CSCE 222
Discrete Structures for Computing

Algorithms

Dr. Philip C. Ritchey

Introduction

- An **algorithm** is a finite sequence of precise instructions for performing a computation or for solving a problem.
  - Searching
  - Sorting
  - Optimizing
  - Etc.

Example

- Describe an algorithm for finding the maximum value in a list (finite sequence) of integers.

- Solution
  - Set the temporary maximum to the first element of the list.
  - For each remaining element in the list, compare it to the temporary maximum. If it is larger, set the temporary maximum to this integer.
  - Return the temporary maximum as the answer.

Psuedocode

- **Psuedocode** is an intermediate between an English description and an implementation in a particular language of a an algorithm.
  - English is very high-level, not always well-suited to precise descriptions of algorithms
  - Programming languages are very precise, but can make algorithms hard to understand.

Psuedocode for Finding the Max

```
procedure max(a_1, a_2, …, a_n)
temp_max = a_1
for i=2 to n do
        if temp_max < a_i
        then temp_max = a_i
return temp_max
```

Properties of Algorithms

- **Input.**
  - Input values from a specified set.
- **Output**
  - Output values from a specified set. The solution to the problem.
- **Definiteness**
  - Steps are defined precisely.
- **Correctness**
  - Produces correct answer for every input.
- **Finiteness**
  - Terminate after a finite number of steps.
- **Effectiveness**
  - Each step performed in finite time.
- **Generality**
  - Works for all problems of the desired form.

Does the max-finding algorithm have all of these properties?

- **Input.**
  - A list of integers
- **Output**
  - The largest integer in the list.
- **Definiteness**
  - Assignments, finite loops, and comparisons all have precise definitions.
- **Correctness**
  - Yes. Informal proof: temp_max is updated every time a large value is seen; all values seen; therefore temp_max is the largest value in the list after the loop ends.
- **Finiteness**
  - Stops after seeing all elements of the list.
- **Effectiveness**
  - Assignments, finite loops, and comparisons all take finite time.
- **Generality**
  - Finds the maximum of any list of integers.

Search

- **Search**
  - Find a given element in a list. Return the location of the element in the list (index), or -1 if not found.
- **Linear Search**
  - Compare key (element being searched for) with each element in the list until a match is found, or the end of the list is reached.
- **Binary Search**
  - Compare key only with elements in certain locations. Split list in half at each comparison. *Requires list to be sorted.*

Linear Search

```
procedure linear_search (key , {a_1,…,a_n})
for index = 1 to n
        if a_i equals key
                return index
return -1
```

Binary Search

```
procedure binary_search (key , {a_1,…,a_n})
left = 1
right = n
while left < right
        middle = ⌊(left + right)/2⌋
        if key == a_middle, then return middle
        elseif key > a_middle, then left = middle + 1
        else right = middle
if key == a_left, then return left
return -1
```

Linear Search Exercise

- Write the numbers 1 to 20 on post-it notes.
  - 1 number per note.
- Randomly order the notes on the table.
- How many comparisons to find:
  - 7?
  - 13?
  - 1?
  - 20?

Binary Search Exercise

- Sort the notes in ascending order
- How many comparisons to find:
  - 7?
  - 13?
  - 1?
  - 20?

Sort

- **Sort:** put the elements of a list in ascending order
  - Example:
    - List: 7,2,1,4,5,9
    - Sorted List: 1,2,4,5,7,9
- Bubble Sort
  - Compare every element to its neighbor and swap them if they are out of order. Repeat until list is sorted.
- Insertion Sort
  - For each element of the unsorted portion of the list, insert it in sorted order in the sorted portion of the list.

Bubble Sort

procedure bubble_sort($\{a_1, \ldots, a_n\}$)

for i = 1 **to** n-1

    for j = 1 **to** n-i

        if $a_j > a_{j+1}$

        then, swap a_j and a_{j+1}

$\{a_1, \ldots, a_n\}$ is in sorted order.

Insertion Sort

```
procedure insertion_sort({a_1, ..., a_n})
for j = 2 to n
        i = 1
        while a_j > a_i
                i = i + 1
        m = a_j
        for k = 0 to j-i-1
                a_{j-k}=a_{j-k-1}
        a_i=m
{a_1, ..., a_n} is in sorted order.
```

Bubble Sort Exercise

- Order the notes on the table as follows:
  - 10, 2, 1, 5, 3, 9, 6, 4, 7, 8
- Sort them using Bubble Sort.
- How many comparisons and swaps did you use?
  - Don’t count condition checks in **for** loops.

