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 27 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

We iterate through elements of the n-element array (from 0 to $n-1$ respectively). When we consider 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. Because $i-1$ smallest elements are in the segment $[0, i-1]$, the ith smallest element of the array is the smallest element in the segment $[i, n-1]$.

Algorithm 1 Selection sort

```
1:  function selectionSort(array, n)
2:      for i ← 0 to n − 2 do
3:          min_id ← i
4:
5:          for j ← i + 1 to n − 1 do
6:              if array[j] < array[min_id] then
7:                  mid_id ← j
8:              end if
9:          end for
10:
11:         temp ← array[i]
12:         array[i] ← array[min_id]
13:         array[min_id] ← temp
14:     end for
15: end function
```

1.1.3 Time complexity

The problem size is: n - the number of elements in 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 go through the cards one by one from left to right.

When we consider 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

The insertion sort simulates considering and inserting cards from the above idea. When we consider the ith element, we need to insert that element into the appropriate position in the first $i-1$ elements.

However, inserting an element is different from inserting a card. To insert an element at position j, we need to shift all the elements $j, j+1, ...$ one position to the right before inserting to position j.

In turn going through the cards one by one from left to right, we will get a sorted array.

Algorithm 2 Insertion sort

```
1: function insertionSort(array, n)
2:     for i ← 1 to n − 1 do
3:         key ← array[i]
4:         j ← i − 1
5:
6:         while j ≥ 0 and key < array[i] do
7:             array[j + 1] ← array[j]
8:             j ← j − 1
9:         end while
10:
11:        array[j + 1] ← key
12:    end for
13: end function
```

1.2.3 Time complexity

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

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

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

- **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

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

- **Best case:** each element in the array does not need to be inserted into a different position. 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 bring the largest element to the end of the array. Next, we move the second largest element next to the largest element, and so on.

Bubble sort consists of $n-1$ rounds. On each round, the algorithm iterates through elements of the array. Whenever two consecutive elements are found that are not in correct order, the algorithm swaps them.

After the first round of the algorithm, the largest element will be in the correct position, and in general, after k rounds, the k largest elements will be in the correct positions. Thus, after $n-1$ rounds, the whole array will be sorted.

1.3.2 Descriptions

Bubble sort consists of two nested loops. The outer loop is used to count the number of rounds. The inner loop is used for "bubbling" elements.

If $n-1$ elements are in the correct position, the remaining element will also be in the correct position. So, the algorithm only need $n-1$ rounds to sort the array.

Moreover, we know that at the kth loop there will be $k-1$ elements in the correct position at the end of the array, so we do not need to consider these elements.

Improvement: if there is no swap operations were performed in a round, the array is now sorted, no further operations need to be done. We can add a flag to check if the swap operation was performed.

Algorithm 3 Bubble sort

```
1: function bubbleSort(array, n)
2:     for i ← 0 to n − 2 do
3:         do_swap ← 0
4:
5:         for j ← 0 to n − i − 2 do
6:             if array[j + 1] < array[j] then
7:                 temp ← array[j + 1]
8:                 array[j + 1] ← array[j]
9:                 array[j] ← temp
10:                do_swap ← 1
11:            end if
12:        end for
13:
14:        if do_swap = 1 then
15:            Stop the outer loop
16:        end if
17:    end for
18: end function
```

1.3.3 Time complexity

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

- **Worst case:** the array is not sorted until the last round of bubbling is performed. With $i = 0, 1, ..., n-2$, there are $n-1, n-2, ..., 1$ time unit respectively. So the total number of time units is

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

- **Best case:** the array is already sorted. In the first round, there will be no swap operation is performed, so there will be only $n-1$ comparisons are made before the algorithm stops.

$$f(n) = n - 1$$

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

Time complexity (best case): $O(n)$ (only when the bubble sort has the improvement).

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

Merge sort is a divide-and-conquer algorithm.

Divide:

Considering the unsorted segment $[l, r]$ of the array, we 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 $array[l, r]$. Then, 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

```
1: function MERGE(array, l, mid, r)
2:     n1 ← mid − l + 1
3:     n2 ← r − mid
4:     array1[n1] ← array[l, mid]
5:     array2[n2] ← array[mid + 1, r]
6:     i ← 0, j ← 0, id ← l
7:
8:     while i < n1 and j < n2 do
9:         if array1[i] < array2[j] then
10:            array[id] ← array1[i]
11:            i ← i + 1
12:            id ← id + 1
13:        else if array1[i] > array2[j] then
14:            array[id] ← array2[j]
15:            j ← j + 1
16:            id ← id + 1
17:        else
18:            array[id] ← array1[i]
19:            array[id + 1] ← array2[j]
20:            i ← i + 1
21:            j ← j + 1
22:            id ← id + 2
23:        end if
24:    end while
25:
26:    while i < n1 do
27:        array[id] ← array1[i]
28:        i ← i + 1
29:        id ← id + 1
30:    end while
31:
32:    while j < n2 do
33:        array[id] ← array2[j]
34:        j ← j + 1
35:        id ← id + 1
36:    end while
37: end function
38:
39: function MERGESORT(l, l, r)
40:    if l < r then
41:        mid ← ⌊(l + r)/2⌋
42:        mergeSort(a, l, mid)
43:        mergeSort(a, mid + 1, r)
44:        Merge(a, l, mid, r)
45:    end if
46: end function
```

1.4.3 Time complexity

The problem size is: n - the number of elements in the array.
The number of division stages is: $\log_2 n$ (sub-arrays' size is reduced by two times each division step).
On each merging 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 merging stage, we will merge exactly n elements, which means the total size of the arrays created in each stage is exactly n elements (they will be 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 ball with value 3;
- 4 balls 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

```
1:  function countingSort(array, n)
2:      max ← array[0]
3:      for i ← 1 to n − 1 do
4:          if max < array[i] then
5:              max ← array[i]
6:          end if
7:      end for
8:
9:      Create an array cnt[] of size n + 1; the initial value of the elements is 0.
10:
11:     for i ← 0 to n − 1 do
12:         cnt[array[i]] ← cnt[array[i]] + 1
13:     end for
14:
15:     id ← 0
16:     for x ← 0 to max do
17:         while cnt[x] > 0 do
18:             array[id] ← x
19:             id ← id + 1
20:             cnt[x] ← cnt[x] − 1
21:         end while
22:     end for
23: end function
```

1.5.3 Time complexity

The problem size are:

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

Choose key operation (comparison) of the pseudocode is: $cnt[x] > 0$.
Define $f(n)$ is the number of key time units. Consider the **for** $x \leftarrow 0$ **to** max **do** loop, for each value of x, there are $cnt[x] + 1$ comparisons will be made. 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.