. For each bolt find a match in the array of nuts.
3. Reposition the nut to be at the same index as of the bolt.

We can clearly observe that this approach runs at $\theta(n^2)$.

As there is a nested loop, the outer loop loops over all the bolts, and the inner loop loops to find a match for the bolt over all the nuts in the worst case.



2.2 Divide and Conquer

1. Sort the array of bolts using a divide and conquer sorting algorithm based on its ascii code.
2. Sort the array of nuts using a divide and conquer sorting algorithm based on its ascii code.
3. Now each bolt has a matching nut at the same index.

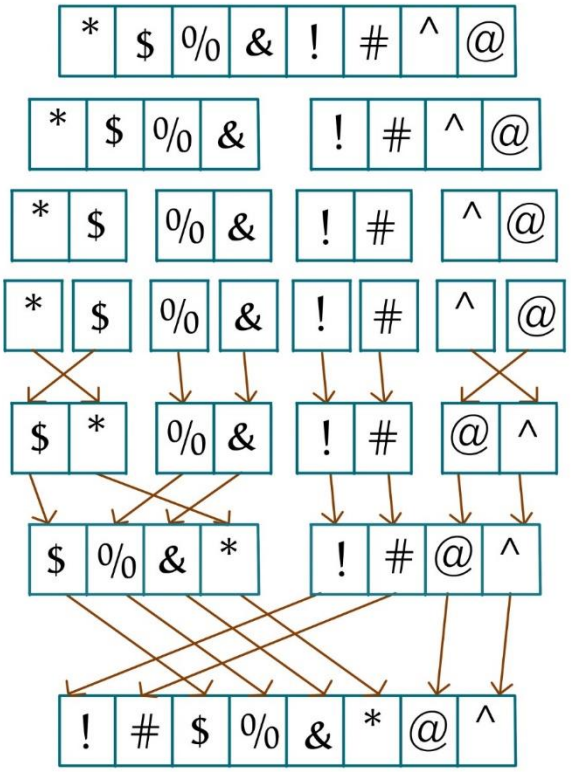
Divide and conquer algorithms such as quick sort & merge sort, run at $\theta(nlog(n))$.

The main difference between these two is the stability; the quick sort algorithm is unstable as it doesn't maintain the relative order of records in the case of equality of keys while the merge sort maintains this property.

However, in our case we do not care about the order of elements, the whole sorting idea is to position the bolts and nuts in such a way, so each bolt has a matching nut at the same index.

So, it comes to whichever has better performance, although quick sort has worse running time at the worst case $\theta(n^2)$, it can easily be avoided with high probability by choosing the right pivot. Moreover, quick sort can be two to three times faster than merge sort, because the quick sort operations in the innermost loop are much simpler, it also does not use extra memory like merge sort.

In conclusion, quick sort is a better choice in this application due to its space and time efficiency.



3.0 IMPLEMENTATION

3.1 Quick Sort

```
package algorithmsproject;

public class AlgorithmsProject {

   public static void main(String[] args) {
      char nuts[] = {'@', '#', '$', '%', '^', '&'};
      char bolts[] = {'$', '%', '&', '^', '@', '#'};

      match(nuts, bolts, 0, 5);

      System.out.println("Matched nuts and bolts are : ");
      printArray(nuts);
      printArray(bolts);
   }

   private static void printArray(char[] arr) {
      for (int i = 0; i < arr.length; i++) {
          System.out.print(arr[i] + " ");
      }
      System.out.print("\n");
   }

    private static void match(char[] nuts, char[] bolts, int low, int high) {
      if (low < high) {
         int pivot = partition(nuts, low, high, bolts[high]);

         partition(bolts, low, high, nuts[pivot]);

         match(nuts, bolts, low, pivot-1);
         match(nuts, bolts, pivot+1, high);
      }
   }
```

```
    private static int partition(char[] arr, int low, int high, char pivot) {
      int i = low;
      char temp1, temp2;
      for (int j = low; j < high; j++) {
        if (arr[j] < pivot){
          temp1 = arr[i];
          arr[i] = arr[j];
          arr[j] = temp1;
          i++;
        }
        else if(arr[j] == pivot){
          temp1 = arr[j];
          arr[j] = arr[high];
          arr[high] = temp1;
          j--;
        }
      }
      temp2 = arr[i];
      arr[i] = arr[high];
      arr[high] = temp2;

      return i;
    }
}
```

