Cấu trúc dữ liệu và giải thuật

Nhóm 15

TỔNG HỢP SẮP XẾP

Họ và tên	**Mã sinh viên**
Vũ Công Tuấn Dương	B22DCKH024

Ngày nộp : 02/03/2024

1 TỔNG HỢP SẮP XẾP

1. Cài đặt thuật toán sắp xếp đã học: 5

2. Khảo sát thời gian thực thi các thuật toán lần lượt với các giá trị n khác nhau của cùng 1 dãy số: 5

3. Thời gian thực thi của các thuật toán với cùng 1 giá trị n(rất lớn, > 10000) với cùng 1 dãy số có sự khác biệt do ảnh hưởng bởi nhiều yếu tố: may mắn(nếu trong trường hợp tốt nhất), độ phức tạp thuật toán về thời gian-nguyên nhân chủ yếu. Cụ thể:

   - Selection sort:
     (a) Trường hợp tốt nhất: $O(n^2)$
     (b) Trường hợp trung bình: $O(n^2)$
     (c) Trường hợp xấu nhất: $O(n^2)$
   - Insertion sort:
     (a) Trường hợp tốt nhất: $O(n)$
     (b) Trường hợp trung bình: $O(n^2)$
     (c) Trường hợp xấu nhất: $O(n^2)$
   - Bubble sort:
     (a) Trường hợp tốt nhất: $O(n)$
     (b) Trường hợp trung bình: $O(n^2)$
     (c) Trường hợp xấu nhất: $O(n^2)$
   - Merge sort:
     (a) Trường hợp tốt nhất: $O(nlog(n))$
     (b) Trường hợp trung bình: $O(nlog(n))$
     (c) Trường hợp xấu nhất: $O(nlog(n))$
   - Quick sort:
     (a) Trường hợp tốt nhất: $O(nlog(n))$
     (b) Trường hợp trung bình: $O(nlog(n))$
     (c) Trường hợp xấu nhất: $O(n^2)$

   Nhìn vào các thuật toán ta dễ thấy 2 thuật toán sắp xếp có độ phức tạp thấp nhất là Quick Sort và Merge Sort, rất có khả năng khi chạy thực nghiệm 2 thuật toán này sẽ có thời gian chạy tăng chậm khi n càng lớn, còn 3 thuật toán còn lại sẽ có thời gian chạy tăng nhanh khi n càng lớn.

4. Vẽ đồ thị thể hiện thời gian thực thi của mỗi thuật toán phụ thuộc vào n: Hình 1. Ta có thể thấy quickSort và mergeSort có thời gian chạy chênh nhau rất nhỏ (hai đường gần như trùng nhau) nên ta sẽ vẽ riêng 2 đường này: Hình 2.

5. Chương trình được viết như sau:

```
#include <iostream>
#include <algorithm>
#include <chrono>
#include <fstream>
#include <random>
using namespace std;
void selectionSort(int *a, int l, int r);
void bubbleSort(int *a, int l, int r);
void insertionSort(int *a, int l, int r);
void merge(int *a, int l, int mid, int r);
void mergeSort(int *a, int l, int r);
int partition(int *a, int l, int r);
void quickSort(int *a, int l, int r);
void generateRandomArray(int *a, int n);
void measureTime(void (*sort)(int*, int, int), const string& algorit
void generateRandomArray1(int *a, int n);
int main () {
    ofstream file("output.csv");
    file << "algorithm,n,time\n";

    // List of sorting algorithms
    vector<pair<string, void(*)(int*, int, int)>> algorithms = {
        {"quickSort", quickSort},
        {"bubbleSort", bubbleSort},
        {"insertionSort", insertionSort},
        {"selectionSort", selectionSort},
        {"mergeSort", mergeSort}
    };

    for (int n = 5; n <= 100000; n *= 2) {
        int *a = new int[n];
        generateRandomArray1(a, n);
        int *b = new int[n];
        for (int i = 0; i < n; i++) {
                b[i] = a[i];
        }
        for (auto& alg : algorithms) {

            measureTime(alg.second, alg.first, a, n, file);
            for (int i = 0; i < n; i++) {
                a[i] = b[i];
            }
        }
        delete[] a;
    }

    file.close();
    return 0;
}
void insertionSort(int *a, int l, int r) {
```

```
    for (int i = l + 1; i <= r; i++) {
        int key = a[i];
        int j = i;
        for(; j > l && a[j - 1] > key; j--) {
            a[j] = a[j - 1];
        }
        a[j] = key;
    }
}
void selectionSort(int *a, int l, int r) {
    for (int i = l; i <= r; i++) {
        int min_index = i;
        for (int j = i + 1; j <= r; j++) {
            if (a[j] < a[min_index]) {
                min_index = j;
            }
        }
        if (min_index != i) {
            swap(a[i], a[min_index]);
        }
    }
}
void bubbleSort(int *a, int l, int r) {
    for (int i = r; i > l; i--) {
        bool swapped = false;
        for (int j = l; j < i; j++) {
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
void mergeSort(int *a, int l, int r) {
    if (l >= r) {
        return;
    }
    int mid = (l + r) / 2;
    mergeSort(a, l, mid);
    mergeSort(a, mid + 1, r);
    merge(a, l, mid, r);
}

void merge(int *a, int l, int m, int r) {
    int tempArray[r - l + 1];
    int i = l;
    int j = m + 1;
```

