Comparisons on Sorting and Searching

VE281 - Data Structures and Algorithms, Xiaofeng Gao, Autumn 2019

This document introduces the implementation details and performance of the most commonly used searching and sorting algorithms, including *Linear Search*, *Binary Search*, *Selection Sort*, *Bubble Sort*, *Insertion Sort*, *Merge Sort*, and *Quick Sort*.

Linear Search

Linear search scan an array sequentially from the very beginning to check whether the key exists, as shown in Alg. 1.

Algorithm 1: LinearSearch($A[\cdot]$, x)

```
Input  : An array $A[1, \cdots, n]$ of $n$ elements, an integer key $x$
Output: First index of key $x$ in $A$, $-1$ if not found
1 $index \leftarrow -1$;
2 for $i \leftarrow 0$ to $n-1$ do
3     if $A[i] = x$ then
4         $index \leftarrow i$;
5 return $index$
```

1. **Best Case**: $\Omega(1)$.

   Appears when the key exists in the first slot of the array.

   Example: $A = [1, 2, 7, 3, 6, 0, 9]$, $x = 1$.

2. **Worst Case**: $O(n)$.

   Appears when the key does not exist in the array (or as the last item).

   Example: $A = [3, 1, 0, 5, 4, 7, 2]$, $x = 6$.

3. **Average Case**: $O(n)$.

   Assume the probability that x appears at $A[i]$ is equal for all i (Note that $i = n$ means x is not found). The expected number of comparisons should be:

$$\begin{aligned} E[\text{total comparison}] &= \sum_{i=0}^{n} Pr(x \text{ appears at } A[i]) \cdot (\text{no. of comparisons in this case}) \\ &= \sum_{i=0}^{n} \frac{i}{n+1} = \frac{n}{2}. \end{aligned}$$

4. **Stable**: $-$ (No change.)

5. **In-Place**: ✓ (The space used remains as a constant number.)

Binary Search

Finding a key in a sorted array is much easier. Compare the middle of the array with x, then go left or right half of this array according to the comparison result. There are two ways to implement Binary search, iteratively and recursively. The details are shown in Alg. 2 and Alg. 3 respectively.

Algorithm 2: BinarySearchIterative($A[\cdot]$, x)

Input : A sorted array $A[1, \cdots, n]$ of n elements, an integer key x
Output: First index of key x in A, -1 if not found

```
1  $low \leftarrow 0$; $high \leftarrow n - 1$; $index \leftarrow -1$;
2  while $low \leq high$ do
3      $mid \leftarrow low + ((high - low)/2)$;
4      if $A[mid] > x$ then
5          $high \leftarrow mid - 1$;
6      else if $A[mid] < x$ then
7          $low \leftarrow mid + 1$;
8      else
9          $index \leftarrow mid$;
10 return $index$;
```

Algorithm 3: BinarySearchRecursive($A[\cdot]$, x, low, $high$)

Input : A sorted array $A[1, \cdots, n]$ of n elements, an integer key x, first index low, last index $high$
Output: First index of key x in A; -1 if not found

```
1  if $high < low$ then
2      return -1;
3  $mid \leftarrow low + ((high - low)/2)$;
4  if $A[mid] > x$ then
5      $mid \leftarrow BinarySearchRecursive(A[\cdot], x, low, mid - 1)$;
6  else if $A[mid] < x$ then
7      $mid \leftarrow BinarySearchRecursive(A[\cdot], x, mid + 1, high)$;
8  else
9      return $mid$;
```

Both iterative and recursive implementation have the same time complexity (that is the meaning of asymptotic algorithm analysis, which ignores the details of execution).

1. **Best Case**: $\Omega(1)$.

   Appears when the key exists in the middle slot of the array.

   Example: $A = [1, 2, 3, 6, 7]$, $x = 3$.

2. **Worst Case**: $O(\log n)$.

   Appears when the key does not exist in the array (or as the last or first item).

   Example: $A = [0, 1, 3, 4, 5, 7, 9]$, $x = 0$.

3. **Average Case**: $O(\log n)$.

