Proposal of the use of a quicksort algorithm to increase DNA strand filtering optimization using C++ as a base

Demetrio Manuel Roa Perdomo

Notas del autor

Demetrio Manuel Roa Perdomo, Facultad de Ingeniería Mecánica y Eléctrica, Universidad Autónoma de Nuevo León

Esta investigación ha sido financiada por el propio alumno

La correspondencia relacionada con esta investigación debe ser dirigida a Demetrio Roa

Universidad Autónoma de Nuevo León, Pedro de Alba S/N, Niños Héroes, Ciudad Universitaria, San Nicolás de los Garza, N.L.

Contacto: demetrio.roap@uanl.edu.mx

Contents

Introduction

The history of sorting algorithms exists several years ago, sorting has always been a fundamental task in various fields, including, engineering and data processing. Some examples from the history of sorting algorithms are as follows.

Beginning with the bubble sort algorithm, also known as exchange sort. Its formal description was introduced by John W. Mauchly and J. Presper Eckert in the context of early computer programming during the 1940s and 1950s.

Other sorting is the selection sort algorithm, also called the exchange sort. It involves repeatedly selecting the minimum or maximum element and placing it in its correct position; selection sort was formally described by Robert W. Floyd in 1962.

Shell sort. It was proposed by Donald Shell in 1959. Shell sort improves the efficiency of insertion sort by comparing elements that are far apart first and gradually reducing the gap between compared elements until the entire list is sorted.

Other example is the merge sort algorithm this was developed by John von Neumann in 1945 and he described it as a method for merging two sorted sequences into a single sorted sequence.

The sorting algorithm quicksort was originally developed by Tony Hoare in 1959. It is a divide-and-conquer algorithm that selects a pivot element and partitions the list into two sub lists, one containing elements smaller than the pivot and the other containing elements larger than the pivot; quicksort is known for its efficiency and is widely used in practice.

This report will talk about quicksort and everything important around this sorting algorithm. The key idea behind quicksort is the efficient partitioning step, which ensures that the pivot element ends up in its correct sorted position. By dividing the list into two sublists and sorting them independently, quicksort achieves a fast average-case time complexity of O(n log n), where n is the number of elements in the list.

Quicksort is widely used due to its efficiency, simplicity, and relatively low memory requirements. It is also known for its good cache performance, making it well-suited for sorting large datasets. Additionally, quicksort can be easily parallelized, allowing for efficient sorting on multi-core systems.

General Overview

The code begins by opening a text file named "rosalind_dna.txt" and reading a single line from it, which reprTo address the bioinformatics challenge of DNA nucleotide counting, our solution involves the development of a program incorporating the Bubble Sort algorithm. The general workings of our implementation are elucidated below.esents the DNA string. The string is then processed into a DNA array for sorting. The DNA array is printed, and the quicksort algorithm is applied to sort the array in ascending order. The sorted DNA array is printed, and then the code counts the occurrences of each nucleotide (A, C, G, T) in the sorted array. The counts are stored in an array called counts, and the result is printed.

To sort the DNA array, the code uses the quicksort algorithm, which is a divide-and-conquer sorting algorithm. It selects a pivot element from the array and partitions the array into two parts: elements smaller than the pivot on the left and elements greater than the pivot on the right. The process is recursively applied to the subarrays before and after the pivot until the entire array is sorted.

After sorting, the code iterates over the sorted DNA array and counts the occurrences of each nucleotide by incrementing the respective count in the counts array. Finally, the counts for each nucleotide are printed, providing the frequency of each nucleotide in the sorted DNA string.

To understand the program, there are several key concepts to grasp:

- Input and Output: The program reads input from a text file named "rosalind_dna.txt," which contains a DNA string. It processes the input and produces output by printing the sorted DNA array and the counts of each nucleotide.
- Arrays: The DNA string is converted into a DNA array, which is a collection of characters representing nucleotides. Arrays are used to store and manipulate data efficiently. The program performs operations on the DNA array, such as sorting and counting.
- Quicksort Algorithm: The quicksort algorithm is a sorting technique used to arrange elements in a particular order. This is what my classmate explained.
- Partitioning: The quicksort algorithm partitions the array by selecting a pivot element and rearranging the elements such that those smaller than the pivot are placed to its left and those greater than the pivot are placed to its right. This process divides the array into two subarrays, and the partitioning step is repeated recursively for each subarray.
- Counting: After the array is sorted, the program counts the occurrences of each nucleotide in the sorted DNA array. It iterates over the array, incrementing the count for each nucleotide encountered. The counts are stored in an array, providing the frequency of each nucleotide in the DNA string.

By understanding these key concepts, you can gain a deeper understanding of how the program processes the input DNA string, sorts it using the quicksort algorithm, and counts the nucleotides in the sorted array.

