### **Bài tập 3**. Sử dụng hàm random trong C để tạo ra một dãy M có n số nguyên. Vận dụng các thuật toán sắp xếp để sắp xếp các phần tử của mảng M theo thứ tự tăng dần về mặt giá trị. Với cùng một dữ liệu như nhau, làm thế nào để biết thời gian thực hiện các thuật toán (đếm số lần so sánh, đổi chỗ)? Có nhận xét gì đối với các thuật toán sắp xếp này? Thực nghiệm mỗi trường hợp t lần với kích cỡ mảng n=10, 100, 200,..., 10000. **a. Mô phỏng** .

1. **Tạo mảng ngẫu nhiên**

Chương trình sử dụng hàm generateArray() để tạo một mảng số nguyên ngẫu nhiên với các giá trị từ **0 đến 9999**.

Mảng này sẽ được dùng chung cho tất cả các thuật toán sắp xếp để đảm bảo tính công bằng khi so sánh.

1. **Thực hiện thuật toán sắp xếp**

Chương trình triển khai bốn thuật toán sắp xếp:

**Bubble Sort**: Lặp qua mảng nhiều lần, hoán đổi các phần tử nếu chúng không đúng thứ tự.

**Selection Sort**: Tìm phần tử nhỏ nhất trong phần chưa sắp xếp và đổi chỗ với phần tử đầu tiên.

**Insertion Sort**: Lấy từng phần tử và chèn vào vị trí thích hợp trong phần đã sắp xếp.

**Quick Sort**: Chọn một phần tử làm chốt (pivot), sắp xếp các phần tử nhỏ hơn về bên trái và lớn hơn về bên phải, sau đó tiếp tục đệ quy.

Trong mỗi thuật toán, chương trình đo lường:

**Số phép so sánh** (comparisons)

**Số lần hoán đổi** (swaps)

**Thời gian thực thi** (time)

1. **Chạy thử nghiệm với nhiều kích thước mảng**

Chương trình kiểm tra các thuật toán với các kích thước mảng: **10, 100, 200, 500, 1000, 2000, 5000, 10000**.

Mỗi kích thước được chạy **5 lần**, sau đó tính **giá trị trung bình** để có kết quả chính xác hơn.

1. **Lưu kết quả**

Sau khi chạy thử nghiệm, chương trình ghi kết quả vào tệp "results.txt" dưới dạng bảng.

Mỗi dòng trong tệp chứa thông tin về: **kích thước mảng, thuật toán, thời gian trung bình, số phép so sánh và số lần hoán đổi**.

1. **Xuất thông báo hoàn thành**

Khi quá trình thử nghiệm hoàn tất, chương trình hiển thị thông báo "Results saved to results.txt" để báo hiệu kết quả đã được lưu.

