**SORTING ALGORITHMS REPORT**

**1. Bubble sort:**

* **Idea**: Bubble sort runs 2 For loops. The inner loop does comparisons between adjacent values and exchanges them if they are not in sorted order. Every time the inner loop has done its execution, the minimum value of the remaining unsorted array is always in the right position.
* **Algorithm**:

void BubbleSort(int \*a, int size)

{

    for (int i = 0; i < size; ++i)

    {

        for (int j = size; j > i; --j)

        {

            if (a[j] < a[j - 1])

                swap(a[j], a[j - 1]);

        }

    }

}

* **Time complexity**:
* Best case: Ω(n)
* Worst case: Ο(n2)
* Average case: Θ(n2)
* **Review**: Bubble sort is one of the simplest sorting algorithms to understand. Since bubble sort is very inefficient, it is best suited for academic purposes and should not be used on lists with a great number of elements. Bubble sort is stable.

**2. Shaker sort:**

* **Idea**: Shaker sort is an improved version of bubble sort. By remembering the last exchange position, shaker sort operates in both directions instead of one direction like bubble sort. Additionally, if there is no exchange in the last pass, shaker sort will terminate immediately.
* **Algorithm**:

void ShakerSort(int \*a, int size)

{

    bool swapped = true;

    int start = 0, end = size - 1, i;

    while (swapped)

    {

        swapped = false;

        for (i = start; i < end; ++i)

        {

            if (a[i] > a[i + 1])

            {

                swap(a[i], a[i + 1]);

                swapped = true;

            }

        }

        if (!swapped)

            break;

        --end;

        for (i = end - 1; i >= start; --i)

        {

            if (a[i] > a[i + 1])

            {

                swap(a[i], a[i + 1]);

                swapped = true;

            }

        }

        ++start;

    }

}

* **Time complexity**:
* Best case: Ω(n)
* Worst case: Ο(n2)
* Average case: Θ(n2)
* **Review**: Despite having some improvements compared to bubble sort, shaker sort only provides marginal performance. Like bubble sort, shaker sort is primarily used for academic purposes. Shaker sort is also stable.

**3. Selection sort:**

* **Idea**: Selection sort always looks for the minimum values in the unsorted list and places it in the right position until the list is completely sorted.
* **Algorithm**:

void SelectionSort(int \*a, int size)

{

    int min, minindex;

    for (int i = 0; i < size; ++i)

    {

        min = a[i];

        minindex = i;

        for (int j = i + 1; j < size; ++j)

        {

            if (a[j] <= min)

            {

                min = a[j];

                minindex = j;

            }

        }

        a[minindex] = a[i];

        a[i] = min;

    }

}

* **Time complexity**:
* Best case: Ω(n2)
* Worst case: Ο(n2)
* Average case: Θ(n2)
* **Review**: Selection sort is inefficient since it has quadratic time complexity, but it makes the minimum exchange operations. Like bubble sort, selection sort is only suitable for sorting lists with a small number of elements.

**4. Insertion sort:**

* **Idea**: Insertion sort divides the lists into two parts: the left part is the sorted list and the right part is the unsorted list. In each pass, the left most element in the unsorted list is inserted into the right position in the sorted list.
* **Algorithm**:

void InsertionSort(int \*a, int size)

{

    int selected, index;

    for (int i = 1; i < size; ++i)

    {

        selected = a[i], index = i - 1;

        while (a[index] >= selected && index >= 0)

        {

            a[index + 1] = a[index];

            --index;

        }

        a[index + 1] = selected;

    }

}

* **Time complexity**:
* Best case: Ω(n)
* Worst case: Ο(n2)
* Average case: Θ(n2)
* **Review**: Compared to other quadratic time complexity sorting algorithms such as bubble sort and selection sort, insertion sort is much more efficient in practice. Insertion sort is suitable for sorting small lists and sometimes even faster than quicksort. But since insertion sort average time complexity is quadratic, it is impractical for sorting lists with a great number of elements.

**5. Binary insertion sort:**

* **Idea**: Binary insertion sort is an improved version of insertion sort. By using binary search instead of sequential search, binary insertion sort will determine the right position to insert the new elements to the sorted list, therefore reduces the number of comparisons.
* **Algorithm**:

void BinaryInsertionSort(int \*a, int size)

{

    int selected, index, loc;

    for (int i = 1; i < size; ++i)

    {

        selected = a[i], index = i - 1;

        loc = BinarySearch(a, selected, 0, index);

        while (index >= loc)

        {

            a[index + 1] = a[index];

            --index;

        }

        a[index + 1] = selected;

    }

}

* **Time complexity**:
* Best case: Ω(n)
* Worst case: Ο(n2)
* Average case: Θ(n2)
* **Review**: Binary insertion sort only provides minor improvement by reducing the number of comparisons in the insertion sort. In the worst case, binary insertion sort time complexity is still quadratic because of the series of swaps required for each insertion.

