CSC2710 Analysis of Algorithms — Fall 2022
Unit 2 - Asymptotic Notation and Analysis

Book: §2.2; §3.1; §A.1 – §A.2

2.1 Analysis Framework

- Recall their are two types of efficiency
  1. **Time**: How much time it takes to for an algorithm to run on an input of size $n \in \mathbb{Z}^*$.
  2. **Space**: How much *extra* space an algorithm need for an input of size $n \in \mathbb{Z}^*$.
- While space complexity is not as important as it once was, we will still consider it though to a less extent than time complexity.
- **Fact:** The time to run an algorithm increases as the size of the input increases.
- How do we measure an input size n?
  - Could be the number of bits.
  - Could be the number of elements in an array.
  - Could be the degree of a polynomial.
- In any case, it is generally easy to find the size parameter.
  - Note: it is possible to have multiple size parameters. For example, consider a $m \times n$ matrix.
- What should the units of our running time be?
  - Should they be seconds or minutes?
  - should they be picoseconds?
- Do any time units actually make sense?
  - I argue "No!". The time is dependent on a specific computer that is not general enough.
  - Instead, lets count the operations on the idealized RAM model of computation we talked about.
    * We always count in terms of the input size. In other words, we are getting a formula *not* a number.
- The book talks about counting the number of times a *basic operation* is executed.
  - A basic operation is an operation that contributes most to the running time.
  - This requires a lot of intuition to spot. Therefore, we are going to start with counting all the operations and work on building our intuition.
- No matter how we choose to count operations, we will *always* ignore constants.
- When we ignore constants and only look at the largest term in a count, we get the *order of growth* of the function.
  - e.g., $10n^2 + 3n + 4$ has the order of growth n^2.
  - This makes sense as we are really concerned with the behavior of algorithms on large values of n. In other words, what term dominates as $n \to \infty$.

- To see common order of magnitudes behavior see the table below:

n	$\lg n$	n	$n \lg n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1,000	1,024	3,628,800
100	6.6	100	660	10,000	1,000,000	1.26×10^{30}	9.3×10^{157}

- One can easily imagine wanting to sort 100 elements in an array do you really want an algorithm that preforms n^2 operations?
- Things to note, we only considered the base two logarithm, lg, as it differs from all other logarithms by only a constant factor. Remember we ignore constants.
  - **Practice**: Why can I say that all logarithms only differ by a constant from the base two logarithm?
- When we consider time efficiency in terms of n we also look at different types of behaviors. In particular we look at:
  1. **Worst-Case Efficiency**: The efficiency for the worst-case input of size n. This is the input of size n for which the algorithm runs the longest (performs the most operations) among all possible inputs of the size in question.
     - This is the most common form of analysis we will be performing.
  2. **Best-Case Efficiency**: The efficiency for the best-case input of size n. his is the input of size n for which the algorithm runs the shortest (performs the least amount of operations) among all possible inputs of the size in question.
     - Generally, we are not so concerned with best-case efficiency.
  3. **Average-Case Efficiency**: The algorithms behavior on a typical random input of size n. This type of analysis depends on assumptions about the structure of the input of size n.
     - We will *not* be performing a lot of this type of analysis. It is more common in graduate school.
  4. **Amortized Efficiency**: Look at the efficiency of a sequence of operations performed on a data structure. It is similar to amortizing costs in a business. In other words, you look some operations will be very expensive while others are very cheap. So, if we have few uses of the expensive operation it becomes cheaper to perform over time as the cost is absorbed in to the long sequence of n operations.
     - It is unlikely that we will do any of this form of analysis this semester.
     - You will see this if you take CSC 5030 Analysis of Algorithms II

2.2 A Search Problem Example

- We have looked at *a lot* of definitions and discussed a lot of pros and cons. Let's look at a simple problem and go through all of our steps (sans implementation).
- The problem we are solving is the *search problem*

  Definition 1 (Search Problem)**.**

  Input: *A sequence of n keys $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a search key k.*

  Output: *The index i, of k in the input sequence if there exists an $a_i = k$ or $\perp$ otherwise.*

