CS F211

Data Structures and Algorithms

Sorting

Problem 1

You are given an array of distinct integers. Use merge sort to count the number of *inversions* in the array. An inversion is a pair of indices i and j such that $i < j$ and $a_i > a_j$.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int merge(vector<int>& arr, vector<int>& temp, int left, int mid, int right) {
    /* ??? */
}

int merge_sort(vector<int>& arr, vector<int>& temp, int left, int right) {
    /* ??? */
}

int count_inversions(vector<int>& arr) {
    vector<int> temp(arr.size());
    return merge_sort(arr, temp, 0, arr.size() - 1);
}

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    cout << count_inversions(a) << endl;
    return 0;
}
```

```
>> 8 7
>> 3 4 7 2 -3 1 4 2
4
```

Next task: Write additional code to generate all $n!$ permutations of the given array. For each permutation, calculate the number of inversions, and then output the average number of inversions. Is there a relation between

the average number of inversions and n? Remember the constraint that all integers in the array are distinct.

Problem 2

Merge sort works well on larger lists, but on smaller ones, simpler procedures like insertion sort empirically work better. Write code for a modified merge sort that first divides the list into k lists of roughly equal size, sorts each smaller list using insertion sort, and then merges the lists into a single sorted list.

```cpp
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

void insertion_sort(vector<int>& arr) {
    /* ??? */
}

vector<int> merge(const vector< vector<int> >& lists) {
    /* ??? */
}

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> v(n);
    for (int i = 0; i < n; i++) cin >> v[i];
    vector< vector<int> > lists(k);
    for (int i = 0; i < n; i++) lists[i % k].push_back(v[i]);
    for (int i = 0; i < k; i++) insertion_sort(lists[i]);
    vector<int> res = merge(lists);
    for (int i = 0; i < n; i++) cout << res[i] << " \n"[i ==
        n - 1];
    return 0;
}
```

```
>> 10 3
>> 8 4 6 1 9 2 7 3 0 5
0 1 2 3 4 5 6 7 8 9
```

Next task: Improve the efficiency of the merge function using heaps.

Problem 3

You are given a list of numeric intervals, each defined by a start and end value (inclusive). You have to merge the intervals, by first sorting them using quick sort and then merging.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Interval {
    int start, end;
};

bool compare(const Interval& a, const Interval& b) {
    /* ??? */
}

int partition(vector<Interval>& arr, int low, int high) {
    /* ??? */
}

void quick_sort(vector<Interval>& arr, int low, int high) {
    /* ??? */
}

int main() {
    int n;
    cin >> n;
    vector<Interval> v;
    for (int i = 0; i < n; i++) {
        Interval in;
        cin >> in.start >> in.end;
        v.push_back(in);
    }
    quick_sort(v, 0, n - 1);
    /* ??? */
    return 0;
}
```

```
>> 5
>> 9 12
```

```
>> 1 2
>> 7 11
>> 2 6
>> 15 20
3
1 6
7 12
15 20
```

Next task: Integrate the interval merging logic into the recursive quick sort function.

Problem 4

Write a bucket sort function that sorts integers. The function should take an argument k that describes the number of buckets. Buckets are created on an equal-width basis – if the minimum and maximum elements of the array are 5 and 20, and $k = 3$, the bucket ranges should be $[5, 10)$, $[10, 15)$, and $[15, 20]$, irrespective of the actual distribution of elements.

```cpp
#include <iostream>
#include <vector>

using namespace std;

void bucket_sort(vector<int>& arr, int k) {
    if (arr.empty() || k <= 0) return;

    int min_val = arr[0], max_val = arr[0];
    for (int i = 0; i < arr.size(); i++) {
        min_val = min(min_val, arr[i]);
        max_val = max(max_val, arr[i]);
    }
    int range = max_val - min_val + 1;

    vector< vector<int> > buckets(k);
    for (int num : arr) {
        int index = ((num - min_val) * k) / range;
        if (index == k) index--;
        buckets[index].push_back(num);
    }

    arr.clear();
    for (int i = 0; i < k; ++i) {
        for (int j = 1; j < buckets[i].size(); ++j) {
            int key = buckets[i][j];
            int l = j - 1;
            while (l >= 0 && buckets[i][l] > key) {
                buckets[i][l + 1] = buckets[i][l];
                l--;
            }
            buckets[i][l + 1] = key;
        }
        for (int num : buckets[i]) {
            arr.push_back(num);
```

```
        }
    }
}

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    bucket_sort(a, k);
    for (int i = 0; i < n; i++) cout << a[i] << " \n"[i == n
        - 1];
    return 0;
}
```

```
>> 8 4
>> 29 25 3 49 9 37 21 43
3 9 21 25 29 37 43 49
```

Next task: Modify the function to create buckets on an equal-frequency basis. To determine bucket thresholds, write a divide and conquer procedure similar to quick sort that can find the kth smallest element in an array in $O(n)$.

Problem 5

You are given a list of n hexadecimal numbers, each as a string. Implement radix sort to sort the numbers.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int hex_to_dec(char c) {
    /* ??? */
}

void sort_by_digit(vector<string>& v, int digit) {
    /* ??? */
}

void radix_sort(vector<string>& v) {
    /* ??? */
}

int main() {
    int n;
    cin >> n;
    vector<string> v(n);
    for (int i = 0; i < n; i++) cin >> v[i];
    radix_sort(v);
    for (int i = 0; i < n; i++) cout << v[i] << " \n"[i == n
        - 1];
}
```

```
>> 8
>> 0 4C ABC F 9A 6A3 D5A8 B43F
0 F 4C 9A 6A3 ABC B43F D5A8
```

Next task: Now that you have obtained the numbers in a sorted list, use the two pointers technique to find the maximum number of disjoint pairs of numbers that can be formed such that the two numbers in each pair have an XOR value of `FFF...FFF` where the number of `F`'s is the number of digits in the largest number.