**6. Shell sort:**

* **Idea**: Shell sort is an improved version of insertion sort. By choosing a gap sequence, shell sort will make multiple small lists whose elements are determined by the gap values and sort them using insertion sort. In each pass, the gap values will progressively reduce. The final gap value is always 1.

void ShellSort(int \*a, int size)

{

    for (int gap = (size / 2); gap > 0; gap /= 2)

    {

        for (int i = gap; i < size; ++i)

        {

            int temp = a[i], j;

            for (j = i; j >= gap && a[j - gap] > temp; j -= gap)

                a[j] = a[j - gap];

            a[j] = temp;

        }

    }

}

* **Time complexity**:
* Best case: Unknown
* Worst case: Ο(n2)
* Average case: Unknown
* **Review**: Shell sort’s performance depends heavily on the gap sequence. Its time complexity remains an open problem. But since shell sort does not use recursion and can be implemented using little code, libraries target at embedded systems use this sorting algorithm. Shell sort is not stable.

**7. Merge sort:**

* **Idea**: Merge sort is a divide and conquer sorting algorithm. It simply divides the list into n sub-lists, which is considered sorted, then repeatedly merges these sub-lists until only one sub-list remaining. This will be the sorted list.
* **Algorithm**:

void merge(int \*a, int start, int mid, int stop)

{

    int \*lst = new int[stop - start];

    int i = start, j = mid, index = 0;

    while (i < mid && j < stop)

    {

        if (a[i] < a[j])

            lst[index++] = a[i++];

        else

            lst[index++] = a[j++];

    }

    while (i < mid)

        lst[index++] = a[i++];

    for (int k = 0; k < index; ++k)

        a[start + k] = lst[k];

    delete[] lst;

}

void MergeSort(int \*a, int start, int stop)

{

    if (start >= (stop - 1))

        return;

    int mid = (start + stop) / 2;

    MergeSort(a, start, mid);

    MergeSort(a, mid, stop);

    merge(a, start, mid, stop);

}

* **Time complexity**:
* Best case: Ω(nlogn)
* Worst case: Ο(nlogn)
* Average case: Θ(nlogn)
* **Review**: Merge sort is a good sorting algorithm. It makes fewer comparisons compared to quicksort in the worst case. The drawback of merge sort is its space complexity since it required Ο(n) extra space. But when it comes to sorting a linked list, merge sort is the best choice. Merge sort is the default sorting algorithm in some programming languages.

**8. Quick sort:**

* **Idea**: Quicksort is a divide and conquer sorting algorithm. By selecting a pivot element, quicksort partitions all other elements into two sub-lists according to whether they are less than or greater than the pivot. After the partition operation, the pivot is always in the right position. The sub-lists are then sorted recursively by the same partition method.
* **Algorithm**:

int partition(int \*a, int start, int stop)

{

    swap(a[start], a[(start + stop) / 2]);

    int pivotValue = a[start], i = start + 1, j = stop;

    while (i <= j)

    {

        while (a[i] < pivotValue && i <= j)

            ++i;

        while (a[j] > pivotValue)

            --j;

        if (i <= j)

        {

            swap(a[i], a[j]);

            ++i;

            --j;

        }

    }

    swap(a[start], a[j]);

    return j;

}

void QuickSort(int \*a, int start, int stop)

{

    if (start >= stop)

        return;

    int pivotindex = partition(a, start, stop);

    QuickSort(a, start, pivotindex - 1);

    QuickSort(a, pivotindex + 1, stop);

}

* **Time complexity**:
* Best case: Ω(nlogn)
* Worst case: Ο(n2)
* Average case: Θ(nlogn)
* **Review**: Quicksort is an efficient sorting algorithm. If quicksort is well implemented, it can run faster than merge sort and heapsort, which have the same time complexity as quicksort. Compared to merge sort, quicksort is an in-place sorting algorithm, so it has better space complexity. In the worst case, where the pivot is always the smallest or biggest value in the list, quicksort suffers from the quadratic time complexity, but this situation is very rare.

