```cpp
#ifndef ALGORITMI_BUBBLESORT_HPP
#define ALGORITMI_BUBBLESORT_HPP


/*
 * Complessità O(n^2)
 * - Best case O(n) se l'array è già ordinato
 * - Average case O(n^2)
 * - Worst case O(n^2) se l'array è ordinato in senso contrario
 */


void bubble_sort(int arr[], int dim) {

    // Ottimizzazione: se in un ciclo non esegue scambi termina
    bool ordinato = false;

    // Compie dim-1 cicli (l'ultimo passaggio non è necessario perchè si ordinano elementi in coppia)
    for (int i = 0; i < dim - 1 && !ordinato; ++i) {

        ordinato = true;

        for (int j = dim - 1; j >= i + 1; --j)

            // Scambia due elementi consecutivi se non ordinati correttamente
            if (arr[j] < arr[j - 1]) {
                int tmp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = tmp;
                ordinato = false;
            }
    }
}


#endif // ALGORITMI_BUBBLESORT_HPP
```

```cpp
#ifndef ALGORITMI_SELECTIONSORT_HPP
#define ALGORITMI_SELECTIONSORT_HPP


/*
 * Complessità in ogni caso di O(n^2)
 */


void selection_sort(int arr[], int dim) {

    for (int i = 0; i < dim; ++i) {

        // Seleziona come minimo l'indice di partenza
        int min = i;

        // Cerca il valore minimo sulla porzione di array restante
        for (int j = i+1; j < dim; ++j) {
            if (arr[j] < arr[i]) min = j;
        }

        // Scambia gli elementi per far apparire l'elemento minore all'inizio
        int tmp = arr[i];
        arr[i] = arr[min];
        arr[min] = tmp;
    }
}

#endif // ALGORITMI_SELECTIONSORT_HPP
```

```cpp
#ifndef ALGORITMI_INSERTIONSORT_HPP
#define ALGORITMI_INSERTIONSORT_HPP


/* Riassumendo: l’algoritmo scorre un array dall’inizio alla fine, assicurandosi che ogni elemento
 *  sia ordinato rispetto a tutti quelli alla sua sinistra, nel caso questo non avvenisse allora
 *  crea un buco in cui inserire l’elemento corrente spostando a destra tutti gli elementi necessari.
 *
 * Complessità di O(n^2)
 * - Best case O(n) se l'array è già ordinato
 * - Average e Worst case O(n^2) se l'array è ordinato in senso contrario
 */


void insertion_sort(int arr[], int dim) {
    int tmp;
    int j;

    // Ignora il primo elemento (già ordinato per definizione) e scorre dal secondo all'ultimo
    for (int i = 1; i < dim; ++i) {

        // Prende e mette da parte l'elemento in esame
        tmp = arr[i];

        // Scorre il vettore al contrario partendo dall'elemento j precedente a quello in esame
        for (j = i-1; j >= 0 && tmp < arr[j]; --j) {
            // Se l'emento in esame è minore di j allora sposto l'elemento j a destra di un posto
            arr[j+1] = arr[j];
        }

        arr[j+1] = tmp;
        /* Perchè rimetto il valore copiato in j+1 ?
         *  Se non ho effettuato scambi allora lo rimetto in i+1-1 ossia in i, ossia dov'era prima
         *  e non cambia nulla, Se ho effettuato scambi allora j è un valore <= di i, dunque il
         *  posto giusto per mettere il valore i è esattamente un posto sopra a j, ossia j+1
         */
    }
}


#endif // ALGORITMI_INSERTIONSORT_HPP
```

```cpp
#ifndef ALGORITMI_QUICKSORT_HPP
#define ALGORITMI_QUICKSORT_HPP


/* Riassumendo: ad ogni chiamata l'algoritmo divide l'array in due parti
 *  (non per forza a metà), una sarà tutta < del perno e una tutta > del perno,
 *  ripete il procedimento sulle parti create finchè hanno almeno due elementi.
