CSc 3102: Analysis of Algorithms

Supplementary Notes

- Mathematics Review
- Asymptotic Behavior of Functions
- The Computational Complexity of Algorithms

The efficient use of both time and memory is important. How do you compare the time efficiency of two algorithms that solve the same problem? One approach might be to write programs but there are some drawbacks:

1. This is dependent on how the program is coded.
2. The type of computer used influences the runtime.
3. The kind of data the program uses also influences the analysis.

These difficulties are resolved by employing mathematical techniques that analyze algorithms independent of specific implementations, computers, or data. This is done by counting the number of significant operations in a particular solution as a function of the input size.

1 Mathematics Review

1.1 Arithmetic and Geometric Series

The arithmetic series often comes up in the analysis of basic algorithm. The arithmetic series is defined as below:

$$S_n = \sum_{i=1}^{n} [a + (i-1)d]$$
$$S_n = a + (a+d) + (a+2d) + ... + (a+(n-1)d)$$
$$S_n = \frac{n[2a+(n-1)d]}{2}$$

a is the initial value, d is the common difference between adjacent terms and n is the number of terms in the series. Here is an example:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n = \frac{n\left[2 + (n-1)\right]}{2} = \frac{n(n+1)}{2}$$

The geometric series also comes up in the analysis of some algorithms and data structures. The geometric series is defined as below:

$$\sum_{i=1}^{n} ar^{i-1} = a + ar + ar^2 + \cdots + ar^{n-1} = a \bullet \frac{r^n - 1}{r - 1},\ where\ r \neq 1$$

Each term, except the first, is obtained by multiplying its predecessor by a fixed value called the common *ratio*.

1.2 Properties of Logarithm

The logarithm function is used extensively in the analysis of basic algorithms. It is the inverse of the exponential function. Specifically, the relationship between the two functions is defined as:

$$b^a = x \quad \leftrightarrow \quad \log_b x = a \quad where \;\; b > 0 \;\; and \;\; x > 0$$

Here are four properties of the logarithm function:

1. $\log_b xy = \log_b x + \log_b y$
2. $\log_b \frac{x}{y} = \log_b x - \log_b y$
3. $\log_b a^n = n \log_b a$
4. $\frac{\log_b x}{\log_b y} = \log_y x$

Logarithms may be used to solve exponential equations. For example, solve $n = 2^k$ for k.

$$n = 2^k \rightarrow \log_2 n = \log_2 2^k \rightarrow \log_2 n = k \log_2 2 \rightarrow k = \log_2 n$$

1.3 Laws of Limits

Suppose $\lim_{x\to c} f(x) = L$ and $\lim_{x\to c} g(x) = M$, where L and M are real numbers. Then the following apply:

1. $\lim_{x\to c}(f(x)+g(x)) = L+M$
2. $\lim_{x\to c}(f(x)-g(x)) = L-M$
3. $\lim_{x\to c}(kf(x)) = kL$
4. $\lim_{x\to c}(f(x)\bullet g(x)) = LM$
5. $\lim_{x\to c}\left(\frac{f(x)}{g(x)}\right) = \frac{L}{M}$, $M \neq 0$
6. $\lim_{x\to c}(f(x))^n = L^n$, $n > 0$
7. $\lim_{x\to c}\left(\sqrt[n]{f(x)}\right) = L^{\frac{1}{n}}$, $n > 0$

2 Asymptotic Behavior of Functions

Let f and g be functions from $\mathbb{Z}^+ \to \mathbb{R}^+$, positive real-valued functions on the domain of positive integers. The *Big* notations describes bounds that are either loose or tight.

2.1 O-notation

If $f(n) \in O(g(n))$, then $g(n)$ is said to be an asymptotic upper bound for $f(n)$. Mathematically, there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)\ \forall\, n \geq n_o$.

2.2 Ω-notation

If $f(n) \in \Omega(g(n))$, then $g(n)$ is said to be an asymptotic lower bound for $f(n)$. Mathematically, there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)\ \forall\, n \geq n_o$.

2.3 Θ-notation

If $f(n) \in \Theta(g(n))$, then $g(n)$ is said to be an asymptotic tight bound for $f(n)$. Mathematically, there are constants $c' > 0$, $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)\ \forall\, n \geq n_o$. Observe that if $f(n) \in \Theta(g(n))$, then $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

For the growth rate function of an algorithm,

1. You can ignore lower order terms. For example, if an algorithm is $O(n^3 + 4n^2 + 3n)$, it is $O(n^3)$.

2. You can ignore a multiplicative constant in the high-order term. For example, if an algorithm is $\Theta(5n^3)$, it is $\Theta(n^3)$.

3. They are additive and you can combine them. For example, if an algorithm is $\Omega(n^2) + \Omega(n)$, it is also $\Omega(n^2 + n)$, which we can write simply as $\Omega(n^2)$.

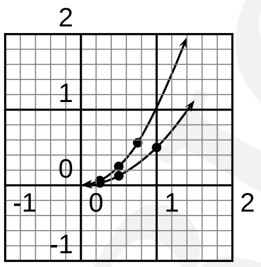

Figure 1: Graph of n^2 and $\frac{n^2}{2}$

$\frac{n^2}{2} \in O(n^2)$ since $\frac{n^2}{2} \leq cn^2$, where $n \geq n_o$. Here we take $c = 1$ and $n_o = 1$ to show that n^2 is an asymptotic upperbound of $\frac{n^2}{2}$.

