Sorting, Searching, & the complexities

Pengurutan, pencarian, dan kompleksitas

Ardiawan Bagus Harisa

Sorting

Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Bubble sort is used if:

- Complexity does not matter.
- Short and simple code is preferred.

Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Code

```
void bubbleSort(int array[], int size) {
  // loop to access each array element
  for (int step = 0; step < size; ++step) {
    // loop to compare array elements
    for (int i = 0; i < size - step; ++i) {
      // compare two adjacent elements
      if (array[i] > array[i + 1]) {
      // swapping elements if elements are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}
```

Selection Sort

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

The selection sort is used when:

- A small list is to be sorted.
- Cost of swapping does not matter.
- Checking of all the elements is compulsory.
- Cost of writing to a memory matters like in flash memory (number of writes/swaps is O(n) as compared to O(n2) of bubble sort).

Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
    swap minimum with first unsorted position
end selectionSort
```

Code

```
void selectionSort(int array[], int size) {
    for (int step = 0; step < size - 1; step++) {
        int min_idx = step;
        for (int i = step + 1; i < size; i++) {
            // To sort in descending order, change > to < in this line.
            // Select the minimum element in each loop.
            if (array[i] < array[min_idx])
                min_idx = i;
        }
        // put min at the correct position
        swap(&array[min_idx], &array[step]);
    }
}
```

Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

The insertion sort is used when:

- The array is has a small number of elements.
- There are only a few elements left to be sorted.

Algorithm

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
      for j <- lastSortedIndex down to 0
        if current element j > X
          move sorted element to the right by 1
      break loop and insert X here
end insertionSort
```

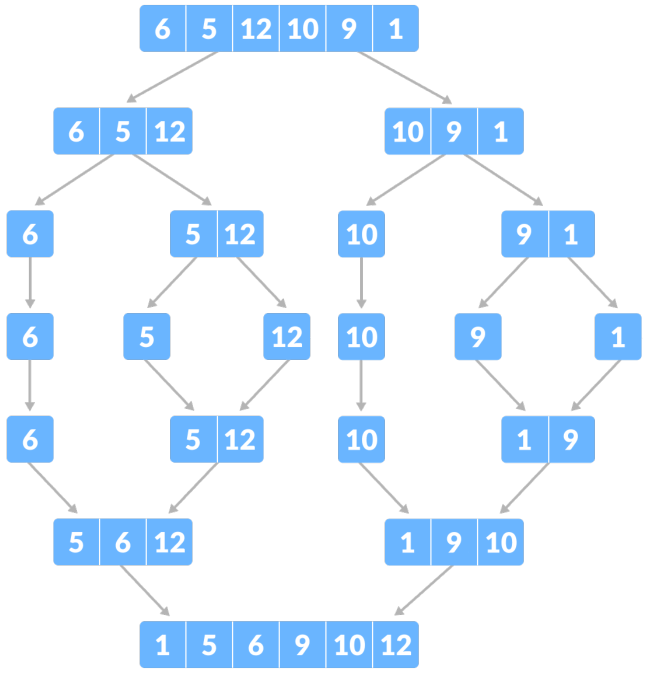
Code

```
void insertionSort(int array[], int size) {
    for (int step = 1; step < size; step++) {
        int key = array[step];
        int j = step - 1;
        // Compare key with each element on the left of it until an element smaller than it is found.
        // For descending order, change key<array[j] to key>array[j].
        while (key < array[j] && j >= 0) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = key;
    }
}
```

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

Merge Sort Applications:

- Inversion count problem
- External sorting
- E-commerce applications

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

$$\begin{tikzcd}
 & & & A[0 .. 3] \arrow[dll] \arrow[drr] & & & \\
 & A[0 .. 1] \arrow[dl] \arrow[dr] & & & & A[2 .. 3] \arrow[dl] \arrow[dr] & \\
A[0 .. 0]^* \arrow[dr] & & A[1 .. 1]^* \arrow[dl] & & A[2 .. 2]^* \arrow[dr] & & A[3 .. 3]^* \arrow[dl] \\
 & A[0 .. 1]^* \arrow[drr] & & & & A[0 .. 1]^* \arrow[dll] & \\
 & & & A[0 .. 1]^* & & &
\end{tikzcd}$$

Algorithm

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)

//The algorithm maintains three pointers, one for each of the two arrays
//and one for maintaining the current index of the final sorted array.