3.2 Merge Sort

```
package algorithmsproject;

public class AlgorithmsProject {

   private static void merge(char arr[], int l, int m, int r) {
      // Fid sizes of two subarrays to be merged
      int n1 = m - l + 1;
      int n2 = r - m;

      /* Create temp arrays */
      char L[] = new char [n1];
      char R[] = new char [n2];

      /Copy data to temp arrays/
      for (int i=0; i<n1; ++i)
         L[i] = arr[l + i];

      for (int j=0; j<n2; ++j)
         R[j] = arr[m + 1+ j];


      /* Merge the temp arrays */

      // Initial indexes of first and second subarrays
      int i = 0, j = 0;

      // Initial index of merged subarry array
      int k = l;
      while (i < n1 && j < n2) {
         if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
         }
         else {
            arr[k] = R[j];
            j++;
         }
         k++;
      }

      /* Copy remaining elements of L[] if any */
      while (i < n1) {
         arr[k] = L[i];
         i++;
         k++;
      }
```

```
    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
private static void sort(char arr[], int l, int r) {
    if (l < r) {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

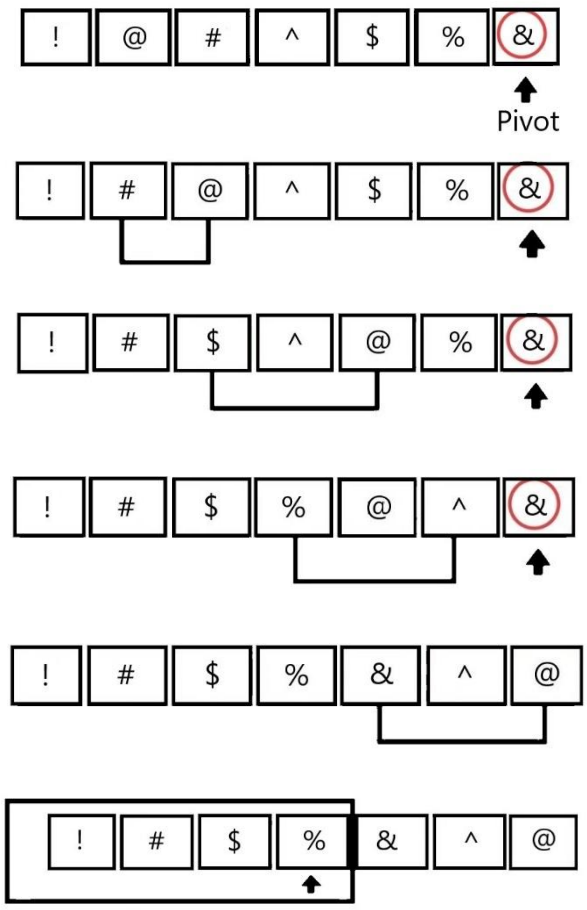
4.0 TEST CASES

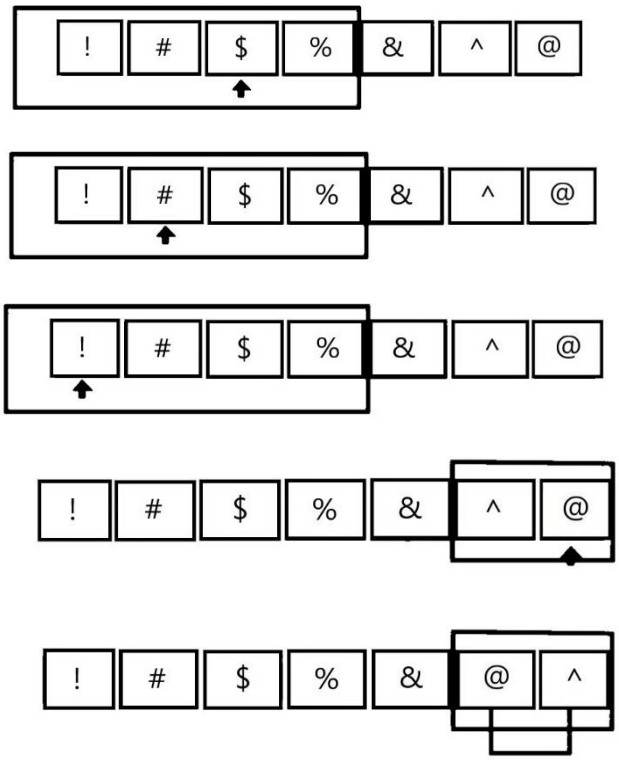
4.1 Quick Sort

- The quick sort algorithm starts by choosing a pivot, in our case the last element in the array.
- It rearranges the array such that all the elements that are smaller than the pivot are on its left and all elements that are greater than the pivot are on its right.
- Apply the partitioning recursively on the left and right sub-arrays.

To sort both arrays bolts and nuts the quick sort is modified a little:

- It picks the last element in the array of bolts as a pivot.
- Rearrange the array of nuts and return the index 'i' such that all the nuts smaller than nuts[i] are on the left side, and all the nuts greater than nuts[i] are on the right side.
- Using nuts[i] partition the array of bolts.
- Apply the partitioning recursively on the left and right sub-arrays of bolts and nuts.





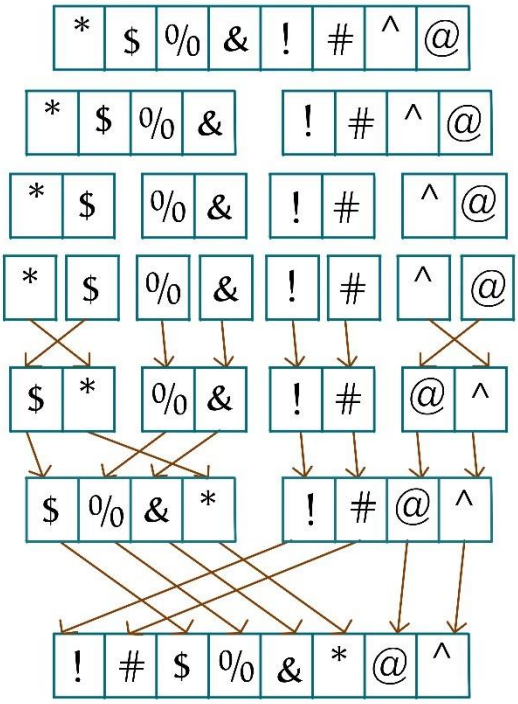
$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$

$a = 2,\ b = 2,\ d = 1$
$a = b^d$
Time Efficiency = Θ (n log n)
Quick sort is **unstable**.

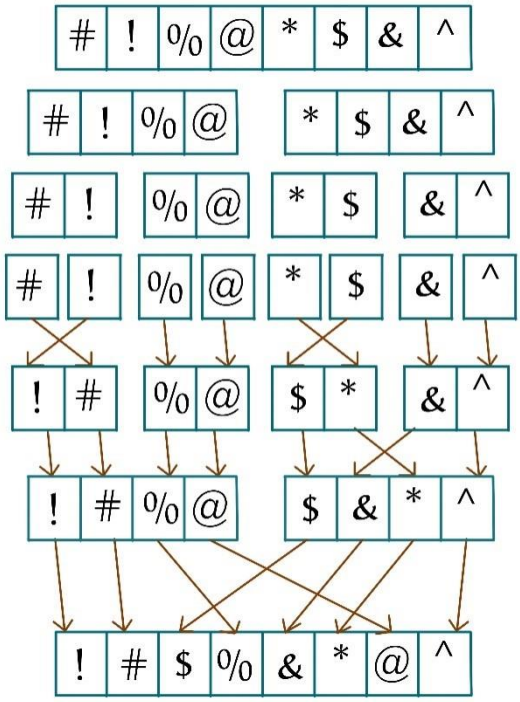
