DSA Sorting

Table of Contents

Introduction

I. Algorithm mode:

```
command 1: running a sorting algorithm on the given input data
prototype: [Execution file] -a [Algorithm] [Given input] [Output params]
Example:   a.exe -a radix-sort input.txt -both

command 2: Run a sorting algorithm on the data generated automatically with
specified size and order
prototype; [Execution file] -a [Algorithm] [Input size] [Input order] [output
params]
```

```
Example:   a.exe -a selection-sort 50 -rand -time

command 3:  Run a sorting algorithm on ALL data arrangements of a specified size.
prototype: [Execution file] -a [Algorithm] [Input size] [Output params]
Example:   a.exe -a binary-insertion-sort 7000 -comp
```

II. Comparison mode:

```
command 4:  Run two sorting algorithms on the given input.
prototype: [Execution file ] -c [Algorithm 1] [Algorithm 2] [Given input]
Example:    a.exe -c heap-sort merge-sort input.txt

command 5:  Run two sorting algorithms on the data generated automatically.
prototype:  [Execution file] -c [Algorithm 1] [Algorithm 2] [Input size] [Input
order]
Example:    a.exe -c quick-sort merge-sort 100000 -nsorted
```

III. Input order ~ Data order : 4 types

```
 -rand: randomized data
 -nsorted: nearly sorted data
 -sorted: sorted data
- rev: reverse sorted data
```

IV. Output params:

```
-time: running time
-comp: number of comparisons
-both:
```

V. Given input:

```
1st line : an interger n, the number of elements in the array
2nd line: n integers, the elements of the array, separated by a single space
```

VI. Output order:

```
-time : algorithm running time
-comp : number of comparisons
-both : both time and comparisons
```

VII. Data size:

```
10 000, 30 000, 50 000, 100 000, 300 000, 500 000
```

Sorting Algorithms

1️⃣ Selection Sort

I. Algorithm:

```
1. Find the smallest element in the array and swap it with the first element.
2. Find the second smallest element in the array and swap it with the second
element.
3. Find the third smallest element in the array and swap it with the third
element.
4. Repeat until the array is sorted.
```

II. Complexity:

```
- Time complexity: O(n^2)
- Space complexity: O(1)
```

III. Pseudocode:

```
    for i = 0 to n - 2
        min = i
        for j = i + 1 to n - 1
            if a[j] < a[min]
                min = j
        swap a[i] and a[min]
```

2️⃣ Insertion Sort

I. Algorithm:

```
1. Insert the second element into the sorted subarray.
2. Insert the third element into the sorted subarray.
3. Insert the fourth element into the sorted subarray.
4. Repeat until the array is sorted.
```

II. Complexity:

```
- Time complexity: O(n^2)
- Space complexity: O(1)
```

III. Pseudocode:

```
    for i = 1 to n - 1
        x = a[i]
        j = i - 1
        while j >= 0 and a[j] > x
            a[j + 1] = a[j]
            j = j - 1
        a[j + 1] = x
```

3 Bubble Sort

I. Algorithm:

```
1. Compare the first element with the second element. If the first element is
greater than the second element, swap them.
2. Compare the second element with the third element. If the second element is
greater than the third element, swap them.
3. Compare the third element with the fourth element. If the third element is
greater than the fourth element, swap them.
4. Repeat until the array is sorted.
```

II. Complexity:

```
- Time complexity: O(n^2)
- Space complexity: O(1)
```

III. Pseudocode:

```
    for i = 0 to n - 2
        for j = n - 1 to i + 1
            if a[j] < a[j - 1]
                swap a[j] and a[j - 1]
```

4 Shaker Sort

I. Algorithm:

```
1. Compare the first element with the second element. If the first element is
greater than the second element, swap them.
2. Compare the second element with the third element. If the second element is
greater than the third element, swap them.
3. Compare the third element with the fourth element. If the third element is
greater than the fourth element, swap them.
4. Repeat until the array is sorted.
```

II. Complexity:

```
- Time complexity: O(n^2)
- Space complexity: O(1)
```

III. Pseudocode:

```
    for i = 0 to n - 2
        for j = n - 1 to i + 1
            if a[j] < a[j - 1]
                swap a[j] and a[j - 1]
        for j = i + 1 to n - 1
            if a[j] < a[j - 1]
                swap a[j] and a[j - 1]
```

5 Shell Sort

I. Algorithm:

```
1. Sort the elements with a gap of n/2.
2. Sort the elements with a gap of n/4.
3. Sort the elements with a gap of n/8.
4. Repeat until the array is sorted.
```

II. Complexity:

```
- Time complexity: O(n^2)
- Space complexity: O(1)
```

III. Pseudocode:

```
    for gap = n/2 to 1
        for i = gap to n - 1
            x = a[i]
            j = i - gap
            while j >= 0 and a[j] > x
                a[j + gap] = a[j]
                j = j - gap
            a[j + gap] = x
```

6 Heap Sort

I. Algorithm:

```
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it
with the last item of the heap followed by reducing the size of heap by 1.
Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.
```

II. Complexity:

```
- Time complexity: O(nlogn)
- Space complexity: O(1)
```

III. Pseudocode:

```
    buildMaxHeap(a, n)
    for i = n - 1 to 1
        swap a[0] and a[i]
        heapify(a, i, 0)

    heapify(a, n, i)
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and a[l] > a[largest]
        largest = l
    if r < n and a[r] > a[largest]
        largest = r
    if largest != i
```

```
        swap a[i] and a[largest]
        heapify(a, n, largest)
```