- In our case our sequence will be an array A.
- What's the most basic algorithm for searching for a key k in array A?
  - Our good friend the linear search
  - The linear search algorithm is given as follows:

```python
#Input: An array A[0 .. n-1], a search key k
#Output: Return the index of the first element in A that matches
#  search key k or None if there is no matching element
def LIN_SEARCH(A, k):
    for i in range( len(A) ):                    # Line 1
        if A[i] == k:                            # Line 2
            return i                             # Line 3
    return None                                  # Line 4
```

- What's the **worst case** instance of the algorithm? It turns out with any search that the worst case is any instance in which the key `k` does not appear in the search structure (in this case, the array `A`).
- What's the worst case *running time*? To compute this we determine the cost for each line and number of times each lines executes. The results are in the table below.

Line	Cost	Count
1	c_1	n
2	c_2	n
3	c_3	0 (in the worst case)
4	c_4	1 (at most)

  - We then total the cost multiplied by the count for each line. Since line 6 or 7 only executes, never both we only add the constant once [1].

$$\begin{aligned} T(n) &= c_1 n + c_2 n + c_4 \\ &= (c_1 + c_2)n + c_4 \\ &= c_1' n + c_2' && \text{Collapse constants into new constants.} \\ &\approx n && \text{Order of growth by dropping constants.} \end{aligned}$$

  - Notice that we drop all low order terms in other words we look at what term in the sum grows quickest. This can be achieved by looking at limits.
    - Sometimes we will need to do this to help us see the fastest growing term.
  - For completeness, line 2 is the *basic operation* of this algorithm.

[1] In the worst case we will not find the key in the array A.

2.3 Asymptotic Notations and Basic Efficiency Classes

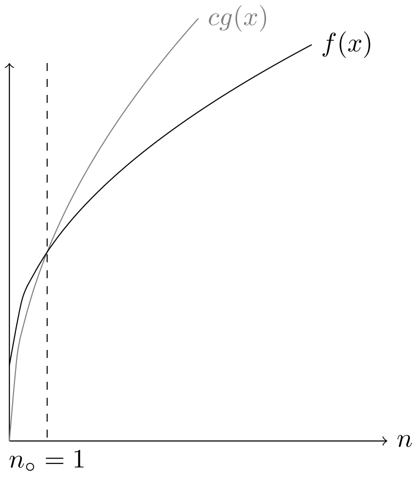
- To bound orders of growth, we will use three different notions making up a study known as *aysmptotics*.
- We will consider three types of asymptotic notations that can be used to bound $T(n)$.
  1. Big-Oh notation
  2. Big-Ω notation
  3. Big-Θ notation
- Big-Oh notation gives us an asymptotic *upper bound* on a function.
- Intuitively, we are looking for a function $g(n)$ that is always greater than our function $T(n)$
- Formally, we define big-Oh notation as follows

  Definition 2 (Big-Oh Notation)**.** *For a given function $g(n)$ we denote by $O(g(n))$ set of functions:*

$$O(g(n)) = \{f(n) \mid \exists c, n_\circ > 0 \text{ s.t. } 0 \le f(n) \le cg(n), \forall n \ge n_\circ\}$$

  Note, that $O(g(n))$ is the set of all *functions bounded above by some constant multiple of $g(n)$.*

- Graphically we have something like:



- Example: If $T(n) = 5n^2 + 3n + 2$ then $T(n) = O(n^2)$. To prove this is true we can simply find a constant. Our goal is to satisfy:

$$5n^2 + 3n + 2 \le cn^2$$

  for all $n > n_\circ$ where $n_\circ > 0$. Since the polynomial is monotonically increasing[2] we know that $n_\circ = 1$. Thus, $10 \le c$.

  - A different answer can be computed as,

$$\begin{aligned} 5n^2 + 3n + 2 &< 5n^2 + 3n^2 && \text{(For } n \ge 2) \\ &\le 8n^2 \le c_1 n^2 && \text{(For } c_1 \ge 8) \end{aligned}$$

[2] A function is monotonically increasing if $m \ge n$ implies that $f(m) \ge f(n)$ for all m and n in the domain.

