Part 5.
Computational Complexity (3)

CS 200 Algorithms and Data Structures

Outline
- Recurrence Relations
- Divide and Conquer
- Understanding the Complexity of Algorithms

Recurrence Relations: An Overview

- What is a recurrence relation?
  - A recursively defined sequence

- Example
  - Arithmetic progression: \(a, a+d, a+2d, \ldots, a+nd\)
    - \(a_0 = a\)
    - \(a_n = a_{n-1} + d\)

Recurrence Relations: Formal Definition

A recurrence relation for the sequence \(\{a_n\}\) is an equation that expresses \(a_n\) in terms of one or more of the previous terms of the sequence, namely, \(a_0, a_1, \ldots, a_{n-1}\), for all integers \(n\) with \(n \geq n_0\) where \(n_0\) is a nonnegative integer.

- Sequence = Recurrence relation + Initial conditions ("base case")

- Example: \(a_n = 2a_{n-1} + 1\), \(a_1 = 1\)

Compound Interest

- You deposit $10,000 in a savings account that yields 10% yearly interest. How much money will you have after 30 years? (\(b\) is balance, \(r\) is rate)
  \[b_n = b_{n-1} + rb_{n-1} = (1 + r)b_0\]
How to Approach Recursive Relations

Recursive Functions

Sequence of Values

• \( f(0) = 0 \) (base case)
• \( f(n) = f(n-1) + 2 \) for \( n > 0 \) (recursive part)

Closed Form (solution, explicit formula)

Solving recurrence relations (1)

\[ a_0 = 2; \quad a_n = 3a_{n-1}, \quad n > 0 \]

(1) What is the recursive function?
(2) What is the sequence of values?

Hint 1: Solve by substitution

Solving recurrence relations (2)

\[ a_0 = 2; \quad a_n = 3a_{n-1} + 2 \quad \text{for} \quad n > 0 \]

(1) What is the recursive function?
(2) What is the sequence of values?

Hint 1: Solve by substitution

Hint 2: Use formula for summing geometric series

\[ 1 + r + r^2 + \ldots + r^n = (r^{n+1} - 1)/(r - 1) \]

if \( r \neq 1 \)

Solving recurrence relations (3)

\[ a_0 = 1; \quad a_n = a_{n-1} + n \quad \text{for} \quad n > 0 \]

Hint 1: Solve by substitution

Modeling with Recurrence

• Suppose that the number of bacteria in a colony triples every hour
  – Set up a recurrence relation for the number of bacteria after \( n \) hours have elapsed.
  – 100 bacteria are used to begin a new colony.

Linear Recurrence Relations

A linear homogeneous recurrence relation of degree \( k \) with constant coefficients is a recurrence relation of a form

\[ a_0 = c_1a_{n-1} + c_2a_{n-2} + \ldots + c_ka_{n-k} \]

where, \( c_1, c_2, c_3, \ldots, c_k \) are real numbers and \( c_k \) is not 0.
Is this linear homogeneous recurrence relation?

- \( f_n = f_{n-1} + f_{n-2} \)
- \( a_n = a_{n-1} + a_{n-2} \)

- Modeling of problems
- They can be systematically solved.

What is the solution of the recurrence relation

- \( a_n = a_{n-1} + 2a_{n-2} \) with \( a_0 = 2 \) and \( a_1 = 7 \)?

Theorem 1

Let \( c_1 \) and \( c_2 \) be. Suppose that \( r_2 - c_1 r - c_2 = 0 \) has two distinct roots \( r_1 \) and \( r_2 \). Then the sequence \( \{a_n\} \) is a solution of the recurrence relation

\[ a_n = c_1 a_{n-1} + c_2 a_{n-2} \] if and only if

\[ a_n = \alpha_1 r_1^n + \alpha_2 r_2^n \] for \( n = 0, 1, 2, \ldots \) where \( \alpha_1 \) and \( \alpha_2 \) are constants. (Rosen 6th edition, section 7.2 theorem 1)

Outline

- Recurrence Relations
- Divide and Conquer
- Understanding the Complexity of Algorithms

Divide-and-Conquer

Basic idea:

- Take large problem and divide it into smaller problems until problem is trivial, then combine parts to make solution.

Recurrence relation for the number of steps required:

\[ f(n) = a \cdot f(n/b) + g(n) \]

- \( n/b \) : the size of the sub-problems solved
- \( a \) : number of sub-problems
- \( g(n) \) : steps necessary to combine solutions to sub-problems

Example: Binary Search

```java
public static int binSearch(int[] myArray, int first, int last, int value) {
    // returns the index of value or -1 if not in the array
    int index;
    if (first > last) { index = -1; }
    else {
        int mno = (first + last)/2;
        if (value == myArray[mno]) { index = mno; }
        else if (value < myArray[mno]) {
            index = binSearch(myArray, first, mno-1, value);
        }
        else {
            index = binSearch(myArray, mno+1, last, value);
        }
        return index;
    }
}
```

What are \( a \), \( b \), and \( g(n) \)?

\[ f(n) = a \cdot f(n/b) + g(n) \]
Estimating big-O (Master Theorem)