Complexity Analysis

There are several common time complexities that algorithms can have, listed here in order from least complex to most complex:

- O(1): Constant time complexity. The running time of the algorithm does not depend on the size of the input.
- O(log n): Logarithmic time complexity. The running time of the algorithm grows slowly as the size of the input increases.
- O(n): Linear time complexity. The running time of the algorithm grows proportionally to the size of the input.
- O(n log n): Linearithmic time complexity. The running time of the algorithm grows slightly faster than linear time but slower than quadratic time as the size of the input increases.
- O(n^2): Quadratic time complexity. The running time of the algorithm grows proportionally to the square of the size of the input.
- O(n^3): Cubic time complexity. The running time of the algorithm grows proportionally to the cube of the size of the input.
- O(2^n): Exponential time complexity. The running time of the algorithm grows exponentially as the size of the input increases.
- O(n!): Factorial time complexity. The running time of the algorithm grows factorially as the size of the input increases.

In general, algorithms with lower time complexities are more efficient and can handle larger inputs than algorithms with higher time complexities.

The quicksort algorithm used in this code works by choosing a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The function then recursively sorts the sub-arrays.

For example, let’s say we have an array [8, 5, 2, 9, 7, 6, 3]. If we choose 7 as the pivot, then we can partition the array into two sub-arrays: [5, 2, 3] and [8, 9, 6]. These sub-arrays are then sorted recursively using the same process.

The time complexity of quicksort depends on how balanced the partitions are. If the partitions are balanced, meaning that they have nearly equal size, then quicksort runs asymptotically as fast as the best comparison-based sorting algorithms. In this case, the time complexity is O(n log n), where n is the number of elements in the array. This is because each partitioning takes O(n) time and there are log n levels of recursion.

However, if the partitions are unbalanced, meaning that one is much smaller than the other, then the time complexity can be as bad as O(n^2) in the worst case. This can happen if we always choose the smallest or largest element as the pivot. In this case, each partitioning still takes O(n) time but there are now n levels of recursion.

On average, quicksort has a time complexity of O(n log n). This is because on average, the partitions are balanced.

The space complexity of quicksort is O(log n) due to the recursive call stack. This is because each recursive call to quicksortHelper requires space on the call stack. In the best case and on average, there are log n levels of recursion so the space complexity is O(log n). In the worst case, there are n levels of recursion so the space complexity is O(n).

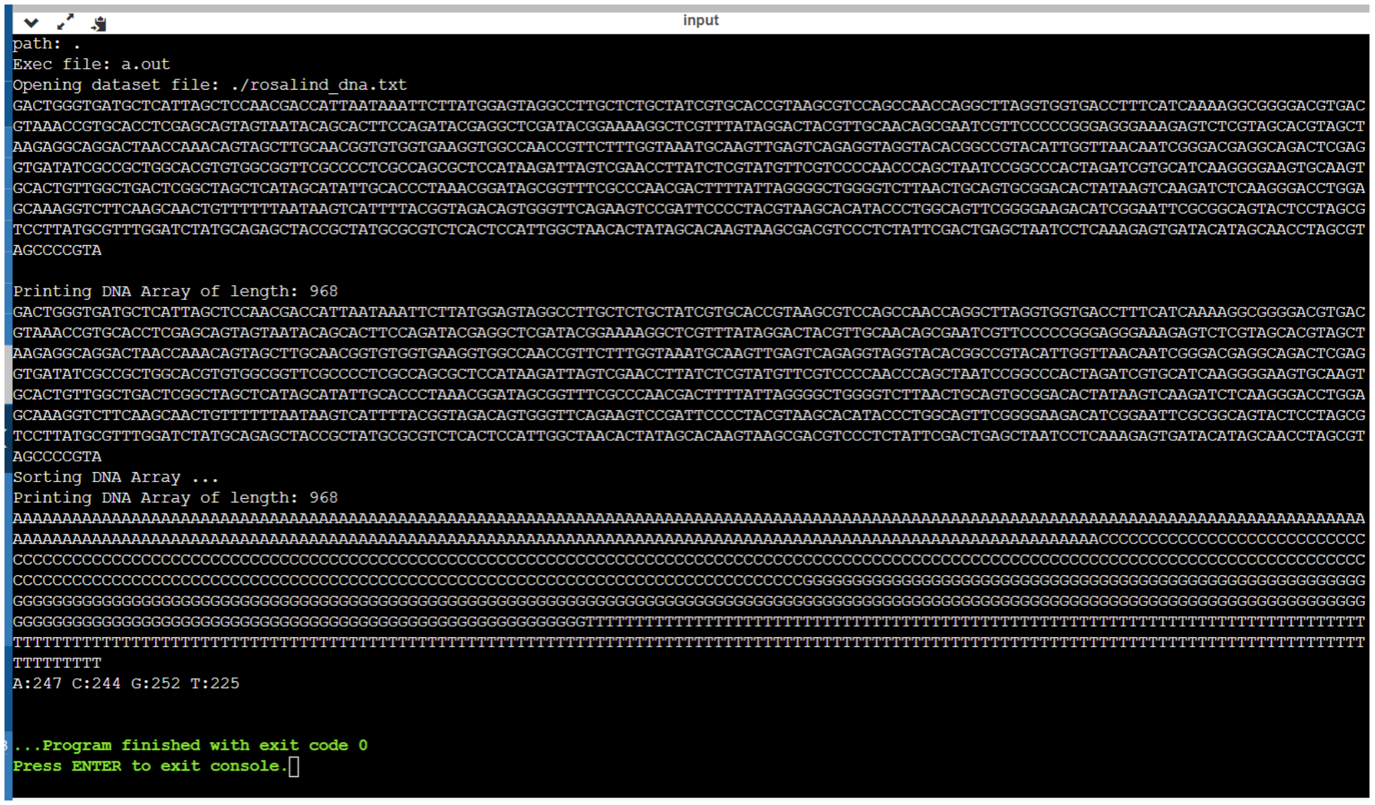
The rest of the code has a time complexity of O(n) for processing the DNA string into an array and counting the occurrences of each type of element. The space complexity is also O(n) for storing the DNA array.

