CS 2500: Algorithms

Lecture 3: Asymptotic Notation (Big-Theta) and Analysis of Non-Recursive Algorithmms

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

August 27, 2024

Asymptotic Notation: Big-Theta(Θ)

Big-Theta: Let f and g be functions from the integers or the real numbers to the real numbers. The function $f(n)$ is $\Theta(g(n))$ if there exist positive constants C_1, C_2, and k such that:

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \text{ for all } n > k$$

- Gives **average-case** time complexity of an algorithm.
- More precise than Big-O and Big-Omega: $f(n) = \Theta(g(n))$ if and only if (iff) $g(n)$ is both an upper and lower bound on $f(n)$. (**Homework: Why?)**

Example 1: Sum of n numbers

Algorithm Sum(a, n)

```
s ← 0
for i ← 1 to n do
    s ← s + a[i]
end for
return s
```

Example 1: Sum of n numbers

Algorithm Sum(a, n)

```
1: s ← 0
2: for i ← 1 to n do
3:     s ← s + a[i]
4: end for
5: return s
```

Time Complexity Analysis:

- **Line 1:** Initialization `s := 0.0;` takes constant time, $\Theta(1)$.
- **Line 2:** The `for` loop runs n times, so it contributes $\Theta(n)$.
- **Line 3:** Inside the loop, `s := s + a[i];` is executed n times, contributing $\Theta(n)$.
- **Line 4:** The `return` statement is a constant time operation, $\Theta(1)$.

Example 1: Sum of n numbers

Algorithm Sum(a, n)

```
s ← 0
for i ← 1 to n do
    s ← s + a[i]
end for
return s
```

Total Time Complexity:

- Adding up the contributions: $\Theta(1) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.
- Therefore, the overall time complexity is $\Theta(n)$, meaning the algorithm's runtime grows linearly with the input size n.

Space Complexity: $\Theta(1)$

Example 2: Matrix addition

Algorithm Add(a, b, c, m, n)

1: **for** $i \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 1$ to n **do**
3:         $c[i,j] \leftarrow a[i,j] + b[i,j]$
4:     **end for**
5: **end for**

Example 2: Matrix addition

Algorithm Add(a, b, c, m, n)

1: **for** $i \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 1$ to n **do**
3:         $c[i,j] \leftarrow a[i,j] + b[i,j]$
4:     **end for**
5: **end for**

Time Complexity Analysis:

- **Line 1:** The outer `for` loop runs m times, contributing $\Theta(m)$.
- **Line 2:** The inner `for` loop runs n times for each iteration of the outer loop, contributing $\Theta(mn)$.
- **Line 3:** Inside the inner loop, the addition operation $c[i,j] \leftarrow a[i,j] + b[i,j]$ is executed mn times, contributing $\Theta(mn)$.

Example 2: Matrix addition

Algorithm Add(a, b, c, m, n)

```
for i ← 1 to m do
    for j ← 1 to n do
        c[i, j] ← a[i, j] + b[i, j]
    end for
end for
```

Total Time Complexity:

- Adding up the contributions: $\Theta(m) + \Theta(mn) + \Theta(mn) = \Theta(mn)$.
- Overall: $\Theta(mn)$. The algorithm's runtime grows with the product of the input dimensions m and n.

Space Complexity: $\Theta(m \times n)$ for matrix c.

Example 3: Fibonacci Series

Algorithm Fibonacci(n)

```
if n ≤ 1 then
    write(n);
else
    fnm2 ← 0;  fnm1 ← 1;
    for i ← 2 to n do
        fn ← fnm1 + fnm2;
        fnm2 ← fnm1;  fnm1 ← fn;
    end for
    write(fn);
end if
```

Example 3: Fibonacci Series

Time Complexity Analysis:

- **Lines 1-2:** The check `if (`$n \leq 1$`)` and `write(`n`)` are constant time operations, $\Theta(1)$.
- **Line 4:** Initialization of $fnm2$ and $fnm1$ takes constant time, $\Theta(1)$.
- **Line 5:** The `for` loop runs from 2 to n, so it executes $(n-1)$ times, contributing $\Theta(n)$.
- **Lines 6-7:** Each iteration of the loop involves constant time operations, hence the total contribution of these lines is $\Theta(n)$.
- **Line 9:** `write(`fn`)` operation takes constant time, $\Theta(1)$.

Total Time Complexity:

- Adding up the contributions:
  $\Theta(1) + \Theta(1) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.
- Overall: $\Theta(n)$.