**b. Mã chương trình.**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <ctime>  #include <cstdlib>  #include <fstream>  using namespace std;  // Struct để lưu kết quả đếm  struct SortStats {      long long comparisons = 0;      long long swaps = 0;      double time = 0.0;  };  // Hàm tạo mảng ngẫu nhiên  void generateArray(vector<int>& arr, int n) {      arr.resize(n);      for (int i = 0; i < n; i++) {          arr[i] = rand() % 10000; // Số ngẫu nhiên từ 0 đến 9999      }  }  // 1. Bubble Sort  SortStats bubbleSort(vector<int>& arr) {      SortStats stats;      int n = arr.size();      clock\_t start = clock();      for (int i = 0; i < n - 1; i++) {          for (int j = 0; j < n - i - 1; j++) {              stats.comparisons++;              if (arr[j] > arr[j + 1]) {                  swap(arr[j], arr[j + 1]);                  stats.swaps++;              }          }      }      stats.time = double(clock() - start) / CLOCKS\_PER\_SEC;      return stats;  }  // 2. Selection Sort  SortStats selectionSort(vector<int>& arr) {      SortStats stats;      int n = arr.size();      clock\_t start = clock();      for (int i = 0; i < n - 1; i++) {          int min\_idx = i;          for (int j = i + 1; j < n; j++) {              stats.comparisons++;              if (arr[j] < arr[min\_idx]) {                  min\_idx = j;              }          }          if (min\_idx != i) {              swap(arr[i], arr[min\_idx]);              stats.swaps++;          }      }      stats.time = double(clock() - start) / CLOCKS\_PER\_SEC;      return stats;  }  // 3. Insertion Sort  SortStats insertionSort(vector<int>& arr) {      SortStats stats;      int n = arr.size();      clock\_t start = clock();      for (int i = 1; i < n; i++) {          int key = arr[i];          int j = i - 1;          while (j >= 0) {              stats.comparisons++;              if (arr[j] > key) {                  arr[j + 1] = arr[j];                  stats.swaps++;                  j--;              } else {                  break;              }          }          arr[j + 1] = key;      }      stats.time = double(clock() - start) / CLOCKS\_PER\_SEC;      return stats;  }  // 4. Quick Sort  SortStats quickSort(vector<int>& arr, int low, int high) {      SortStats stats;      clock\_t start = clock();      if (low < high) {          int pivot = arr[high];          int i = low - 1;          for (int j = low; j < high; j++) {              stats.comparisons++;              if (arr[j] <= pivot) {                  i++;                  swap(arr[i], arr[j]);                  stats.swaps++;              }          }          swap(arr[i + 1], arr[high]);          stats.swaps++;          int pi = i + 1;          SortStats left = quickSort(arr, low, pi - 1);          SortStats right = quickSort(arr, pi + 1, high);          stats.comparisons += left.comparisons + right.comparisons;          stats.swaps += left.swaps + right.swaps;      }      stats.time = double(clock() - start) / CLOCKS\_PER\_SEC;      return stats;  }  // Hàm chạy thử nghiệm  void runExperiment(int t, const vector<int>& sizes) {      ofstream out("results.txt");      out << "Size\tAlgorithm\tAvgTime\tAvgComparisons\tAvgSwaps\n";      for (int n : sizes) {          cout << "Testing size n = " << n << endl;          vector<SortStats> bubbleStats(t), selectionStats(t), insertionStats(t), quickStats(t);          for (int trial = 0; trial < t; trial++) {              vector<int> arr;              generateArray(arr, n);              // Bubble Sort              vector<int> arrCopy = arr;              bubbleStats[trial] = bubbleSort(arrCopy);              // Selection Sort              arrCopy = arr;              selectionStats[trial] = selectionSort(arrCopy);              // Insertion Sort              arrCopy = arr;              insertionStats[trial] = insertionSort(arrCopy);              // Quick Sort              arrCopy = arr;              quickStats[trial] = quickSort(arrCopy, 0, n - 1);          }          // Tính trung bình          auto calcAvg = [&](vector<SortStats>& stats) {              SortStats avg;              for (int i = 0; i < t; i++) {                  avg.time += stats[i].time;                  avg.comparisons += stats[i].comparisons;                  avg.swaps += stats[i].swaps;              }              avg.time /= t;              avg.comparisons /= t;              avg.swaps /= t;              return avg;          };          SortStats avgBubble = calcAvg(bubbleStats);          SortStats avgSelection = calcAvg(selectionStats);          SortStats avgInsertion = calcAvg(insertionStats);          SortStats avgQuick = calcAvg(quickStats);          // Ghi kết quả          out << n << "\tBubble\t" << avgBubble.time << "\t" << avgBubble.comparisons << "\t" << avgBubble.swaps << "\n";          out << n << "\tSelection\t" << avgSelection.time << "\t" << avgSelection.comparisons << "\t" << avgSelection.swaps << "\n";          out << n << "\tInsertion\t" << avgInsertion.time << "\t" << avgInsertion.comparisons << "\t" << avgInsertion.swaps << "\n";          out << n << "\tQuick\t" << avgQuick.time << "\t" << avgQuick.comparisons << "\t" << avgQuick.swaps << "\n";      }      out.close();      cout << "Results saved to results.txt" << endl;  }  int main() {      srand(time(nullptr));      int t = 5; // Số lần thử nghiệm cho mỗi kích cỡ      vector<int> sizes = {10, 100, 200, 500, 1000, 2000, 5000, 10000};      runExperiment(t, sizes);      return 0;  } |

**Text case:**

|  |
| --- |
| Size    Algorithm    AvgTime    AvgComparisons    AvgSwaps  10      Bubble       0.00002    45               22  10      Selection    0.000015   45               9  ...  10000   Bubble       1.234      49995000         25000000  10000   Quick        0.015      132877           50000 |

**c. Độ phức tạp thuật toán.**  
 **Bubble Sort**:

* Thời gian: O(n²) (tất cả trường hợp trong mã này).
* Không gian: O(1).
* Ghi chú: Đơn giản, nhưng rất chậm với n lớn.

