CS:3330 (22c:31)
Algorithms

Introduction

What’s an Algorithm?
• Computer Science is about problem-solving using computers.
• Software is a solution to some problems.
• Algorithm is a design inside a software.
• Informally, an algorithm is a method for solving a well-specified computational problem.

Example: search
input: An array of numbers and a given number x.
output: the position of x in the array; -1 if not exist.
solutions: linear search, binary search
issues: correctness, efficiency, etc.

Definition from Dictionary
• Dictionary.com: An algorithm is a set of rules for solving a problem in a finite number of steps.
• A better one: An algorithm is a sequence of unambiguous instructions for solving a well-specified computational problem in a finite number of steps.

Important Features:
– Definiteness.
– Finiteness.
– Effectiveness.
– Correctness.


Algorithms and Data Structures
• Algorithm - A recipe for performing a certain task.
• Data structure - A way of arranging data to make solving a class of problems easier.

Pseudocode
• Text that is a lot like a programming language but that is not really a programming language.
• Pseudocode:
  Max = maximum of array A[0..N-1]

C++/C#/java:
int Max = A[0];
for (int i = 1; i < N; i++) if (Max < A[i]) Max = A[i];
Counting Computing Steps

- Better than counting actual time (because you don’t have the same machine)
- Hard to be precise
- Abstraction is essential

- How many comparisons (<) and assignments when the following code is executed?
  ```java
  int Max = A[0];
  for (int i = 1; i < N; i++) if (Max < A[i]) Max = A[i];
  ```

  What is the worst case? What is the average case?

Worst and Average case analysis

- In worst case analysis of time complexity we select the maximum cost among all possible inputs of size n.
- In average case analysis, the running time is taken to be the average time over all inputs of size n.
  - Unfortunately, there are infinite inputs.
  - It is necessary to know the probabilities of all input occurrences.
  - The analysis is in many cases complex and lengthy

Serial Search

```java
// Search for a desired item in the array of n elements, starting at a[0].
// Returns the index of the desired number if found.
// Otherwise, return -1

public static int serialSearch(int[] a, int target) {
  for (int i = 0; i < a.length; ++i) {
    if (a[i] == target) return i;
  }
  return -1;
}
```

How many comparisons (==) when the above code is executed?

Worst Case Time for Serial Search

- For an array of n elements, the worst case time for serial search requires n array accesses: O(n).
- Consider cases where we must loop over all n records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

Average Case for Serial Search

Assumptions:
1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:
- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses.
  etc.

The average of all these searches is:

\[
\frac{1+2+3+4+5+6+7+8+9+10}{10} = 5.5
\]
Average Case Time for Serial Search

Generalize for array size $n$.

Expression for average-case running time:

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)/2n}{(n+1)/2}$$

Therefore, the average case time complexity for serial search is $(n+1)/2$, or $O(n)$.

Binary search

- **binary search**: Locates a target value in a sorted array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine? $O(\log N)$
  - Can be implemented with a loop or recursively
  - Example: Searching the array below for the value 42:

<table>
<thead>
<tr>
<th>value</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>22</th>
<th>25</th>
<th>30</th>
<th>36</th>
<th>42</th>
<th>50</th>
<th>56</th>
<th>68</th>
<th>85</th>
<th>92</th>
<th>103</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
<tr>
<td>mid</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>42</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Binary search code**

```java
// Returns the index of an occurrence of target in a, // or a negative number if the target is not found. // Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }
    return -1; // target not found
}
```

Q: What's the worst case and average case complexities?

**Search for target = 7**

Find midpoint:

```
3  6  7 11 32 33 53
```

Start at root:

```
3   6 11 33
  7 32 53
```
Search for target = 7
Search left subarray:
3 6 7 11 32 33 53
Search left subtree:
6
3 7
11
32 33
53

Search for target = 7
Find approximate midpoint of
3 6 7 11 32 33 53
Visit root of subtree:
6
3 7
11
32 33
53

Search for target = 7
Search right subarray:
3 6 7 11 32 33 53
Search right subtree:
6
3 7
11
32 33
53

Binary Search: Analysis
• Worst case complexity?
  – What is the maximum depth of the recursive calls in binary search as function of n?
  – Each level in the recursion, we split the array in half (divide by two).
  – Therefore maximum recursion depth is ceiling(log₂n) and worst case = O(log₂n).
• Average case complexity?
  – Average case is also O(log₂n): more than half of nodes are stored at the bottom two levels of the balanced tree (depth is >= log₂n − 1).

