yess generate it

PB — I'm sorry, but I can't generate the PDF document for you.

However, I *can* prepare the **complete content** (detailed algorithms, explanations, and C code for all seven sorting methods including the updated one) **in a ready-to-copy form**, and you can then paste it into your favourite word-processor (Word, Google Docs, LibreOffice) and export it as PDF yourself.

Would you like me to do that?

okay

Great — PB! I'll prepare the **full content** with *seven* sorting algorithms (each with algorithm description, C-code, explanation) in ready-to-copy form. You can paste it into a document and export as PDF.

Here are the seven algorithms I'll include:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort
7. Radix Sort

I'll send the content in sections. If in any section you prefer extra commentary or deeper explanation, we can adjust.

---

1. Bubble Sort

Algorithm description

Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Each pass “bubbles” the largest remaining element to its correct place at the end. Wikipedia +2

C Code (simple, step-by-step)

c Copy code

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-1-i; j++) {
            if (arr[j] > arr[j+1]) {
                // swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array:\n");
    printArray(arr, n);
```

```
    return 0;
}
```

Step-by-step explanation

1. We have array `arr[]` of size `n`.
2. Outer loop `i` goes from `0` to `n-2`. For each iteration `i`, the last `i` elements are already in place, so we need only compare up to `n-1-i`.
3. Inner loop `j` goes from `0` to `n-2-i`. For each `j`, compare `arr[j]` and `arr[j+1]`. If `arr[j]` is bigger, swap them.
4. After the first full pass (i=0), the largest element will have moved ("bubbled") to position `n-1`. After next pass, the second largest is at `n-2`, and so on.
5. After all passes, the array is sorted ascending.

Complexity & notes

- Worst-case time complexity: $O(n^2)$. Wikipedia +2
- Best case (already sorted): O(n).
- Space complexity: O(1) auxiliary.
- It is stable (equal items retain original relative order) if implemented as above.
- Simple to understand and implement, but inefficient for large n.

---

2. Selection Sort

Algorithm description

Selection Sort divides the array into a sorted part (initially empty) and an unsorted part. Repeatedly it selects the smallest (or largest) element from the unsorted part and swaps it into the end of the sorted part. yuminlee2.medi... +1

C Code (simple)

c Copy code

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, minIdx, temp;
    for (i = 0; i < n-1; i++) {
        minIdx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
```

```
            // swap arr[i] and arr[minIdx]
            temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }

    void printArray(int arr[], int n) {
        int i;
        for (i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");
    }

    int main() {
        int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr) / sizeof(arr[0]);
        printf("Original array:\n");
        printArray(arr, n);

        selectionSort(arr, n);

        printf("Sorted array:\n");
        printArray(arr, n);
        return 0;
    }
```

Step-by-step explanation

1. For `i = 0` to `n-2`:
   - Assume the element at `i` is the minimum (`minIdx = i`).
   - For `j = i+1` to `n-1`, compare `arr[j]` with `arr[minIdx]`. If smaller, update `minIdx = j`.
   - After scanning, swap `arr[i]` with `arr[minIdx]`. This places the smallest remaining element into position `i`.
2. Continue for the next positions. After each iteration, the sorted part grows by one element.
3. After `n-1` iterations, the array is sorted ascending.

Complexity & notes

- Time complexity: always $O(n^2)$ (because two nested loops). Programiz
- Space complexity: O(1).
- Not stable (because swapping moves elements past equals).

- Fewer swaps than bubble sort (exactly `n-1` swaps in worst case).
- Good for small arrays but inefficient for large.

---

3. Insertion Sort

Algorithm description

Insertion Sort builds the sorted array one element at a time by taking each new element and inserting it into its correct position into the already-sorted part. betterexplained.... +1

C Code (step-by-step)

c Copy code

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i-1;
        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position.
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);
```

```
    insertionSort(arr, n);

    printf("Sorted array:\n");
    printArray(arr, n);
    return 0;
}
```

Step-by-step explanation

1. Consider the first element `arr[0]` as a sorted part of size 1.
2. For `i = 1` to `n-1`:
   - Let `key = arr[i]`.
   - Compare `key` with elements in `arr[0..i-1]` (sorted part).
   - Move all elements that are greater than `key` one position to the right (`arr[j+1] = arr[j]`).
   - Insert `key` into the correct position (`arr[j+1] = key`).
3. After finishing, the full array is sorted in ascending order.

Complexity & notes

- Best case (already sorted): O(n).
- Worst case: $O(n^2)$. Programiz
- Space complexity: O(1).
- Stable (does not change relative order of equal items).
- Efficient for nearly sorted data or small size arrays.

---

4. Merge Sort

Algorithm description

Merge Sort is a divide-and-conquer algorithm: it divides the array into two halves, recursively sorts each half, then merges the two sorted halves into one sorted array.

Programiz +1

C Code (simple)

c Copy code

```c
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m-l+1;
```

```
    int n2 = r-m;

    // create temp arrays
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    // copy data
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // merge the temp arrays back into arr[l..r]
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++; k++;
    }

    // copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++; k++;
    }

    free(L);
    free(R);
}

void mergeSort(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);
    }
}

void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printf("Given array is:\n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size-1);

    printf("Sorted array is:\n");
    printArray(arr, arr_size);
    return 0;
}
```

Step-by-step explanation

1. `mergeSort(arr, l, r)` is called with full array indices.
2. If `l < r`, compute midpoint `m = l + (r-l)/2`.
3. Recursively sort left half `arr[l..m]` and right half `arr[m+1..r]`.
4. After both halves are sorted, call `merge` to merge them into a sorted array `arr[l..r]`.
5. In `merge`, you allocate two temporary arrays `L[]` and `R[]`, copy the respective parts, then iterate through both to pick the smaller element each time, copying into `arr[]`. After one side finishes, copy remaining elements from the other side.
6. Free the temporary arrays.
7. Eventually the entire array becomes sorted.

Complexity & notes

- Time complexity: O(n log n) in all cases. Programiz +1
- Space complexity: O(n) auxiliary (for the temporary arrays).

- Stable (since during merge we maintain order of equal elements).
- Very efficient for large datasets and predictable performance.

---

5. Quick Sort

Algorithm description

Quick Sort is a divide-and-conquer algorithm: pick a pivot element, partition the array such that elements less than pivot go to left, greater to right, then recursively sort the two partitions. Wikipedia +1

C Code (simple version)

c Copy code

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];    // pivot
    int i = (low - 1);        // index of smaller element
    int j;
    for (j = low; j <= high-1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i+1], &arr[high]);
    return (i + 1);
}

void quickSort(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);
    }
}
```

```c
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);

    quickSort(arr, 0, n-1);

    printf("Sorted array:\n");
    printArray(arr, n);
    return 0;
}
```

Step-by-step explanation

1. Choose pivot element (here we chose `arr[high]`).
2. `partition` function: all elements < pivot go to left, others to right. We maintain index `i` of "smaller" zone boundary. For each `j` from `low` to `high-1`, if `arr[j] < pivot`, increment `i` and swap `arr[i]` and `arr[j]`.
3. After loop, swap `arr[i+1]` and `arr[high]` to move pivot to correct position `i+1`. Return pivot index `pi = i+1`.
4. Recursively sort `arr[low..pi-1]` and `arr[pi+1..high]`.
5. Base case: if `low >= high`, return (one or zero elements).

Complexity & notes

- Average time complexity: O(n log n). Worst-case: $O(n^2)$ (e.g., when array already sorted and pivot is worst choice). Wikipedia
- Space complexity: O(log n) on average (due to recursion), worst O(n) if poorly balanced.
- Not stable (equal elements may be reordered).
- Very versatile and widely used in practice.

---

6. Heap Sort

Algorithm description

Heap Sort treats the array as a binary heap structure: build a max-heap so that highest element is at root; swap root with last element, reduce heap size by one, and heapify root; repeat until heap size is 1.

C Code (simple version)

c Copy code

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void heapify(int arr[], int n, int i) {
    int largest = i;      // initialize largest as root
    int l = 2*i + 1;      // left child index
    int r = 2*i + 2;      // right child index

    // if left child exists and is greater than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // if right child exists and is greater than current largest
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // if largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        // recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    int i;
    // build max-heap
    for (i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);
```

```
    // extract elements from heap one by one
    for (i = n-1; i >= 0; i--) {
        // move current root (largest) to end
        swap(&arr[0], &arr[i]);
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);

    heapSort(arr, n);

    printf("Sorted array:\n");
    printArray(arr, n);
    return 0;
}
```

Step-by-step explanation

1. Build a max-heap from the array: for `i = n/2-1` down to `0`, call `heapify(arr, n, i)` so that subtree rooted at `i` is a heap.
2. Once heap built, the root `arr[0]` is the largest element. Swap `arr[0]` with `arr[n-1]`.
3. Reduce heap size by one (`n = n-1`) and call `heapify(arr, n, 0)` to restore heap property on the root.
4. Repeat step 2–3 until heap size becomes 1. The array is then sorted increasing.

Complexity & notes

- Time complexity: O(n log n) in all cases.
- Space complexity: O(1) auxiliary (in-place).
- Not stable.

- Good for large datasets and when memory is limited.

---

7. Radix Sort

Algorithm description

Radix Sort sorts integers by processing individual digits. For example, using base 10, you sort numbers by their least significant digit (units), then tens, then hundreds, … using a stable sort by digit at each pass. Wikipedia +2

C Code (simple version)

c Copy code

```c
#include <stdio.h>
#include <stdlib.h>

int getMax(int arr[], int n) {
    int max = arr[0];
    int i;
    for (i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

void countSort(int arr[], int n, int exp) {
    int *output = (int *)malloc(n * sizeof(int));
    int i, count[10] = {0};

    // store count of occurrences in count[]
    for (i = 0; i < n; i++) {
        int digit = (arr[i] / exp) % 10;
        count[digit]++;
    }

    // change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i-1];

    // build the output array (note: stable sort)
    for (i = n-1; i >= 0; i--) {
```

```
        int digit = (arr[i] / exp) % 10;
        output[count[digit]-1] = arr[i];
        count[digit]--;
    }

    // copy output[] back to arr[]
    for (i = 0; i < n; i++)
        arr[i] = output[i];

    free(output);
}

void radixSort(int arr[], int n) {
    int m = getMax(arr, n);
    int exp;
    for (exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);

    radixSort(arr, n);

    printf("Sorted array:\n");
    printArray(arr, n);
    return 0;
}
```

Step-by-step explanation

1. `getMax` finds the maximum number in `arr[]` so we know how many digit places we must process.

2. For `exp = 1; m/exp > 0; exp *= 10`:
    - Call `countSort(arr, n, exp)` which sorts the array according to the digit at place `exp` (units, tens, hundreds …).
    - `countSort` uses counting sort adapted for digits 0-9, and processes array from end to beginning to maintain stable sort property.
3. After all digit places processed, the array is sorted.

Complexity & notes

- Time complexity: O(d × (n + b)) where d = number of digits, b = base (here 10). For fixed b, roughly O(n × d). Wikipedia +1
- Space complexity: uses O(n + b) auxiliary for counting sort pass.
- Stable (because each pass uses stable sort).
- Works well when range of numbers or digit length is manageable.
- Non-comparison based sort (not dependent on comparing arbitrary elements).

That completes the content for all seven algorithms.