Adam Gluck CSE 2321 Exam 2

Log Rules

- $\log(ab) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^k) = k\log(a)$
- $\log(1) = 0$
- $\log_b(b) = 1$
- $\log_b(b^k) = k$
- $b^{\log_b(k)} = k$
- $\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$

Summation Identities

$$\sum_{i=a}^{b} f(x) = \sum_{i=a}^{b}(x) + \sum_{i=a}^{b}(y)$$

$$\sum_{i=a}^{b} f(x) = \sum_{i=0}^{b} f(x) - \sum_{i=0}^{a-1} f(x)$$

$$\sum_{i=a}^{b} (c)\, f(i) = c\sum_{i=0}^{a-1} f(i)$$

$$\sum_{i=0}^{n-1} c = cn$$

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

$$\sum_{i=0}^{n-1} a\,r^i = \frac{a(r^n - 1)}{r-1} \qquad r \neq 1$$

Asymptomatic Analysis Definitions

- $f(n) = O(g(n))$
  - $O(g(n)) = \{f \in \mathcal{F}: \exists c, n_o > 0, \forall n \geq, f(n) \leq cg(n)\}$
  - f grows no faster than g, i.e. the growth of g is an upper bound to the growth of f.
- $f(n) = \Omega(g(n))$
  - $\Omega(g(n)) = \{f \in \mathcal{F}: \exists c, n_o > 0, \forall n \geq, f(n) \geq cg(n)\}$
  - f grows at least as fast as g, i.e. g is a lower bound to the growth of f.
- $f(n) = \Theta(g(n))$
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
  - The set of $f(n)$ where f grows as fast as g
- $f(n) = o(g(n))$
  - $o(g(n)) = O(g(n)) \backslash \Theta(g(n)$
  - f grows noticeably slower than g
- $f(n) = \omega(g(n))$
  - $\omega(g(n)) = \Omega(g(n)) \backslash \Theta(g(n)$
  - f grows noticeably faster than g

Upper Bound / Lower Bound Method Examples

- $f(n) = 3^{n+1} + 5n^4$

Upper Bound

$$\begin{aligned} 3^{n+1} + 5n^4 &\leq 3^{n+1} + 5(3^{n+1}) \\ &= 3 * 3^n + 15 * 3^n \\ &= 18 * 3^n \\ &= O(3^n) \end{aligned}$$

Lower Bound

$$\begin{aligned} 3^{n+1} + 5n^4 &\geq 3^{n+1} \\ &= 3 * 3^n \\ &= \Omega(3^n) \end{aligned}$$

Limit Method

Let f and g be monotonically increasing

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \begin{cases} 0 & \text{then} f(n) = O(g(n)) \\ c > 0 & \text{then} f(n) = \Theta(g(n)) \\ \infty & \text{then} f(n) = \Omega(g(n)) \end{cases}$$

Helpful Lemma

If $f(n) = O(g(n))$ then $\ln(f(n)) = O\left(\ln(g(n))\right)$.

For Loop Examples

```
Function T1(n):
x = 0
for i=1 to n do
    for j=1 to i do
        x = x + (i-j)
end
```

$$T2(n) = \sum_{a=1}^{n}\left(\sum_{b=1}^{a} 1\right) = \sum_{a=1}^{n} a = \frac{n(n+1)}{2} = \Theta(n^2)$$

```
Function T2(n):
x = 0
for i=1 to n do
    for j=1 to √n do
        x = x + (i-j)
end
```

*Note $n\lfloor\sqrt{n}\rfloor \leq n\sqrt{n}$

$$T2(n) = \sum_{a=1}^{n}\sum_{b=1}^{\sqrt{n}} 1 = \sum_{a=1}^{n} \sqrt{n} = n\sqrt{n} = \Theta(n^{1.5})$$

While Loop Examples

```
Function T3(n):
x = 0
i = 1
while i < n do
    x = (x + 1)^2
    i = 2i
end
```

*k is number of iterations

$$\begin{aligned} i = 1 * 2^k &< n \\ \lg(2^k) &< \lg(n) \\ k &< \lg(n) \end{aligned}$$

$$T3(n) = \sum_{a=1}^{\lg(n)} 1 = \lg(n) = \Theta(\lg(n))$$

For and While Loop Examples

```
Function T4(n):
for i = 1 to n do
    j = 1
    while j < n do
        x = (x + 1)^2
        j = 2j
end
```

$$\begin{aligned} j = 2^k &< n \\ \lg(2^k) &< \lg(n) \\ k &< \lg(n) \end{aligned}$$

$$T4(n) = \sum_{a=1}^{n} \sum_{b=1}^{\lg(n)} 1 = \sum_{a=1}^{n} \lg(n) = n\lg(n) = \Theta\big(n\lg(n)\big)$$

Recursion Trees

Expression

$$\begin{aligned} T(n) &= 2T(n-1) + 1 && \forall n > 1 \\ T(1) & && = 1 \end{aligned}$$

General Expression

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2(2(2T(n-3) + 1) + 1) + 1 \\ &= 2^3 T(n-3) + 4 + 2 + 1 \\ T(n) &= 2^{k+1} T\big(n - (k+1)\big) + \sum_{i=0}^{k} 2^i \end{aligned}$$

Find Number of Expansions

$$\begin{aligned} n - (k+1) &= 1 \\ k &= n - 2 \end{aligned}$$

Solve Runtime

$$\begin{aligned} T(n) &= 2^{n-2+1} T\big(n - (n-2+1)\big) + \sum_{i=0}^{n-2} 2^i \\ &= 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i \\ &= 2^{n-1} + \frac{1 - 2^{n-1}}{1-2} \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \\ T(n) &= \Theta(2^n) \end{aligned}$$

Recurrence Relation From Code

$$T(n) = f(n) T\big(g(n)\big) + h(n)$$

1. f(n) is the number of recursive calls that will happen (f(n) = 1 or 2 for most algorithms you will see)
2. g(n) describes how the size of the problem changes from one call to the next
3. h(n) describes the amount of work happening before and after the recursive calls

Example

```
int BinarySearch(Array, low, high, value)
    if low>high
        index = -1
    else
        midpt = (low+high)/2
        if value = Array[midpt]
            index = midpt
        else if k < Array[midpt]
            index = BinarySearch(Array, low, midpt - 1
, value)
        else
            index = BinarySearch(Array, midpt + 1,
high, value)
    return index
```

$$\begin{aligned} T(n) & & &= T(n/2) + c \\ T(n) & &= 1 & \quad n < 1 \end{aligned}$$

Sorting Algorithms

A sorting algorithm is any algorithm which solves this problem:

Input A sequence of n numbers $a_1, a_2, \dots, a_n$

Output A permutatuion $a_1', a_2', \dots, a_n'$ of the input sequence such that $a_1' \le a_2' \le \dots \le a_n'$

Comparison Sort

```
InsertionSort(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Best Case: A is already sorted. Then the while loop never runs, because the A[i]>key condition always fails. Thus, the running time of InsertionSort for an already sorted list is $\Theta(n)$

Worst Case: If A comes to us sorted in reverse order, then the while loop will run the maximum number of times in each iteration of the for loop. In this case, the running time will be

$$\begin{aligned} \sum_{j=2}^{n} \sum_{i=1}^{j-1} 1 &= \sum_{j=2}^{n} (j-1) = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \sum_{j=2}^{n} j - (n-1) = \sum_{j=1}^{n} j - 1 \\ &= \frac{n(n+1)}{2} - 1 - (n-1) \end{aligned}$$

Therefore, the running time of InsertionSort for a list in reverse sorted order is $\Theta(n^2)$