University of Houston

Mid-Term Review

COSC 3320
Algorithms and Data Structures

Gopal Pandurangan

This page intentionally left blank.

1 Asymptotic Notation

1.1 Big-O

Definition: Big-O.

A function f is "Big-O" of g, written $f(n) = O(g(n))$ if and only if there exist constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = O(g(n))$ if and only if

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

Definition: Little-o.

A function f is "Little-o" of g, written $f(n) = o(g(n))$ if and only if there exist constants c and n_0 such that

$$f(n) < c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = o(g(n))$ if and only if

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

1.2 Big-Ω

Definition: Big-Ω.

A function f is "Big-Ω" of g, written $f(n) = \Omega(g(n))$ if and only if there exist constants c and n_0 such that

$$f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = O(g(n))$ if and only if

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

Definition: Little-ω.

A function f is "Little-ω" of g, written $f(n) = \omega(g(n))$ if and only if there exist constants c and n_0 such that

$$f(n) > c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = \omega(g(n))$ if and only if

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

1.3 Big-Θ

Definition: Big-Θ.

A function f is "Big-Θ" of g, written $f(n) = \Theta(g(n))$ if and only if there exist constants a, c, and n_0 such that

$$a \cdot g(n) \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = \Theta(g(n))$ if and only if

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

where $0 < c < \infty$.

Equivalently, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$.

Definition: Little-θ.

A function f is "Little-θ" of g, written $f(n) = \theta(g(n))$ if and only if there exist constants a, c, and n_0 such that

$$a \cdot g(n) < f(n) < c \cdot g(n) \text{ for all } n \geq n_0$$

Equivalently, $f(n) = \theta(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Equivalently, $f(n) = \theta(g(n))$ if and only if $f(n) = o(g(n))$ **and** $f(n) = \omega(g(n))$.

1.4 Summary

Behavior	Written	Definition	Limit
Big-O	$f(n) = O(g(n))$	$f(n) \leq c \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
Big-Ω	$f(n) = \Omega(g(n))$	$f(n) \geq c \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
Big-Θ	$f(n) = \Theta(g(n))$	$a \cdot f(n) \leq f(n) \leq c \cdot g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
Little-o	$f(n) = o(g(n))$	$f(n) < c \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
Little-ω	$f(n) = \omega(g(n))$	$f(n) > c \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
Little-θ	$f(n) = \theta(g(n))$	$a \cdot g(n) < f(n) < c \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

1.5 Examples

Exercise 1: Show that $n^3 + 25n^2 + n + \log n + 15 = O(n^3)$.

Solution. There are two straightforward solutions to this problem. One is to show this using the Big-O definition directly. Since n^2, n, $\log n$, and 1 (the constant function) are all less than or equal to n^3 for $n \geq 1$, we have

$$\begin{aligned} n^3 + 25n^2 + n + 15 + \log n &\leq n^3 + 25n^3 + n^3 + n^3 + n^3 \\ &= 29n^3 \end{aligned}$$

Another is to take the limit of

$$\frac{n^3 + 25n^2 + n + \log n + 15}{n^3}$$

$$\begin{aligned} \lim_{n\to\infty} \frac{n^3 + 25n^2 + n + \log n + 15}{n^3} &= \lim_{n\to\infty} \frac{3n^2 + 50n + 1 + \frac{1}{n}}{3n^2} \text{ by L'Hôpital's rule} \\ &= \lim_{n\to\infty} \frac{6n + 50 + \frac{-1}{n^2}}{6n} \text{ by L'Hôpital's rule} \\ &= \lim_{n\to\infty} \frac{6 + \frac{2}{n^3}}{6} \text{ by L'Hôpital's rule} \\ &= \lim_{n\to\infty} 1 + \frac{1}{3}n^3 \\ &= 1 \end{aligned}$$

□

Exercise 2: Let $f(n) = n!$ and $g(n) = 7^n$. Determine which function has the greater asymptotic behavior, i.e., whether $f(n) = O(g(n))$ or $g(n) = O(f(n))$ (or both).