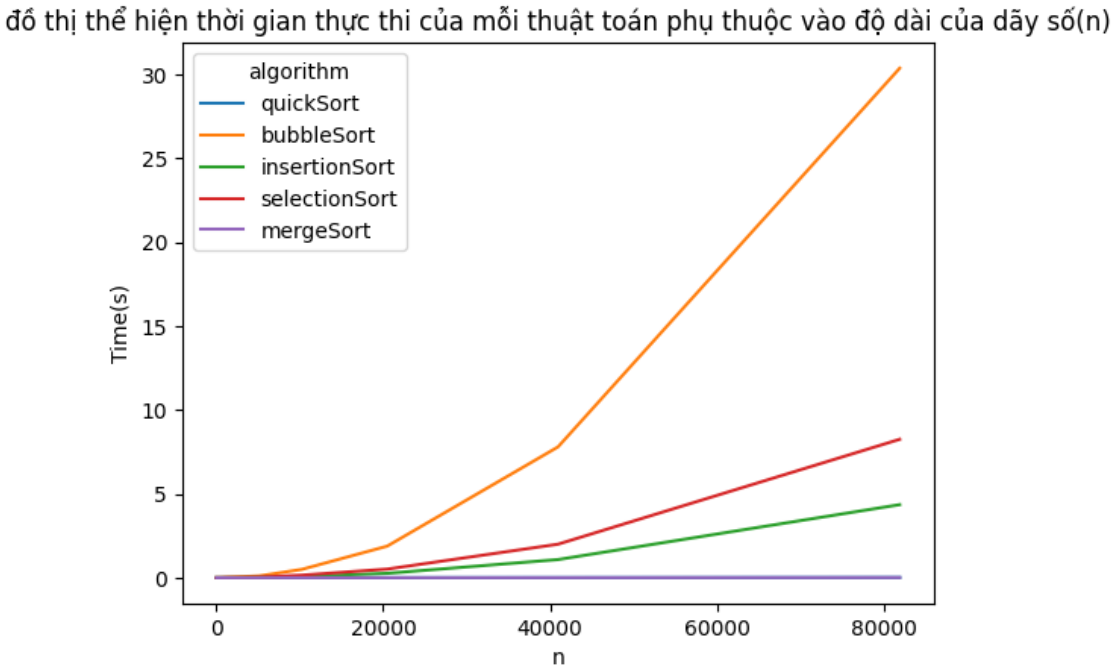
```
    int k = 0;
    while (i <= m && j <= r) {
        if (a[i] < a[j]) {
            tempArray[k++] = a[i++];
        } else {
            tempArray[k++] = a[j++];
        }
    }
    while (i <= m) {
        tempArray[k++] = a[i++];
    }
    while (j <= r) {
        tempArray[k++] = a[j++];
    }
    for (i = l, k = 0; i <= r; i++, k++) {
        a[i] = tempArray[k];
    }
}

//quicksort
int partition(int *a, int l, int r) {
    int pivot = a[l];
    int lo = l + 1;
    int hi = r;
    while (true) {
        while (lo <= hi && a[lo] <= pivot) {
            lo++;
        }
        while (lo <= hi && a[hi] >= pivot) {
            hi--;
        }
        if (lo >= hi) {
            break;
        }
        swap(a[lo], a[hi]);
    }
    if (a[hi] < pivot) {
        swap(a[l], a[hi]);
    }
    return hi;
}
void quickSort(int *a, int l, int r) {
    if (l >= r) {
        return;
    }
    int pivotIndex = partition(a, l, r);
    quickSort(a, l, pivotIndex - 1);
    quickSort(a, pivotIndex + 1, r);
}
void generateRandomArray(int *a, int n) {
```