Let \( f \) be an increasing function that satisfies
\[
f(n) = a \cdot f(n/b) + c \cdot n^d
\]
whenever \( n = b^k \), where \( k \) is a positive integer, \( a \gg 1 \), \( b \) is an integer \( > 1 \), and \( c \) and \( d \) are real numbers with \( e \) positive and \( d \) nonnegative. Then
\[
f(n) =
\begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{d+e}) & \text{if } a > b^d 
\end{cases}
\]

From section 8.3 (for 6th Edition 7.3) in Rosen

Example: Binary Search using the Master Theorem

\[
f(n) = a f(n/b) + n^d
\]
for the binary search algorithm
\[
f(n) =
\begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{d+e}) & \text{if } a > b^d 
\end{cases}
\]

Therefore, \( d = 0 \) (to make \( n^d \) a constant), \( b = 2 \), \( a = 1 \).

It satisfies the second condition of the Master theorem.

So, \( f(n) = O(n^d \log n) = O(n^d n) = O(\log n) \)

Step-by-step Master Theorem

- Modeling the divide-conquer with a recurrence relation.
- \( f(n) \) is a recurrence function representing the number of operations with size of input \( n \).

Step-by-step Master Theorem

```plaintext
Method_A(array)
Method_A(left_half); Method_A(right_half); Complete_the_Method(array);

Method A

Method A

Method A

Method A

Method A
```

Step-by-step Master Theorem

```plaintext
Method_C(array)
Method_C(first_one_third); Method_C(second_one_third); Method_C(last_one_third); Complete_method(array);

Method C

Method C

Method C

Method C

Method C

Method C
```

Step-by-step Master Theorem

```plaintext
Method_B(array)
Method_B(possible_half);
Do_nothing_Method(array);

Method B

Method B
```

Step-by-step Master Theorem

```plaintext
Method_B(array)
Method_B(left_half);
Method_B(right_half); Complete_the_Method(array);

Method B

Method B
```

Step-by-step Master Theorem

```plaintext
Method_B(array)
Method_B(left_half);
Method_B(right_half); Complete_the_Method(array);

Method B

Method B
```
Goal 1: What is the complexity per level?
Goal 2: What is the total complexity?

“Method” is called with input size n. “Method” calls “Method” recursively a times “Method” calls “Method” recursively with input size n/b.

“Method” includes software fragment on top of calling methods recursively, and it takes g(n) operations.

Therefore, in LEVEL 1, “Method” requires, 
\[ f(n) = a f(n/b) + g(n) \]

This is also true for LEVEL 2
\[ f(n) = a f(n/b) + g(n) \]

This is also true for LEVEL i
\[ f(n) = a f(n/b) + g(n) \]
More generally,

Let $f$ be an increasing function that satisfies

$$f(n) = a f(n/b) + g(n) = a \cdot f(n/b) + n^c$$ if $n > 1$

$$f(n) = d$$ if $n = 1$

whenever $n = a^k$, where $a$ is a positive integer, $a \geq 1$, $b$ is an integer > 1, and $c$ and $d$ are real numbers with $c$ positive and $d$ nonnegative.

Base case

- We will set $d = 1$
  - The bottom level of the tree is equally well computed.
  - Base case
  - It is straightforward to extend the proof for the case when $d \neq 1$.

Goal 1. complexity per level

- Let’s think about the recursion tree.
- There will be $\log_b n$ levels.
- At each level, the number of subproblems will be multiplied by $a$.
- Therefore, the number of subproblems at level $i$ will be $a^i$.
- Each subproblem at level $i$ is a problem of size $(n/b)^i$.
- A subproblem of size $(n/b)^i$ requires $(n/b)^i$ additional work.

The total number of units of work on level $i$

- $a^i (n/b)^i = n^c (a/b)^i$
- $= n^c (a/b)^i$

For the level $i$, the work per level is decreasing, constant, or increasing exactly when $(a/b)^i$ is decreasing, constant, or increasing.

Observation

- Now, observe that,
  $$(a/b)^i = 1$$
  $$a = b^c$$

- Therefore, the relations,
  (1) $a < b^c$ (2) $a = b^c$, (3) $a > b^c$

are the conditions deciding the types of the growth function.

Goal 2: Bounding $f(n)$ in the different cases.

- In general, we have that the total work done is,
  $$\sum_{i=0}^{\lfloor \log_b n \rfloor} (a/b)^i = n^c \sum_{i=0}^{\lfloor \log_b n \rfloor} (a/b)^i$$

(1) $a < b^c$
(2) $a = b^c$
(3) $a > b^c$
**Case 1.** $a < b^c$

\[
\sum_{i=0}^{\log_b n} (a/b^c)^i = n^c \sum_{i=0}^{\log_b n} (a/b^c)^i
\]

- $n^c$ times a geometric series with a ratio of less than 1.
- First item is the biggest one.

\[n^c \sum_{i=0}^{\log_b n} (a/b^c)^i = O(n^c)\]

**Case 2.** $a = b^c$

\[
\sum_{i=0}^{\log_b n} (a/b^c)^i = n^c \sum_{i=0}^{\log_b n} (a/b^c)^i
\]