Code Used

```
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

/** Prints content of DNA array*/
void printDNAarray (const char *dnaArray, int length)
{
  cout << "Printing DNA Array of length: " << length << endl;
  for (int i = 0; i < length; i++)
    cout << dnaArray[i];
  cout << endl;
}

/** Converts the DNA string into a DNA array, which can be used for sorting */
char *
processDNAString (string line)
{
  char *arrayToSort = new char[line.length ()];
  for (int i = 0; i < line.length (); i++)
    arrayToSort[i] = line[i];
  return arrayToSort;
}
```

```
/** Partition function for quicksort*/
template <typename T>
int partition(T* array, int low, int high) {
    T pivot = array[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++;
            swap(array[i], array[j]);
        }
    }
    swap(array[i + 1], array[high]);
    return i + 1;
}

/** Recursive implementation of quicksort */
template <typename T>
void quicksortHelper(T* array, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(array, low, high);
        quicksortHelper(array, low, pivotIndex - 1);
        quicksortHelper(array, pivotIndex + 1, high);
    }
}

template <typename T>
void quicksort(T* array, int length) {
    quicksortHelper(array, 0, length - 1);
}
/** Main procedure for reading input text file that contains DNA string*/

int main(int argc, char* argv[]) {
    ifstream fin;
    string line;
    std::string cur_dir(argv[0]);

    // DNA file is at the same folder than your executable
    int pos = cur_dir.find_last_of("/\\");
    string path = cur_dir.substr(0, pos);
    std::cout << "path: " << cur_dir.substr(0, pos) << std::endl;
    std::cout << "Exec file: " << cur_dir.substr(pos + 1) << std::endl;
    string filename = path + "/rosalind_dna.txt";
    cout << "Opening dataset file: " << filename << endl;
    fin.open(filename);

    if (!fin) {
        cerr << "Unable to open rosalind_dna.txt file..." << endl;
        cerr << "Review your current working directory!!" << endl;
        exit(1);
    }

    while (fin) {
        //Read a line from file
        getline(fin, line);
        //Print line in console
        cout << line << endl;
        //There is a single line in the file, so we stop reading
        break;
    }
  cout << endl;

    //Process the DNA string into an array such that we can sort it.
    char* dnaArray = processDNAString(line);
    printDNAarray(dnaArray, line.length());
    cout << "Sorting DNA Array ..." << endl;
    quicksort(dnaArray, line.length());
    printDNAarray(dnaArray, line.length());


    /** Count the occurrences of each type of element in the DNA array*/
    int counts[4] = { 0, 0, 0, 0 };
    for (int i = 0; i < line.length(); i++) {
        if (dnaArray[i] == 'A') {
            counts[0]++;
        }
        else if (dnaArray[i] == 'C') {
            counts[1]++;
        }
        else if (dnaArray[i] == 'G') {
            counts[2]++;
        }
        else if (dnaArray[i] == 'T') {
            counts[3]++;
        }
    }
    /* Print the counts for each type of element**/
    cout << "A:" << counts[0] << " C:" << counts[1] << " G:" << counts[2] << " T:" <<
        counts[3] << endl;
    fin.close();
    delete[] dnaArray;
    return 0;
}
```