Insertion Sort Exercise

- Order the notes on the table as follows:
  - 10, 2, 1, 5, 3, 9, 6, 4, 7, 8
- Sort them using Insertion Sort.
- How many comparisons and swaps did you use?
  - Don’t count condition checks in **for** loops.

Binary Insertion Sort Exercise

- Order the notes on the table as follows:
  - 10, 2, 1, 5, 3, 9, 6, 4, 7, 8
- Sort them using Binary Insertion Sort.
  - Use binary search, instead of linear search, when searching for the correct place to insert each number.
- How many comparisons and swaps did you use?
  - Don’t count condition checks in **for** loops.

The Growth of Functions

- The time required to solve a problem using a procedure depends on:
  - Number of operations used
    - Depends on the size of the input
  - Speed of the hardware and software
    - Does not depend on the size of the input
    - Can be accounted for using a constant multiplier
- The growth of functions refers to the number of operations used by the function to solve the problem.

Big-*O* Notation

- Estimate the growth of a function without worrying about constant multipliers or smaller order terms.
  - Do not need to worry about hardware or software used
- Assume that different operations take the same time.
  - Addition is actually much faster than division, but for the purposes of analysis we assume they take the same time.

Big-O

- Let f and g be functions from $\mathbb{Z}$ or $\mathbb{R}$, to $\mathbb{R}$.
- We say that $f(x)$ is $O\big(g(x)\big)$ if there are constants C and k such that $|f\ (x)|\ \le\ C|g(x)|$ whenever $x > k$.
  - "$f(x)$ is bounded above by $g(x)$"
  - "$f(x)$ grows slower than C$g(x)$, as x grows without bound"
  - Constants C and k are called *witnesses*.

Example: Max

- Let $f(n)$ be the number of operations to find the maximum value in a list of n elements.

procedure max(a_1, a_2, …, a_n)	- assign = depending on implementation, 1 or n op.	
temp_max = a_1	- assign = 1 op.	
for i=2 **to** n **do**	- assign + compare = 1+1 = 2 ops.	
if temp_max < a_i	- access + comparison = 1+1 = 2 ops.	(n-1) times
then temp_max = a_i	- access + assign = 1+1 = 2 ops.	(n-1) times
	- increment + compare = 1+1 = 2 ops.	(n-1) times
return temp_max	- return = 1 op.	

$$f(n) = 1 + 1 + 2 + (n - 1)(2 + 2 + 2) + 1$$
$$f(n) = 6n - 1$$

Example: Max

- Let $f(n)$ be the number of operations to find the maximum value in a list of n elements.
  - $f(n) = 6n - 1$
  - $f(n) \leq Cg(n),\ \forall n > k$
  - $6n - 1 \leq 6n,\ \forall n > 0$
  - Let $g(n) = n$
  - $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(n)}$. Witnesses: $\boldsymbol{C = 6,\ k = 0}$

Example: Sort

- Let $f(n)$ be the number of operations to sort a list of n elements.

procedure bubble_sort($\{a_1, \dots, a_n\}$)

for i = 1 **to** n-1

  for j = 1 **to** n-i

    if $a_j > a_{j+1}$

    then, swap a_j and a_{j+1}

- 1: assign
- 2: assign and compare in loop1
- $2(n-1)$: assign and compare in loop2
- $\sum_{i=1}^{n-1} 3(n-i)$: accesses and compare
- $\sum_{i=1}^{n-1} 3(n-i)$: assigns
- $\sum_{i=1}^{n-1} 2(n-i)$: increment and compare in loop 2
- $2(n-1)$: increment and compare in loop 1

$$f(n) = 1 + 2 + 2(n-1) + \sum_{i=1}^{n-1} 3(n-i) + \sum_{i=1}^{n-1} 3(n-i) + \sum_{i=1}^{n-1} 2(n-i) + 2(n-1)$$

$$f(n) = 4n^2 - 1$$

Example: Sort

- Let $f(n)$ be the number of operations to sort a list of n elements.
  - $f(n) = 4n^2 - 1$
  - $f(n) \leq Cg(n),\ \forall n > k$
  - $4n^2 - 1 \leq 4n^2, \forall n > 0$
  - Let $g(n) = n^2$
  - **$\boldsymbol{f(n)}$ is $\boldsymbol{O(n^2)}$. Witnesses: $\boldsymbol{C = 4,\ k = 0}$**

