Lebanese American University
Department of Computer Science and Mathematics
CSC310: Algorithms and Data Structures
Fall 2020

Instructor: Faisal N. Abu-Khzam

1 Introduction

We review some prerequisite material and cover basic definitions needed for algorithmic analysis.

1.1 Algorithmic Analysis

The running time of an algorithm is a function of the input size or, sometimes, the number of key data elements in the input.

We are mostly concerned with the behavior of the algorithm as the input size increases. The main question is: how well does the running time scale as a function of the input size?

If the input size is n, then a running time that is bounded above by $100n$ is assumed to be better than one that is bounded above by n^2 or even $n \log n$.

An important remark here is: what if the upper bound is loose? In such cases one might be more interested in the average-case behavior.

In this course, when we analyze the running time of an algorithm, we mostly consider worst-case behavior.

1.2 Asymptotic Notation

We consider functions of the natural numbers.

For $f, g : \mathcal{N} \rightarrow \mathcal{R}$ we say:

- $f(n) \in \mathcal{O}(g(n))$ iff: $\exists c, n_0$ such that $f(n) \leq cg(n) \forall n \geq n_0$

  Alternatively:

  $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)}$ is finite.

- $f(n) \in \Omega(g(n))$ iff: $\exists c, n_0$ such that $f(n) \geq cg(n) \forall n \geq n_0$

  Alternatively:

  $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$

- $f(n) \in \Theta(g(n))$ iff: $\exists c_1, c_2, n_0$ s. t. $c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0$

  Alternatively:

  $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

- $f(n) \in o(g(n))$ iff: $\exists c, n_0$ such that $f(n) < cg(n) \forall n \geq n_0$.

  Alternatively:

  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

- $f(n) \in \omega(g(n))$ iff: $\exists c, n_0$ such that $f(n) > cg(n) \forall n \geq n_0$

  Alternatively:

  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$

Example 1. *The following examples are straightforward.*

1. $\frac{n}{10000} \in \Theta(n)$
2. $\log^{10} n \in o(n^{0.0001}) \in o(\sqrt{n}) \in o(n)$
3. $\log^{10} n \in \mathcal{O}(n^{0.0001}) \in \mathcal{O}(\sqrt{n}) \in \mathcal{O}(n)$
4. $225 \in \mathcal{O}(1)$
5. $n \log n \in \Omega(n)$

Example 2. *The function* $f(n) = 980n^{3.102} + n^3 logn^{3.102}$ *is in:*

1. $\mathcal{O}(n^4)$*?*
2. $\Theta(n^3 logn)$*?*
3. $\Theta(n^{3.102} logn)$*?*
4. $\Omega(n^{3.102} + n^3 logn)$*?*
5. $\Omega(\sqrt{n})$*?*
6. $o(n^{3.2})$*?*

Example 3. *Give a* **tight** *worst-case asymptotic running-time of the most time-efficient algorithm for each of the following:*

1. *Sequential search through an array.*
2. *Binary search in a sorted array.*
3. *Inserting an element after position i in a sorted array (i is arbitrary).*
4. *Inserting an element after a given node (with a given reference) in a sorted linked list.*
5. *Inserting an element before a given node in a singly-linked list.*
6. *Deleting the last node of a singly-linked list.*

1.3 Bubble Sort

Let us analyze the running time of Bubble Sort. Here is the algorithm:

```
void BubbleSort(int[] A)
{
    for(int i=A.length-1; i>0; i--)
        for(int j=0; j<i; j++)
            if(A[j]>A[j+1])
                Swap(A,j,j+1));
}
```

Since the number of steps performed by a single comparison is constant, we assume the running time is proportional to the the number of comparisons performed on the third line (above).

To analyze the running time we therefore count the total number of comparisons. We may assume this is equivalent to counting the number of swaps in a worst-case scenario.

In the worst case, the function swap is called on each iteration of the j-loop. This happens when the array is sorted in reverse.

The total number of iterations of the j-loop is:
$(n-1)+(n-2)+\cdot+2+1=\frac{n(n-1)}{2}$

Therefore the running time $T(n)$ is in $\mathcal{O}(n^2)$.

In the worst-case $T(n)$ is in $\Theta(n^2)$.

However, in the above Bubble Sort algorithm, the best-case and worst-case exhibit an equal number of comparisons (they differ by the number of swaps). Therefore we can safely say (in this case) that the running time is in $\Theta(n^2)$.

1.4 Sequential Search

A simple, familiar, method/function you wrote in CP1 is sequential search:

```
int sSearch(int[] A, int x)
{
    for(int i = 0; i < A.length; i++)
        if(A[i] == x)
            return i;
    return -1;
}
```

What is a:

best-case scenario? What is the corresponding running time?

worst-case scenario? What is the corresponding running time?

1.5 Find Min

One of the simplest functions you wrote in CP1 was findMin: find the location of a minimum element in an array if numbers. Here is a findMin function:

```
int findMin(int[] A)
{
    int ind = 0;
    for(int i = 1; i < A.length; i++)
        if(A[i] < A[ind])
            ind = i;
    return ind;
}
```

What is a:

best-case scenario? What is the corresponding running time?

worst-case scenario? What is the corresponding running time?

1.6 The Fibanacci Numbers

Consider the following function to compute the nth Fibanacci number.

```
int Fib(int n)
{
   if((n == 1) || (n == 2))
      return 1;
   return Fib(n-1) + Fib(n-2);
}
```

```
What is Fib(100)?

                          Fib(100)

=             Fib(99)          +            Fib(98)

=    Fib(98)    +    Fib(97)       Fib(97)   +   Fib(96)

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
```

The running time, $T(n)$, is proportional to the number of leaves in the above tree. In general, therefore: $T(n) = T(n-1) + T(n-2)$

How do we solve the above equation?

First observe that the number of leaves is between $2^{n/2}$ and 2^n. Why?!

Therefore we know $T(n) = x^n$ for some $x \in (\sqrt{2}, 2)$. Then we simply solve the following equation for x:

$$x^n = x^{n-1} + x^{n-2}$$

Or

$$x^2 = x + 1$$

Get

$$T(n) = (\tfrac{1+\sqrt{5}}{2})^n \in \mathcal{O}(1.62^n)$$

1.7 Homework

Write a pseudocode or code for the following sorting algorithms and analyze it:
Insertion Sort
Selection Sort

1.8 Binary Search Trees

In the lab!

1.9 Merge Sort

Next time!