Notes for ECE 36800 - Data Structures

Zeke Ulrich

February 4, 2025

Contents

Course Description

Provides insight into the use of data structures. Topics include stacks, queues and lists, trees, graphs, sorting, searching, and hashing.

Introduction

Algorithms

An algorithm is simply a method to solve a class of problems. There are algorithms to make toast, to solve a Rubik's cube, and to plot the optimal path for a mailmain. The two most important qualities of any algorithm are effectiveness and secondly efficiency. Our solution must work, preferably in as little time as possible. A working algorithm is no good if the heat death of the universe occurs before it finds a solution. We describe the cost of an algorithm using Big O notation. As an example, consider listing 1 If each operation has cost C_i, then

```
int total = 0; // C_1
for (int i = 0; i++; i <= n){ // C_2
    total = total + i; // C_3
}
return total; // C_4
```

Figure 1: Sum of first n numbers

the total cost of the program is

$$C_1 + C_2(n+1) + C_3 n + C_4 = n(C_2 + C_3) + (C_1 + C_4). \qquad (1)$$

Big O notation considers only the largest power of n, so the Big O complexity for this algorithm would be $O(n)$.

Data Structures

A data structure is an organization of information for ease of manipulation. For example, a dictionary, a checkout line, and an org chart.

Sorting

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Listing 1: Bubble Sort

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int swapped = 0;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1;
            }
        }
        if (swapped == 0)
            break;
    }
}
```

Insertion Sort

Insertion Sort builds the sorted array one item at a time. It picks an element and places it into its correct position in the sorted part of the array.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Listing 2: Insertion Sort

```
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Selection Sort

Selection Sort divides the array into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region.

Time Complexity:

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Listing 3: Selection Sort

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Shell Sort

Shell Sort is an optimization over Insertion Sort. It first sorts elements that are far apart and then progressively reduces the gap between elements to be sorted.

Time Complexity:

- Best Case: $O(n \log n)$
- Average Case: depends on the gap sequence, commonly $O(n^{3/2})$ or $O(n \log^2 n)$
- Worst Case: $O(n^2)$

Listing 4: Shell Sort

```
void shell_sort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

A sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Insertion sort, merge sort, and bubble sort are all stable sorting algorithms.

Asymptotic Notation

Big O

Big O notation describes the upper bound of an algorithm's growth rate. It provides an asymptotic measure of the time complexity or space complexity of an algorithm as a function of the input size. Informally, big O notation just picks out the highest order term from your runtime expression. Formally, Big O is defined as follows:

$$f(n) \in O(g(n)) \iff \exists\, c > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, often representing the dominant term of $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big O notation focuses on the dominant term of $g(n)$, ignoring lower-order terms and constant factors, to provide a simplified representation of the algorithm's efficiency.

Big Ω

Big Ω notation describes the lower bound of an algorithm's growth rate. It provides an asymptotic measure of the minimum time complexity or space complexity of an algorithm as a function of the input size. While Big O focuses on the upper bound, Big Ω focuses on the lower bound, ensuring that the algorithm's growth rate is at least as fast as a specified function, up to constant factors. Formally, Big Ω is defined as follows:

$$f(n) \in \Omega(g(n)) \iff \exists\, c > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq c \cdot g(n) \leq f(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the lower bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ from below.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big Ω notation is used to describe the best-case or minimum growth rate of an algorithm. For example, if an algorithm's runtime is $\Omega(n \log n)$, it means that the algorithm cannot perform better than $n \log n$ in the best case, up to constant factors. This notation is particularly useful when proving lower bounds on the complexity of problems or algorithms.

Big Θ

Big Θ notation describes both the upper and lower bounds of an algorithm's growth rate. It provides a tight asymptotic bound on the time complexity or space complexity of an algorithm as a function of the input size. Unlike Big O, which only provides an upper bound, Big Θ ensures that the growth rate of the algorithm is bounded both above and below by the same function, up to constant factors. Formally, Big Θ is defined as follows:

$$f(n) \in \Theta(g(n)) \iff \exists\, c_1, c_2 > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the tight bound on $f(n)$.
- $c_1, c_2 > 0$: Constants that scale $g(n)$ to bound $f(n)$ from below and above.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big Θ notation is used when the growth rate of an algorithm is known to be tightly constrained by a specific function, providing a precise characterization of its efficiency. For example, if an algorithm's runtime is both $O(n^2)$ and $\Omega(n^2)$, then its runtime is $\Theta(n^2)$.

Little o

Little o notation describes an upper bound that is not tight. It provides a stricter asymptotic measure of the growth rate of an algorithm compared to Big O notation. While Big O allows for the possibility that the growth rates are the same up to constant factors, Little o ensures that the growth rate of the function is strictly less than the comparison function. Formally, Little o is defined as follows:

$$f(n) \in o(g(n)) \iff \forall\, c > 0,\ \exists\, n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) < c \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.

- $g(n)$: The comparison function, representing the upper bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ strictly from above.
- n_0: A constant representing the threshold beyond which the inequality holds.

Little o notation is used to describe situations where the growth rate of an algorithm is strictly smaller than the comparison function. For example, if $f(n) = 2n + 3$, then $f(n) \in o(n^2)$, because $2n + 3$ grows strictly slower than n^2 as n becomes large. This notation is particularly useful when emphasizing that an algorithm's growth rate is asymptotically insignificant compared to another function.

Little ω

Little ω notation describes a lower bound that is not tight. It provides a stricter asymptotic measure of the growth rate of an algorithm compared to Big Ω notation. While Big Ω allows for the possibility that the growth rates are the same up to constant factors, Little ω ensures that the growth rate of the function is strictly greater than the comparison function. Formally, Little ω is defined as follows:

$$f(n) \in \omega(g(n)) \iff \forall\, c > 0,\ \exists\, n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq c \cdot g(n) < f(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the lower bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ strictly from below.
- n_0: A constant representing the threshold beyond which the inequality holds.

Little ω notation is used to describe situations where the growth rate of an algorithm is strictly larger than the comparison function. For example, if $f(n) = n^2$, then $f(n) \in \omega(n)$, because n^2 grows strictly faster than n as n becomes large. This notation is particularly useful when emphasizing that an algorithm's growth rate is asymptotically dominant compared to another function.

Stack

A stack is last in, first out. The most recent entry is "pushed" to the top of the stack, and to retrieve it we "pop" it off the stack. A natural choice of data structure for a stack is a linked list.

Queue

A queue is first in, first out. Entries are "enqueued" to insert them into the queue and "dequeued" to take them off the queue. Like a stack, queues can be implemented using linked lists.

Priority Queue

Each element in a priority queue has an associated priority. In a priority queue, elements with high priority are served before elements with low priority. An example of this is airplane loading.