Big-O for Polynomials

- Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.
- Then, $f(x)$ is $O(x^n)$.
- Example: $f(x) = 5x^2 - 18x + 20$
  - $5x^2 - 18x + 20 \leq 5x^2 + 20$ for $x > 0$
  - $5x^2 + 20 \leq 5x^2 + 20x^2$ for $x > 1$
  - $5x^2 + 20x^2 = 25x^2 \leq Cg(x)$ for $x > 1$
  - Let $g(x) = x^2$
  - **$\boldsymbol{f(x)}$ is $\boldsymbol{O\left(x^2\right)}$. Witnesses: $\boldsymbol{C = 25}$, $\boldsymbol{k = 1}$**

Exercise

- Give a big-O estimate for the sum of the first n positive integers.

- Solution:
- $1 + 2 + \cdots + n \leq n + n + \cdots + n = n^2$
- $1 + 2 + \cdots + n$ is $O(n^2)$, $C = 1, k = 1$

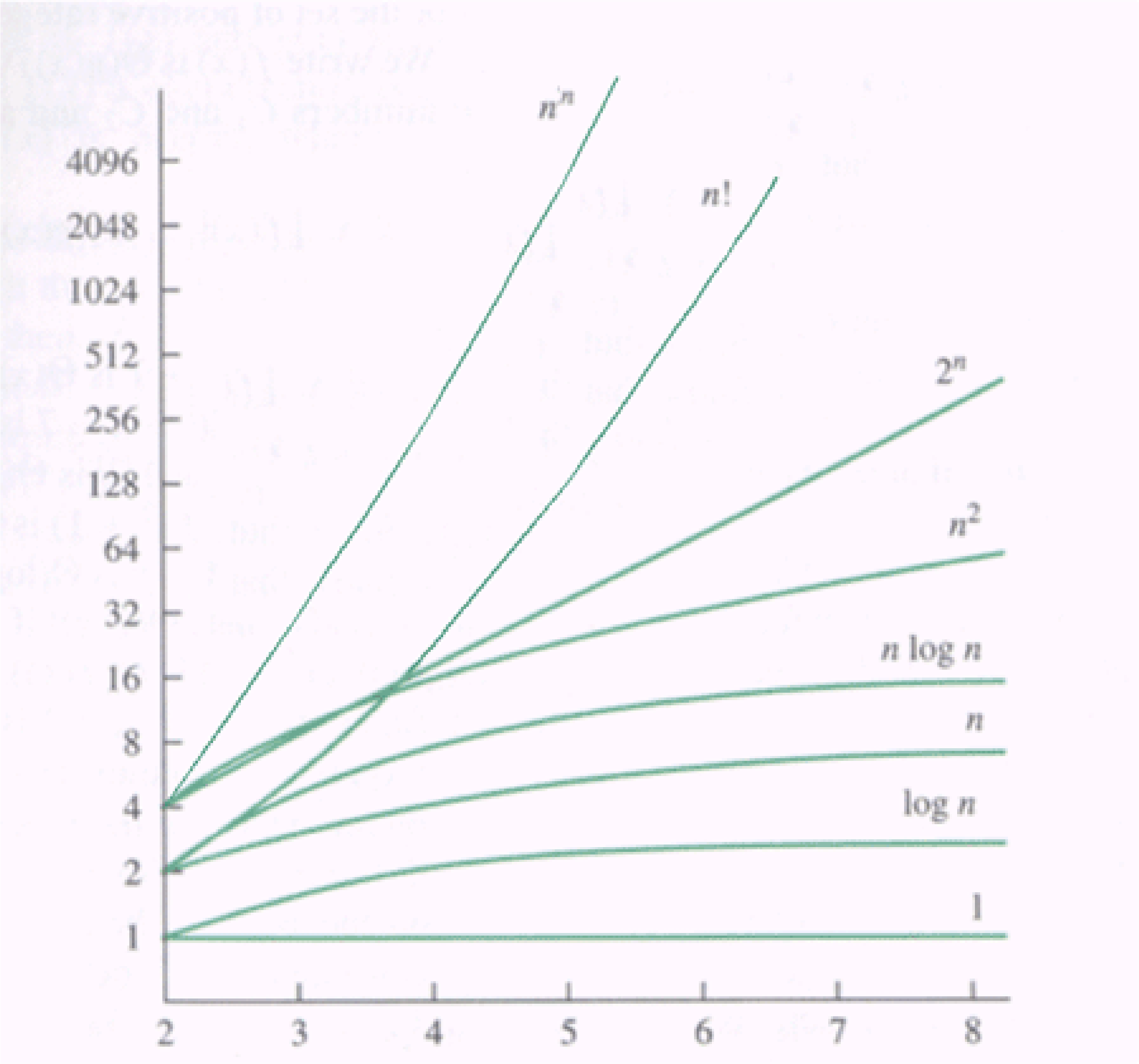
Exercise

- Give a big-O estimate for the factorial function, $\mathrm{f}(n) = n!$, and the logarithm of the factorial.

- Solution:
- $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \leq n \cdot n \cdot n \cdot \cdots \cdot n = n^n$
  - $n!$ is $O(n^n)$
- $\log(n!) \leq \log(n^n) = n \log n$
  - $\log(n!)$ is $O(n \log n)$

Basic Growth Functions

Constant:	$O(1)$
Logarithmic:	$O(\log n)$
Linear:	$O(n)$
Linearithmic:	$O(n \log n)$
Polynomial:	$O(n^c)$
Exponential:	$O(2^n)$
Factorial:	$O(n!)$



Useful Big-O Estimates

- n^c is $O\left(n^d\right)$, but n^d is **not** $O(n^c)$, $\ d > c > 1$
- $(\log_b n)^c$ is $O\left(n^d\right)$, but n^d is **not** $O\left((\log_b n)^c\right)$, $b > 1, c, d > 0$
- n^d is $O(b^n)$, but b^n is **not** $O\left(n^d\right)$, $d > 0, b > 1$
- b^n is $O(c^n)$, but c^n is **not** $O(b^n)$, $c > b > 1$

