inzva Algorithm Programme 2018-2019

Bundle 2

Algorithms - 1

Editor
Kadir Emre Oto

Reviewers
Muhammed Burak Buğrul
Tahsin Enes Kuru

Contents

1 Search Algorithms **3**
1.1 Linear Search 3
1.2 Binary Search 4
1.3 Ternary Search 6

2 Sorting Algorithms **7**
2.1 Insertion Sort 7
2.2 Merge Sort 8
2.3 Quick Sort 9
2.4 Radix Sort 10

3 Divide and Conquer **11**

1 Search Algorithms

It may be necessary to determine if an array or solution set contains a specific data, and we call this finding proccess **searching**. In this article, three most common search algorithms will be discussed: linear search, binary search, and ternary search.

This visualization may help you understand how the search algorithms work: Link.

1.1 Linear Search

Simplest search algorithm is *linear search*, also know as *sequential search*. In this technique, all elements in the collection of the data is checked one by one, if any element matches, algorithm returns the index; otherwise, it returns -1.

In the worse case scenario, the search key may not be in the array, or it may be placed towards the end of it. Under these circumstances, the algorithms may check almost all the elements, it means the time complexity is $O(N)$

0	1	2	3	4	5	6	7	8	9	Search Key: 70
7	29	48	53	63	70	76	89	94	96	7 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	29 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	48 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	53 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	63 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 1: Example for linear search

```c
int linearSearch(int *array, int size, int key){
    for (int i=0; i < size; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```

1.2 Binary Search

We know linear search is quite a slow algorithm because it compares each element of the set with search key, and there is a high-speed searching technique for **sorted** data instead of linear search, which is **binary search**. After each comparison, the algorithm eliminates half of the data using the sorting property.

We can also use binary search on increasing functions in the same way.

Procedure:

- Compare the key with the middle element of the array,
- If it is a match, return the index of middle.
- If the key is bigger than the middle, it means that the key must be in the right side of the middle. We can eliminate the left side.
- If the key is smaller, it should be on the left side. The right side can be ignored.

Comlexity:

$$\begin{aligned} T(N) &= T(N/2) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

L				M					R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	63 < 70, shift L

					L		M		R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	89 > 70, shift R

					L M	R				Search Key: 70
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 2: Example for binary search

```c
int binarySearch(int *array, int size, int key){
    int left = 0, right = size, mid;

    while (left < right){
        mid = (left + right) / 2;

        if (array[mid] >= key)
            right = mid;
        else
            left = mid + 1;
    }
    return array[left] == key ? left : -1 ;
}
```

1.3 Ternary Search

Suppose that we have a unimodal function, $f(x)$, on an interval $[l, r]$, and we are asked to find the local minimum or the local maximum value of the function according to the behavior of it.

There are two types of unimodal functions:

1. The function, $f(x)$ strictly increases for $x \leq m$, reaches a global maximum at $x = m$, and then strictly decreases for $m \leq x$. There are no other local maxima.

2. The function, $f(x)$ strictly decreases for $x \leq m$, reaches a global minimum at $x = m$, and then strictly increases for $m \leq x$. There are no other local minima.

In this document, we will implement the first type of unimodal function, and the second one can be solved using the same logic.

Procedure:

1. Choose any two points m_1, and m_2 on the interval $[l, r]$, where $l < m_1 < m_2 < r$.

2. If $f(m_1) < f(m_2)$, it means the maxima should be in the interval $[m_1, r]$, so we can ignore the interval $[l, m_1]$, move l to m_1

3. Otherwise, $f(m_1) \geq f(m_2)$, the maxima have to be in the interval $[l, m_2]$, move r to m_2

4. If $r - l < \epsilon$, where ϵ is a negligible value, stop the algorithm, return l. Otherwise turn to the step 1.

m_1 and m_2 can be selected by $m_1 = l + (r - l)/3$ and $m_2 = r - (r - l)/3$ to avoid increasing the time complexity.

Comlexity:

$$\begin{aligned} T(N) &= T(2 \cdot N/3) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