- $(a/b^c) = 1$
- \(n^c(1+1+\ldots+1) = n^c \log_b n\)

\[n^c \sum_{i=0}^{\log_b n} (a/b^c)^i = O(n^c \log_b n)\]

**Case 3.** $a > b^c$

\[
\sum_{i=0}^{\log_b n} (a/b^c)^i = n^c \sum_{i=0}^{\log_b n} (a/b^c)^i
\]

- $(a/b^c) > 1$
- Therefore the largest term is the last one.

\[n^c (a/b^c)^{\log_b n} = n^c (a^{\log_b n} / (b^c)^{\log_b n}) = n^c (a^{\log_b n} / n^c) = n^c (n^{\log_b n} / n^c) = n^{\log_b n} \]

\[n^c \sum_{i=0}^{\log_b n} (a/b^c)^i = O(n^{\log_b n})\]

**Complexity of MergeSort with Master Theorem (1/2)**

- Mergesort splits a list to be sorted twice per level.
- Uses fewer than $n$ comparisons to merge the two sorted lists of $n/2$ items each into one sorted list.
- Function $M(n)$ satisfying the divide-and-conquer recurrence relation
  - $M(n) = 2M(n/2) + n$

**Complexity of MergeSort with Master Theorem (2/2)**

For the mergesort algorithm

\[f(n) = a f(n/b) + n^c \quad \text{for} \quad f(n) = O(n^{\log_b a}), \quad \text{if} \quad a < b^c\]

Therefore, $c = 1$, $b = 2$, $a = 2$.

\[b^c = 2^c = 2\]

It satisfies the second condition of the Master theorem.

So, $f(n) = O(n \log n)$

\[= O(n \log n)\]

\[= O(n \log n)\]

**Quicksort?**
### The Closest-Pair Problem

- Consider the problem of determining the **closest pair of points** in a set of $n$ points $(x_1, y_1)\ldots(x_n, y_n)$ in the plain, where the distance between two points $(x_i, y_i)$ and $(x_j, y_j)$ is the usual Euclidean distance.

\[
\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}
\]

### Solution (1/5)

- Calculate the distance between every pair of points and then find the smallest of these distances
  - It will take $O(n^2)$

### Solution (2/5)

- Assume that we have $n$ points such that $n = 2^k$.
- When $n = 2$, the distance between two points is the minimum distance.
- We sort the points in order of increasing $x$ coordinates, and also sort the points in order of increasing $y$ coordinates using mergesort.
  - It will take $O(n \log n)$
- Using sorted list by $x$ coordinate, divide the group of points into two (**Right** and **Left**)

### Solution (3/5)

![Diagram of the closest pair problem with points and regions](image)

**Closest Pair**

### Solution (4/5)

- Find the closest pair within $R$ and $L$.
- Merge the region $R$ and $L$ by examining the points on the boarder area.
  - Find closest pair between pairs from $R$ and $L$. ($d = \min(d_r, d_l)$)
  - Check the points within the strip of the maximum distance of $d$ from the border.
  - Sort the points within the strip with associated $y$ coordinate.
  - Beginning with a point in the strip with the smallest $y$ coordinate.

### Solution (5/5)

- We need at most 7 comparisons for each of the points in the strip.

![Diagram of the closest pair problem with points and regions](image)
Analysis

• We find the function \( f(n) \) satisfying the recurrence relations

\[
f(n) = 2f(n/2) + 7n
\]

where \( f(2) = 1 \)

By the master theorem, \( f(n) = O(n \log n) \)

Outline

• Recurrence Relations
• Divide and Conquer
• Understanding the Complexity of Algorithms

Understanding the Complexity of Algorithm

• Commonly used terminology for the complexity of algorithms

\( O(1) \) : Constant complexity
\( O(\log n) \) : Logarithmic complexity
\( O(n) \) : Linear complexity
\( O(n \log n) \) : Linearithmic complexity
\( O(n^b) \) : Polynomial complexity
\( O(b^n) \), where \( b > 1 \) : Exponential complexity
\( O(n!) \) : Factorial complexity

Tractability

• A problem that is solvable using an algorithm with polynomial worst-case complexity is called \textit{tractable}.

• If estimation has high degree or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem.

Intractable problems and Unsolvable problems

• If the problem \textit{cannot be solved} using an algorithm with worst-case polynomial time complexity, such problems are called \textit{intractable}.

• If it can be shown that no algorithm exists for solving them, such problems are called \textit{unsolvable}.

Class NP and NP-Complete

• Problems for which a solution can be \textit{checked} in polynomial time are said to belong to the \textit{class NP}.
  
  – Tractable problems belong to class \textit{P}.

• \textit{NP-complete} problems are an important class of problems
  
  – If any of these problems can be solved by a polynomial worst-case time algorithm then ...
    
    • All problems in the class \textit{NP} can be solved by polynomial worst-case time algorithms.
### NP-Hard

- **NP-Hard** problems are *at least as hard as the hardest problems in NP*

### Next Reading

- Section 11.1, 11.2 in Prichard