Solution. A good rule of thumb to remember is that $x^n < n! < n^n$ for sufficiently large n. Unfortunately, using the limit definition to prove this result is difficult. Instead, we'll show it using the definition of Big-O. Take for granted that for some $k > 7$, $k! \geq 7^k$. Then, for all $n > k$, we have

$$\begin{aligned} 7^n &= 7^k \cdot 7^{n-k} \\ &\leq k! 7^{n-k} \end{aligned}$$

Now,

$$7^{n-k} = \underbrace{7 \cdot 7 \cdot \ldots \cdot 7}_{n-k+1 \text{ multiplications}}$$

and

$$n! = \underbrace{1 \cdot 2 \cdot \ldots \cdot k}_{k!} \cdot \underbrace{(k+1) \cdot (k+2) \cdot \ldots \cdot n}_{n-k+1 \text{ multiplications}}$$

Since $k > 7$, those $n - k + 1$ multiplications in the factorial are greater than 7^{n-k}, so $7^n \leq n!$.

Now, a more clever solution is to let $h(n) = \frac{7^n}{n!}$ and apply the ratio test. Since

$$\lim_{n \to \infty} \frac{h(n+1)}{h(n)} = \lim_{n \to \infty} \frac{7}{n+1} = 0$$

then, by the ratio test, the series

$$\sum_{n=0}^{\infty} \frac{7^n}{n!}$$

converges. A series $\sum a_n$ converges only if $a_n \to 0$, thus

$$\lim_{n \to \infty} \frac{7^n}{n!} = 0$$

This means $7^n = O(n!)$. In fact, this argument works for any $a > 1$ to show that $a^n = o(n!)$. □

2 Mathematical Induction

2.1 Weak Induction

Definition: Weak Induction.

Let $P(n)$ be some statement on the natural numbers and suppose $P(i)$ holds for some non-negative integer i (the base-case). Then, if, for any $k \geq i$, $P(k)$ implies $P(k+1)$, then $P(n)$ holds for all $n \geq i$.

Usually (but not always), the base-case is 0 or 1. We show the base-case (it is quite often trivial) and then show, if the statement holds for some k, it must hold for $k+1$. Then the statement holds for all values greater than or equal to the base-case.

2.2 Strong Induction

Definition: Strong Induction.

Let $P(n)$ be some statement on the natural numbers and suppose $p(i)$ holds for some non-negative integer i (the base-case). Then, if, for any $k \geq i$, $\{P(j) | i \leq j \leq k\}$ implies $P(k+1)$, then $P(n)$ holds for all $n \geq i$.

Again, usually (but not always), the base-case is 0 or 1. We show the base-case (it is quite often trivial) and then show, if the statement holds for all k, it must hold for $k+1$. Then the statement holds for all values greater than or equal to the base-case.

2.3 Summary

Weak Induction and Strong Induction are *logically equivalent*. In other words, any statement that can be proven with Weak Induction can be proven with Strong Induction and vice-versa. If you are unsure which to use, choose Strong Induction. The general strategy for induction is

1. Show the base case, $P(0)$.
2. Assume the induction step $P(k)$ (Weak Induction) or $P(0), P(1), \ldots, P(k)$ (Strong Induction).
3. Show that $P(k+1)$ holds.
4. Then $P(n)$ is true by Induction.

2.4 Examples

Exercise 1: Show that $1 + 2 + 3 + \ldots + (n-1) + n = \frac{(n)(n+1)}{2}$ by Weak Induction.

Solution. The base case is trivial: $1 = \frac{(1)(1+1)}{2}$. Suppose that, for some $k > 1$, $1 + 2 + \ldots + k = \frac{(k)(k+1)}{2}$. Then, for $n = k + 1$, we have

$$\begin{aligned} 1 + 2 + \ldots + k + (k+1) &= (1 + 2 + \ldots + k) + (k+1) \\ &= \frac{(k)(k+1)}{2} + (k+1) \text{ by our Induction Hypothesis} \\ &= \frac{k^2 + k}{2} + \frac{2k+2}{2} \\ &= \frac{k^2 + 3k + 2}{2} \\ &= \frac{(k+1)(k+2)}{2} \end{aligned}$$

This completes the proof. □

Exercise 2: Let $T(n) = T(n-1) + T(n-2) + T(n-3)$ and $T(0) = 1$, $T(1) = 1$, and $T(2) = 3$. Show that $T(n) < 2^n$ by Strong Induction.

