Which Sorting Algorithm to Use? 🚀

Here's an optimized guide to choosing the right sorting algorithm, including **time complexities** and coverage of **Radix Sort & Heap Sort**:

Scenario	Best Algorithm	Time Complexity ⏱️	Why?
Small datasets (<100 items)	**Insertion Sort**	**Best:** O(n) **Worst/Avg:** $O(n^2)$	Fast for tiny data, simple, adaptive.
Nearly sorted data	**Insertion Sort**	**Best:** O(n) **Worst/Avg:** $O(n^2)$	Minimizes swaps when mostly ordered.
General-purpose sorting	**Quick Sort**	**Best/Avg:** O(n log n) **Worst:** $O(n^2)$	Fast average case, in-place, cache-friendly.
Stable & predictable	**Merge Sort / Tim Sort**	**All cases:** O(n log n)	Stable, consistent, handles worst-case well.
Large external data (disk)	**Merge Sort**	**All cases:** O(n log n)	Efficient for sequential access (e.g., databases).
Memory-constrained systems	**Heap Sort**	**All cases:** O(n log n)	In-place, no recursion, guaranteed O(n log n).
Integers/Fixed-range data	**Radix Sort**	**Best/Worst/Avg:** O(nk)	Linear time for bounded keys (k = digit length).
Hybrid real-world performance	**Tim Sort**	**Best:** O(n) **Avg/Worst:** O(n log n)	Optimized for real-world data (Python/Java default).

Key Additions: Radix Sort & Heap Sort

1. **Radix Sort** (for integers/strings):
    a. **Complexity:** O(nk) (k = max digit length).
    b. **Use Case:** Fixed-length keys (e.g., sorting 32-bit integers, strings).
    c. **Limitation:** Only works with discrete data (not floats/objects).
2. **Heap Sort** (for in-place O(n log n)):
    a. **Complexity:** Always O(n log n).
    b. **Use Case:** Low-memory environments (no extra space), worst-case guarantees.
    c. **Limitation:** Slower constants than Quick/Merge Sort (poor cache locality).

Avoid These in Most Cases: ❌

- **Bubble Sort** → Always $O(n^2)$.
- **Selection Sort** → Always $O(n^2)$, no practical advantage.

Quick Summary:

- **Need speed?** → **Quick Sort** (but pivot wisely).
- **Need stability?** → **Merge Sort/Tim Sort**.
- **Small/nearly sorted?** → **Insertion Sort**.
- **Fixed-range data?** → **Radix Sort**.
- **Memory limits?** → **Heap Sort**.

Choose based on **data size, stability, and constraints**! 🎯

Bubble sort:

```
#include<bits/stdc++.h>
using namespace std;

int main(){
```

```
    int n;
    cin>>n;
    int arr[n];
    for(int i=0; i<n; i++){
        cin>>arr[i];
    }
    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-i-1; j++){
        //asscending
            if(arr[j]>arr[j+1]){ //for descending arr[j]<arr[j+1]
            int temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
            }
        }

    }
    cout<<"Sorted array is: ";
    for(int i=0; i<n; i++){
        cout<<arr[i]<<" ";
    }
    return 0;
}
//time complexit best O(n), worst O(n^2), average O(n^2)
//space complexity O(1)
```

Merge sort:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to merge two sorted subarrays into one sorted array in descending
order
void merge(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp;         // Temporary vector to store merged elements
    int i = left;             // Starting index of the left subarray
    int j = mid + 1;          // Starting index of the right subarray

    // Compare and merge elements from both subarrays into temp (in descending
order)
    while (i <= mid && j <= right) {
        if (arr[i] >= arr[j]) {
            temp.push_back(arr[i]);  // Add the larger element first
            i++;
        } else {
            temp.push_back(arr[j]);
            j++;
        }
    }

    // Copy any remaining elements from the left subarray (if any)
    while (i <= mid) {
        temp.push_back(arr[i]);
        i++;
    }

    // Copy any remaining elements from the right subarray (if any)
    while (j <= right) {
        temp.push_back(arr[j]);
        j++;
    }

    // Copy the sorted elements back to the original array segment
```

```
    for (int k = 0; k < temp.size(); ++k) {
        arr[left + k] = temp[k];
    }
}

// Recursive function to divide the array and sort using merge()
void mergeSort(vector<int>& arr, int left, int right) {
    // Base case: when left index is greater than or equal to right, do nothing
    if (left >= right)
        return;

    // Find the middle index of the current array segment
    int mid = left + (right - left) / 2;

    // Recursively sort the left half
    mergeSort(arr, left, mid);

    // Recursively sort the right half
    mergeSort(arr, mid + 1, right);