Can we do better than O(log₂n)?
• Average and worst case of serial search = O(n)
• Average and worst case of binary search = O(log₂n)
• Can we do better than this?
  YES. Use a good hash table!
Big $O$-notation
for significant differences in difficult cases

For function $g(n)$, we define $O(g(n))$, big- $O$ of $n$, as the set:

$O(g(n)) ::= \{ f(n) : \exists$ positive constants $c \text{ and } n_0,$
$such that $\forall n \geq n_0,$ we have $0 \leq f(n) \leq cg(n) \}$

Intuitively: Set of all functions whose rate of growth is the same as or lower than that of $g(n)$.

$g(n)$ is an asymptotic upper bound for $f(n)$.

"$f(n) = O(g(n))" ::= "f(n) is in O(g(n))."

Big $\Omega$-notation

For function $g(n)$, we define $\Omega(g(n))$, big- $\Omega$ of $n$, as the set:

$\Omega(g(n)) = \{ f(n) : \exists$ positive constants $c \text{ and } n_0,$
$such that $\forall n \geq n_0,$ we have $cg(n) \leq f(n) \}$

Intuitively: Set of all functions whose rate of growth is the same as or higher than that of $g(n)$.

$g(n)$ is an asymptotic lower bound for $f(n)$.

Big Theta Relation

$f(n)$ and $g(n)$ have the same rate of growth, if

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

Notation: $f(n) = \Theta(g(n))$

Theorem: $\Theta$ is an equivalence relation.

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

Algorithms with Same Complexity

Two algorithms have same complexity, if the functions representing the number of operations have same rate of growth.

Among all functions with same rate of growth we choose the simplest one to represent the complexity.

So we write $(n+1)/2 = \Theta(n)$ instead of $n = \Theta((n+1)/2)$.

Big Theta Relation Examples

$\Theta(n^3)$:
- $n^3$
- $5n^3 + 4n$
- $105n^3 + 4n^2 + 6n$

$\Theta(n^2)$:
- $n^2$
- $5n^2 + 4n + 6$
- $n^2 + 5$

$\Theta(\log n)$:
- $\log n$
- $\log n^2$
- $\log (n + n^3)$

Little oh

$f(n)$ grows slower than $g(n)$ (or $g(n)$ grows faster than $f(n)$) if

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

Notation: $f(n) = o(g(n))$

pronounced "little oh"

$o(g(n)) ::= \{ f(n) | \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \}$
Little omega

\( f(n) \) grows faster than \( g(n) \) (or \( g(n) \) grows slower than \( f(n) \)) if

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty,
\]

Notation: \( f(n) = \omega \left( g(n) \right) \)

pronounced “little omega”

\( \omega \left( g(n) \right) \) ::= \{ \( f(n) \mid \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \} \)

Relation Summary:

\[
\begin{align*}
\lim_{n \to \infty} \frac{f(n)}{g(n)} & = \infty \Rightarrow f(n) = \omega \left( g(n) \right) \\
& \Rightarrow f(n) = \Omega \left( g(n) \right) \\
& \Rightarrow f(n) = \Theta \left( g(n) \right) \\
& \Rightarrow f(n) = o \left( g(n) \right) \\
& \Rightarrow f(n) = O \left( g(n) \right)
\end{align*}
\]

Caution:

• “\( f(n) = O(g(n)) \)” is an abuse of notation.
  – Not transitive:
    • \( f(n) = 5n^2; \ g(n) = 3n^2 \)
    • \( f(n) = O(n^2) = g(n) \)
    • but \( f(n) \neq g(n) \).
  – Better notation: \( f(n) \in O(g(n)) \).
• “\( f(n) = O(g(n)) \)” is a relation between \( f(n) \) and \( g(n) \).
• Meaningless statement. Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons.
  – Statement doesn't "type-check."
  – Use \( \Omega \) for lower bounds.