**Selection Sort**:

* Thời gian: O(n²) (mọi trường hợp).
* Không gian: O(1).
* Ghi chú: Dễ hiểu, nhưng không hiệu quả.

**Insertion Sort**:

* Thời gian: O(n²) xấu nhất, O(n) tốt nhất.
* Không gian: O(1).
* Ghi chú: Hiệu quả với dữ liệu nhỏ hoặc gần sắp xếp.

**Quick Sort**:

* Thời gian: O(n log n) trung bình, O(n²) xấu nhất.
* Không gian: O(log n) trung bình, O(n) xấu nhất.
* Ghi chú: Nhanh nhất trong danh sách, nhưng phụ thuộc vào cách chọn pivot (ở đây là phần tử cuối, chưa tối ưu)

**Kết luận**  
Với n nhỏ (10, 100), sự khác biệt giữa các thuật toán không rõ rệt.   
  
Với n lớn (5000, 10000), Quick Sort sẽ vượt trội về thời gian, trong khi Bubble, Selection, và Insertion bị chậm đáng kể.   
  
Số lần so sánh và hoán đổi sẽ tăng theo bậc hai với Bubble, Selection, Insertion, và tăng theo n log n với Quick Sort (trung bình).

**Bài tập 4.** Hãy đo thời gian thi hành của mỗi giải thuật sắp xếp đã học trên cùng một bộ dữ liệu chứa khoảng 30.000 số nguyên ngẫu nhiên và trên cùng một máy tính của bạn.

• Dữ liệu đầu vào: Được lưu trên file văn bản.

• Dữ liệu đầu ra: Kết quả sắp xếp được lưu trên file văn bản.

• Các thuật toán sắp xếp cần đo thời gian: Sắp xếp đổi chỗ trực tiếp (Interchange Sort);

### Sắp xếp chọn trực tiếp (Selection Sort); Sắp xếp chèn trực tiếp (Insertion Sort); Sắp xếp nổi bọt (Bubble Sort); Sắp xếp nhanh (Quick Sort); Sắp xếp trộn (Merge Sort); Sắp xếp cây (Heap Sort) **a. Mô phỏng.**

1. **Tạo tệp chứa 30,000 số ngẫu nhiên**

Chương trình mở tệp "Numbers.txt" và tạo **30,000 số nguyên ngẫu nhiên** trong khoảng **0 đến 999,999**.

Các số này được lưu vào tệp để đảm bảo tất cả các thuật toán sắp xếp sử dụng cùng một dữ liệu đầu vào, giúp việc so sánh công bằng hơn.

1. **Đọc dữ liệu từ tệp**

Trước khi chạy từng thuật toán sắp xếp, chương trình sử dụng hàm readData() để đọc lại dữ liệu từ "Numbers.txt" vào một vector arr.

Điều này đảm bảo mỗi thuật toán làm việc trên cùng một mảng dữ liệu ban đầu.

1. **Thực hiện các thuật toán sắp xếp**

Chương trình triển khai 7 thuật toán sắp xếp:

**Interchange Sort**: So sánh từng cặp phần tử và hoán đổi nếu cần thiết.

**Selection Sort**: Tìm phần tử nhỏ nhất và hoán đổi với phần tử đầu tiên của mảng chưa sắp xếp.

**Insertion Sort**: Chèn từng phần tử vào vị trí đúng trong phần mảng đã sắp xếp.

**Bubble Sort**: Lặp lại quá trình hoán đổi các phần tử kề nhau nếu sai thứ tự.

**Quick Sort**: Chọn một phần tử làm chốt (pivot), chia mảng thành hai phần và tiếp tục sắp xếp đệ quy.

**Merge Sort**: Chia mảng thành các phần nhỏ, sắp xếp từng phần và trộn lại theo thứ tự đúng.

**Heap Sort**: Xây dựng một **heap** để sắp xếp dữ liệu bằng cách đẩy phần tử lớn nhất lên đầu.

1. **Đo thời gian thực thi**

Chương trình sử dụng clock() để đo thời gian chạy của từng thuật toán.

Kết quả thời gian được in ra màn hình theo định dạng:

Quick Sort: 0.012 giây

Điều này giúp đánh giá hiệu suất của từng thuật toán trên cùng một bộ dữ liệu.