Solution. Our base cases $T(0)$, $T(1)$ and $T(2)$ are given. Suppose that, for all $k \leq n-1$, $T(k) < 2^k$. Then

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + T(n-3) \\ &< 2^{n-1} + 2^{n-2} + 2^{n-3} \text{ by our Induction Hypothesis} \\ &= 2^{n-3}(2^2 + 2^1 + 2^0) \\ &= 7 \cdot 2^{n-3} \\ &< 8 \cdot 2^{n-3} \\ &= 2^n \end{aligned}$$

This completes the proof. □

3 Recursion

3.1 Recursive Functions

A recursive function has two conditions

1. A termination condition.
2. A recursive call.

For this reason, there is an obvious parallel between recursion and Mathematical Induction. An example of a recursion is the factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

Our base case is when $n = 0$. If, for example, we wish to calculate 4! recursively, it would be as follows:

$$\begin{aligned} 4! &= 4 \cdot 3! \\ &= 4 \cdot (3 \cdot 2!) \\ &= 4 \cdot (3 \cdot (2 \cdot (1!))) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot (0!)))) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot (1)))) \\ &= 4 \cdot (3 \cdot (2 \cdot (1))) \\ &= 4 \cdot (3 \cdot (2)) \\ &= 4 \cdot (6) \\ &= 24 \end{aligned}$$

3.2 Recurrences

A way to evaluate the runtime of a recursive algorithm is to set up a recurrence relation. In the factorial example above, let T denote the runtime of the algorithm. Clearly, $T(0) = 0$. Each step of the recursion involves a single multiplication and then calls the algorithm on the previous number. Our recurrence is therefore $T(n) = T(n-1) + 1$. To determine the asymptotic runtime of this algorithm, we unroll the recursion:

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 2 \\ &\vdots \\ &= T(1) + (n-1) \\ &= T(0) + n \\ &= n \end{aligned}$$

Hence $T(n) = O(n)$.

However, often our recurrences are more involved. Consider the recurrence for Merge Sort. Each step involves splitting the array in half and then performing the Merge Sort on each half, then combining the result in $O(n)$ time. This gives the following recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Another way to solve a recurrence is by "guess-and-check". For example, let's try to show that $T(n) = O(n)$. Proceed by Strong Induction. The base-case is given. Then

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2} + c'n \\ &= (c + c')n \end{aligned}$$

This does not work, because it is crucial that the c we use in the induction step (that $T(n) \leq cn$) is the *same* c at the end. Showing $T(n) \leq (c + c')n$ is *not* the desired result.

Thus, we revise our guess. Let's try $T(n) = O(n \log n)$. Then

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2}\log\frac{n}{2} + c'n \\ &= cn\log n + c'n - cn\log 2 \\ &= cn\log n + c'n - cn \\ &= cn\left(\log n + c' - c\right) \end{aligned}$$

Now, for sufficiently large c, we have $c' < c$. Then $c' - c < 0$, and

$$\begin{aligned} T(n) &\leq cn\,(\log n + c' - c) \\ &\leq cn \log n \end{aligned}$$

3.3 Divide and Conquer

Divide and Conquer is an algorithm technique in which you split your problem into several subproblems then combine the solutions on those subproblems. Merge Sort is a classic example: sort an array by dividing it in half, sort each half recursively, then combine the sorted sub-arrays into one sorted array.

3.4 The DC Recurrence Theorem

The **DC Recurrence Theorem** applies for any recursion of the form $T(n) = aT(\frac{n}{b}) + f(n)$. It provides a straightforward method of determining the asymptotic runtime of recurrences that are in this form.

Theorem. Let $T(n) = aT(\frac{n}{b}) + f(n)$ be a recurrence and let $af(\frac{n}{b}) = cf(n)$. Then

1. If $c < 1$ then $T(n) = \Theta(f(n))$.
2. If $c > 1$ then $T(n) = \Theta(n^{\log_b a})$.
3. If $c = 1$ then $T(n) = \Theta(f(n) \log_b n)$.

For the above example of Merge Sort, we have $T(n) = 2T(\frac{n}{2}) + O(n)$. In this case, $f(n) = c'(n)$. Then $af(\frac{n}{b}) = 2c'(\frac{n}{2}) = c'n = f(n)$. Thus, $c = 1$ and we have $T(n) = \Theta(n \log n)$.

Note that this theorem *only* applies to recurrences like the above. For example, to determine the asymptotic runtime of $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + n$, the DC Recurrence Theorem does *not* apply. We must use induction or analyze the recursion tree to determine its runtime.

3.5 Examples

Exercise 1: Show that $T(n) = 4T(\frac{n}{2}) + n^3 = \Theta(n^3)$.