The Growth of Combinations of Functions

- Suppose $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - $(f_1 + f_2)(n)$ is $O(\max(g_1(n), g_2(n)))$
    - If $g_1(n) = g_2(n) = g(n)$, then $(f_1 + f_2)(n)$ is $O\big(g(n)\big)$
  - $(f_1 f_2)(n)$ is $O(g_1(n) g_2(n))$

Exercise

- Which of these functions is $O(x)$?
  - $f(x)$ = 10
    - $C = 1, k = 10$
  - $f(x) = 3x + 7$
    - $C = 4, k = 7$
  - $f(x) = x^2 + x + 1$
    - Not $O(x)$
  - $f(x) = 5 \log x$
    - $C = 5, k = 2$
  - $f(x) = \lfloor x \rfloor$
    - $C = 1, k = 0$
  - $f(x) = \left\lceil \frac{x}{2} \right\rceil$
    - $C = 1, k = 0$

Exercise

- Find the least integer c such that $f(n)$ is $O(n^c)$:
  - $f(n) = 2n^3 + n^2 \log n$
    - $c = 3$
    - $C = 3, k = 1$
  - $f(n) = \frac{n^4+n^2+1}{n^3+1}$
    - $c = 1$
    - $C = 1.5, k = 1$

Big-Ω

- Big-O
  - $\exists C, k \ \ \forall n > k \ \ \ f(n) \leq Cg(n)$
- Big- Ω (big omega)
  - $\exists C, k \ \ \forall n > k \ \ f(n) \geq Cg(n)$
  - C must be **positive.**
  - $f(n)$ is $\Omega\big(g(n)\big) \leftrightarrow g(n)$ is $O\big(f(n)\big)$
  - "$f(x)$ is bounded below by $g(x)$"

Big-Θ

- Big- Θ (big theta)
  - $f(n)$ is $O\big(g(n)\big)$ and $\Omega\big(g(n)\big)$
  - $f(n)$ is $O\big(g(n)\big)$ and $g(n)$ is $O\big(f(n)\big)$
  - $f(n)$ is $\Theta\big(g(n)\big) \leftrightarrow$ g(n) is $\Theta\big(f(n)\big)$
  - $\exists C_1, C_2, k \;\; \forall n > k \quad C_1 g(n) \leq f(n) \leq C_2 g(n)$
  - $f(n)$ is of *order* $g(n)$
  - $f(n)$ and $g(n)$ are of the *same order*