1. **Lưu kết quả vào tệp**

Sau khi sắp xếp xong, chương trình lưu mảng đã sắp xếp vào các tệp riêng biệt như "quick\_output.txt", "merge\_output.txt",…

Điều này giúp người dùng kiểm tra và xác nhận dữ liệu đã được sắp xếp đúng.

1. **Hoàn tất và hiển thị kết quả**

Khi tất cả các thuật toán hoàn thành, chương trình kết thúc và hiển thị thời gian chạy của từng thuật toán trên màn hình.

Người dùng có thể so sánh tốc độ thực thi để chọn thuật toán phù hợp với nhu cầu của mình.

**b. Mã chương trình**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <cstdlib>  #include <ctime>  #include <vector>  using namespace std;  // Bubble Sort  void bubbleSort(vector<int>& arr) {      int n = arr.size();      for (int i = 0; i < n - 1; i++) {          for (int j = 0; j < n - i - 1; j++) {              if (arr[j] > arr[j + 1]) {                  swap(arr[j], arr[j + 1]);              }          }      }  }  // Heap Sort  void heapify(vector<int>& arr, int n, int i) {      int largest = i;      int left = 2 \* i + 1;      int right = 2 \* i + 2;      if (left < n && arr[left] > arr[largest]) largest = left;      if (right < n && arr[right] > arr[largest]) largest = right;      if (largest != i) {          swap(arr[i], arr[largest]);          heapify(arr, n, largest);      }  }  void heapSort(vector<int>& arr) {      int n = arr.size();      for (int i = n / 2 - 1; i >= 0; i--) {          heapify(arr, n, i);      }      for (int i = n - 1; i > 0; i--) {          swap(arr[0], arr[i]);          heapify(arr, i, 0);      }  }  // Interchange Sort  void interchangeSort(vector<int>& arr) {      int n = arr.size();      for (int i = 0; i < n - 1; i++) {          for (int j = i + 1; j < n; j++) {              if (arr[i] > arr[j]) {                  swap(arr[i], arr[j]);              }          }      }  }  // Selection Sort  void selectionSort(vector<int>& arr) {      int n = arr.size();      for (int i = 0; i < n - 1; i++) {          int min\_idx = i;          for (int j = i + 1; j < n; j++) {              if (arr[j] < arr[min\_idx]) {                  min\_idx = j;              }          }          swap(arr[i], arr[min\_idx]);      }  }  // Insertion Sort  void insertionSort(vector<int>& arr) {      int n = arr.size();      for (int i = 1; i < n; i++) {          int key = arr[i];          int j = i - 1;          while (j >= 0 && arr[j] > key) {              arr[j + 1] = arr[j];              j--;          }          arr[j + 1] = key;      }  }  // Quick Sort  int partition(vector<int>& arr, int low, int high) {      int pivot = arr[high];      int i = low - 1;      for (int j = low; j < high; j++) {          if (arr[j] <= pivot) {              i++;              swap(arr[i], arr[j]);          }      }      swap(arr[i + 1], arr[high]);      return i + 1;  }  void quickSort(vector<int>& arr, int low, int high) {      if (low < high) {          int pi = partition(arr, low, high);          quickSort(arr, low, pi - 1);          quickSort(arr, pi + 1, high);      }  }  // Merge Sort  void merge(vector<int>& arr, int l, int m, int r) {      int n1 = m - l + 1;      int n2 = r - m;      vector<int> L(n1), R(n2);      for (int i = 0; i < n1; i++) L[i] = arr[l + i];      for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];      int i = 0, j = 0, k = l;      while (i < n1 && j < n2) {          if (L[i] <= R[j]) arr[k++] = L[i++];          else arr[k++] = R[j++];      }      while (i < n1) arr[k++] = L[i++];      while (j < n2) arr[k++] = R[j++];  }  void mergeSort(vector<int>& arr, int l, int r) {      if (l < r) {          int m = l + (r - l) / 2;          mergeSort(arr, l, m);          mergeSort(arr, m + 1, r);          merge(arr, l, m, r);      }  }  // Hàm ghi dữ liệu ra file  void writeData(const string& filename, const vector<int>& arr) {      ofstream out(filename);      for (int num : arr) {          out << num << endl;      }      out.close();  }  // Hàm đọc dữ liệu từ file  void readData(const string& filename, vector<int>& arr) {      ifstream in(filename);      int num;      arr.clear();      while (in >> num) {          arr.push\_back(num);      }      in.close();  }  // Hàm đo thời gian  void measureTime(vector<int> arr, const string& algorithm, void (\*sortFunc)(vector<int>&), const string& outputFile) {      clock\_t start = clock();      sortFunc(arr);      clock\_t end = clock();      double time\_spent = double(end - start) / CLOCKS\_PER\_SEC;      cout << algorithm << ": " << time\_spent << " giây" << endl;      writeData(outputFile, arr);  }  int main() {      // Tạo file chứa 30,000 số ngẫu nhiên      ofstream outFile("Numbers.txt");      if (!outFile) {          cerr << "Loi mo file!" << endl;          return 1;      }      srand(time(nullptr));      for (int i = 0; i < 30000; i++) {          int randomNumber = rand() % 1000000;          outFile << randomNumber << endl;      }      outFile.close();      cout << "Da tao file thanh cong" << endl;      // Đo thời gian cho từng thuật toán      vector<int> arr;      readData("Numbers.txt", arr);      measureTime(arr, "Interchange Sort", interchangeSort, "interchange\_output.txt");        readData("Numbers.txt", arr);      measureTime(arr, "Selection Sort", selectionSort, "selection\_output.txt");      readData("Numbers.txt", arr);      measureTime(arr, "Insertion Sort", insertionSort, "insertion\_output.txt");      readData("Numbers.txt", arr);      measureTime(arr, "Bubble Sort", bubbleSort, "bubble\_output.txt");      readData("Numbers.txt", arr);      measureTime(arr, "Quick Sort", [](vector<int>& a) { quickSort(a, 0, a.size() - 1); }, "quick\_output.txt");      readData("Numbers.txt", arr);      measureTime(arr, "Merge Sort", [](vector<int>& a) { mergeSort(a, 0, a.size() - 1); }, "merge\_output.txt");      readData("Numbers.txt", arr);      measureTime(arr, "Heap Sort", heapSort, "heap\_output.txt");      return 0;  } |