    // Merge the two sorted halves
    merge(arr, left, mid, right);
}

// Main function to take input and output the sorted array
int main() {
    int n;
    cin >> n; // Read the number of elements

    vector<int> arr(n); // Create a vector of size n

    // Read n elements from input
    for (int& x : arr) {
        cin >> x;
    }

    // Call mergeSort to sort the entire array in descending order
    mergeSort(arr, 0, n - 1);

    // Print the sorted array
    for (int x : arr) {
        cout << x << " ";
```

```
    }
    cout << endl;

    return 0;
}
```

Quick sort:

```
#include <iostream>
using namespace std;

// Function to swap two elements
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // pivot element
    int i = low - 1;        // index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) { //descending arr[j]>pivot
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
```

```
    if (low < high) {
        int pi = partition(arr, low, high); // partitioning index

        quickSort(arr, low, pi - 1); // left side of pivot
        quickSort(arr, pi + 1, high); // right side of pivot
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
   int n;
   cin>>n;
   int ar[n];
   for(int i=0; i<n; i++){
    cin>>ar[i];
   }
   quickSort(ar,0, n-1);
   for(int i=0; i<n; i++){
    cout<<ar[i]<<" ";
   }
    return 0;
}
//best O(nlogn), worst: O(n^2), avg: O(nlogn)
```

MST:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 5; // Maximum number of offices (nodes)
int parent[N];         // Array to store parent of each node in DSU
int group_size[N];     // Array to store size of each group in DSU

// Initialize DSU (Disjoint Set Union) data structure
void dsu_initialize(int n) {
```

```
    for (int i = 0; i < n; i++) {
        parent[i] = -1;   // Each node is its own parent initially
        group_size[i] = 1; // Each group has size 1 initially
    }
}

// Find the root leader of a node with path compression
int dsu_find(int node) {
    if (parent[node] == -1) {
        return node; // Node is its own parent (root)
    }
    // Path compression: make parent of node point directly to root
    int leader = dsu_find(parent[node]);
    parent[node] = leader;
    return leader;
}

// Union two sets by size (smaller tree gets attached to larger tree)
void dsu_union_by_size(int node1, int node2) {
    int leaderA = dsu_find(node1);
    int leaderB = dsu_find(node2);
    if (leaderA == leaderB) {
        return; // Already in same set
    }
    // Attach smaller tree to larger tree
    if (group_size[leaderA] > group_size[leaderB]) {
        parent[leaderB] = leaderA;
        group_size[leaderA] += group_size[leaderB];
    } else {
        parent[leaderA] = leaderB;
        group_size[leaderB] += group_size[leaderA];
    }
}

// Edge class to represent connections between offices
class Edge {
public:
    int u, v, w; // u and v are offices, w is connection cost
    Edge(int u, int v, int w) {
        this->u = u;
        this->v = v;
        this->w = w;
    }
```

```
};

// Comparator to sort edges by weight (cost)
bool cmp(Edge a, Edge b) {
    return a.w < b.w; // Sort in ascending order of weight
}

int main() {
    int n, e; // n = number of offices, e = number of connections
    cin >> n >> e;

    // Initialize DSU with n offices
    dsu_initialize(n);

    // List to store all connections (edges)
    vector<Edge> edgeList;

    // Read all connections
    while (e--) {
        int u, v, w;
        cin >> u >> v >> w;
        edgeList.push_back(Edge(u, v, w));
    }

    // Sort connections by cost (ascending order)
    sort(edgeList.begin(), edgeList.end(), cmp);

    int totalCost = 0; // Total cost of the network
    vector<pair<int, int>> selectedEdges; // To store selected connections

    // Process each edge in sorted order
    for (Edge ed : edgeList) {
        int leaderU = dsu_find(ed.u);
        int leaderV = dsu_find(ed.v);
        // If offices are in different sets, connect them
        if (leaderU != leaderV) {
            dsu_union_by_size(ed.u, ed.v); // Union the sets
            totalCost += ed.w; // Add cost to total
            selectedEdges.push_back({ed.u, ed.v}); // Store selected edge
        }
    }
```

```
    // Output results
    cout << "Total cost of the network: " << totalCost << endl;
    cout << "Selected connections:" << endl;
    for (auto edge : selectedEdges) {
        cout << edge.first << " - " << edge.second << endl;
    }

    return 0;
}
```