2.4 *o*-notation

The o-notation provides an asymptotically stricter upper bound than the O-notation; that is, it cannot be a tight bound. Mathematically, for every constant constant $c > 0$ there is an integer constant $n_0 \geq 1$ such that $f(n) < cg(n)\ \forall\, n \geq n_o$. Also,

$$f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

2.5 ω-notation

The ω-notation provides an asymptotically stricter lower bound than the Ω-notation; that is, it cannot be a tight bound. Mathematically, for every constant $c > 0$ there is an integer constant $n_0 \geq 1$ such that $f(n) > cg(n)\ \forall\, n \geq n_o$. Also,

$$f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

3 Common Complexity Classes

Commonly Used Terminology	
Complexity Class	**Terminology**
O(1)	Constant complexity
O($\log n$)	Logarithmic complexity
O(n)	Linear Complexity
O($n \log n$)	log-linear Complexity
O(n^b)	Polynomial Complexity
O(b^n), where $b > 1$	Exponential Complexity
O($n!$)	Factorial Complexity

Table 1: Some Asymptotic Time Complexity Classes

Definition 1. An algorithm is said to be asymptotically order-optimal for a given class of problems if its order of complexity is best for that class of problems. For example, *binary search* which has a time complexity of O($\log n$) is an order-optimal comparison-based search for a random key in a sorted array.

4 Asymptotic Complexities of Some Algorithms

We now derive the asymptotic time complexities of some algorithms:

4.1 Selection Sort

```
Algorithm selectionSort(A)
{Input: An array with n elements
Output: A is sorted.}
for i <- 0 to n-2 do
   MinPosition <- i;
   for j <- i + 1 to n-1 do
      if A[j] < A[MinPosition] then
         MinPosition <- j
      endif
   temp <- A[i]
   A[i] <- A[MinPosition]
   A[MinPosition] <- temp
endfor
{A is sorted}
```

This algorithm consists of a nested loop. For every iteration of the outer loop, the inner loop iterates i times, where i is the index of the outer loop. Thus, if we count the number of comparisons made in the inner loop, the predominant operation, we get the following number of operations:

$$(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2} \in \mathrm{O}(n^2)$$

Alternatively, we can obtain the same result by counting the number of visits to elements of the array. How and why?

Problem 1. In what order does this version of selectionSort sort A (ascending or descending)? Rewrite the algorithm so that it sorts A in the opposite order.

4.2 Insertion Sort

```
Algorithm insertionSort(A)
 {Input: An array with n elements
  Output: A is sorted.}
for i <- 1 to n - 1 do
   next <-  A[i]
   j <- i
   while j > 0 AND A[j-1] > next do
      A[j] <- A[j-1]
      j <- j - 1
   endwhile
   A[j] <- next
endfor
```

Initially the array is partitioned into two regions: the sorted region containing only the first element and the unsorted region containing the rest of the elements. On each iteration of the for-loop, insertionSort moves the left-most element of the unsorted region into its correct position in sorted region and expands the size of the sorted region by 1. Therefore, after at most $n-1$ passes, all n keys will be in their final sorted order. During the i^{th} iteration of the for-loop of insertionSort, the while-loop could make no iteration if the left-most element in the unsorted region is not less the right-most element in the sorted region and as many as i iterations if it is smaller than the left-most element in the sorted region. Using these facts, convince yourself that in the best-case, the insertionSort algorithm is O(n) and in the worst-case it is O(n^2).

Problem 2. In what order does this version of insertionSort sort A (ascending or descending)? Rewrite the algorithm so that it sorts A in the opposite order.

4.3 Sequential Search

```
Algorithm equentialSearch(K,A)
{Input A an array of n elements
       K the target to search for
Output: the index of K in A or -1
        indicating K is not found}
   i <- 0
   while i<n and K <> A[i] do
      i <- i+1
   if i < n then
      location = i
   else
      location = -1
   return location;
```

In the best-case, the key is at A[0] and only three comparisons are performed. Thus the best-case performance of seqentialSearch is O(1). The worst-case performance occurs when the key is not in the array. The algorithm will access all n array elements and perform O(n) comparisons. Specifically, it performs $2n + 1$ comparisons. We will present the average-case analysis in class.

4.4 Binary Search

```
Algorithm binarySearch(K, A, L,R)
Input L, the target to search for,
      A, a sorted array of n elements
      L, the left-most index
      R, the right-most index
Output: index of the target if found or -1.
   if L > R then
      return -1
   else if K = A[(L+R)/2] then
      return (L+R)/2;
   else if K > [(L+R)/2] then
      BinarySearch(K,A,(L+R)/2+1,R)
   else
      BinarySearch(K,A,L,(L+R)/2-1)
```

Observe that this algorithm is a recursive algorithm. More importantly, observe also that every time that a call is made to the function the array is approximately halved. So the best-case performance occurs when the key is at the midpoint of the list. In this instance, the function is called exactly once. Thus the best-case performance of BinarySearch is O(1). The worst-case performance of the algorithm occurs when the key is not in the array. The algorithm searches the array, repeatedly halving it, until the size of the array is 1. Thus the number of calls made to the function in the worst-case would be:

$$\begin{aligned}
&T(n) = 1 + T\left(\tfrac{n-1}{2}\right)\\
&T(n) = 2 + T\left(\tfrac{n-3}{4}\right)\\
&T(n) = 3 + T\left(\tfrac{n-7}{8}\right)\\
&...\\
&T(n) = k + T\left(\tfrac{n-2^k+1}{2^k}\right)
\end{aligned}$$

Thus the worst-case performance is $T\left(\frac{n-2^k+1}{2^k}\right) = 1$. Solving $\frac{n-2^k-1}{2^k} = 1$, we get $k = \log_2(n+1) - 1$. Thus, binarySearch is $\mathrm{O}(\log n)$. We will present the average-case analysis in class. We will also present an alternate, less rigorous, worst-case analysis.