ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN CÔNG NGHỆ TRI THỨC

Đề tài: Sorting Algorithms

Môn học: DSA

Sinh viên thực hiện: TpG (23127244)

Giáo viên hướng dẫn: BHT

Ngày 26 tháng 6 năm 2024

Mục lục

1 Algorithms Presentation

1.1 Selection sort

1.1.1 Ideas

The meaning of an ascending sorted array is that the smallest element is placed first, the second smallest element is placed second, ..., the largest element is placed last.
For the first position in the array, we find the smallest element and swap it with the first element. For the second position, find the second smallest and swap it with the second element in the array, and so on.

1.1.2 Descriptions

Define the $0th$ smallest element as the smallest element, the $1st$ as the second smallest,...
We browse and sort each element of the n-element array (from 0 to $n-1$ respectively). When browsing to position i, we have the elements in the segment $[0, i-1]$ already sorted, the remaining elements are not sorted.
At this point, we need to find the ith smallest element to swap with element i. We have $i-1$ smallest element that is before i, so the ith smallest element of the array will be in the segment $[i, n-1]$ which is also the smallest element in the segment $[i, n-1]$.

Algorithm 1 Selection sort

```
function SELECTIONSORT(array, n)
    for i ∈ [0, n − 2] do
        min_id ← i

        for j ∈ [i + 1, n − 1] do
            if array[j] < array[min_id] then
                mid_id ← j
            end if
        end for

        temp ← array[i]
        array[i] ← array[min_id]
        array[min_id] ← temp
    end for
end function
```

1.1.3 Time complexity

The problem size is: n - the number of elements of the array.

Choose key operation (comparison) of the pseudocode is: $array[j] < array[min\_id]$.

Define $f(n)$ is the number of key time units. With $i = 0, 1, ..., n-2$, there are $n-1, n-2, ..., 1$ time unit respectively. So the total number of time units is

$$f(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$$

Time complexity: $O(n^2)$.

1.1.4 Space complexity

Since selection sort is an in-place sorting algorithm, it does not require additional storage.

Space complexity: $O(n)$.

1.2 Insertion sort

1.2.1 Ideas

Suppose we have a deck of cards in our hand, each card has an integer written on it. To sort the deck, we examine the cards one by one from left to right.

When checking the ith card, it means that the first $i-1$ cards have been sorted. So we just need to insert the ith card into the appropriate position in the first $i-1$ cards, then the first i cards will be sorted. Same with positions $i+1, i+2, ...$

1.2.2 Descriptions

To sort the array using insertion sort, we simulate checking and inserting cards from the above idea. When browsing to the ith element, we need to insert that element into the appropriate position in the first $i-1$ element. However, inserting elements is different from inserting cards. To place an element at position j, we need to shift elements $j, j+1, ...$ one position to the right before assigning the value to element j.

In turn checking and inserting elements from left to right, we will get a sorted array.

Algorithm 2 Insertion sort

```
function insertionSort(array, n)
    for i ∈ [1, n − 1] do
        key ← array[i]
        j ← i − 1

        while j ≥ 0 and key < array[i] do
            array[j + 1] ← array[j]
            j ← j − 1
        end while

        array[j + 1] ← key
    end for
end function
```

1.2.3 Time complexity

The problem size is: n - the number of elements of the array.

Choose key operation (comparision) of the pseudocode is: $key < array[i]$.

Define $f(n)$ is the number of key time units.

- **The worst case:** each element in the array needs to be inserted at the beginning. Specifically, the original array is sorted descending while we need to sort the array ascending.

  With $i = 1, 2, ..., n-1$, there are $1, 2, ..., n-1$ time unit respectively. So the total number of time units is

$$f(n) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$$

- **The best case:** each element in the array does not need to be inserted into a different location. Specifically, the initial array is sorted.

  Each element need 1 time unit. So the total number of time units is

$$f(n) = \underbrace{1 + 1 + ... + 1}_{n-1 \text{ times}} = n - 1$$

Time complexity (worst case): $O(n^2)$.

Time complexity (best case): $O(n)$.

1.2.4 Space complexity

Since insertion sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$.