<u>Output of the code on the Online C++ Compiler Website</u>



```
path: .
Exec file: a.out
Opening dataset file: ./rosalind_dna.txt
GACTGGGTGATGCTCATTAGCTCCAACGACCATTAATAAATTCTTATGGAGTAGGCCTTGCTCTGCTATCGTGCACCGTAAGCGTCCAGCCAACCAGGCTTAGGTGGTGACCTTTCATCAAAAGGCGGGGACGTGAC
GTAAACCGTGCACCTCGAGCAGTAGTAATACAGCACTTCCAGATACGAGGCTCGATACGGAAAAGGCTCGTTTATAGGACTACGTTGCAACAGCGAATCGTTCCCCCGGGAGGGAAAGAGTCTCGTAGCACGTAGCT
AAGAGGCAGGACTAACCAAACAGTAGCTTGCAACGGTGTGGTGAAGGTGGCCAACCGTTCTTTGGTAAATGCAAGTTGAGTCAGAGGTAGGTACACGGCCGTACATTGGTTAACAATCGGGACGAGGCAGACTCGAG
GTGATATCGCCGCTGGCACGTGTGGCGGTTCGCCCCTCGCCAGCGCTCCATAAGATTAGTCGAACCTTATCTCGTATGTTCGTCCCCAACCCAGCTAATCCGGCCCACTAGATCGTGCATCAAGGGGAAGTGCAAGT
GCACTGTTGGCTGACTCGGCTAGCTCATAGCATATTGCACCCTAAACGGATAGCGGTTTCGCCCAACGACTTTTATTAGGGGCTGGGGTCTTAACTGCAGTGCGGACACTATAAGTCAAGATCTCAAGGGACCTGGA
GCAAAGGTCTTCAAGCAACTGTTTTTTAATAAGTCATTTTACGGTAGACAGTGGGTTCAGAAGTCCGATTCCCCTACGTAAGCACATACCCTGGCAGTTCGGGGAAGACATCGGAATTCGCGGCAGTACTCCTAGCG
TCCTTATGCGTTTGGATCTATGCAGAGCTACCGCTATGCGCGTCTCACTCCATTGGCTAACACTATAGCACAAGTAAGCGACGTCCCTCTATTCGACTGAGCTAATCCTCAAAGAGTGATACATAGCAACCTAGCGT
AGCCCCGTA

Printing DNA Array of length: 968
GACTGGGTGATGCTCATTAGCTCCAACGACCATTAATAAATTCTTATGGAGTAGGCCTTGCTCTGCTATCGTGCACCGTAAGCGTCCAGCCAACCAGGCTTAGGTGGTGACCTTTCATCAAAAGGCGGGGACGTGAC
GTAAACCGTGCACCTCGAGCAGTAGTAATACAGCACTTCCAGATACGAGGCTCGATACGGAAAAGGCTCGTTTATAGGACTACGTTGCAACAGCGAATCGTTCCCCCGGGAGGGAAAGAGTCTCGTAGCACGTAGCT
AAGAGGCAGGACTAACCAAACAGTAGCTTGCAACGGTGTGGTGAAGGTGGCCAACCGTTCTTTGGTAAATGCAAGTTGAGTCAGAGGTAGGTACACGGCCGTACATTGGTTAACAATCGGGACGAGGCAGACTCGAG
GTGATATCGCCGCTGGCACGTGTGGCGGTTCGCCCCTCGCCAGCGCTCCATAAGATTAGTCGAACCTTATCTCGTATGTTCGTCCCCAACCCAGCTAATCCGGCCCACTAGATCGTGCATCAAGGGGAAGTGCAAGT
GCACTGTTGGCTGACTCGGCTAGCTCATAGCATATTGCACCCTAAACGGATAGCGGTTTCGCCCAACGACTTTTATTAGGGGCTGGGGTCTTAACTGCAGTGCGGACACTATAAGTCAAGATCTCAAGGGACCTGGA
GCAAAGGTCTTCAAGCAACTGTTTTTTAATAAGTCATTTTACGGTAGACAGTGGGTTCAGAAGTCCGATTCCCCTACGTAAGCACATACCCTGGCAGTTCGGGGAAGACATCGGAATTCGCGGCAGTACTCCTAGCG
TCCTTATGCGTTTGGATCTATGCAGAGCTACCGCTATGCGCGTCTCACTCCATTGGCTAACACTATAGCACAAGTAAGCGACGTCCCTCTATTCGACTGAGCTAATCCTCAAAGAGTGATACATAGCAACCTAGCGT
AGCCCCGTA
Sorting DNA Array ...
Printing DNA Array of length: 968
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTT
A:247 C:244 G:252 T:225


...Program finished with exit code 0
Press ENTER to exit console.
```

Pseudocode

1. Define a function “printDNAarray” that takes a DNA array and its length from the Rosalind file as input:

a. Print the message "Printing DNA Array of length: [length]". It will count the elements and it will be showed in the output.

b. Iterate over the elements of the DNA array and print each element.

c. Print a new line.