- Example: If $T(n) = 7n\lg n + 3n$ and we want to show that $T(n) = O\left(n^2\right)$ this is a common abuse of notation [3]. We must show that there exists an $n_\circ \geq 0$ and c such that for all $n \geq n_\circ$

$$7n\lg n + 3n \leq cn^2.$$

$$\begin{aligned} 7n\lg n + 3n &\leq cn^2 && (n \geq n_\circ = 2 \text{ otherwise } \lg n = 0) \\ \implies \frac{7\lg n}{n} + \frac{3}{n} &\leq c && (\text{Can be done since } n \geq 2) \\ \implies 5 &\leq c \end{aligned}$$

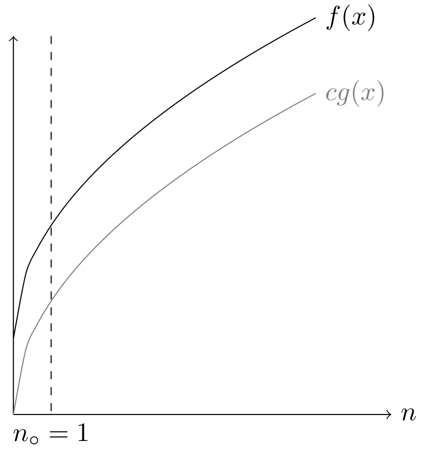
- Big-Ω notation gives us an asymptotic *lower bound* on a function.
- Intuitively, we are looking for a function $g(n)$ that is always less than our function $T(n)$
- Formally, we define big-Ω notation as follows

  Definition 3 (Big-Ω Notation)**.** *For a given function $g(n)$ we denote by $\Omega(g(n))$ set of functions:*

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_\circ > 0 \ s.t.\ 0 < cg(n) \leq f(n), \forall n \geq n_\circ\}$$

  Note, that $\Omega(g(n))$ is the set of all *functions bounded below by some constant multiple of $g(n)$.*

- Graphically we have something like:



- Example: If $T(n) = 5n^2 + 3n + 2$ then $T(n) = \Omega\left(n^2\right)$. To prove this is true we can simply find a constant. Our goal is to satisfy:

$$5n^2 + 3n + 2 \geq cn^2$$

  We proceed as follows

$$\begin{aligned} 5n^2 + 3n + 2 &\geq cn^2 \\ \implies 5 + \frac{3}{n} + \frac{2}{n^2} &\leq c && (\text{provided } n \geq n_\circ = 1.) \\ \implies 5 &\geq c && (\text{as } n \to \infty \text{ for the ratio.}) \end{aligned}$$

  We can select $c = 5$ and our result holds for $n_\circ = 1$.

[3]We will abuse notation by saying $T(n) = O\left(n^2\right)$ instead of using the more correct $T(n) \in O\left(n^2\right)$

- Alternatively, we could proceed as follows:

$$\begin{aligned} 5n^2 + 3n + 2 &\geq 5n^2 && \text{(For all } n \geq 1\text{)} \\ &\geq c_1 n^2 && \text{(For } c_1 \leq 5\text{)} \end{aligned}$$

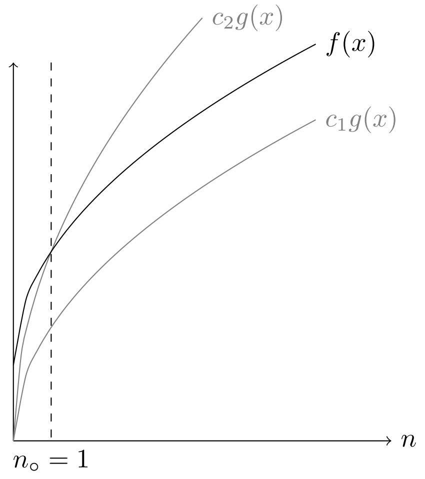
- Big-Θ notation gives us an asymptotic *tight bound* on a function.
- Intuitively, we are looking for a function $g(n)$ that bounds $T(n)$ both above and below.
- Formally, we define big-Θ notation as follows