Big-Θ for Polynomials

- Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.
- Then, $f(x)$ is of order x^n.
  - "$f(x)$ is bounded [above and below] by $g(x)$"
- Example:
  - $3x^8 + 10x^7 + 221x^2 + 1444$ is of order x^8
    - Witnesses: $C = 6,\ k = 10$

Complexity of Algorithms

- **Computational complexity** is the amount of time and space an algorithm uses to solve a problem.
  - **Space complexity**
    - Depends on data structures used to implement the algorithm
  - **Time complexity**
    - Depends on the number of operations used by the algorithm.
    - Use big-O (or big-Θ, if possible) to specify

Time Complexity

- Elementary operations have constant time ($\Theta(1)$) complexity:
  - Assignment
  - Arithmetic operations
  - Boolean operations
  - Comparisons
  - Array access

Time Complexity

- Blocks of statements
  - $Block_1$; `// takes` T_1 `time`
  - $Block_2$; `// takes` T_2 `time`
  - …
  - $Block_k$; `// takes` T_k `time`
- To execute the sequence of Blocks 1 through k takes $O(T_1 + T_2 + \cdots + T_k)$ time.

Time Complexity

- Control Structures
  - `if(BoolExpr) // takes` T_B `time`
    - $Block_1$`; // takes` T_1 `time`
  - `else`
    - $Block_2$`; // takes` T_2 `time`
- To execute the control structures takes $O(\mathrm{T_B} + \max(T_1, T_2))$ time.

Time Complexity

- For Loops
  - `for i=a to b`
    - $Block_1$; `// takes` $T_1(k)$ `time when i=k`
- To execute the loop takes $T_1(a) + T_1(a+1) + \cdots + T_1\,(b)$ time
- If $T_1(k)$ is $\Theta(1)$, then the loop takes $O\big((b-a+1)\cdot T_1\big)$ time

Time Complexity

- Function Calls
  - `def f(params)` `// takes` T_p `time to assign params`
    - $Block_1$`; // takes` T_1 `time`
  - To execute the function takes $O(T_p + T_1)$ time

Bubble Sort Revisited

```
procedure bubble_sort({a_1, …, a_n})     // O(T_p) = O(1)
for i = 1 to n-1                          // Block_1
        for j = 1 to n-i                  //     Block_2
                if a_j > a_{j+1}          //         Block_3
                then, swap a_j and a_{j+1}
```

$Block_3$ is a control structure which takes $O(3+3) = O(1)$ time

$Block_2$ is a for loop, which takes $O\big((n-i-1+1)\cdot O(1)\big) = O(n-i)$ time

$Block_1$ is a for loop, which takes $O\big(O(n-1)+O(n-2)+\cdots+O(1)\big) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$ time

Therefore, the procedure takes $O\big(O(1)+O(n^2)\big) = O(n^2)$ time

Tractability

- A problem which can be solved by an algorithm with worst-case polynomial time complexity ($\Theta(n^c)$) is called **tractable.**
  - Does not guarantee that it can be solved in any reasonable amount of time.
  - Reasonable input sizes can be solved in relatively short time.
- A problem which cannot be solved by any algorithm with worse-case polynomial polynomial time complexity is called **intractable**.
  - Average case complexity may be better.
  - Many important problems are intractable, but still get solved everyday.
    - Approximate solutions.
- A problem for which there does not exists any algorithm is called **unsolvable.**
  - The first unsolvable, proved by Turing: The halting problem.

P vs NP

- All the **tractable** problems belong to a set called **P**.
  - **Can be solved** in worst-case polynomial time.
- All the problems whose **solutions can be verified** in polynomial time belong to a set called **NP**.
  - Example: Boolean Satisfiability (SAT) – find an assignment of truth values that satisfies some Boolean expression.
    - Solution can be verified very easily.
    - Finding a solution for n variables requires $\Omega(2^n)$ operations

NP-Complete

- It turns out that a bunch of problems in **NP** are actually the same problem. These are called **NP-complete** problems.
  - Every problem in **NP** can be reduced in polynomial time to an **NP-complete** problem.
    - SAT was the first to be proved to be NP-complete.
  - If **any** NP-complete problem can be solved in polynomial time, then **every** NP problem can, too.
    - **P** = **NP**.
- $1,000,000 prize for proof of whether **P** = **NP.**
  - General consensus is that **P** ≠ **NP.**