2. Define a function “processDNAString” that takes a DNA string as input and converts it into a DNA array:

a. Create a new array of characters with a length equal to the length of the DNA string.

b. Iterate over the characters of the DNA string and assign each character to the corresponding position in the array.

c. Return the DNA array. As is given in the Rosalind file, it isn’t sorted yet.

3. Define a function “partition” that takes an array, a low index, and a high index as input and performs partitioning for quicksort:

a. Set the pivot as the element at the high index of the array.

b. Initialize a variable i = low-1.

c. Iterate over the elements from the low index to the high index - 1:

- If the current element is less than the pivot:

    - Increment i.

    - Swap the elements at positions i and j in the array.

d. Swap the element at position i + 1 with the pivot element.

e. Return the index of the pivot element after partitioning.

4. Define a recursive function “quicksortHelper” that performs the quicksort algorithm:

a. If the low index is less than the high index:

- Call the “partition” function to get the pivot index.

- Recursively call “quicksortHelper” with the subarray to the left of the pivot.

- Recursively call “quicksortHelper” with the subarray to the right of the pivot.

5. Define a function “quicksort” that takes an array and its length as input and performs quicksort:

   a. Call “quicksortHelper” with the array, the low index (0), and the high index (length-1).

6. In the main function:

   a. Open the input file "rosalind_dna.txt".

   b. Read a line from the file.

   c. Print the line.

   d. Process the DNA string into a DNA array using the “processDNAString” function.

   e. Print the DNA array.

   f. Sort the DNA array using the “quicksort” function.

   g. Print the sorted DNA array.

   h. Count the occurrences of each type of element (A, C, G, T) in the DNA array.

   i. Print the counts for each type of element.

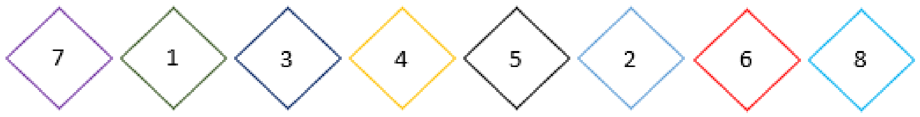
   j. Close the file.

   k. Free the memory allocated for the DNA array.

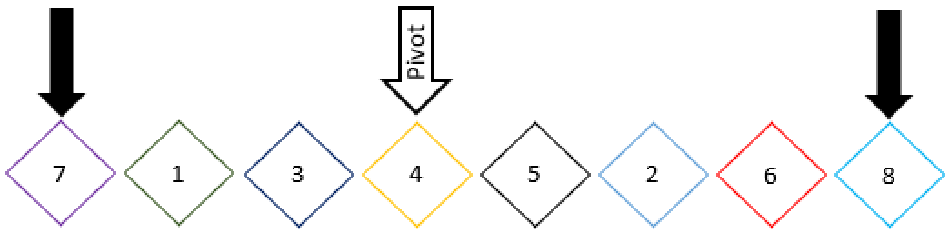
Example

The QuickSort algorithm is based on the "divide and conquer" technique whereby in each recursion, the problem is divided into smaller subproblems, and they are solved separately (applying the same technique) to be joined again once solved.

To explain this algorithm, let's imagine we have a list of 8 numbers that are out of order.



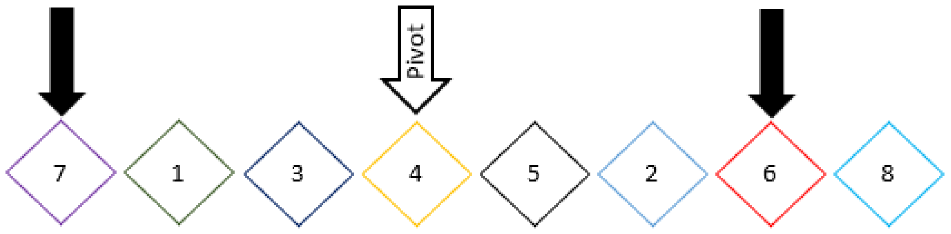
The first thing we must do is assign an element that we will call "pivot". The pivot can be any element in the array, although it is common to select the first element, the last element, or a random element. We also need to assign two elements called pointers.



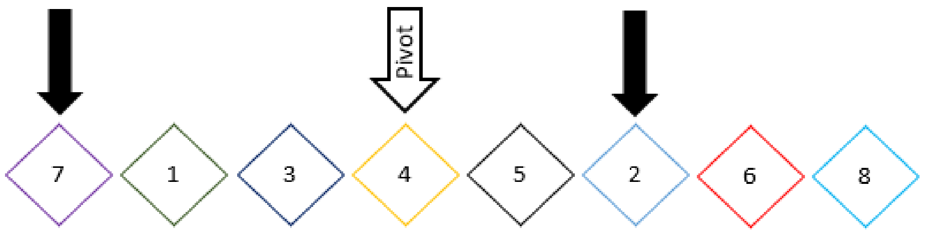
Now we need to arrange the elements of the array so that all elements less than the pivot are before it, and all elements greater are after it.