1.3 Bubble sort

1.3.1 Ideas

The meaning of an ascending sorted array is that the smallest element is placed first, the second smallest element is placed second, ..., the largest element is placed last.
First, we find the largest element and bring it to the end of the array. Next, we find and put the second largest element next to the largest element and so on. Suppose the position of the largest element is i. To move element i to the end of the array, we move element i one position to the right until it reaches the end of the array.

1.3.2 Descriptions

When performing the operation of swapping $array[i]$ and $array[i+1]$, element i will be moved 1 position to the right. To bring element i to the end, we swap elements $(i, i+1)$, $(i+1, i+2), ...$
Obviously, we only swap $(i, i+1)$ when $array[i] > array[i+1]$ and this is called bubbling. When bubbling positions $(1, 2); (2, 3); (3, 4); ...$, the largest element (position i) is always brought to the end of the array because $array[i]$ is larger than any other element after it (it will be bubbled towards the end of the array).
First round of bubbling $(1, 2); (2, 3); ...$ will bring the largest element to the end of the array; The second round of bubbling will bring the second largest element next to the largest element; ...

Algorithm 3 Bubble sort

```
function bubbleSort(array, n)
    for i ∈ [0, n − 2] do
        for j ∈ [0, n − i − 2] do
            if array[j + 1] < array[j] then
                temp ← array[j + 1]
                array[j + 1] ← array[j]
                array[j] ← temp
            end if
        end for
    end for
end function
```

1.3.3 Time complexity

The problem size is: n - the number of elements of the array.

Choose key operation (comparison) of the pseudocode is: $array[j+1] < array[j]$.

Define $f(n)$ is the number of key time units. With $i = 0, 1, ..., n-2$, there are $n-1, n-2, ..., 1$ time unit respectively. So the total number of time units is

$$f(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$$

Time complexity: $O(n^2)$.

1.3.4 Space complexity

Since bubble sort is an in-place sorting algorithm, it does not require additional storage.

Space complexity: $O(n)$.

1.4 Merge sort

1.4.1 Ideas

Suppose we have two sorted arrays, we can merge them into one sorted array in $O(n)$. Conversely, to sort an array, we can divide it into two sub-arrays of equal length (or a difference of 1), sort those two sub-arrays and then merge them back to the original.

To sort the two sub-arrays, we continue to divide them into smaller sub-arrays as we did. The division repeats until each sub-array has the size one element. After that, we merge them back.

1.4.2 Descriptions

The Merge sort algorithm is a divide-and-conquer algorithm.

Divide:

Considering the unsorted segment $[l, r]$ of the array, we will divide it into two segments $[l, mid]$ and $[mid + 1, r]$ with $mid = \lfloor \frac{l + r}{2} \rfloor$ is the middle position of the segment.
Sorting segment $[l, mid]$ and $[mid + 1, r]$ is the same as $[l, r]$.

Conquer:

Merging two sub-arrays $array[l, mid]$ and $array[mid + 1, r]$ to $array[l, r]$

- Step 1: Create two arrays $array1[]$ of size $n1 = mid - l + 1$ and $array2[]$ of size $n2 = r - mid$.
- Step 2: Simultaneously traverse $array1[]$ and $array2[]$. Pick smaller of current elements in $array1[]$ and $array2[]$, copy this smaller element to next position in $array[l, r]$ and move ahead in $array[l, r]$ and the array whose element is picked.
- Step 3: If there are remaining elements in $array1[]$ or $array2[]$, copy them also in $array[l, r]$.

Algorithm 4 Merge sort

```
function MERGE(array, l, mid, r)
    n1 ← mid − l + 1
    n2 ← r − mid
    array1[n1] ← array[l, mid]
    array2[n2] ← array[mid + 1, r]
    i ← 0, j ← 0, id ← l

    while i < n1 and j < n2 do
        if array1[i] < array2[j] then
            array[id] ← array1[i]
            i ← i + 1
            id ← id + 1
        else if array1[i] > array2[j] then
            array[id] ← array2[j]
            j ← j + 1
            id ← id + 1
        else
            array[id] ← array1[i]
            array[id + 1] ← array2[j]
            i ← i + 1
            j ← j + 1
            id ← id + 2
        end if
    end while
end function

function MERGESORT(l, l, r)
    if l < r then
        mid ← ⌊(l + r)/2⌋
        mergeSort(a, l, mid)
        mergeSort(a, mid + 1, r)
        Merge(a, l, mid, r)
    end if
end function
```

1.4.3 Time complexity

The problem size is: n - the number of elements of the array.

