**Algorithm Unit 1**

**110403518 林晉宇**

1. **Prove each sorting algorithm is stable and in-place or not.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Bubble** | **Insertion** | **Quick** | **Heap** | **Selection** |
| **Stable** | YES | YES | NO | NO | NO |
| **In place** | YES | YES | YES/NO | YES | YES |

1. **Bubble Sort:**

* **Stable**, we swap elements only when A is less than B. If A is equal to B, we do not swap them. Thus, relative order between equal elements will be maintained.
* **In place**, there’s only fixed amount of extra space for flag(i,j), size variable and tmp to store swap value. Thus, the space complexity is O(1).

***for i=0 ~ length(A)-1***

***for j=0 ~ length(A)-1***

***if(A[j]>A[j+1])***

***swap(A[j],A[j+1])***

1. **Insertion Sort:**

* **Stable**, we only swap the ordering of any two items if the item on the right is less than the item to its left. Thus, the ordering of two equivalent items will always maintained.
* **In place**, it only need fixed amount of extra space for flag(i,j) and a key value for comparison. Thus, the space complexity is O(1).

***for i=0 ~ length(A)-1***

***j=i;***

***while j>0 and A[j-1]>A[j]***

***swap(A[j],A[j-1])***

***j--***

1. **Quick Sort:**

* **Not Stable**, it swaps non-adjacent elements. Quick sort choose some element to be its pivot, and then divides the supplied array around it.
* **In place**,

1. **Heap Sort:**

* **Not Stable**,
* **In place**,

1. **Selection Sort:**

* **Not Stable**, it swaps non-adjacent elements. Selection sort works by finding the minimum element and then inserting it in its correct position by swapping with the element to the right position. This is what makes it unstable.

**Counter example:** [4\* 2 3 4 1] -> [1 2 3 4 4\*], the relative order of 4’s has changed after operate selection sort.

* **In place**, it only need fixed amount of extra space for flag**(i,j)**, **tmp** for swaping value, and **min** to store the minimum value for each iteration.

***for i=0 ~ length(A)-1***

***min=i***

***for j=i+1 ~ length(A)-1***

***if A[j]<A[min]***

***min=j***

***if min!=j***

***swap(A[j],A[min])***

1. **Design a data structure to represent a set with elements being positive integers, then design algorithms for the following operations.**

I use hash table, which each integer is stored as a key in the table. Hash table are **O(1)** average and amortized case complexity(worst case is **O(n)**).

Here I use **unordered\_set** in C++ since it is implemented using a hash table.

I assume **set1** size is **n**, and **set2** size is **m**.

1. **Compute the union of two sets.**

To compute the union of two sets, we iterate over all elements in two sets and add them to a new set if they haven’t existed yet.

Time complexity: **O(n+m)**

***unordered\_set<int> hash //hash table***

***res={} //empty set***

***for i : set1***

***hash.insert(1)***

***for j : set2***

***hash.insert(2)***

***for k : hash***

***res.insert(k) //union of 2 sets***

***return res***

1. **Compute the intersection of two sets.**

To compute the intersection of two sets, we can iterate over each element in the first set and check if it exist in other set. If it does, insert that element to **res** set.

Time complexity: **O(n+m)**

***unordered\_set<int> hash //hash table***

***res={} //empty set***

***for i : set1***

***hash.insert(1)***

***for j : set2***

***if j existed in hash***

***res.insert(j)***

***return res***

1. **Determine if a given element is in a given list.**

To determine this, we can simply check whether or not the element exists in the hash table.

Time complexity: **O(n)** or **O(m)**

***bool exist(element,set)***

***unordered\_set<int> hash //hash table***

***for i : set***

***hash.insert(i)***

***return element in hash***

**3. Given two sorted array, design an algorithm to compute min|x[i]-y[j]|.**

1. **Pseudo code**

***i=0,j=0,res=INF***

***while(i<m&&j<n)***

***res=min(res,abs(x[i]-y[j]))***

***if(x[i]<=y[j])  i++***

***else    j++***

***return res***

1. I use two-pointer to implement this problem. First, initialize two pointers(**i and j**) point to the first element of two array. Also initialize **res** as infinity value. While i<m and j<n, we update **res** if smaller answer appear. If **x[i]** is less than **y[j]** we increment I, else increment j.
2. This method work because two arrays are sorted, the elements that are close to each other will be close to each other in two arrays as well. The time complexity will be **O(m+n)** which is a linear method since we only need to scan through two arrays once.

**4. There’s a sequence with n integers and many duplications, the number of distinct integers in the sequence is O(logn).**

1. We use a self-balancing binary search tree (eg. AVL / RB…) to implement, and we can achieve **O(logN)** time in insertion (N is the number of the elements in the tree).
2. For each element in the sequence, we try to insert it into the binary search tree.

* If the element is existed, we append a count variable stored at that node.
* Else, we insert the node and set the count to 1.

1. Traverse the binary search tree in inorder, and add elements according to the count.

There are **O(logn)** distinct elements to be inserted, so the total insertion time takes **O(loglogn)**. Then, we traverse the tree in inorder by also adding elements according to the count, which takes **O(n)**. Thus, the total time complexity will be **O(nloglogn)**.

1. **Why is the lower bound of sorting (nlogn) not satisfied in this case?**

The major reason is due to the number of distinct integers in the sequence is **O(logn),** we only need to insert distinct element to our binary search tree instead of whole sequence.

**5. Selection Sort**

1. **Pseudo code**

***void selection\_sort(a[])***

***for i=1~a.length()-1 //1-based***

***minIndex=i***

***for j=i+1~a.length()***

***if a[j]<a[minIndex]***

***minIndex=j***

***swap(a[i],a[minIndex])***

1. **What loop invariant does this algorithm maintain?**

The start of each iteration of the outer for loop, the subarray a[1]~a[i-1] contains i-1 smallest elements of array a, sorted in increasing order.

1. **Why does it need to run for only the first n-1 elements rather than n elements?**

Because by compare only the first n-1 elements, the last iteration will compare a[n] with the minimum eleme nt in a[1]~a[n-1] and swap if it fits the condition. So, there is no need to continue one more iteration.

1. **Give the worst-case running time of selection sort in theta notation.**

Θ(), the worst-case will be to sort a reverse sorted array. We will take one element at a time and compare it with all the other elements. So, each n elements will be compared with rest of the n-1 elements. Thus, the running time will be Θ().

1. **Is the best case running time any better?**

No, the best-case means the original array is already sorted, however, selection sort will still do the same thing as in worst-case. Therefore, the running time for both scenario is Θ(). The best-case does not run any better.