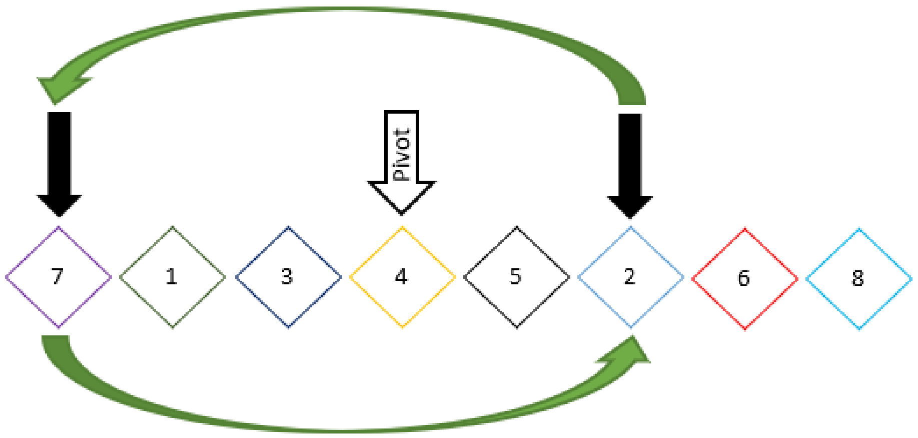
In this case, it is evaluated as follows:

- 7 less than 4, since this is false, we move to the right pointer.
- 8 greater than 4, since this is true the pointer is traversed to the next element.



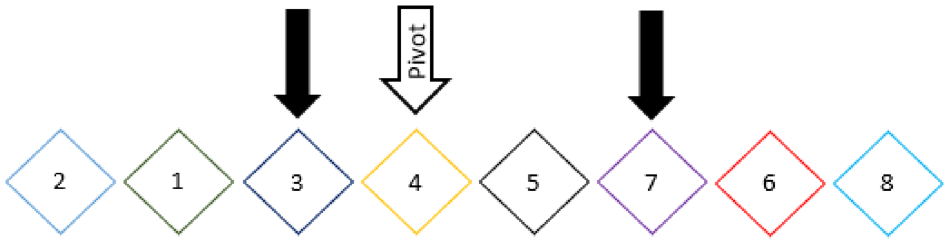
- Since 6 is still greater than 4, we move the right pointer to the next element.
- Now we find the number 2, since 2 is not greater than 4, we must make an exchange with the last value of the pointer on the left.





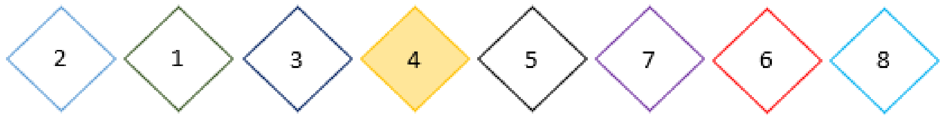
Now we return with the pointer on the left and make the comparison:

- 1 less than 4, this is true, and we loop through the pointer.
- 3 less than 4, this is also true.
- 4 less than 4, this is false and now we move to the pointer to the right.

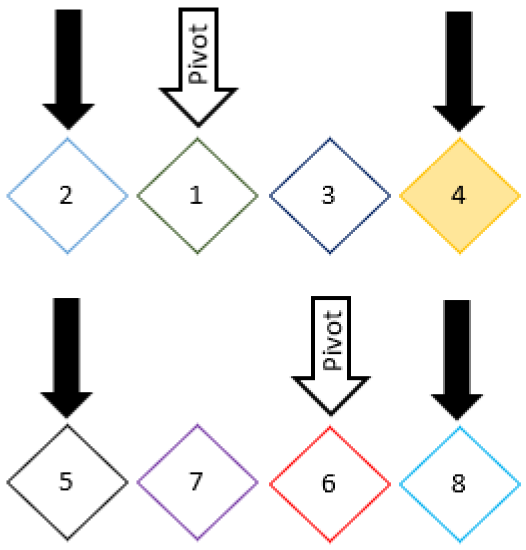


- 7 is greater than 4, this true and we loop through the pointer.
- Since 5 is greater than 4, we can conclude that the number 4 is in its position.

As we can see, we now have two subsets, the one on the left is less than the pivot and the one on the right is greater than the pivot.

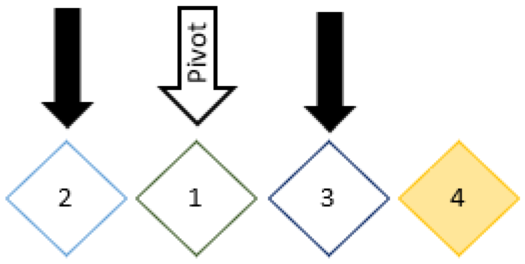


We must divide these subsets and then apply the same method and thus be able to order them.