Space Complexity: $\Theta(1)$

Example 4: Magic Square

- **Magic Square**. Problem from recreational mathematics.
- A square array of numbers, usually positive integers, is called a magic square if the sums of the numbers in each row, each column, and both main diagonals are the same.
- We will look at an algorithm for creating an $n \times n$ magic square.

Example 4: Magic Square

```
Algorithm Magic(n)
// Create a magic square of size n, n being odd.
{
    if ((n mod 2) = 0) then
    {
        write ("n is even"); return;
    }
    else
    {
        for i := 0 to n − 1 do  // Initialize square to zero.
            for j := 0 to n − 1 do  square[i, j] := 0;
        square[0, (n − 1)/2] := 1; // Middle of first row
        // (i, j) is the current position.
        j := (n − 1)/2;
        for key := 2 to n^2 do
        {
            // Move up and left. The next two if statements
            // may be replaced by the  mod  operator if
            // −1 mod n has the value n − 1.
            if (i ≥ 1) then k := i − 1; else k := n − 1;
            if (j ≥ 1) then l := j − 1; else l := n − 1;
            if (square[k, l] ≥ 1) then i := (i + 1) mod n;
            else // square[k, l] is empty.
            {
                i := k; j := l;
            }
            square[i, j] := key;
        }
        // Output the magic square.
        for i := 0 to n − 1 do
            for j := 0 to n − 1 do write (square[i, j]);
    }
}
```

Example 4: Magic Square

Time Complexity Analysis:

- **Line 4-7:** If n is even, the algorithm terminates early, resulting in $\Theta(1)$ time.
- **Line 10-13:** Initializing the square matrix of size $n \times n$ takes $\Theta(n^2)$ time.
- **Line 14:** Initializing the middle element of the first row takes constant time, $\Theta(1)$.
- **Line 15-28:** The loop runs $n^2 - 1$ times, and each iteration involves constant time operations, contributing $\Theta(n^2)$.
- **Line 30-32:** Outputting the magic square requires iterating over the $n \times n$ matrix, which takes $\Theta(n^2)$ time.

Total Time Complexity:

- Adding up the contributions: $\Theta(n^2) + \Theta(1) + \Theta(n^2) + \Theta(n^2) = \Theta(n^2)$.
- Overall: $\Theta(n^2)$.

Space Complexity: $\Theta(n^2)$

Example 5: Maximum Element in a List

Problem: Find the value of the largest element in a list of n numbers.

Algorithm MaxElement(a)

```
maxval ← a[0]
for i ← 1 to n − 1 do
    if a[i] > maxval then
        maxval ← a[i]
    end if
end for
return maxval
```

Example 5: Maximum Element in a List

Time Complexity Analysis:

- **Line 1:** Initializing *maxval* with the first element takes constant time, $\Theta(1)$.
- **Line 2:** The `for` loop runs $n - 1$ times, so its contribution is $\Theta(n)$.
- **Line 3-6:** Each iteration checks if the current element is greater than *maxval* and updates *maxval* if necessary. This comparison and possible assignment are constant time operations, $\Theta(1)$, but since they run inside the loop, their total contribution is $\Theta(n)$.
- **Line 7:** The return statement is a constant time operation, $\Theta(1)$.

Example 5: Maximum Element in a List

Total Time Complexity

- Adding up the contributions from all lines:

$$\Theta(1) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$$

- Therefore, the overall time complexity is $\Theta(n)$.
- This means the algorithm's runtime grows linearly with the size of the input array n.

Space Complexity: $\Theta(1)$

Example 6: Check Unique Element in a List

Problem: Check whether all the elements in a given array of n elements are distinct.

Algorithm UniqueElements(a)