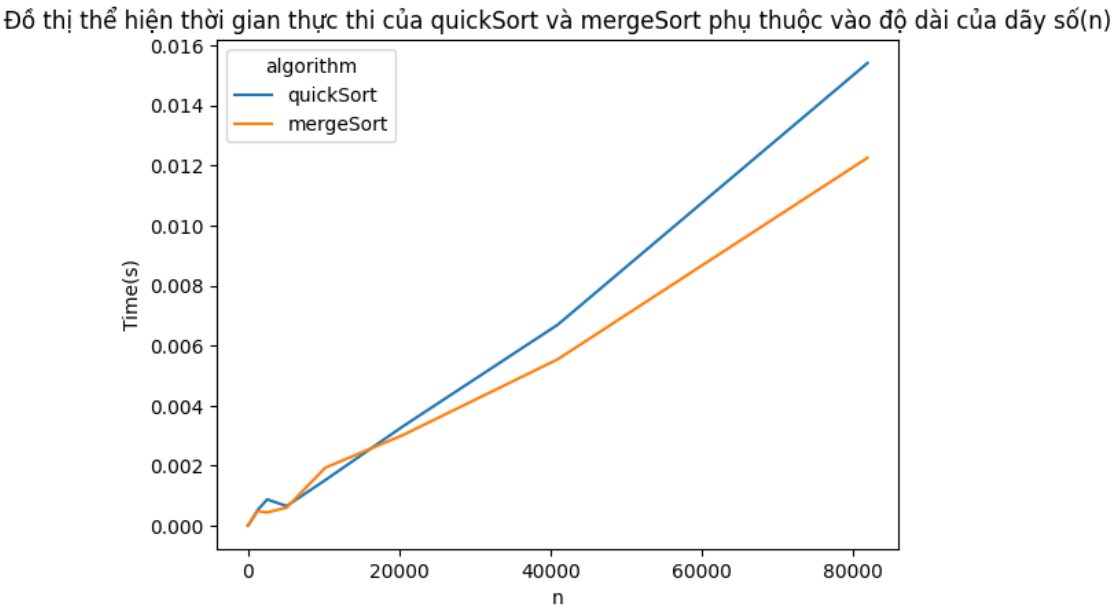
```
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 100);
    for (int i = 0; i < n; i++) {
        a[i] = dis(gen);
    }
}

void measureTime(void (*sort)(int*, int, int), const string& algorit
    auto start = chrono::high_resolution_clock::now();
    sort(a, 0, n - 1);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    file << algorithm << "," << n << "," << elapsed.count() << "\n";
}
void generateRandomArray1(int *a, int n) {
    random_device rd;
    mt19937 gen(rd());
    vector<double> weights(1000, 1); // 1000 possible outcomes each
    discrete_distribution<> dis(weights.begin(), weights.end());
    for (int i = 0; i < n; i++) {
        a[i] = dis(gen) + 1; // +1 because discrete_distribution gen
    }
}
```

6. Các đồ thị:



Hình 1: Đồ thị thể hiện thời gian thực thi của mỗi thuật toán phụ thuộc vào độ dài của dãy số(n)



Hình 2: Đồ thị thể hiện thời gian thực thi của quickSort và mergeSort phụ thuộc vào độ dài của dãy số(n)