Have we reached the end of any of the arrays?
    No:
        Compare current elements of both arrays
        Copy smaller element into sorted array
        Move pointer of element containing smaller element
    Yes:
        Copy all remaining elements of non-empty array
```

Code

```
// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // m is the point where the array is divided into two subarrays
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the sorted subarrays
        merge(arr, l, m, r);
    }
}
```

```c
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {
    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1], M[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];
    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0; j = 0; k = p;
    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
         if (L[i] <= M[j]) {
             arr[k] = L[i];
             i++;
         } else {
             arr[k] = M[j];
             j++;
         }
         k++;
    }
    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
     }
}
```

Quick Sort

Quicksort is a sorting algorithm based on the divide and conquer approach where:

- An array is divided into subarrays by selecting a pivot element (element selected from the array).
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

2. Rearrange the Array

- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

3. Divide Subarrays

- Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.

Quicksort algorithm is used when:

- The programming language is good for recursion.
- Time complexity matters.
- Space complexity matters.

Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
    if element[i] < pivotElement
      swap element[i] and element[storeIndex]
      storeIndex++
    swap pivotElement and element[storeIndex+1]
  return storeIndex + 1
```

```c
// function to rearrange array (find the partition point)
int partition(int array[], int low, int high) {
    // select the rightmost element as pivot
    int pivot = array[high];
    // pointer for greater element
    int i = (low - 1);
    // traverse each element of the array
    // compare them with the pivot
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            // if element smaller than pivot is found
            // swap it with the greater element pointed by i
            i++;
            // swap element at i with element at j
            swap(&array[i], &array[j]);
        }
    }
    // swap pivot with the greater element at i
    swap(&array[i + 1], &array[high]);
    // return the partition point
    return (i + 1);
}
```

Code

```
void quickSort(int array[], int low, int high) {
  if (low < high) {
    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on righ of pivot
    int pi = partition(array, low, high);
  // recursive call on the left of pivot
  quickSort(array, low, pi - 1);
  // recursive call on the right of pivot
  quickSort(array, pi + 1, high);
  }
}
```

Searching

Linear Search

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

When to use?

- For searching operations in smaller arrays (<100 items).

Code

```
int search(int array[], int n, int x) {
   // Going through array sequencially
   for (int i = 0; i < n; i++)
      if (array[i] == x)
         return i;
   return -1; // If not found
}
```

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Algorithm can be implemented in two ways which are discussed below.

- Iterative Method
- Recursive Method

The recursive method follows the divide and conquer approach.

- In libraries of Java, .Net, C++ STL
- While debugging, the binary search is used to pinpoint the place where the error happens.

Algorithms

```
Iteration method
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid]) // x is on the right side
        low = mid + 1
    else // x is on the left side
        high = mid - 1
```

```
Recursive method
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid] // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else // x is on the left side
            return binarySearch(arr, x, low, mid - 1)
```

Code

```
int binarySearch(int array[], int x, int low, int high) {
    if (high >= low) {
        int mid = low + (high - low) / 2;
        // If found at mid, then return it
        if (array[mid] == x)
            return mid;
        // Search the left half
        if (array[mid] > x)
            return binarySearch(array, x, low, mid - 1);
        // Search the right half
        return binarySearch(array, x, mid + 1, high);
    }
    return -1;
}
```

Algorithm’s Complexity

Complexity of Sorting Algorithms

The efficiency of any sorting algorithm is determined by the time complexity and space complexity of the algorithm.

1. Time Complexity: Time complexity refers to the time taken by an algorithm to complete its execution with respect to the size of the input. It can be represented in different forms:

- Big-O notation (O)
- Omega notation (Ω)
- Theta notation (Θ)

2. Space Complexity: Space complexity refers to the total amount of memory used by the algorithm for a complete execution. It includes both the auxiliary memory and the input.

The auxiliary memory is the additional space occupied by the algorithm apart from the input data. Usually, auxiliary memory is considered for calculating the space complexity of an algorithm.

Asymptotic Analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Complexity comparisons

Sorting Algorithm	Time Complexity - Best	Time Complexity - Worst	Time Complexity - Average	Space Complexity
Bubble Sort	n	n^2	n^2	1
Selection Sort	n^2	n^2	n^2	1
Insertion Sort	n	n^2	n^2	1
Merge Sort	nlog n	nlog n	nlog n	n
Quicksort	nlog n	n^2	nlog n	log n
Counting Sort	n+k	n+k	n+k	max
Radix Sort	n+k	n+k	n+k	max
Bucket Sort	n+k	n^2	n	n+k
Heap Sort	nlog n	nlog n	nlog n	1
Shell Sort	nlog n	n^2	nlog n	1