  Definition 4 (Big-Θ Notation)**.** *For a given function $g(n)$ we denote by $\Theta(g(n))$ a set of functions:*

  $$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_\circ > 0 \text{ s.t. } 0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_\circ\}$$

  Note, that $\Theta(g(n))$ is the set of all *functions bounded above and below by constant multiples of $g(n)$.*

- Graphically we have something like:



- Example: If $T(n) = 5n^2 + 3n + 2$ then $T(n) = \Theta(n^2)$. To prove this is true we need to find a c_1, c_2, and $n_\circ$. One way to solve this problem is to break the inequality in half. Solve for c_1 and then solve for c_2 the $n_\circ$ value becomes the maximum of the two sides.
  - This hints at the following theorem

    Theorem 1. *$T(n) \in \Theta(g(n))$ iff $T(n) \in O(g(n))$ and $T(n) \in \Omega(n)$.*

- Let's prove the theorem. I'm an advocate of the *"How does that story go again?"* proof approach.
  - What does it mean for a function $f(n) \in \Theta(g(n))$?
    * There exists a $c_1, c_2, n_\circ > 0$ such that

      $$0 < c_1 g(n) \leq f(n) \leq c_2 g(n),$$

      for all $n \geq n_\circ$.
  - What does it mean for a function $f(n) \in O(g(n))$?

    * There exists a $c, n_\circ > 0$ such that

$$0 < f(n) \leq cg(n),$$

for all $n \geq n_\circ$.

  - What does it mean for a function $f(n) \in \Omega(g(n))$?
    * There exists a $c, n_\circ > 0$ such that

$$0 < cg(n) \leq f(n),$$

for all $n \geq n_\circ$.

- Since the theorem has an if and only if form, we break it up into two lemmas:

Lemma 1. *If $T(n) \in \Theta(g(n))$ then, $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$.*

Lemma 2. *If $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$ then, $T(n) \in \Theta(g(n))$.*

- Both directions are fairly straightforward. We will prove lemma 2 and leave lemma 1 as an exercise to the reader.

Proof. Assume that $T(n) \in \Omega(g(n))$ this implies that there exists a $a, n_1 > 0$ such that $0 < ag(n) \leq f(n)$. We further assume that $T(n) \in O(g(n))$ this implies that there exists a $b, n_2 > 0$ such that $T(n) \leq bg(n)$.

In order to show that $T(n) \in \Theta(g(n))$ one must find a $c_1, c_2, n_\circ > 0$ such that

$$0 < c_1 g(n) \leq T(n) \leq c_2 g(n),$$

for all $n \geq n_\circ$. We select $c_1 = a$ and $c_2 = b$ given above. We know that our constants are only good when n is greater then their respective n_i so we set $n_\circ = \max\{n_1, n_2\}$. We have thus, proved the theorem. □

- Another useful theorem when dealing with asymptotics is the following theorem about consecutive execution of algorithms.

Theorem 2. *If $T_1(n) \in O(g_1(n))$ and $T_2(n) \in O(g_2(n))$, then $T_1(n) + T_2(n) \in O(\max\{g_1(n), g_2(n)\})$.*

The theorem also holds when dealing with Ω and Θ.

2.4 The Limit Approach

- We can relate two functions using limits as $n \to \infty$. In particular,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & \text{Implies } f(n) \in o(g(n)) \\ c, & \text{Implies } f(n) \in \Theta(g(n)) \\ \infty, & \text{Implies } f(n) \in \Omega(g(n)) \end{cases}$$

  - Technically the first case is $o(g(n))$, this is not a typo, that should be little "o". I will not however, mark you down for writing big "O".
  - It should be noted, that if the limit does not exist, the definitions of the appropriate asymptotic notation must be used.
- An additional common application of the limit notion is to order functions by growth rate.
- If the limit ends up of the form $\frac{\infty}{\infty}$, we need to use L'Hôpital's Rule. Recall L'Hôpital's Rule is:

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

  - Note, you can apply L'Hôpital's Rule as many times as you would like.
- **Practice**: Let's compare the function $f(n) = n^2$ and $g(n) = n \lg n$.
- **Practice**: Compare $f(n) = \lg n$ and $g(n) = \sqrt{n}$.
- **Practice**: Order the following functions in increasing order by the growth rate.