 *
 * Complessità di O(n^2)
 * - Best e Average case O(nlog(n)) se la scelta del perno divide sempre l'array in modo equo
 * - Worst case O(n^2) se la scelta del perno divide l'array in un elemento e tutto il resto
 */


void quick_sort(int arr[], int inf, int sup) {

    // La scelta del perno in questa implementazione è l'elemento centrale dell'array
    int perno = arr[(inf + sup)/2];

    // Dichiara due slider temporanei
    int S = inf, D = sup;

    // Finchè i due slider non si toccano
    while (S < D) {

        // Fa avanzare a destra lo slider S finchè non trova un elemento maggiore del perno
        while (arr[S] < perno) ++S;
        // Fa avanzare a sinistra lo slider D finchè non trova un elemento minore del perno
        while (arr[D] > perno) --D;
        // Se i due slider si sovrappongono mi posso fermare
        if (S > D) break;

        // Scambia i due elementi non al lato giusto del perno
        int tmp = arr[S];
        arr[S] = arr[D];
        arr[D] = tmp;

        // Fa avanzare gli slider
        ++S;
        --D;
    }

    // Chiama ricorsivamente sulle parti in cui è stato diviso l'array
    if (D > inf) quick_sort(arr, inf, D);
    if (S < sup) quick_sort(arr, S, sup);
}


#endif // ALGORITMI_QUICKSORT_HPP
```

```cpp
#ifndef ALGORITMI_MERGESORT_HPP
#define ALGORITMI_MERGESORT_HPP


/* Riassumendo: L'algoritmo divide un array in sottoarray sempre grandi la metà del precedente,
 *  fino ad ottenere svariati sottoarray di dimensione uno, che sono per definizione ordinati,
 *  allora combina in modo ordinato i vari sottoarray partendo dai più piccoli.
 *
 * Complessità in ogni caso di O(nlog(n)) con l'utilizzo di memoria ausiliaria (NOT IN PLACE)
 */

#include <vector>
#include "quickSort.hpp"


void merge(int* arr, int inf, int mid, int sup) {

    // Dichiara un vettore di appoggio sul quale ordinare gli elementi
    std::vector<int> buffer;

    // Dichiara due slider temporanei
    int S = inf, D = mid;

    // Ordina gli elementi prendendo dai sottoarray come se fossero delle pile
    while(S < mid && D <= sup) {
        if (arr[S] < arr[D]) buffer.push_back(arr[S++]);
        else buffer.push_back(arr[D++]);
    }

    // Gestione degli ultimi elementi
    while (S < mid) buffer.push_back(arr[S++]);
    while (D <= sup) buffer.push_back(arr[D++]);

    // Ricopia degli elementi ordinati dal vettore di appoggio
    for (int i = inf; i <= sup; i++) arr[i] = buffer[i-inf];
}

void merge_sort(int* arr, int inf, int sup) {

    // Finchè ci sono sottoarray da ordinare (ossia con almeno due elementi)
    if (inf < sup) {

        // Dividili in due
        int mid = (inf + sup)/2;

        // Dividi ancora le due metà
        merge_sort(arr, inf, mid);
        merge_sort(arr, mid+1, sup);

        // Riunisci ordinatamente i vari pezzettini
        merge(arr, inf, mid+1, sup);
    }
}


/*
 * Versione alternativa del merge sort che applica lo stesso principio ma ordinando
 * con un altro algoritmo i sottoarray quando sono abbastanza piccoli
```

```
 */

void hybrid_sort(int* arr, int inf, int sup, const int target) {

    // Finchè ci sono sottoarray da ordinare abbastanza grandi
    if (sup-inf > target) {

        // Dividili in due
        int mid = (inf + sup)/2;

        // Dividi ancora le due metà
        hybrid_sort(arr, inf, mid, target);
        hybrid_sort(arr, mid+1, sup, target);

        // Riunisci ordinatamente i vari pezzettini
        merge(arr, inf, mid+1, sup);
    }
        // Altrimenti, se i sottoarray sono abbastanza piccoli
    else {
        // Ordinali con un altro algoritmo
        quick_sort(arr, inf, sup);
    }
}


/*
 * Versione alternativa del merge sort applicata alle liste
 * Non utilizza memoria aggiuntiva!