References

Sorting Algorithm	**Stability**
Bubble Sort	Yes
Selection Sort	No
Insertion Sort	Yes
Merge Sort	Yes
Quicksort	No
Counting Sort	Yes
Radix Sort	Yes
Bucket Sort	Yes
Heap Sort	No
Shell Sort	No

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0
such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

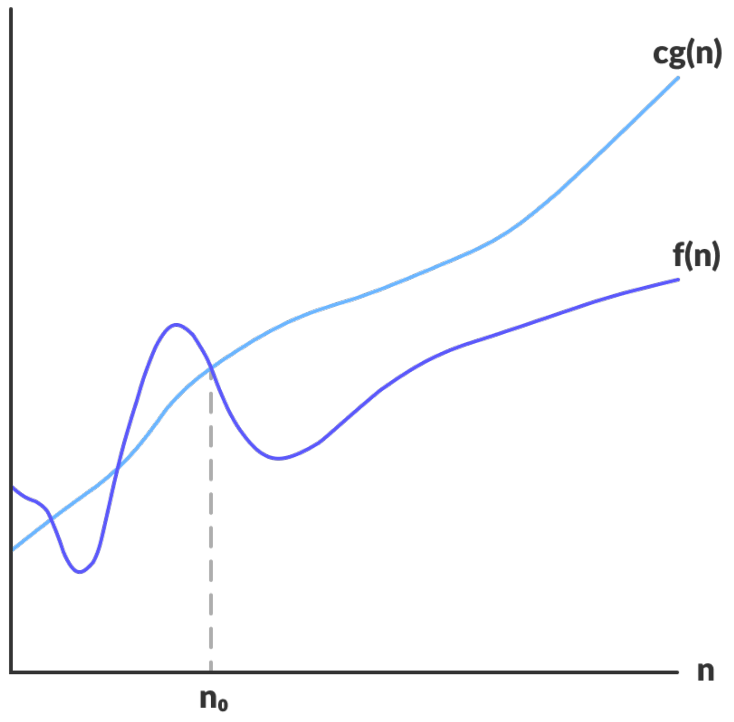
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Big-O Notation (O-notation)

Big-O gives the upper bound of a function



f(n) = O(g(n))

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

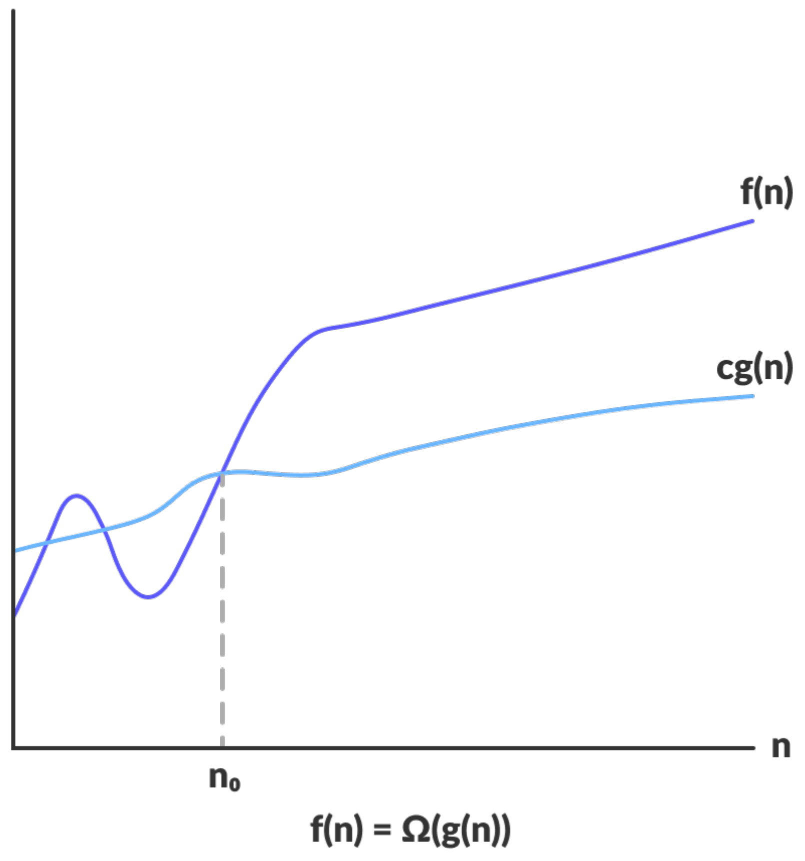
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and $n0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

Omega Notation (Ω-notation)

Omega gives the lower bound of a function



Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

For a function g(n), Θ(g(n)) is given by the relation:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

References

Programmiz.com

w3schools.com