7 Merge Sort

I. Algorithm:

```
1. If the list is empty or has one item, it is sorted.
2. Divide the list recursively into two halves until it can no more be divided.
3. Merge the smaller lists into new list in sorted order.
```

II. Complexity:

```
- Time complexity: O(nlogn)
- Space complexity: O(n)
```

III. Pseudocode:

```
    mergeSort(a, l, r)
    if l < r
        m = (l + r) / 2
        mergeSort(a, l, m)
        mergeSort(a, m + 1, r)
        merge(a, l, m, r)

    merge(a, l, m, r)
    n1 = m - l + 1
    n2 = r - m
    L[1..n1], R[1..n2]
    for i = 1 to n1
        L[i] = a[l + i - 1]
    for j = 1 to n2
        R[j] = a[m + j]
    i = 1
    j = 1
    for k = l to r
        if L[i] <= R[j]
            a[k] = L[i]
            i = i + 1
        else
            a[k] = R[j]
            j = j + 1
```

8 Quick Sort

I. Algorithm:

```
1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements with values less than the pivot come
before the pivot, while all elements with values greater than the pivot come after
it (equal values can go either way). After this partitioning, the pivot is in its
final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-list of elements with smaller
values and separately to the sub-list of elements with greater values.
```

II. Complexity:

```
- Time complexity: O(nlogn)
- Space complexity: O(nlogn)
```

III. Pseudocode:

```
    quickSort(a, l, r)
    if l < r
        p = partition(a, l, r)
        quickSort(a, l, p - 1)
        quickSort(a, p + 1, r)

    partition(a, l, r)
    pivot = a[r]
    i = l - 1
    for j = l to r - 1
        if a[j] <= pivot
            i = i + 1
            swap a[i] and a[j]
    swap a[i + 1] and a[r]
    return i + 1
```

9 Counting Sort

I. Algorithm:

```
1. Find the largest element in the array.
2. Create a count array of size equal to the largest element and initialize it
with all zeros.
3. Store the count of each element at its respective index in count array.
4. Find cumulative sum of elements in the count array.
5. Find the index of each element of the original array in the count array. This
```

```
gives the cumulative count.
6. Place the element at the index calculated in step 5 in the output array.
7. Reduce the count by 1 for each index calculated.
```

II. Complexity:

```
- Time complexity: O(n+k)
- Space complexity: O(n+k)
```

III. Pseudocode:

```
    countingSort(a, n)
    max = findMax(a, n)
    count[0..max]
    output[1..n]
    for i = 0 to max
        count[i] = 0
    for i = 1 to n
        count[a[i]] = count[a[i]] + 1
    for i = 1 to max
        count[i] = count[i] + count[i - 1]
    for i = n downto 1
        output[count[a[i]]] = a[i]
        count[a[i]] = count[a[i]] - 1
    for i = 1 to n
        a[i] = output[i]
```

🔟 Radix Sort

I. Algorithm:

```
1. Find the maximum number to know number of digits.
2. Do counting sort for every digit. Note that instead of passing digit number,
exp is passed. exp is 10^i where i is current digit number.
3. The idea of Radix Sort is to do digit by digit sort starting from least
significant digit to most significant digit. Radix sort uses counting sort as a
subroutine to sort.
```

II. Complexity:

```
- Time complexity: O(nk)
- Space complexity: O(n+k)
```

III. Pseudocode:

```
    radixSort(a, n)
    m = findMax(a, n)
    for exp = 1 to m
        countingSort(a, n, exp)

    countingSort(a, n, exp)
    output[1..n]
    count[0..9]
    for i = 0 to 9
        count[i] = 0
    for i = 1 to n
        count[(a[i] / exp) % 10] = count[(a[i] / exp) % 10] + 1
    for i = 1 to 9
        count[i] = count[i] + count[i - 1]
    for i = n downto 1
        output[count[(a[i] / exp) % 10]] = a[i]
        count[(a[i] / exp) % 10] = count[(a[i] / exp) % 10] - 1
    for i = 1 to n
        a[i] = output[i]
```

1️⃣1️⃣ Flash Sort

I. Algorithm:

```
1. Find the minimum and maximum values in the array.
2. Calculate the m value using the following formula: m = (n * alpha) / (max -
min)
3. Create a new array of size m and initialize it with zeros.
4. Iterate over the array and calculate the index of each element using the
following formula: index = (array[i] - min) * m / (max - min)
5. Increment the value at the index calculated in step 4.
6. Iterate over the new array and calculate the cumulative sum.
7. Iterate over the array in reverse order and calculate the index of each element
using the following formula: index = (array[i] - min) * m / (max - min)
8. Place the element at the index calculated in step 7 in the output array.
9. Reduce the count by 1 for each index calculated.
```

II. Complexity:

```
- Time complexity: O(n)
- Space complexity: O(n)
```

III. Pseudocode:

```
flashSort(a, n)
max = findMax(a, n)
min = findMin(a, n)
m = (n * alpha) / (max - min)
l[m]
for i = 0 to m
    l[i] = 0
for i = 0 to n - 1
    index = (a[i] - min) * m / (max - min)
    l[index] = l[index] + 1
for i = 1 to m
    l[i] = l[i] + l[i - 1]
for i = n - 1 downto 0
    index = (a[i] - min) * m / (max - min)
    output[l[index]] = a[i]
    l[index] = l[index] - 1
for i = 0 to n - 1
    a[i] = output[i + 1]
```

:jigsaw: Experimental Results

Data Tables

Line Graphs

Bar Charts

References

- 1. tính thời gian chạy thuật toán
- 2. Space complexity