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if a[i] = a[j] then
            return false
        end if
    end for
end for
return true
```

Example 6: Check Unique Element in a List

Worst-Case Input:

- The worst-case occurs when the algorithm does not exit the loop prematurely.
- Two kinds of worst-case inputs:
  1. Arrays with no equal elements.
  2. Arrays in which the last two elements are the only pair of equal elements.
- In these cases, one comparison is made for each repetition of the innermost loop.

Question: What is the number of comparisons for a fixed i?

Example 6: Check Unique Element in a List

What is the number of comparisons for a fixed i?
Answer: $n - i - 1$

- The `UniqueElements` algorithm compares each element $a[i]$ with every subsequent element in the array.
- For a fixed i, the algorithm compares $a[i]$ with elements $a[i+1], a[i+2], \ldots, a[n-1]$.
- The elements that come after $a[i]$ start at index $i+1$ and end at index $n-1$.
- The total number of elements after $A[i]$ is: $(n-1) - (i+1) + 1 = n - i - 1$
- Therefore, the number of comparisons for a fixed i is $n - i - 1$.

Example 6: Check Unique Element in a List

Total Number of Comparisons in the Worst Case:

$$\begin{aligned}
f_{\text{worst}}(n) &= \sum_{i=0}^{n-2}(n-1-i) \\
&= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i \\
&= (n-1) \times (n-1) - \frac{(n-2)(n-1)}{2} \\
&= \frac{2(n-1)^2 - (n-2)(n-1)}{2} \\
&= \frac{n^2 - n}{2} \\
&= \frac{n(n-1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)
\end{aligned}$$

Example 6: Check Unique Element in a List

Summary:

- The `UniqueElements` algorithm checks whether all elements in an array are distinct by comparing every pair of elements.
- The worst-case time complexity is $\Theta(n^2)$, which occurs when the algorithm must compare all possible pairs of elements.
- This quadratic time complexity arises because the algorithm examines every pair of elements in the worst case.

Space Complexity: $\Theta(1)$

Can we improve the time complexity any further? How?

Example 6: Check Unique Element in a List

Improved Algorithm: Using a Hash Set

- Use a hash set to track the elements we've seen as we iterate through the array.
- For each element, check if it already exists in the hash set.
- If it does, the array has duplicates; otherwise, continue.

Algorithm UniqueElements2(a)

```
Initialize an empty hash set seen
for i ← 0 to n − 1 do
    if a[i] is in seen then
        return false
    else
        Insert a[i] into seen
    end if
end for
return true
```

Example 6: Check Unique Element in a List

Time Complexity:

- Hash set operations (insertion and lookup) take $\Theta(1)$ time on average.
- Iterating through the array takes $\Theta(n)$ time.

Total Time Complexity: $\Theta(n)$
Space Complexity: $\Theta(n)$ for the hash set.

Example 7: Matrix Multiplication

Problem:

- Given two $n \times n$ matrices a and b, compute their product $c = ab$.
- The product matrix c is also an $n \times n$ matrix, where each element $c[i, j]$ is the dot product of the i-th row of a and the j-th column of b.

Example 7: Matrix Multiplication

Algorithm MatrixMultiplication(a,b)

for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    $c[i,j] \leftarrow 0.0$
    for $k \leftarrow 0$ to $n-1$ **do**
      $c[i,j] \leftarrow c[i,j] + a[i,k] \times b[k,j]$
    end for
  end for
end for
return c

Example 7: Matrix Multiplication

Time Complexity:

- The algorithm consists of three nested loops:
  - Outer loop over i runs n times.
  - Middle loop over j runs n times for each i.
  - Inner loop over k runs n times for each pair (i, j).
- Each iteration of the innermost loop involves a constant-time operation (multiplication and addition).
- Total number of operations: $n \times n \times n = n^3$.

Total Time Complexity: $\Theta(n^3)$
Space Complexity: $\Theta(n^2)$ for matrix ‘c‘

Example 8: Counting Binary Digits

Problem:

- Given a positive decimal integer n, determine the number of binary digits (bits) in its binary representation.
- Example:
  - The binary representation of 13 is 1101, which has 4 digits.
  - The binary representation of 8 is 1000, which has 4 digits.
  - The binary representation of 1 is 1, which has 1 digit.

Algorithm Binary(n)

$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
end while
return $count$

Example 8: Counting Binary Digits

Key Insight: Why the Algorithm Works

Binary Representation and Powers of 2:

- Binary is a base-2 system: each digit represents a power of 2.
- Example: $1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$.
- The number of binary digits is the number of powers of 2 needed to represent n.

Dividing by 2:

- Each division by 2 shifts the binary digits to the right, reducing the number of digits by 1.
- The algorithm counts how many times n can be divided by 2 until it becomes 1.

Example 8: Counting Binary Digits

Time Complexity Analysis

Loop Analysis:

- The loop continues until n is no longer greater than 1.
- Each iteration of the loop divides n by 2.
- The number of iterations equals the number of times n can be halved before it becomes 1.

Number of Divisions:

- The number of divisions is equal to the number of binary digits in n.
- This is $\lfloor \log_2 n \rfloor + 1$, where $\log_2 n$ is the number of times 2 can be multiplied to reach n.

Example 8: Counting Binary Digits

Time Complexity:

- Each iteration involves constant-time operations: incrementing `count` and dividing n by 2.
- Since the loop runs approximately $\log_2 n$ times, the overall time complexity is $\Theta(\log n)$.

Summary:

- The `Binary` algorithm efficiently counts the number of binary digits in n with a time complexity of $\Theta(\log n)$.
- This logarithmic growth makes the algorithm highly efficient, even for large values of n.

Space Complexity: $\Theta(1)$

Example 9: Mystery

Algorithm Mystery(n)

$S \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $S \leftarrow S + i \times i$
end for
return S

1. What does this algorithm compute?
2. What is its basic operation?
3. How many times is the basic operation executed?
4. What is the efficiency class of this algorithm?
5. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Example 9: Mystery

1. **What does this algorithm compute?**
   - The algorithm computes the sum of the squares of the first n natural numbers.
   - Specifically, it calculates:

$$S = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

   - This can be written as:

$$S = \sum_{i=1}^{n} i^2$$

Example 9: Mystery

2. What is its basic operation?

- The basic operation of the algorithm is the multiplication $i \times i$, which computes the square of the integer i.

Example 9: Mystery

3. How many times is the basic operation executed?

- The basic operation (squaring of i) is executed once for each iteration of the ‘for‘ loop.
- The loop runs from $i = 1$ to $i = n$, so the basic operation is executed n times.

Example 9: Mystery

4. What is the efficiency class of this algorithm?

- The algorithm has a time complexity of $\Theta(n)$.
- The loop runs n times, and each iteration involves a constant amount of work (multiplication and addition).

5. Can we improve the algorithm?

- The sum of the squares of the first n natural numbers has a known closed-form formula:

$$S = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

- This formula allows us to compute the sum in constant time, $\Theta(1)$, rather than iteratively summing the squares.

Improved Algorithm: ImprovedMystery(n)

1: $S \leftarrow n \times (n+1) \times (2n+1)/6$
2: **return** S

Efficiency Class: $\Theta(1)$

Real-world Example: RSA Encryption

RSA Encryption and Decryption

- RSA is a widely used public-key cryptosystem for secure data transmission.
- Encryption: Given a message M, compute the ciphertext C as:

$$C = M^e \mod n$$

- Decryption: To recover M, compute:

$$M = C^d \mod n$$

- Here, e is the public key exponent, and d is the private key exponent.

Question: How well can we implement this?

Example 10: Computing x^n

Task. The problem is to compute x^n for any real number x and integer $n \geq 0$.

Naive Algorithm:

Algorithm NaiveExponentiate(x, n)