 */

struct elem {
    int info;
    elem* next;
};

void split_list(elem* &p1, elem* &p2) {

    // Se ho una lista vuota o di un solo elemento non posso dividerla
    if (p1 == nullptr || p1->next == nullptr) return;

    // p2 è la lista degli elementi dispari della lista, inizialmente vuota
    // p1 è la lista degli elementi pari della lista, inizialmente è l'intera lista

    // Inserisco in coda alla lista degli elementi pari il nuovo elemento
    elem* tmp = p1->next; // Salvo il primo elemento in un puntatore ausiliario
    p1->next = tmp->next; // Collego l'elemento zero all'elemento due, in modo da ottenere una lista di elementi pari s1

    // Insericso in testa alla lista degli elementi dispari il nuovo elemento
    tmp->next = p2; // Prendo l'elemento salvato (primo) e lo metto in testa alla lista degli elementi dispari s2
    p2 = tmp; // Aggiorno la testa della lista dispari

    // Ripeto il procedimento partendo dal secondo elemento (elementi 0 e 1 sono andati dove volevo)
    split_list(p1->next, p2);
}

void merge_list(elem* &p1, elem* p2) {

    // Se p2 è vuota non devo unire nulla e ho finito
    if (p2 == nullptr) return;
```

```
118
119     // Se p1 è vuota (e p2 no) prendo gli elementi di p2 e li sposto su p1
120     if (p1 == nullptr) {
121         p1 = p2; //
122         return;
123     }
124
125     if (p1->info <= p2->info) {
126         // Scelgo p1 e lascio stare p2
127         merge_list(p1->next, p2);
128     }
129     else {
130         // Scelgo p2 e lascio stare p1
131         merge_list(p2->next, p1);
132         // A p1 collego l'elemeto appena scelto (p2)
133         p1 = p2;
134     }
135 }
136
137 void merge_sort_list(elem* &p1) {
138
139     // Se ho una lista vuota o di un solo elemento è già ordinata
140     if (p1 == nullptr || p1->next == nullptr) return;
141
142     // Elemento ausiliario per dividere la lista
143     elem* p2 = nullptr;
144
145     split_list(p1, p2);
146     merge_sort_list(p1);
147     merge_sort_list(p2);
148     merge_list(p1, p2);
149 }
150
151
152 #endif // ALGORITMI_MERGESORT_HPP
```

```cpp
#ifndef ALGORITMI_HEAPSORT_HPP
#define ALGORITMI_HEAPSORT_HPP


/*
 * Complessità in ogni caso di O(nlog(n)) senza l'utilizzo di memoria ausiliaria (IN PLACE)
 */


inline void exchange(int *arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

// Anche detta heapify
void down(int* arr, int i, int last) {

    int son = 2*i + 1;

    // Se i ha un solo figlio
    if (son == last) {

        // Allora lo scambia col padre se è maggiore
        if (arr[son] > arr[i]) {
            exchange(arr, i, son);
        }
    }

    // Altrimenti, se i ha due figli
    else if (son < last) {

        // Prende il figlio maggiore
        if (arr[son] < arr[son+1]) son++;

        // Lo scambia col padre se è maggiore di esso
        if (arr[son] > arr[i]) {
            exchange(arr, i, son);

            // Controlla se è necessario farlo scendere ancora
            down(arr, son, last);
        }
    }
}


void extract(int* arr, int &last) {
    /*
     * Scambia la radice con l'ultimo elemento dell'array e ne decrementa la dimensione
     * In questo modo l'ultima parte dell'array per lo heap sarà ordinata
     * Ricrea le proprietà dello heap applicando la down alla nuova radice
     */
    exchange(arr, 0, last--);
    down(arr, 0, last);
}

void buildHeap(int* arr, int last) {
    /*
     * Si basa sul concetto che la seconda metà di un vettore che rappresenta uno heap è
     * costituita sempre e solo da foglie, che quindi sono già degli heap per definizione.