$$n^2, \sqrt{n}, n^3 \lg n, 2^{\lg n}$$

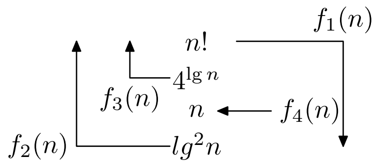
- Sometimes you might be asked to determine if one function is asymptotically bounded by another function when you are only given asymptotic information about the functions. For example, say you are given:

$$f_1(n) \in O(n!) \quad f_2(n) \in \Omega\left(\lg^2 n\right) \quad f_3(n) \in \Omega\left(4^{\lg n}\right) \quad f_4(n) \in \Theta(n)$$

  1. Order the asymptotic classes in asymptotically increasing order. You will want to write the functions on top of each other with the largest asymptotic class on top. Using our example we will have:

$$\begin{array}{r} n! \\ 4^{\lg n} \\ n \\ \lg^2 n \end{array}$$

     - Note here that $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n. This is known as *Stirling's Formula.* This will come up occasionally. Something you generally look up when you need it.
  2. Draw an arrow at each function that points the corresponding asymptotic class as follows:
     (a) directly at it if $f(n)$ is tightly bound to the asymptotic class
     (b) an arrow pointing up if $f(n)$ is bound below by the asymptotic class.
     (c) an arrow pointed down if $f(n)$ is bounded above by the asymptotic class
- This will help you in answering questions of the form given below
  - **Practice**: Is it always true that $f_3(n) \in \Omega(f_4(n))$? Justify your answer.
  - **Practice**: Is it always true that $f_4(n) \in O(f_1(n))$? Justify your answer.
  - **Practice**: Is it always true that $f_3(n) \in \Omega(f_2(n))$? Justify your answer.



2.5 Classes

- Here is a table of the common efficiency classes we will "bump in to" during the course.

Class	**Name**	**Comment**
1	constant	Any algorithm that performs a fixed number of operations regardless of n.
$\lg n$	logarithmic	
n	linear	
$n \lg n$	polylogarithmic (sometimes linearithmic)	A polylogarithmic function is any function of the form $a_k \lg^k n + a_{k-1} \lg^{k-1} n + \ldots a_0$.
n^2	quadratic	
n^3	cubic	
2^n	exponential	
$n!$	factorial	

2.6 Mathematical Analysis of Non-recursive Algorithms

- We are going to do a bunch of examples to apply the methods of analysis we learned.
- Our text has the following generic process, I won't be strictly following all the of the steps as it relies on intuition in some parts. As you get more comfortable with analysis you will find it easier to rely on your intuition.
    1. Decide on a parameter (or parameters) indicating the inputs size.
    2. Identify the algorithms basic operation. Most likely in the innermost loop. *This is an intuition step.*
    3. Check whether the number of times the basic operation is executed is only dependent on the size of input n. If not, your best-case, worst-case, and average-case analysis will all be different.
    4. Set up a sum expressing the number of times the basic operation is executed.
    5. Using standard formulas and rules for sum manipulation, either
        - find a closed form of the sum[4], or
        - establish an order of growth.

2.6.1 Example 1: Maximum Element Problem

- The maximum element problem is:

  Definition 5 (Maximum Element Problem)**.**
  Input: *An array of* n *elements* $A = \{a_1, a_2, \ldots, a_n\}$.
  Output: *An element* $a_j \in A$ *such that* $a_j \geq a_i, \forall i \in \{1, \ldots, n\} \setminus \{j\}$.

- How might you solve this problem?
- The solution I want to consider is the natural algorithm below
    - The intuition is that we walk left to right through the list and update the current maximum element we have seen. At the end of the loop we return the maximum element.

```
# Input: An array A of 1 or more integers
# Output: The returned element A[j] is the largest element in A
def MaxElement(A):
    largest = A[0]                 # Line 1
    for i in range( 1, len(A) ):   # Line 2
        if A[i] > largest:         # Line 3
            largest = A[i]         # Line 4
    return largest                 # Line 5
```

- Using the formal strategy from the book, what is our input size?
    - Answer: n.
- Lets build a table for the cost and count of each line in our algorithm (i.e. our time $T(n)$).