**9. Heap sort:**

* **Idea**: Heapsort converts the list into a heap data structure which is a binary tree that has a simple property: every root node of each subtree is greater than its leaf nodes. This operation is often called “Heapify”. The root of the heap is then removed from the heap. The heap is updated to maintain its property after each removal. When all elements are removed from the heap, the list is sorted.
* **Algorithm**:

void Heapify(int \*a, int left, int right)

{

    int i = left, j = (i \* 2) + 1, x = a[i];

    while (j <= right)

    {

        if (j < right)

        {

            if (a[j] < a[j + 1])

                ++j;

        }

        if (x > a[j])

            break;

        a[i] = a[j];

        i = j;

        j = (i \* 2) + 1;

    }

    a[i] = x;

}

void HeapSort(int \*a, int size)

{

    int i, right = size - 1, mid = right / 2;

    for (i = mid; i >= 0; --i)

        Heapify(a, i, right);

    while (right > 0)

    {

        swap(a[0], a[right]);

        --right;

        Heapify(a, 0, right);

    }

}

* **Time complexity**:
* Best case: Ω(nlogn)
* Worst case: Ο(nlogn)
* Average case: Θ(nlogn)
* **Review**: Heapsort is an efficient sorting algorithm. Compared to quicksort, despite typically running slower than quicksort, heapsort has better worst-case time complexity, which is Ο(nlogn). Quicksort however has a quadratic time complexity which is unacceptable for large data sets. Compared to merge sort, heapsort is better in space complexity since it only requires a constant amount of memory. Heapsort is not stable.

**10. Counting sort:**

* **Idea**: Counting sort is a distribution sorting algorithm. Hence, at first, it must look for the data range. Then, counting sort defines an auxiliary array, whose size is the same as the data range, to count the number of elements with distinct value. Finally, arithmetic is used on these counts to determine each elements’ sorted position.
* **Algorithm**:

int getMax(int \*a, int size)

{

    int max = a[0];

    for (int i = 1; i < size; ++i)

        if (a[i] > max)

            max = a[i];

    return max;

}

void CountingSort(int \*a, int size)

{

    int max = getMax(a, size), i;

    int \*count = new int[max + 1]{};

    int \*output = new int[size];

    for (i = 0; i < size; ++i)

        ++count[a[i]];

    for (i = 1; i <= max; ++i)

        count[i] += count[i - 1];

    for (i = 0; i < size; ++i)

    {

        output[count[a[i]] - 1] = a[i];

        --count[a[i]];

    }

    for (i = 0; i < size; ++i)

        a[i] = output[i];

    delete[] count;

    delete[] output;

}

* **Time complexity**:
* Worst case: Ο(n+ k)
* **Review**: Counting sort is a great sorting algorithm only if the data range is not significantly greater than the number of elements. Counting is often used as a subroutine sorting algorithm in radix sort. Counting sort is not stable.

**11. Radix sort:**

* **Idea**: Radix sort is a non-comparative sorting algorithm. It sorts the list’s elements according to the elements’ radix by distributing these elements into an array of 10 indexes from 0 to 9 corresponding to their digit. This process is repeated for each digit until all digits have been considered.
* **Algorithm**:

void rsCountingSort(int \*a, int size, int exp)

{

    int \*count = new int[10]{}, \*output = new int[size], i;

    for (i = 0; i < size; ++i)

        ++count[(a[i] / exp) % 10];

    for (i = 1; i < 10; ++i)

        count[i] += count[i - 1];

    for (i = 0; i < size; ++i)

    {

        output[count[(a[i] / exp) % 10] - 1] = a[i];

        --count[(a[i] / exp) % 10];

    }

    for (i = 0; i < size; ++i)

        a[i] = output[i];

    delete[] count;

    delete[] output;

}

void RadixSort(int \*a, int size)

{

    int m = getMax(a, size);

    for (int exp = 1; (m / exp) > 0; exp \*= 10)

        rsCountingSort(a, size, exp);

}

* **Time complexity**:
* Worst case: Ο(w\*n)
* **Review**: Radix sort is a very efficient sorting algorithm compared to many general-purpose sorting algorithms. Since it sorts elements according to their radix, radix sort can be applied to data that can be sorted lexicographically.

**12. Flash sort:**

* **Idea**: Flash sort is a distribution sorting algorithm. Each element of the list is classified into an auxiliary class for further sorting. With a known distribution, the class's ID of each element can be determined by computing simple arithmetic. Flash sort uses insertion sort as a subroutine sorting algorithm to sort these classes and finally recombine these sorted classes and copy back to the original list.
* **Algorithm**:

void FlashSort(int \*a, int size)

{

    const int m = 1000;

    int max = a[0], min = a[0], i, j, k;

    int count[m]{}, count\_copy[m]{};

    int \*\*classify\_arr = new int \*[m];

    //Find min max of the array

    for (i = 1; i < size; ++i)

    {

        if (a[i] > max)

            max = a[i];

        if (a[i] < min)

            min = a[i];