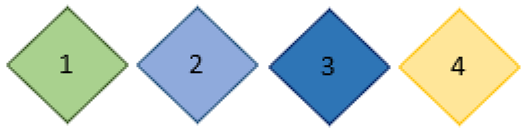
Let's evaluate the first subset. We make the corresponding comparisons:

- 2 less than 1, as this is false, and we must switch to the pointer to the right.
- 4 greater than 1, this is true, and we loop through the pointer.
- 3 greater than 1, this is also true and therefore we traverse the pointer.

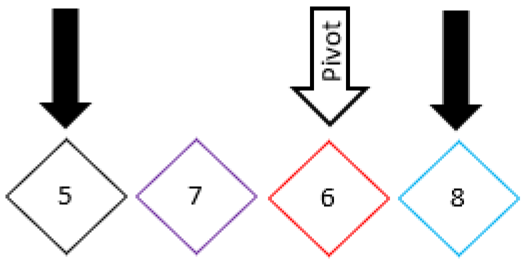


Now we find the following:

1 greater than 1, since this is false, we must swap the elements of the subset. This is the pivot (1) with the last pointer element to the left (2). And in this way, we have the first completely ordered subset elements.

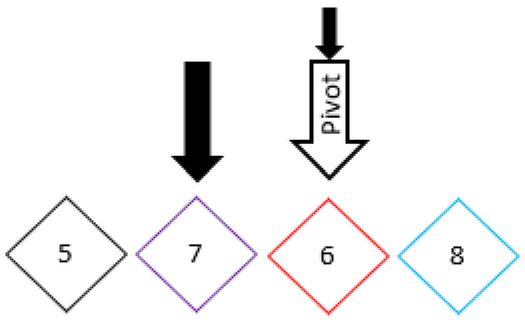


Finally, we need to order the elements of the second subset.



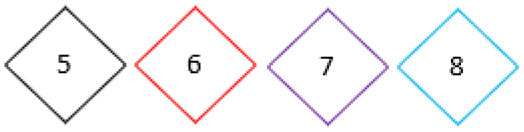
We make the corresponding comparisons:

- 5 less than 6, this is true, and we transverse the pointer.
- 7 less than 6, this is false and now we use the right pointer.
- 8 greater than 6, this is true, and we move the pointer.

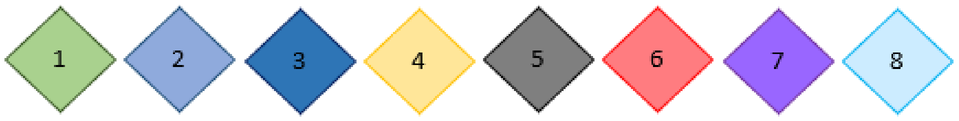


Now, we have the following:

- 6 greater than 6, this is false, so we must interchange the elements of the subset. This is the right pointer element (6) with the last left pointer element (7).



And in this way, we have all the elements of the array ordered.