Number of division stages is: $\log_2 n$ (sub-arrays' size is reduced by two times each division step).

On each merge stage, there are n elements are merged:

- Stage 1: $n \times 1$ elements.

- Stage 2: $n/2 \times 2$ elements.
- Stage 3: $n/4 \times 4$ elements.
- ...

Time complexity: $O(n \log n)$.

1.4.4 Space complexity

Each merge stage, we will merge exactly n elements, which means the total size of the arrays created in each stage is exactly n (and then being deleted after each stage).
Space complexity: $O(n)$.

1.5 Counting sort

1.5.1 Ideas

Consider n balls, each labeled with a non-negative integer with the maximum value is max. Take turns counting the number of balls with values $0, 1, 2, ..., max$.
Suppose $n = 10, max = 4$, and there are:

- 2 balls with value 0;
- 3 balls with value 1;
- 0 ball with value 2;
- 1 balls with value 3;
- 4 ball with value 4;

To sort the balls, we place them in turn:

$$\{\overbrace{0, 0}^{2 \text{ times}}, \underbrace{1, 1, 1}_{3 \text{ times}}, \overbrace{3}^{1 \text{ time}}, \underbrace{4, 4, 4, 4}_{4 \text{ times}}\}$$

.

1.5.2 Descriptions

With non-negative arrays, we do the same idea above. First, we find the max value of the array. Then, creates an array $cnt[]$ of size $max+1$ (from 0 to max) with the meaning: $cnt[x]$ is the number of elements with value x in the array.

After counting the number of each value, we them place back in the array in order of increasing value, each value x repeated exactly $cnt[x]$ times.

Algorithm 5 Counting sort

function COUNTINGSORT($array$, n)
    $max \leftarrow array[0]$
    for $i \in [1, n-1]$ **do**
        if $max < array[i]$ **then**
            $max \leftarrow array[i]$
        end if
    end for

    Creates an array $cnt[]$ has size of $max + 1$, each element has value 0.

    for $i \in [0, n-1]$ **do**
        $cnt[array[i]] \leftarrow cnt[array[i]] + 1$
    end for

    $id \leftarrow 0$
    for $x \in [0, max]$ **do**
        while $cnt[x] > 0$ **do**
            $array[id] \leftarrow x$
            $id \leftarrow id + 1$
            $cnt[x] \leftarrow cnt[x] - 1$
        end while
    end for
end function

1.5.3 Time complexity

The problem size are:

- n - the number of elements of the array;
- max - the maximum value in the array.

Choose key operation (comparison) of the pseudocode is are: $cnt[x] > 0$.

Define $f(n)$ is the number of key time units. Consider the **for** $x \in [0, max]$ loop, for each value of x, there are $cnt[x] + 1$ comparisons will be done. So the total number of time units is:

$$f(n) = (cnt[0] + 1) + (cnt[1] + 1) + ... + (cnt[max] + 1)$$

$$= \underbrace{cnt[0] + cnt[1] + ... + cnt[max]}_{\text{the number of elements in the array}} + \underbrace{1 + 1 + ... + 1}_{max \text{ times}} = n + max$$

Time complexity: $O(n + max)$.

1.5.4 Space complexity

Since we need additional storage for counting elements, the total storage is the size of the array to be sorted plus the size of the array for counting elements.

Space complexity: $O(n + max)$.

Tài liệu

A Phụ lục

- Template này **không phải** là template chính thức của Khoa Công nghệ thông tin - Trường Đại học Khoa học Tự nhiên.
- Các hình ảnh, bảng biểu, thuật toán trong template chỉ mang tính chất ví dụ.
- Nhóm tác giả phân phối **miễn phí** template này trên GitHub và trên Overleaf với Giấy phép GNU General Public License v3.0. Nhóm tác giả không chịu trách nhiệm với các bản phân phối không nằm trong hai kênh phân phối chính thức nêu trên.