Solution. In this case, $a = 4$, $b = 2$ and $f(n) = n^3$. Then $af(\frac{n}{b}) = 4\left(\frac{n}{2}\right)^3 = \frac{4}{8}n^3 = \frac{1}{2}n^3$. Then $c = \frac{1}{2} < 1$ and, by the DC Recurrence Theorem, $T(n) = \Theta(n^3)$. □

Exercise 2: Determine the runtime of $T(n) = 3T(\frac{n}{2}) + n$.

Solution. In this case, $a = 3$, $b = 2$, and $f(n) = n$. Then $af(\frac{n}{b}) = 3\frac{n}{2} = \frac{3}{2}n$. Then $c = \frac{3}{2} > 1$ and, by the DC Recurrence Theorem, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log 3})$. □

Exercise 3: You are given n stones (assume that n is a power of 2) each having a distinct weight. You are also given a two-pan balance scale (no weights are given). For example, given two stones, you can use the scale to compare which one is lighter by placing the two stones on the two different pans. The goal is the find the heaviest and the lightest stone by using as few weighings as possible. Give a divide and conquer strategy that uses only $\frac{3n}{2} - 2$ weighings.

Solution. Write $n = 2^m$. Perform $\frac{n}{2}$ comparisons, noting that the heavier stones are candidates for the heaviest stone, and lighter stones are the candidates for the lightest stone. Perform comparisons on the remaining $\frac{n}{4}$ heavier stones, then $\frac{n}{8}$ heavier stones, and so on. This will total

$$2^m + 2^{m-1} + \ldots + 4 + 2 + 1 = 2^m - 1 = n - 1 \text{ comparisons}$$

and will determine the heaviest stone. From the remaining $\frac{n}{2}$ stones in the pile of candidates for the lightest stone, we similarly perform

$$2^{m-1} + 2^{m-2} + \ldots + 4 + 2 + 1 = 2^{m-1} - 1 = \frac{n}{2} - 1 \text{ comparisons}$$

to determine the lightest stone. In total, this is

$$\overbrace{n-1}^{\text{test on heavy stones}} + \overbrace{\frac{n}{2}-1}^{\text{test on light stones}} = \frac{3n}{2} - 2 \text{ comparisons}$$

□

4 Dynamic Programming

Dynamic programming involves breaking a problem into *overlapping* subproblems, then combining the solutions to those subproblems to determine the solution to the greater problem.

4.1 Fix-One-Index Problems

One way to break a problem P_n into subproblems is to consider subproblems of the form $P_{0,i}$ for all i, where the left-most index is fixed. We then combine those solutions to determine $P_{0,n}$, our original problem. Suppose, for example, that we wish to transform a string x into a string y. We consider the subproblems of transforming x_i (the first i characters) into y_j (the first j characters), and determine the solutions to these subproblems to determine the solution to the original problem.

4.2 Vary-Both-Indices Problems

Another way to break a problem P_n into subproblems is to consider subproblems of the form $P_{i,j}$ for all i and j. In this case, *neither* index is fixed. Suppose, for example, that we have a product of matrices $M_0 M_2 \dots M_{n-1}$ and we wish the minimize the total number of operations performed on this multiplication by grouping our matrices. Our subproblems are $P_{i,j}$ where i is the index of the left-most matrix and j is the index of the right-most matrix.

4.3 Iterative Solutions

Suppose we have broken our problem P into subproblems P_i. We let i iterate from 0 to n and solve each problem iteratively. This usually done by storing our values P_i into an array. This commonly referred to as a *Bottom-Up* approach.

4.4 Recursive Solutions

In this case, we have a recursive formulation for P_n in terms of previous values of P_i. We recursively calculate $P(n)$ to determine our solution. However, this will usually have sub-optimal runtime due to the overlapping-subproblems. To reduce the number of recursive calls, we create a *lookup-table*. The general format for this is

1. Examine $P(n)$ in our lookup table. If it has been evaluated, return the value from the table.
2. Else, evaluate $P(n)$ and store the value in the table.

This is usually done with a *sentinal* value. This is something that indicates that the problem has not yet been solved. For example, if you are determining the minimum value, you can set $T[n] = -\infty$. When looking up a value, if $T[n] = -\infty$, the value has not been determined, and thus must be computed. If the value is anything else, it has been calculated already and can be returned.