```
double f(double x);

double ternarySearch(double left, double right, double eps=1e-7){
    while (right - left > eps){
        double mid1 = left + (right - left) / 3;
        double mid2 = right - (right - left) / 3;

        if (f(mid1) < f(mid2))
            left = mid1;
        else
            right = mid2;
    }
    return f(left);
}
```

2 Sorting Algorithms

Sorting algorithms are used to put the elements of an array in a certain order according to the comparison operator. Numerical order or lexcographical orders are the most common ones, and there are a large number of sorting algorithms, but we discuss four of them: *Insertion Sort*, *Merge Sort*, *Quick Sort*, *Radix Sort*.

For a better understanding, you are strongly recommended to go into this visualization site after reading the topics: Link

2.1 Insertion Sort

Think that you are playing a card game and want to sort them before the game. Your sorting strategy is simple: you have already sorted some part and every time you pick up the next card from unsorted part, you insert it into the correct place in sorted part. After you apply this process to all cards, the whole deck would be sorted.

This is the basic idea for sorting an array. We assume that the first element of the array is the sorted part, and other elements are in the unsorted part. Now, we choose the leftmost element of the unsorted part, and put it into the sorted part. In this way the left part of the array always remains sorted after every iteration, and when no element is left in the unsorted part, the array will be sorted.

```
void insertionSort(int *ar, int size){
    for (int i=1; i < size; i++)
        for (int j=i-1; 0 <= j and ar[j] > ar[j+1]; j--)
            swap(ar[j], ar[j+1]);
}
```

2.2 Merge Sort

Merge Sort is one of the fastest sorting algorithms that uses *Divide and Conquer* paradigm. The algorithm **divides** the array into two halves, solves each part **recursively** using same sorting function and **combines** them in linear time by selecting the smallest value of the arrays every time.

Procedure:

1. If the size of the array is 1, it is sorted already, stop the algorithm (base case),
2. Find the middle point of the array, and split it in two,
3. Do the algorithm for these parts separately from the first step,
4. After the two halves got sorted, merge them in linear time and the array will be sorted.

Comlexity:

$$\begin{aligned} T(N) &= T(N/2) + O(N) \\ T(N) &= O(N \cdot logN) \end{aligned}$$

```
void mergeSort(int *ar, int size){
    if (size <= 1)  // base case
        return;

    mergeSort(ar, size / 2);  // divide the array into two almost equal parts
    mergeSort(ar + size / 2, size - size / 2);

    int index = 0, left = 0, right = size / 2;  // merge them
    int *temp = new int [size];

    while (left < size / 2 or right < size){
        if (right == size or (left < size / 2 and ar[left] < ar[right]))
            temp[index++] = ar[left++];
        else
            temp[index++] = ar[right++];
    }
    for (int i=0; i < size; i++)
        ar[i] = temp[i];
    delete [] temp;
}
```

2.3 Quick Sort

Quick Sort is also a *Divide and Conquer* algorithm. The algorithm chooses an element from the array as a pivot and partitions the array around it. Partitioning is arranging the array that satisfies those: the pivot should be put to its correct place, all smaller values should be placed before the pivot, and all greater values should be placed after the pivot. The partitioning can be done in linear time, and after the partitioning, we can use the same sorting function to solve the left part of the pivot and the right part of the pivot recursively.

If the sellected pivot cannot divide the array uniformly after the partitioning, the time complexity can reach $O(n^2)$ like insertion sort. To avoid this, the pivot can generally be picked randomly.

Procedure:

1. If the size of the array is 1, it is sorted already, stop the algorithm (base case),
2. Choose a pivot randomly,
3. For all values in the array, collect smaller values in the left of the array and greater values in the right of array,
4. Move the pivot to the correct place,
5. Repeat the same algorithm for the left partition and the right partition.

Comlexity:

$$\begin{aligned} T(N) &= T(N/10) + T(9 \cdot N/10) + O(N) \\ T(N) &= O(N \cdot logN) \end{aligned}$$