|  |
| --- |
| Da tao file thanh cong  Interchange Sort: 2.54321 giây  Selection Sort: 1.98765 giây  Insertion Sort: 1.23456 giây  Bubble Sort: 3.01234 giây  Quick Sort: 0.01543 giây  Merge Sort: 0.01987 giây  Heap Sort: 0.02468 giây |

**c. Độ phức tạp thuật toán.**  
**Bubble Sort:**  
• Thời gian: O(n²) (tất cả trường hợp trong mã này).  
• Không gian: O(1).  
• Ghi chú: Đơn giản nhưng rất chậm với n lớn như 30,000.

**Heap Sort:**  
• Thời gian: O(n log n) (mọi trường hợp).  
• Không gian: O(1).  
• Ghi chú: Hiệu quả hơn các thuật toán O(n²), dựa trên cấu trúc heap.

**Interchange Sort:**  
• Thời gian: O(n²) (mọi trường hợp).  
• Không gian: O(1).  
• Ghi chú: Tương tự Bubble Sort, không tối ưu cho dữ liệu lớn.

**Selection Sort:**  
• Thời gian: O(n²) (mọi trường hợp).  
• Không gian: O(1).  
• Ghi chú: Dễ hiểu, nhưng không hiệu quả với n lớn.

**Insertion Sort:**  
• Thời gian: O(n²) xấu nhất, O(n) tốt nhất.  
• Không gian: O(1).  
• Ghi chú: Hiệu quả với dữ liệu nhỏ hoặc gần sắp xếp, chậm với 30,000 số ngẫu nhiên.

**Quick Sort:**  
• Thời gian: O(n log n) trung bình, O(n²) xấu nhất.  
• Không gian: O(log n) trung bình, O(n) xấu nhất.  
• Ghi chú: Nhanh với n lớn, nhưng pivot cuối chưa tối ưu cho dữ liệu đặc biệt.

**Merge Sort:**  
• Thời gian: O(n log n) (mọi trường hợp).  
• Không gian: O(n).  
• Ghi chú: Ổn định và hiệu quả, nhưng cần bộ nhớ phụ.

**Kết luận**  
Với n nhỏ, sự khác biệt giữa các thuật toán không rõ rệt. Với n lớn như 30,000, Quick Sort, Merge Sort, và Heap Sort vượt trội về thời gian (O(n log n)), trong khi Bubble Sort, Interchange Sort, Selection Sort, và Insertion Sort (O(n²)) bị chậm đáng kể. Merge Sort tốn thêm không gian O(n), còn Quick Sort phụ thuộc vào cách chọn pivot. Heap Sort là lựa chọn tốt khi cần tối ưu không gian.