```
power ← x
for i ← 1 to n − 1 do
    power ← power × x
end for
return power
```

Time Complexity: $\Theta(n)$

Example 10: Computing x^n

Improved Approach: Repeated Squaring. A better approach is to employ the "repeated squaring" trick.

Case 1: When n is a power of 2:

- If $n = 2^k$ for some integer k, compute x^n using the following algorithm:

Algorithm PowerOf2Exponentiate(x, k)

```
power ← x
for i ← 1 to k do
    power ← power^2
end for
return power
```

Time Complexity: $\Theta(\log n)$

Example 10: Computing x^n

Case 2: General case: When n is not a power of 2:

- **Key Idea:** Represent n in binary form.
- Let $b_k b_{k-1} \ldots b_1 b_0$ be the binary representation of n. This means:

$$n = \sum_{q=0}^{k} b_q 2^q$$

- Therefore,

$$x^n = x^{\sum_{q=0}^{k} b_q 2^q} = (x^{2^0})^{b_0} \times (x^{2^1})^{b_1} \times \cdots \times (x^{2^k})^{b_k}$$

Example 10: Computing x^n

Exponentiate Algorithm. The general case is handled by the following algorithm:

Algorithm Exponentiate(x, n)

$m \leftarrow n$; $power \leftarrow 1$; $z \leftarrow x$
while $m > 0$ **do**
  while $(m \mod 2) = 0$ **do**
    $m \leftarrow m/2$; $z \leftarrow z^2$
  end while
  $m \leftarrow m - 1$; $power \leftarrow power \times z$
end while
return $power$

Example 10: Computing x^n

Time Complexity Analysis

Outer While Loop:

- The loop runs as long as $m > 0$. Each iteration either halves m or decreases it by 1.
- Since m decreases by at least a factor of 2 in each iteration, the number of iterations is at most $\Theta(\log n)$.

Inner While Loop:

- The inner loop also runs $\Theta(\log n)$ times, squaring z at each step.

Final Time Complexity: $\Theta(\log n)$

Example 10: Computing x^n

Summary:

- The repeated squaring method significantly reduces the number of multiplications needed to compute x^n.
- The naive method takes $\Theta(n)$ time, whereas the improved method takes $\Theta(\log n)$ time.
- The algorithm leverages binary representation to break down the problem into smaller steps, making it efficient for large n.

Applications: This method is widely used in cryptography and other fields requiring fast exponentiation.