Line	Cost	Count	
1	c_1	1	
2	c_2	$n-1$	
3	c_3	$n-1$	
4	c_4	$n-1$	(in the worst case)
5	c_5	1	

[4] You will find appendix A and the Theoretical Computer Scientist's Cheat Sheet (on Classroom) helpful with this task.

- What is our sum?
  - $T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5$.
- Lets determine the asymptotic growth of our sum.

$$\begin{aligned}
T(n) &= c_1 + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \\
&= c_1 + (c_2 + c_3 + c_4)(n-1) + c_5 \\
&= c_6 + c_7 n && \text{where } c_6 = c_1 + c_5 - (c_2 + c_3 + c_4) \text{ and} \\
& && c_7 = c_2 + c_3 + c_4. \\
&\leq c_6 n + c_7 \\
&= c_8 n && \text{where } c_8 = c_6 + c_7 \\
&\in \Theta(n)
\end{aligned}$$

2.6.2 Example 2: Element Uniqueness Problem

- Formally, the problem is:

  Definition 6 (Element Uniqueness Problem)**.**
  Input: *A set of n elements $A = \{a_1, a_2, \ldots, a_n\}$.*
  Output: *True if every element in A is unique. An element a_i is unique if $a_i \neq a_j$ for all $j \in \{1, 2, \ldots, n\} \setminus \{i\}$.*

- Can you think of a way to solve it?
- Here is my solution
  - The intuition is that we work left to right comparing an element to all elements that follow it. We do not need to worry about the elements that preceed the current element as these have already been checked (think induction)

```python
# Input: An array of A of inegers
# Output: True if the array elements are unique, False otherwise
def UnitElements(A):
    for i in range(0, len(A) - 1):                  # Line 1
        for j in range(i + 1, len(A)):              # Line 2
            if A[i] == A[j]:                        # Line 3
                return False                        # Line 4
    return True                                     # Line 5
```

- Using the formal strategy from the book, what is our input size?
  - Answer: n.
- What is the worst case instance?
  - When the elements *are* unique.[5]
- Lets build a table for the cost and count of each line in our algorithm (i.e. our time $T(n)$).

Line	Cost	Count	
1	c_1	$n-1$	
2	c_2	$\sum_{i=0}^{n-1} t_i$	t_i number of times the loop executes for a given i.
3	c_3	$\sum_{i=0}^{n-1} t_i$	
4	c_4	0	(in the worst case)
5	c_5	1	

- What is our sum?
  - $T(n) = c_1(n-1) + (c_2 + c_3)\sum_{i=0}^{n-1} t_i + c_5$
  - Whats t_i for a given i then?
    - ∗ at a given iteration i of the outer loop the inner loop runs from $i+1$ to $n-1$.
    - ∗ What are the number of iterations between $i+1$ and $n-1$ inclusive?
      - · Answer: $n - 1 - (i+1) + 1$.
      - · We simply to get, $n - i - 1$.

[5]This may seem unintuitive, but "worst case" does not often align with "undesired outcome".

– Lets work on just computing $\sum_{i=0}^{n-1} t_i$.

$$\begin{aligned}
\sum_{i=0}^{n-1} t_i &= \sum_{i=0}^{n-1} (n-i-1) \\
&= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 && \text{(Appendix A Su Manipulation Rule 2)} \\
&= n\left(n-1\right) - \sum_{i=0}^{n-1} i - \left(n-1\right) \\
&= n^2 - n - \sum_{i=0}^{n-1} i - n + 1 \\
&= n^2 - n - \left(\frac{(n)\left(n-1\right)}{2}\right) - n + 1 && \text{(Gaussian Sum – Appendix A)} \\
&= n^2 - n - \frac{1}{2}n^2 + \frac{n}{2} - n + 1 \\
&= \frac{1}{2}n^2 - \frac{1}{2}n + 1 \\
&\leq n^2 && \text{(By upper bounding)}
\end{aligned}$$

– This means $T(n) \leq n^2$ beyond some n_0. This implies $T\left(n\right) \in \Theta\left(n^2\right)$.