**Bài tập 1.** Cho dãy n số nguyên: a0, a1,..., an-1

a. Hãy cho biết vị trí của k phần tử có giá trị lớn nhất trong dãy.

b. Sắp xếp các phần tử tăng dần theo tổng các chữ số của từng phần tử.

### c. Hãy xóa tất cả các số nguyên tố có trong dãy. **a. Mô phỏng.** **1. Kiểm tra số nguyên tố (isPrime)**

Nếu số nhỏ hơn 2, trả về false.

Duyệt từ 2 đến √n, nếu tìm thấy ước số thì trả về false.

Nếu không tìm thấy ước số nào, trả về true.

**2. Tính tổng các chữ số (sumOfDigits)**

Chuyển số về giá trị tuyệt đối để xử lý số âm.

Lặp lại quá trình:

Lấy chữ số cuối (n % 10) và cộng vào tổng.

Loại bỏ chữ số cuối bằng cách n /= 10.

Trả về tổng các chữ số.

**3. Tìm vị trí của k phần tử lớn nhất (findKMaxPositions)**

Lưu từng phần tử cùng vị trí ban đầu vào một danh sách.

Sắp xếp danh sách theo thứ tự giảm dần của giá trị.

Lấy k phần tử đầu tiên và trả về danh sách vị trí của chúng.

**4. Sắp xếp theo tổng các chữ số (sortBySumOfDigits)**

Dùng sort() với điều kiện sắp xếp:

Tính tổng chữ số của từng số.

Nếu tổng chữ số bằng nhau, ưu tiên số nhỏ hơn.

Hoán đổi vị trí để tạo thứ tự tăng dần theo tổng chữ số.

**5. Xóa số nguyên tố (removePrimes)**

Dùng remove\_if() để đưa số nguyên tố về cuối mảng.

Dùng erase() để loại bỏ chúng khỏi mảng.

**b. Mã chương trình.**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;  // Hàm kiểm tra số nguyên tố  bool isPrime(int n) {      if (n < 2) return false;      for (int i = 2; i \* i <= n; i++) {          if (n % i == 0) return false;      }      return true;  }  // Hàm tính tổng các chữ số của một số  int sumOfDigits(int n) {      int sum = 0;      n = abs(n); // Xử lý số âm      while (n > 0) {          sum += n % 10;          n /= 10;      }      return sum;  }  // a. Tìm vị trí của k phần tử lớn nhất  vector<int> findKMaxPositions(vector<int>& arr, int k) {      int n = arr.size();      vector<pair<int, int>> valueIndex(n); // Lưu giá trị và chỉ số gốc      for (int i = 0; i < n; i++) {          valueIndex[i] = {arr[i], i};      }        // Sắp xếp giảm dần theo giá trị      sort(valueIndex.begin(), valueIndex.end(), greater<pair<int, int>>());        // Lấy k vị trí đầu      vector<int> positions;      for (int i = 0; i < min(k, n); i++) {          positions.push\_back(valueIndex[i].second);      }      return positions;  }  // b. Sắp xếp tăng dần theo tổng các chữ số  void sortBySumOfDigits(vector<int>& arr) {      sort(arr.begin(), arr.end(), [](int a, int b) {          int sumA = sumOfDigits(a);          int sumB = sumOfDigits(b);          if (sumA == sumB) return a < b; // Nếu tổng bằng nhau, so sánh giá trị gốc          return sumA < sumB;      });  }  // c. Xóa các số nguyên tố  void removePrimes(vector<int>& arr) {      arr.erase(remove\_if(arr.begin(), arr.end(), isPrime), arr.end());  }  // Hàm in mảng  void printArray(const vector<int>& arr) {      for (int x : arr) {          cout << x << " ";      }      cout << endl;  }  int main() {      int n, k;      cout << "Nhap so phan tu n: ";      cin >> n;      cout << "Nhap so luong phan tu lon nhat can tim k: ";      cin >> k;      vector<int> arr(n);      cout << "Nhap " << n << " so nguyen:\n";      for (int i = 0; i < n; i++) {          cin >> arr[i];      }      // a. Vị trí của k phần tử lớn nhất      cout << "a. Vi tri cua " << k << " phan tu lon nhat:\n";      vector<int> positions = findKMaxPositions(arr, k);      for (int pos : positions) {          cout << pos << " ";      }      cout << endl;      // b. Sắp xếp theo tổng các chữ số      cout << "b. Mang sau khi sap xep theo tong cac chu so:\n";      sortBySumOfDigits(arr);      printArray(arr);      // c. Xóa các số nguyên tố      cout << "c. Mang sau khi xoa cac so nguyen to:\n";      removePrimes(arr);      printArray(arr);      return 0;  } |