    }

    //Classification

    for (i = 0; i < size; ++i)

    {

        j = (m - 1) \* ((float)(a[i] - min) / (max - min));

        ++count[j];

        ++count\_copy[j];

    }

    for (i = 0; i < m; ++i)

        classify\_arr[i] = new int[count[i]];

    //Put array value into corresponding class

    for (i = 0; i < size; ++i)

    {

        j = (m - 1) \* ((float)(a[i] - min) / (max - min));

        classify\_arr[j][--count\_copy[j]] = a[i];

    }

    for (i = 0; i < m; ++i)

        InsertionSort(classify\_arr[i], count[i]);

    k = 0;

    for (i = 0; i < m; ++i)

    {

        for (j = 0; j < count[i]; ++j)

            a[k++] = classify\_arr[i][j];

    }

    for (i = 0; i < m; ++i)

        delete[] classify\_arr[i];

    delete[] classify\_arr;

    return;

}

* **Time complexity**:
* Best case: Ω(n)
* Worst case: Ο(n2)
* Average case: Θ(n)
* **Review**: Flash sort is a sorting algorithm that makes linear time complexity sorting possible. Insertion sort is very good for sorting small lists, hence in the ideal case, flash sort could achieve linear time complexity. In the worst case, flash sort has the same worst-case time complexity as its subroutine sorting algorithm, in this case, for insertion sort is Ο(n2). Choosing the right number of auxiliary class can speed up flash sort’s performance. Additionally, choosing better subroutine sorting algorithm can reduce the worst-case time complexity of flash sort. Flash sort is not stable.

**SORTING ALGORITHMS’ RUNTIME VISUALIZATIONS**

**Comment (Random order data):**

* Elementary sorting algorithms such as such as bubble sort, shaker sort, selection sort, insertion sort, and binary insertion sort have very bad runtime since they have average time complexity of Θ(n2). Their runtimes grow steadily when the size of the data is above 100000.
* Shaker sort has better runtime than bubble sort since shaker sort reduces redundant passes of bubble sort.
* Insertion sort and its improved version binary insertion sort appear to be the best elementary sorting algorithms compared to others. They are 10 times faster than bubble sort and over 2 times faster than selection sort.
* Other non-elementary sorting algorithms have very low runtime since most of them have average time complexity of Θ(nlogn). With a data size of 300000, all of them have less than 0.1s runtime, where shell sort is slowest with a runtime of 0.078s and counting sort is the fastest with merely 0.003s.

**Comment (Sorted data)**

* Bubble sort and selection are the only elementary sorting algorithm that has bad runtime. Bubble sort time complexity is great because no matter the data is sorted or not, it still runs all passes. Selection sort also runs all sequential search passes, so it has a similar result with bubble sort.
* For shaker sort and insertion sort, this is the best-case scenario. Shaker sort only runs 1 pass then terminated. Insertion sort does not have to do those costly exchange operations.
* Binary insertion sort has worse runtime compared to insertion sort since the binary search operations, in this case, is redundant.
* Other non-elementary sorting algorithms have very low runtime. Heapsort and flash sort have the worst runtime compared to the other. Heapsort’s heapify operation and all of flash sort’s operations for sorted data is completely redundant and costly.

**Comment (Nearly sorted data):**

* The runtime of sorting algorithms for nearly sorted data is similar to sorted data for the same reason as sorted data.-

**Comment (Reversed data):**

* Bubble sort and shaker sort have nearly the same runtime since reversed data is the worst-case scenario for shaker sort, whose improvements are completely redundant.
* This is the worst-case scenario for both insertion sort and binary insertion sort since in each pass, there are (n – 1) swaps.
* Other non-elementary sorting algorithms have very low runtime, they all have less than 0.1s runtime for a data size of 300000.

**SUMMARY:**

* For elementary sorting algorithms, bubble sort’s and selection sort’s runtime are predictable since in all cases, all passes will be executed.
* Shaker sort has minor improvements compared to bubble sort only if the data is not reversed.
* Insertion sort and binary insertion sort is faster than all other elementary sorting algorithms despite having the same worst-case time complexity. Their worst-case scenario is when the list is already sorted.
* For other sorting algorithms, they are all quite efficient since they have less than or equal to 0.1s runtime for a data size of 300000.
* Flash sort appears to be the worst in non-elementary sorting algorithms since its number of class constant is not optimal.
* Radix, counting, and flash sort are all fast but required auxiliary space.
* Elementary sorting algorithms have Ο(n2) time complexity.
* Most non-elementary algorithms have Ο(nlogn) time complexity. Flash sort could achieve linear time complexity Ο(n).