   To simplify the calculation, let $n = 2^k - 1$ so that we can have $k = \log n$. The average number of iterations for successful find is shown below,

$$\begin{aligned}
E[\text{total comparison}] &= \sum_{i=1}^{n} Pr(x \text{ appears at } A[i]) \cdot (\text{no. of comparisons in this case}) \\
&= \frac{1}{n}\sum_{i=1}^{\log n}(\text{no. of iterations in case } i) \cdot (\text{no. of nodes in case } i) \\
&= \frac{1}{n}\sum_{i=1}^{\log n} i \times 2^{i-1} \\
&= \frac{1}{n} \times (\log n 2^{\log n} - (2^1 + 2^2 + 2^3 + \cdots + 2^{\log n - 1}) - 1) \\
&= \frac{1}{n} \times (n \log n - n + 1) \\
&= \log n - 1 + \frac{1}{n}.
\end{aligned}$$

Then it is obvious to see that the time complexity should be $O(\log n)$.

4. **Stable**: $-$ (No change.)

5. **In-Place**:

   Iterative way: ✓. The space used remains as a constant number.

   Recursive way: $\times$. For each recursive call of original function, the current invocation will be left on the stack. Thus we can see that stack usage will consume $S(n) = O(\log n)$.

Selection Sort

Array is divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

Algorithm 4: SelectionSort($A[\cdot]$)

```
Input  : An array $A[1, \cdots, n]$ of $n$ elements.
Output: $A[1, \cdots, n]$ in nondecreasing order.
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow i + 1$ to $n - 1$ do
        if $A[i] > A[j]$ then
            swap $A[i]$ and $A[j]$;
```

1. **Best Case**, **Average Case**, **Worst Case**: $\Theta(n^2)$.

   Whatever the input array is, Selection Sort will always go through the whole array.

$$\text{total comparisons} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

2. **Stable**: $\times$ (The selection sort will change the order in some cases.)

   Example: $A = [5, 8, 5, 2, 9]$, when we go through the whole array, we will interchange the position of the first 5 and the 2. In this way the order between the two 5s will be changed.

3. **In-Place**: ✓ (The space used remains as a constant number.)

Bubble Sort

Like a bubble underwater, bubble sort compares each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them. It is simple and well-known but also inefficient.

Algorithm 5: BubbleSort($A[\cdot]$)

```
input  : An array $A[1, \cdots, n]$ of $n$ elements.
output: $A[1, \cdots, n]$ in nondecreasing order.
1 for $i \leftarrow n-2$ downto 0 do
2     for $j \leftarrow 0$ to $i$ do
3         if $A[j] > A[j+1]$ then
4             interchange $A[j]$ and $A[j-1]$;
```

Algorithm 6: OptimizedBubbleSort($A[\cdot]$)

```
input  : An array $A[1, \cdots, n]$ of $n$ elements.
output: $A[1, \cdots, n]$ in nondecreasing order.
1 $i \leftarrow 1$; $sorted \leftarrow false$;
2 while $i \leq n-1$ and not $sorted$ do
3     $sorted \leftarrow true$;
4     for $j \leftarrow n$ downto $i+1$ do
5         if $A[j] < A[j-1]$ then
6             interchange $A[j]$ and $A[j-1]$;
7             $sorted \leftarrow false$;
8     $i \leftarrow i+1$;
```

1. **Best Case**, **Average Case**, **Worst Case**: $\Theta(n^2)$.

   The bubble sort will always go through the whole array. Notice that even the original array is already sorted, the bubble sort will also go through the whole process. Thus,

$$\text{total comparisons} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

2. **Stable**: ✓ (The bubble sort only interchange the position when the two elements are unequal.)

3. **In-Place**: ✓ (The space used remains as a constant number.)

4. **Optimized BubbleSort**: In this version, the best case happens when the input is a sorted array, and the time complexity is $O(n)$.