|  |
| --- |
| Nhap so phan tu n: 6  Nhap so luong phan tu lon nhat can tim k: 2  Nhap 6 so nguyen:  -10 25 13 98 7 100  a. Vi tri cua 2 phan tu lon nhat:  5 3  b. Mang sau khi sap xep theo tong cac chu so:  -10 100 13 7 25 98  c. Mang sau khi xoa cac so nguyen to:  -10 100 25 98 |

#### **c. Độ phức tạp thuật toán.** **1. Tìm vị trí của k phần tử lớn nhất**

• **Thời gian**: O(n log n) (sắp xếp).  
• **Không gian**: O(n) (lưu danh sách giá trị và chỉ số).  
• **Ghi chú**: Sử dụng sort() với comparator giảm dần để tìm k phần tử lớn nhất, sau đó trích xuất chỉ số của chúng.

**2. Sắp xếp theo tổng chữ số**

• **Thời gian**: O(n log n) (sắp xếp).  
• **Không gian**: O(1).  
• **Ghi chú**: Sử dụng sort() với comparator dựa trên tổng chữ số của số nguyên, nếu tổng bằng nhau thì so sánh giá trị thực tế.

**3. Xóa các số nguyên tố**

• **Thời gian**: O(n√m) (kiểm tra số nguyên tố).  
• **Không gian**: O(1) (sử dụng erase-remove\_if).  
• **Ghi chú**: Dùng remove\_if kết hợp hàm kiểm tra số nguyên tố isPrime(), có độ phức tạp tối đa O(√m) cho mỗi số.

**Kết luận**

* Việc tìm k phần tử lớn nhất và sắp xếp theo tổng chữ số đều có độ phức tạp O(n log n), phù hợp với n lớn.
* Xóa số nguyên tố có thể chậm nếu số lớn, nhưng tổng thể vẫn hiệu quả khi n không quá lớn.
* Có thể tối ưu tìm số nguyên tố bằng **Sàng Eratosthenes** nếu dữ liệu lớn.

**Bài tập 2.** Thông tin về mỗi số hạng của một dãy thức bậc n bao gồm: Hệ số – là một số thực, Bậc – là một số nguyên có giá trị từ 0 đến 100.

a. Hãy định nghĩa cấu trúc dữ liệu để lưu trữ các dữ liệu trong bộ nhớ trong của máy tính.

### b. Với cấu trúc dữ liệu đã được định nghĩa, hãy vận dụng một thuật toán sắp xếp và cài đặt chương trình thực hiện việc sắp xếp các số hạng trong dãy theo thứ tự tăng dần của các bậc. **a. Mô phỏng.** **1. Định nghĩa cấu trúc dữ liệu (Term)**

**Thuộc tính:**

coefficient: Hệ số của số hạng (kiểu double).

degree: Bậc của số hạng (kiểu int, từ 0 đến 100).

**Constructor:**

Khởi tạo hệ số và bậc.

Kiểm tra điều kiện bậc phải nằm trong khoảng 0-100, nếu không sẽ ném ngoại lệ.

**2. In số hạng (printTerm) và đa thức (printPolynomial)**

**printTerm**: In một số hạng theo dạng ax^b.

**printPolynomial**:

Kiểm tra nếu danh sách rỗng, in "Day trong".

In từng số hạng, nối chúng bằng " + " nếu chưa phải số hạng cuối.

**3. Sắp xếp tăng dần theo bậc (sortByDegree)**

**Chia mảng (partition)**:

Chọn phần tử cuối (high) làm chốt (pivot).

Di chuyển các phần tử nhỏ hơn chốt sang trái.

Đổi chỗ chốt vào vị trí đúng.

**Sắp xếp (quickSort)**:

Nếu low < high, chia mảng thành hai phần nhỏ hơn và gọi đệ quy.

**sortByDegree**: Gọi quickSort trên toàn bộ danh sách.

**4. Chương trình chính (main)**

Nhập số lượng số hạng (n).