```
void quickSort(int *ar, int size){
    if (size <= 1)  // base case
        return;

    int position = 1;  // find the correct place of pivot
    swap(ar[0], ar[rand() % size]);

    for (int i=1; i < size; i++)
        if (ar[0] > ar[i])
            swap(ar[i], ar[position++]);
    swap(ar[0], ar[position-1]);

    quickSort(ar, position-1);
    quickSort(ar + position, size - position);
}
```

2.4 Radix Sort

Quick Sort and *Merge Sort* are comparison-based sorting algorithms and cannot run better than $O(logN)$. However, *Radix Sort* works in linear time ($O(N+K)$, where K is $log(max(ar))$).

Procedure:

1. For each digit from the least significant to the most, sort the array using *Counting Sort* according to corresponding digit. *Counting Sort* is used for keys between specific range, and it counts the number of elements which have different key values. After counting the number of distict key values, we can determine the position of elements in the array.

Comlexity:

$$T(N) \quad = \quad O(N)$$

```
void radixSort(int *ar, int size, int base=10){
    int *temp = new int [size];
    int *count = new int [base]();

    for (int e=1; count[0] != size; e *= base){
        memset(count, 0, sizeof(int) * base);

        for (int i=0; i < size; i++)
            count[(ar[i]/e) % base]++;

        for (int i=1; i < base; i++)
            count[i] += count[i-1];

        if (count[0] == size)
            break;

        for (int i=size-1; 0 <= i; i--)
            temp[--count[(ar[i]/e) % base]] = ar[i];

        for (int i=0; i < size; i++)
            ar[i] = temp[i];
    }

    delete [] temp;
    delete [] count;
}
```

3 Divide and Conquer

Divide and Conquer is a well-known paradigm that **breaks** up the problem into several parts, **solves** each part independently, and finally **combines** the solutions to the subproblems into the overall solution. Because each subproblem is solved recursively, they should be the smaller versions of the original problem; and the problem must have a base case to end the recursion.

Some example algorithms that use divide and conquer technique:

- Merge Sort
- Count Inversions
- Finding the Closest Pair of Points
- Others

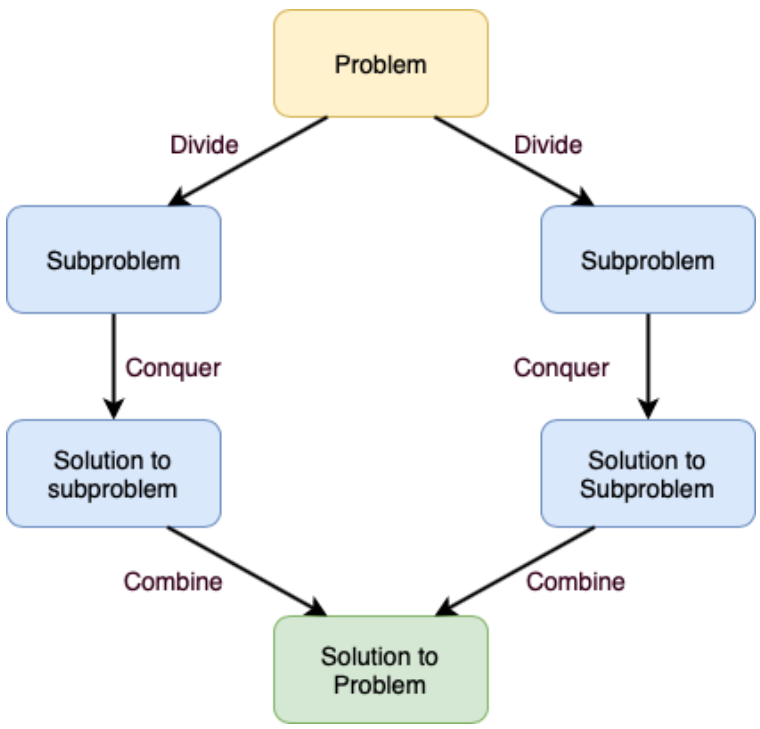

Figure 3: The Flow of *Divide and Conquer*