Insertion Sort

Array is divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains first element of the array and unsorted one contains the rest. At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one. When unsorted part becomes empty, algorithm stops.

```
Algorithm 7: InsertionSort($A[\cdot]$)
  input  : An array $A[1, \cdots, n]$ of $n$ elements.
  output: $A[1, \cdots, n]$ in nondecreasing order.
1 for $j = 1$ to $n - 1$ do
2     $key \leftarrow A[j]$;
3     $i \leftarrow j - 1$;
4     while $i \geq 0$ and $A[i] > key$ do
5         $A[i+1] \leftarrow A[i]$;
6         $i \leftarrow i - 1$;
7     $A[i+1] = key$;
```

1. **Best Case**: $\Omega(n)$.
   The best case happens when the array is already sorted. So for each element in the array, it enters the loop and exists at once. The total amount of comparison will be n.

2. **Worst Case**: $O(n^2)$.
   The worst case happens when the array is reverse ordered. So for each element in the array, it will always be moved to the top of the array. Thus the total amount of comparison will be

$$\text{total comparison} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

3. **Average Case**: $O(n^2)$.
   For the average case, we first can hold two basic assumptions.

   - $A[1, \cdots, n]$ contains the numbers 0 through $n - 1$.
   - All $n!$ permutation are equally likely.

   Suppose $A[i]$ should be inserted at position j ($0 \leq j \leq i$). When $j = 0$, we need i comparisons to insert $A[i]$. Otherwise, we need $i - j + 1$ comparisons. Notice that when $j = 1$ we still need i comparisons to determine its proper position. Since any integer in $[0, i]$ is equally likely to be taken by j, i.e., we can have

$$P(j = 0) = P(j = 1) = \cdots = P(j = i) = \frac{1}{i+1}.$$

   Then the expectation number of comparisons for inserting elements $A[i]$ in its proper position, can be written as

$$\frac{i}{i+1} + \sum_{j=1}^{i} \frac{i-j+1}{i+1} = \frac{i}{i+1} + \sum_{j=1}^{i} \frac{j}{i+1} = \frac{i}{2} - \frac{1}{i+1} + 1.$$

   The average number of comparisons performed by insertion sort is

$$\sum_{i=1}^{n-1}(\frac{i}{2} - \frac{1}{i+1} + 1) = \frac{n^2}{4} + \frac{3n}{4} - \sum_{i=0}^{n-1} \frac{1}{i+1}.$$

   It is now clear for us to see that the time complexity of insertion sort in average case should be $\Theta(n^2)$.

4. **Stable**: ✓ (The insertion sort only change the positions when the two elements are unequal.)
5. **In-Place**: ✓ (The space used remains as a constant number.)

Merge Sort

By applying divide-and-conquer concept, the general sorting question is divided into several small subproblems. We recursively sort the small array and finally merge them together.

Algorithm 8: MergeSort($A[\cdot], left, right$)

```
input : An array $A[1, \cdots, n]$ of $n$ elements, the first index $left$ and the last index $right$.
output: $A[1, \cdots, n]$ in nondecreasing order.
1 if $left \geq right$ then
2     return;
3 $mid \leftarrow (left + right)/2$;
4 MergeSort($A, left, mid$);
5 MergeSort($A, mid + 1, right$);
6 Merge($A, left, mid, right$);
```

1. **Best Case**, **Average Case**, **Worst Case**: $\Theta(n \log n)$.

   The merge sort always go through a same process. We can easily see that the depth of the recursion is $\log n$. The target array is separated in half until the length becomes 1. We consider the merge of n arrays whose length is 1, the time cost should be $\Theta(n)$. The merge of $n/2$ arrays whose length is 2, the time cost is $\Theta(n)$. Then we have,

$$T(n) = \sum_{i=1}^{\log n} \frac{n}{2^i} \times 2^i = \Theta(n \log n).$$

2. **Stable**: ✓
3. **In-Place**: ×

   Space complexity should be $O(n)$. Since we always use extra arrays to do the merge. As for the stack usage of recursive calls, if you draw the space tree out, it will seem as though the space complexity is $O(n \log n)$. However, as the code is a Depth First code, you will always only be expanding along one branch of the tree. As a result, when we plus the additional array used to merge, the total space complexity is

$$S(n) = O(n) + O(\log n) = O(n).$$

Quick Sort

Quick Sort is also a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many way to choose the pivot. The performance may differ when different ways of choosing pivots are applied.