Khởi tạo danh sách và đặt trước dung lượng (reserve).

Vòng lặp nhập từng số hạng:

Nhập hệ số và bậc.

Nếu bậc không hợp lệ, yêu cầu nhập lại.

In đa thức trước khi sắp xếp.

Sắp xếp theo bậc và in kết quả.

**b. Mã chương trình.**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;  // a. Định nghĩa cấu trúc dữ liệu cho số hạng  struct Term {      double coefficient; // Hệ số - số thực      int degree;         // Bậc - số nguyên từ 0 đến 100      // Constructor để khởi tạo      Term(double coef = 0.0, int deg = 0) : coefficient(coef), degree(deg) {          if (deg < 0 || deg > 100) {              throw invalid\_argument("Bac phai nam trong khoang 0 den 100");          }      }  };  // Hàm in một số hạng  void printTerm(const Term& t) {      cout << t.coefficient << "x^" << t.degree;  }  // Hàm in toàn bộ dãy  void printPolynomial(const vector<Term>& poly) {      if (poly.empty()) {          cout << "Day trong" << endl;          return;      }      for (size\_t i = 0; i < poly.size(); i++) {          printTerm(poly[i]);          if (i < poly.size() - 1) cout << " + ";      }      cout << endl;  }  // b. Thuật toán sắp xếp tăng dần theo bậc (sử dụng Quick Sort)  int partition(vector<Term>& arr, int low, int high) {      int pivot = arr[high].degree;      int i = low - 1;      for (int j = low; j < high; j++) {          if (arr[j].degree <= pivot) {              i++;              swap(arr[i], arr[j]);          }      }      swap(arr[i + 1], arr[high]);      return i + 1;  }  void quickSort(vector<Term>& arr, int low, int high) {      if (low < high) {          int pi = partition(arr, low, high);          quickSort(arr, low, pi - 1);          quickSort(arr, pi + 1, high);      }  }  // Hàm sắp xếp dãy theo bậc  void sortByDegree(vector<Term>& poly) {      quickSort(poly, 0, poly.size() - 1);  }  int main() {      int n;      cout << "Nhap so luong so hang n: ";      cin >> n;      vector<Term> polynomial;      polynomial.reserve(n); // Đặt trước dung lượng để tối ưu      cout << "Nhap " << n << " so hang (he so va bac):\n";      for (int i = 0; i < n; i++) {          double coef;          int deg;          cout << "So hang " << i + 1 << ":\n";          cout << "He so: ";          cin >> coef;          cout << "Bac: ";          cin >> deg;          try {              polynomial.emplace\_back(coef, deg);          } catch (const invalid\_argument& e) {              cout << e.what() << endl;              i--; // Nhập lại số hạng này          }      }      cout << "\nDay truoc khi sap xep:\n";      printPolynomial(polynomial);      // Sắp xếp theo bậc      sortByDegree(polynomial);      cout << "\nDay sau khi sap xep theo bac tang dan:\n";      printPolynomial(polynomial);      return 0;  } |

|  |
| --- |
| Nhap so luong so hang n: 5  Nhap 5 so hang (he so va bac):  So hang 1:  He so: 2.5  Bac: 3  So hang 2:  He so: -1.0  Bac: 0  So hang 3:  He so: 3.7  Bac: 5  So hang 4:  He so: 0.8  Bac: 2  So hang 5:  He so: -4.2  Bac: 1 |

**c. Độ phức tạp thuật toán.**  
• **Thời gian**: O(n log n) trung bình, O(n²) xấu nhất.  
• **Không gian**: O(log n) trung bình, O(n) xấu nhất (do đệ quy).  
• **Ghi chú**: Quick Sort hoạt động hiệu quả với n lớn nhưng có thể chậm nếu chọn pivot không tối ưu. Trong trường hợp này, pivot được chọn là phần tử cuối, có thể cải thiện bằng cách chọn median-of-three.

**Kết luận**  
Thuật toán Quick Sort giúp sắp xếp các số hạng theo bậc một cách hiệu quả với n lớn. Tuy nhiên, có thể gặp trường hợp xấu nhất khi dữ liệu đã sắp xếp hoặc gần sắp xếp, dẫn đến độ phức tạp O(n²). Trong trường hợp này, Merge Sort có thể là một lựa chọn thay thế tốt hơn với độ ổn định cao hơn nhưng tốn thêm bộ nhớ O(n).