4.2 Merge Sort

- Number of elements to be sorted: n=r-p+1, where 'p' and 'r' are pointers to the first and last elements in the array.
- Base Case: n = 1, the array contains a single element (which is trivially "sorted"), The program terminates because (p=r).
- The algorithm divides the array as in the following figures into almost two equal halves and then calling the merge sort again till reaching the base case, then the algorithm merges the elements by sorting level by level.

Bolts Array



Nuts Array



$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$

$a = 2,\ b = 2,\ d = 1$
$a = b^d$
Time Efficiency = Θ (n log n)
Merge sort is **stable**.

5.0 BETTER ALTERNATIVE

5.1 Hashmap

6.1.1 Analysis

1. Iterate through the array of nuts.
2. Add each nut into the hashmap where (key = nut, value= indexOf(nut)).
3. Iterate through the array of bolts.
4. For each bolt, check if it is found in the hashmap,
    - If true, reposition the nut to be at the same index as of the bolt.
    - If false, do nothing.

6.1.2 Time Complexity

Hashmap is very popular data structure and found useful for solving many problems due to O(1) time complexity for both get and put operation.

We can clearly observe that this approach runs at O(n).

As the algorithm runs on two consecutives for loops, each of n iterations.

So, we can consider that the algorithm implementation using hashmap as a better alternative.

6.1.3 Implementation

```
package algorithmsproject;
import java.util.HashMap;

public class BoltsAndNuts {
   public static void main(String[] args) {
      char bolts[] = {'$', '%', '&', '^', '@', '#'};
      char nuts[] = {'@', '#', '$', '%', '^', '&'};

      BoltAndNuts(bolts, nuts);

      System.out.println("Matched bolts and nuts are : ");
      printArray(nuts);
      printArray(bolts);
   }

   private static void printArray(char[] arr) {
      for (int i = 0; i < arr.length; i++) {
          System.out.print(arr[i] + " ");
      }
      System.out.print("\n");
   }

   private static void BoltAndNuts(char[] bolts, char[] nuts){
      HashMap<Character, Integer> map = new HashMap();

      for (int i = 0; i < nuts.length; i++) {
         map.put(nuts[i], i);
      }

      for (int i = 0; i < bolts.length; i++) {
         if( map.get(bolts[i]) != null ){
            nuts[i] = bolts[i];
         }
      }
   }
}
```