Conclusions

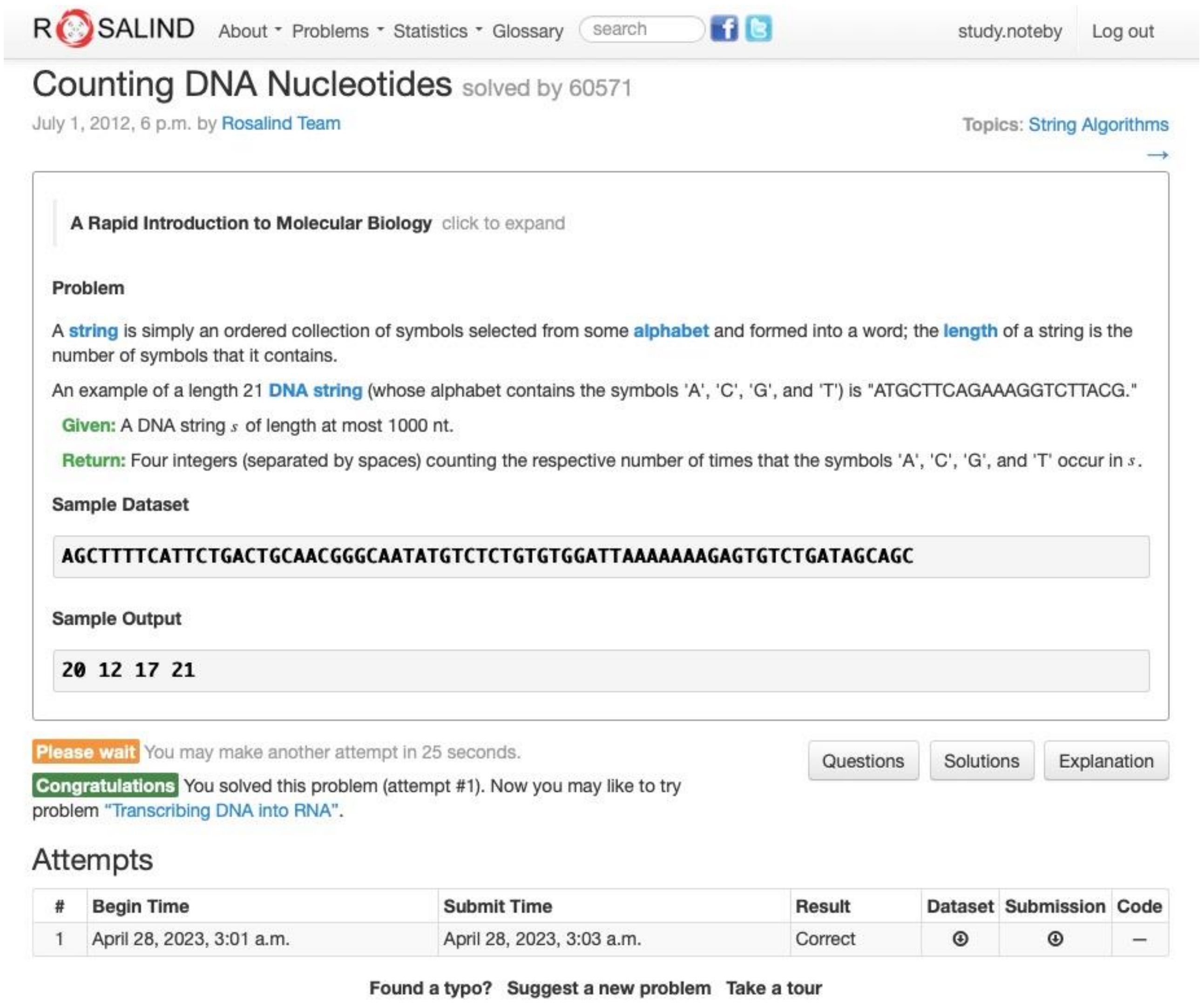
The provided code implements the quicksort algorithm in C++, with a partition function and a recursive function to apply the partition. It also includes reading a DNA strand from a file, processing it into an array, and counting the occurrences of each nucleotide.

Complexity analysis shows that on average quicksort execution time is O(n log n), but in the worst case it can be O(n^2) due to unbalanced partitioning. The spatial complexity is O(log n) on average and in the best case, and O(n) in the worst case.

To improve understanding, it is suggested to divide the content into sections and provide a detailed example with a step-by-step visualization of the algorithm execution. Overall, the explanation covers the nitty-gritty of the Quick Sort algorithm and provides a good foundation for understanding it.

QuickSort is an efficient sorting algorithm based on the divide-and-conquer principle. By selecting a pivot and partitioning the list into smaller subsets, it manages to recursively sort the list until you get the final sorted list. It is widely used due to its good performance in most cases and its simplicity of implementation.

Running the code on the ROSALIND Website



Counting DNA Nucleotides solved by 60571

July 1, 2012, 6 p.m. by Rosalind Team

Topics: String Algorithms

→

A Rapid Introduction to Molecular Biology click to expand

Problem

A **string** is simply an ordered collection of symbols selected from some **alphabet** and formed into a word; the **length** of a string is the number of symbols that it contains.

An example of a length 21 **DNA string** (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Sample Output

```
20 12 17 21
```

Please wait You may make another attempt in 25 seconds.

Congratulations You solved this problem (attempt #1). Now you may like to try problem "Transcribing DNA into RNA".

Attempts

#	Begin Time	Submit Time	Result	Dataset	Submission	Code
1	April 28, 2023, 3:01 a.m.	April 28, 2023, 3:03 a.m.	Correct	⊕	⊕	—

Bibliography

1. C++ Users. (n.d.). qsort function. Retrieved from https://cplusplus.com/reference/cstdlib/qsort/
2. cppreference.com. (n.d.). std::qsort. Retrieved from https://en.cppreference.com/w/cpp/algorithm/qsort
3. GeeksforGeeks. (n.d.). QuickSort. Retrieved from https://www.geeksforgeeks.org/quick-sort/
4. Iliopoulos, V. (2015). The Quicksort algorithm and related topics. arXiv preprint arXiv:1503.02504.
5. Sedgewick, R. (1977). The analysis of quicksort programs. Act Informatica, 7, 327-355.
6. Sedgewick, R. (1978). Implementing quicksort programs. Communications of the ACM, 21(10), 847-857.
7. Online C++ Compiler - online editor. (s. f.). GDB online Debugger. https://www.onlinegdb.com/online_c++_compiler
8. ROSALIND | Problems | Locations. (s. f.). https://rosalind.info/problems/locations/