Algorithm 9: QuickSort($A[\cdot], left, right$)

```
input  : An array $A[1, \cdots, n]$ of $n$ elements, the first index $left$ and the last index
         $right$.
output: $A[1, \cdots, n]$ in nondecreasing order.
1  if $left \geq right$ then
2      return;
3  Choose one element as $pivot$;
4  $pivotat \leftarrow$ the index of $pivot$ in the array;
5  for $i \leftarrow 0$ to $n-1$ do
6      if $A[i] < pivot$ then
7          Put $A[i]$ at the left side of $pivot$;
8      else
9          Put $A[i]$ at the right side of $pivot$;
10 QuickSort($A, left, pivotat - 1$);
11 QuickSort($A, pivotat + 1, right$);
```

1. **Best Case**: $\Omega(n \log n)$.
   The best case appears when every time the pivot separates the array into two equally-sized arrays. Under this circumstance, the Quick Sort will separate the array totally $\log n$ times. The time complexity can be calculated as

$$T(n) = \sum_{i=1}^{\log n} \frac{n}{2^i} \times 2^i = n \log n.$$

2. **Worst Case**: $O(n^2)$.
   The worst case happens when every time the pivot always separates the array into 1 and $n-1$ sized arrays. In this situation, the divide and conquer concepts fails to perform well. So generally the time complexity will be $O(n^2)$. It will go through something like the double loops. We have the following recursive equation,

$$T(n) = n + T(n-1) = n + (n-1) + (n-2) + ... + 1 = n(n+1)/2.$$

3. **Average Case**: $O(n \log n)$.
   For random pivots, it will end up near the middle on average. The precise recurrence formula for the number of comparisons is

$$T(n) = cn + \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-1-i)], T(0) = T(1) = c.$$

   We can solve the equation in the following way,

$$T(n) = cn + \frac{2}{n} \sum_{i=0}^{n-1} T(i),$$

$$nT(n) = cn^2 + 2 \sum_{i=0}^{n-1} T(i).$$

By subtract the same for $n-1$, we can have

$$\begin{aligned} nT(n) &= c(2n-1) + (n+1)T(n-1), \\ \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{c(2n-1)}{n(n+1)} \\ &= \frac{T(n-2)}{n-1} + \frac{c(2n-1)}{n(n+1)} + \frac{c(2n-3)}{n(n-1)} \\ &= ... \\ &= \frac{T(2)}{3} + c\sum_{k=3}^{n} \frac{2k-1}{k(k+1)}. \end{aligned}$$

Then we can take some approximation, and get

$$\frac{T(n)}{n+1} \approx c\sum_{k=3}^{n} \frac{2k-1}{k(k+1)} \approx 2c\int_0^{n-1} \frac{1}{k+1} = 2c\ln n.$$

Then it is clear to see that the time complexity should be

$$T(n) = O(n\log n).$$

4. **Stable**: $\times$.

5. **In-Place**: $\times$ (weakly).

   Quick sort normally uses $O(\log n)$ extra memory, stored on the stack caused be the recursive calls. It's not $O(n)$, because the binary tree is never explicit in memory, we just do a post-order traversal of it (only one path in the tree is ever stored at a given time). Since the in-place partition is originally designed to perform a "in-place" algorithm, however, the extra stack usage is unavoidable in recursive functions, we may consider the Quick Sort as a "weakly" in-place algorithm.

Summary and Comparison

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Stable**	**In-Place**	$S(n)$
Linear Search	$\Omega(1)$	$O(n)$	$O(n)$	–	✓	$O(1)$
Binary Search	$\Omega(1)$	$O(\log n)$	$O(\log n)$	–	$\times$[*]	$O(\log n)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\times$	✓	$O(1)$
Bubble Sort	$\Theta(n^2)$[†]	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓	$O(1)$
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	✓	$O(1)$
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	✓	$\times$	$O(n)$
Quick Sort	$\Omega(n\log n)$	$O(n\log n)$	$O(n^2)$	$\times$	$\times$	$O(\log n)$

[*]When Binary Sort is implemented in iterative way, it is in-place.
[†]With optimized implementation, the best case complexity for bubble sort can be $O(n)$.