     *
```

```
     * In questo modo prendendo gli elementi della prima metà e
     * facendoli scendere, se necessario, costruisco uno heap.
     */
    for (int i = last/2; i >= 0; --i) {
        down(arr, i, last);
    }
}

void heap_sort(int arr[], int dim) {
    buildHeap(arr, dim-1);
    int i = dim-1;
    while (i > 0) extract(arr, i);
}

#endif //ALGORITMI_HEAPSORT_HPP
```

```cpp
#ifndef ALGORITMI_COUNTINGSORT_HPP
#define ALGORITMI_COUNTINGSORT_HPP

/*
 * Complessità di O(n + k) utilizzando memoria ausiliaria (NOT IN PLACE)
 *  Dove k è la lunghezza del vettore di supporto, questo algoritmo presuppone di sapere
 *  il minimo ed il massimo teorici (non per forza assunti) dei NATURALI da ordinare,
 *  conviene solamente quando k è O(n), nel senso che k è <~ n
 */

void counting_sort(int *arr, int dim, int max) {

    // Crea un array ausiliario grande quanto la differenza tra minimo e massimo
    // teorici, per ospitare il conteggio dei numeri, inizialmente tutti a zero
    int gap = max + 1, aux[gap];
    for (int i = 0; i < gap; ++i) aux[i] = 0;

    // Per ogni occorrenza di un numero ne aumento il conteggio
    for (int i = 0; i < dim; ++i) aux[arr[i]]++;

    // Riscrive l'array secondo il conteggio
    int j = 0;
    for (int i = 0; i < gap; ++i) {
        while (aux[i]) {
            arr[j] = i;
            --aux[i];
            ++j;
        }
    }
}


#endif //ALGORITMI_COUNTINGSORT_HPP
```

```cpp
#ifndef ALGORITMI_RADIXSORT_HPP
#define ALGORITMI_RADIXSORT_HPP


/*
 * Complessità di O(d(n + k)) [ossia O(dn) poichè solitamente k è trascurabile] utilizzando memoria ausiliaria (NOT IN PLACE)
 *  Dove d è il numero massimo di cifre e k è il numero di possibili valori di ogni cifra (base numerica)
 *  Conviene solamente quando d << n
 *
 *  Un esempio concreto di utilizzo del radix sort è l'ordinamento di sequenze di
 *  caratteri (d = 1 e k = 26 se considero solo le lettere) dunque O(1(n+26)).
 */

#include <cmath>

struct Elem {
    int n;
    Elem* next;
};

void radix_sort(int *arr, int dim, int cifre, int base = 10) {

    // Tante passate quante le cifre, partendo dalla meno significativa
    for (int i = 0; i < cifre; ++i) {

        // Crea un array con una lista per ogni cifra possibile
        Elem* bucket[base];
        for (int j = 0; j < base; ++j) bucket[j] = nullptr;

        // Popola le liste con le cifre corrispondenti
        for (int j = 0; j < dim; ++j) {

            // Dato un intero arr[j] trova la sua i esima cifra
            int cifra = arr[j] / pow(10, i);
            cifra = cifra % 10;

            Elem* num = new Elem;
            num->n = arr[j];
            num->next = nullptr;

            // Lista vuota
            if (bucket[cifra] == nullptr) {
                bucket[cifra] = num;
            }
            // Inserimento in coda
            else {
                Elem* aux = bucket[cifra];
                for (Elem* k = bucket[cifra]; k; k = k->next ) aux = k;
                aux->next = num;
            }
        }

        // Rimetto nell'array i numeri di ogni secchio
        int k = 0;
        for (int j = 0; j < base; ++j) {

            for (Elem* e = bucket[j]; e; e = e->next ) {
                arr[k] = e->n;
                ++k;
```

```
            }

            // Elimino le liste
            while (bucket[j]) {
                Elem* e = bucket[j];
                bucket[j] = bucket[j]->next;
                delete e;
            }
        }
    }
}


#endif //ALGORITMI_RADIXSORT_HPP
```