2.6.3 Example 3: Binary Digit Counting Problem

- The binary digit counting problem is formally defined as:

  Definition 7 (Binary Digit Counting Problem)**.**
  ***Input**: An integer $n \in \mathbb{Z}^+$.*
  ***Output**: The b the number of bits in the binary expansion of n.*

- How might you solve this?
  - Intuitively, I would count the number of powers of two in the number n.
- What is the worst case instance?
- Here is one formal algorithm that solves the problem.

```
import math

# Input: A positive integer n
# Output:  The number of bits in the binary expansion of n is returned
def BinaryDigitCount( n ):
    count = 1                   # Line 1
    while n > 1:                # Line 2
        count = count + 1       # Line 3
        n = math.floor(n / 2)   # Line 4
    return count                # Line 5
```

  - Recall $\lfloor x \rfloor$ denotes the floor of a number x. The floor of a number is the largest integer less than or equal to x.
    * Example: $\lfloor 3.2 \rfloor = 3$.
    * Example: $\lfloor 4 \rfloor = 4$.
    * Example: $\lfloor 8.9 \rfloor = 8$.
  - Using the formal strategy from the book, what is our input size?
    * Answer: n. Since, the matrices are square our input size is actually only n even though each element has n^2 entries.
  - Lets build a table for the cost and count of each line in our algorithm (i.e. our time $T(n)$).

Line	Cost	Count
1	c_1	1
2	c_2	t
3	c_3	t
4	c_4	t
5	c_5	1

  - What is t?
    * It depends on the value that n is currently (i.e. the loop control variable).
    * What part of the loop updates the loop control and how is it updated?
    * Notices that each time through the loop, n is divide by two. Repeating the process results in dividing n by the t-th power of two.
    * This means we perform exactly $\frac{n}{2^t} = 1$ iterations so what is t?
      · Let's use those logarithms!

$$\frac{n}{2^t} = 1$$
$$n = 2^t$$
$$\lg n = t$$

    * Technically, we do not require that n be an exact power of two so $t = \lfloor \lg n \rfloor + 1$. Notice $(\lfloor \lg n \rfloor + 1) \in \Theta(\lg n)$. We will therefore use $t = \lg n$ in practice.

- What is the running time of this algorithm on an input of size n?

Challenge Problem: Common Dictionary Algorithms

Write the following standard data structure functions using Python. The input for each should be a list `A` (along with any other information you may need). Determine the (tight) Big-Oh of each of them based on n, the number of elements currently in the array. If you decide to do something beyond the trivial for each, justify in words why the Big-Oh is what you have claimed based on the construction of the data structure.

- `push` and `pop` (treat it like a stack)
- `enqueue` and `dequeue` (treat it like a queue)
- `removeDuplicates` (turn it into a set)
- `insert` (treat it as a binary search tree, assume its initial configuration is already a valid BST)

Challenge Problem: Matrix Multiplication

- The matrix multiplication problem is as follows:

  Definition 8 (Matrix Multiplication Problem)**.**
  Input: *An $n \times n$ matrix A and an $n \times n$ matrix B.*
  Output: *The product, C, of A and B.*

- How do you multiply matrices on paper?
- Here is the formal algorithm

```
#Input: Two n x n matrices A and B
#Output Matrix C = AB
def MatrixMultiply(A, B, n):
    # Initialize C, an n x n matrix full of 0s
    C = [[0 for a in range(n)] for b in range(n)]       # Line 1
    for i in range(0, n-1):                             # Line 2
        for j in range(0, n-1):                         # Line 3
            C[i][j] = 0                                 # Line 4
            for k in range(0, n-1):                     # Line 5
                C[i][j] = C[i][j] + A[i][k] * B[k][j]   # Line 6
    return C                                            # Line 7
```

- Using the formal strategy from the book, what is our input size?
- What is the running time function ($T(n)$) of this algorithm? Defend your answer.
- What is the Big-Oh of the running time of this algorithm? Defend your answer.