4.5 Exercises

Exercise 1: Let x and y be strings of symbols from some alphabet set of length m and n respectively (m need not be equal to n). Consider the operations of deleting a symbol from x, inserting a symbol into x and replacing a symbol in x by another symbol (belonging to the alphabet set). Your goal is to design an efficient algorithm using dynamic programming to find the minimum number of such operations (as well as to find the operations that transform x to y that correspond to the minimum number) needed to transform x into y. Your algorithm should run in time O(mn).

Solution. Let $x = x_0x_1 \dots x_{m-1}$ and $y = y_0y_1 \dots y_{n-1}$. Denote by $S_{i,j}(x, y)$ the minimum number of operations to transform the prefix $x_0x_1 \dots x_{i-1}$ into the prefix $y_0y_1 \dots y_{j-1}$ for some $(i, j) \in \{0, 1, \dots, m\} \times \{0, 1, \dots, n\}$.[1] Our subproblems can be stored in the following $(m + 1) \times (n + 1)$ matrix

$$\begin{bmatrix} S_{0,0}(x,y) & S_{0,1}(x,y) & \dots & S_{0,n-1}(x,y) & S_{0,n}(x,y) \\ S_{1,0}(x,y) & S_{1,1}(x,y) & \dots & S_{1,n-1}(x,y) & S_{1,n}(x,y) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ S_{m-1,0}(x,y) & S_{m-1,1}(x,y) & \dots & S_{m-1,n-1}(x,y) & S_{m-1,n}(x,y) \\ S_{m,0}(x,y) & S_{m,1}(x,y) & \dots & S_{m,n-1}(x,y) & S_{m,n}(x,y) \end{bmatrix}$$

Define $C(a, b)$ on characters a and b as follows

$$C(a,b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

Now, consider $S_{i,j}(x, y)$. To transform $x_0x_1 \dots x_{i-1}$ into $y_0y_1 \dots y_{j-1}$, we can

1. Transform $x_0x_1 \dots x_{i-2}$ into $y_0y_1 \dots y_{j-1}$ using $S_{i-1,j}(x, y)$ operations, then perform one deletion, for a total of $S_{i-1,j}(x, y) + 1$ operations.

$$\begin{array}{l} \underbrace{x_0x_1 \dots x_{i-2}}_{\text{Transform}}\,\underbrace{x_{i-1}}_{\text{Delete}} \\ y_0y_1 \dots y_j \end{array}$$

2. Transform $x_0x_1 \dots x_{i-1}$ into $y_0y_1 \dots y_{j-2}$ using $S_{i,j-1}(x, y)$ operations, then perform one insertion, for a total of $S_{i,j-1}(x, y) + 1$ operations.

$$\begin{array}{l} \underbrace{x_0x_1 \dots x_{i-1}}_{\text{Transform}} \\ y_0y_1 \dots y_{j-2}\,\underbrace{y_{j-1}}_{\text{Insert}} \end{array}$$

3. Transform $x_0x_1 \dots x_{i-2}$ into $y_0y_1 \dots y_{j-2}$ using $S_{i-1,j-1}(x, y)$ operations, then
   (a) if $x_{i-1} = y_{j-1}$, zero replacements
   (b) if $x_{i-1} \neq y_{j-1}$, one replacement

   for a total of $S_{i-1,j-1}(x, y) + C(x_{i-1}, y_{j-1})$ operations.

$$\begin{array}{ll} \underbrace{x_0x_1 \dots x_{i-2}}_{\text{Transform}} & \underbrace{x_{i-1}}_{\text{replacement}} \\ y_0y_1 \dots y_{j-2} & y_{j-1} \end{array}$$

Thus, setting

$$\begin{aligned} m_1 &= S_{i-1,j}(x, y) + 1 \\ m_2 &= S_{i,j-1}(x, y) + 1 \\ m_3 &= S_{i-1,j-1}(x, y) + C(x_{i-1}, y_{j-1}) \end{aligned}$$

gives our recursion

$$S_{i,j}(x, y) = \min(m_1, m_2, m_3)$$

Now, the base cases are when i or j are 0. $S_{0,j}(x, y)$ is the minimum number of operations to tranform an empty string into $y_0, y_1, \dots y_{j-1}$, which is clearly just j. Similarly, $S_{i,0}(x, y) = i$. □

[1] Noting that $S_{0,j}$ and $S_{i,0}$ correspond to cases with empty strings.