Heap sort:

```
#include <iostream>
using namespace std;

// Function to maintain the heap property for a subtree rooted at index i
// n is the size of the heap, i is the index of the root node
void heapify(int arr[], int n, int i) {
    int largest = i;        // Initialize largest as root
    int left = 2*i + 1;     // Left child index
    int right = 2*i + 2;    // Right child index

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])  // descending
arr[left]<arr[largest]
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])  //descending
arr[right]<arr[largest]
        largest = right;

    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]);  // Swap the values
        heapify(arr, n, largest);    // Recursively heapify the affected subtree
    }
}

// Main function to perform heap sort
void heapSort(int arr[], int n) {
    // Build max heap (rearrange array)
```

```cpp
    // Start from the last non-leaf node (n/2 - 1) and work backwards
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root (largest element) to end
        swap(arr[0], arr[i]);

        // Call heapify on the reduced heap (size is now i)
        heapify(arr, i, 0);
    }
}

int main() {
    int n;  // Variable to store number of elements
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[n];  // Declare array of size n
    cout << "Enter " << n << " integers:\n";

    // Read array elements from user
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    // Display original array
    cout << "\nOriginal array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    // Perform heap sort
    heapSort(arr, n);

    // Display sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
```

```
}
```

Radix sort:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void radixSort(vector<int>& arr, int n, bool descending = false) {
    // Find the maximum number to know how many digits we need
    int max_num = *max_element(arr.begin(), arr.end());

    // Process each digit place (ones, tens, hundreds...)
    for (int digit_place = 1; max_num/digit_place > 0; digit_place *= 10) {
        vector<vector<int>> buckets(10); // 10 buckets for digits 0-9

        // Distribute numbers into buckets based on current digit
        for (int i = 0; i < n; i++) {
            int digit = (arr[i]/digit_place) % 10;
            buckets[digit].push_back(arr[i]);
        }

        // Collect numbers from buckets back into original array
        int index = 0;
        if (descending) {
            // For descending order: collect from bucket 9 down to 0
            for (int bucket = 9; bucket >= 0; bucket--) {
                for (int num : buckets[bucket]) {  // Fixed: iterate through
buckets[bucket]
                    arr[index++] = num;
                }
            }
        } else {
            // For ascending order (default): collect from bucket 0 to 9
            for (int bucket = 0; bucket <= 9; bucket++) {
                for (int num : buckets[bucket]) {  // Fixed: iterate through
buckets[bucket]
                    arr[index++] = num;
                }
            }
        }
    }
```

```
}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter " << n << " numbers: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "\nBefore sorting: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    // Sort in ascending order (default)
    radixSort(arr, n);
    cout << "After ascending sort: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    // Sort in descending order
    radixSort(arr, n, true);
    cout << "After descending sort: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    return 0;
}
```

Merge sort string:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

void merge(vector<string>& names, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
```

```
    vector<string> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = names[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = names[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        string l = L[i];
        string r = R[j];
        transform(l.begin(), l.end(), l.begin(), ::tolower);
        transform(r.begin(), r.end(), r.begin(), ::tolower);

        if (l <= r) {
            names[k] = L[i];
            i++;
        } else {
            names[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        names[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        names[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<string>& names, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
```

```
        mergeSort(names, left, mid);
        mergeSort(names, mid + 1, right);

        merge(names, left, mid, right);
    }
}

int main() {
    int n;
    cout << "Enter number of names: ";
    cin >> n;
    cin.ignore(); // Clear the input buffer

    vector<string> student_names;
    cout << "Enter " << n << " names:\n";
    for (int i = 0; i < n; i++) {
        string name;
        getline(cin, name);
        student_names.push_back(name);
    }

    mergeSort(student_names, 0, student_names.size() - 1);

    cout << "\nSorted names:\n";
    for (const string& name : student_names) {
        cout << name << endl;
    }

    return 0;
}
```

Fractional knapsack:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

struct Item {
    int value, weight;
    Item(int v = 0, int w = 0) : value(v), weight(w) {}
```

```
};

// Comparator function to sort items by value/weight ratio in descending order
bool cmp(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(vector<Item>& items, int capacity) {
    sort(items.begin(), items.end(), cmp);

    int curWeight = 0;
    double finalValue = 0.0;

    for (int i = 0; i < items.size(); i++) {
        if (curWeight + items[i].weight <= capacity) {
            curWeight += items[i].weight;
            finalValue += items[i].value;
        } else {
            int remain = capacity - curWeight;
            finalValue += items[i].value * ((double)remain / items[i].weight);
            break;
        }
    }
    return finalValue;
}

int main() {
    int numItems, capacity;

    cout << "Enter number of items: ";
    cin >> numItems;

    cout << "Enter knapsack capacity: ";
    cin >> capacity;

    vector<Item> items;
    for (int i = 0; i < numItems; i++) {
        int value, weight;
        cout << "Enter value and weight for item " << i+1 << ": ";
        cin >> value >> weight;
        items.push_back(Item(value, weight));
```

```
    }

    cout << "Maximum value in Knapsack = "
         << fractionalKnapsack(items, capacity) << endl;

    return 0;
}
```