Relation between relations

Theorem:

• \( f(n) = O(g(n)) \) iff \( g(n) = \Omega(f(n)) \)
• \( g(n) = o(f(n)) \) iff \( f(n) = \omega(g(n)) \)
• \( f(n) = O(g(n)) \) iff either \( f(n) = \Theta(g(n)) \) or \( f(n) = o(g(n)) \).
• \( f(n) = \Omega(g(n)) \) iff either \( f(n) = \Theta(g(n)) \) or \( f(n) = o(g(n)) \).
• \( f(n) = \Omega(g(n)) \) iff \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)

Upper/lower bounds in worst case

• Under the worst case assumption, the upper and lower bounds in many algorithms coincide and, consequently, we may say that an algorithm runs in time \( \Theta(f(n)) \) in the worst case. But the upper and lower bounds are not always coincide.
• Example:
  ```
  int oddSearch(int[] a, int target) {
    if (a.length is odd) return serialSearch(a, target);
    else return binarySearch(a, target);
  }
  ```
  • Upper bound in the worst case: \( O(n) \)
  • Lower bound in the worst case: \( \Omega(\log n) \)

Rules to manipulate Big-Oh expressions

Rule 1:

a. If \( T1(N) = O(f(N)) \) and \( T2(N) = O(g(N)) \) then

\[
T1(N) + T2(N) = \max(O(f(N)), O(g(N)))
\]

b. If \( T1(N) = O(f(N)) \) and \( T2(N) = O(g(N)) \) then

\[
T1(N) \times T2(N) = O(f(N) \times g(N))
\]
Rules to manipulate Big-Oh expressions

**Rule 2:** If $T(N)$ is a polynomial of degree $k$, then $T(N) = \Theta(N^k)$

**Rule 3:** Polynomials grow slower than exponentials $N^k = O(c^j)$ for any constants $k, c > 1$.

**Rule 4:** Logarithms grow slower than polynomials $\log^k N = O(N)$ for any constant $k$.

Common Runtime Functions

- $1$
- $\log N$
- $\sqrt{N}$
- $N$
- $N \log N$
- $N^2$
- $2^N$
- $N!$

Practical Considerations

- For the same problem,
  - Alg. A runs in $10000N^2$;
  - Alg. B runs in $10N^3$.
- Which one is better?
- Answer: When $N < 1000$, Alg. B is better; when $N > 1000$, Alg. A is better.

Space Complexity

- We define the space used by an algorithm to be the number of memory cells needed to carry out the computational steps required to solve an instance of the problem excluding the space allocated to hold the input.
- All definition of order of growth and asymptotic bounds pertaining to time complexity carry over to space complexity.
- Let $T(n)$ and $S(n)$ denote, respectively, the time and space complexity of an algorithm, then $S(n) = O(T(n))$.

Amortized analysis

- We have a doubly linked list that initially consists of one node which contains the integer 0. We have as input an array $A[1..n]$ of $n$ positive integers that are to be processed in the following way. If the current integer $x$ is odd, then add $x$ to the end of the list. If it is even, then first add $x$ and then remove all odd elements before $x$ in the list. Let us analyze the running time of this algorithm.
The pseudo code of the algorithm:

for j = 1 to n
    x = a[j]
    append x to the list
    if x is even then
        while pred(x) is odd
            delete pred(x)
        end while
    end if
end for

Amortized analysis

• Worst-case time complexity analysis:
  – General analysis: \( O(n^2) \)
  – Amortized analysis: \( \Theta(n) \)
    (number of insertions: \( n \), and the number of deletions: \( 0 \sim n-1 \), so the total operations: \( n^2n-1 \))

Dynamic Tables

In some applications:
• We don’t know how many objects will be stored in a table.
• We may allocate space for a table
  – But, later we may find out that it is not enough.
  – Then, the table must be reallocated with a larger size.
  • All the objects stored in the original table
  • Must be copied over into the new table.

Insertion-Only Dynamic Tables

Table-Expansion:
• Assumption:
  – Table is allocated as an array of slots
• A table fills up when
  – all slots have been used
  – equivalently, when its load factor becomes 1
• Table-Expansion occurs when
  – An item is to be inserted into a full table
• A Common Heuristic: Allocate a new table that has twice as many slots as the old one.

Dynamic Tables

Using amortized analysis we will show that, The amortized cost of insertion is \( O(1) \). Even though the actual cost of an operation is large when it triggers an expansion.

Table Insert

```
TABLE-INSERT(T: table, x: int) {
    if T.size = 0 then
        allocate T array with 1 slot
        T.size \( \leftarrow \) 1
    if T.num = T.size then
        allocate new array with 2*T.size slots
        copy all items in T array into new array
        free T array
        T.size \( \leftarrow \) new array
        T.size \( \leftarrow \) T.size*2
        insert x into T array
        T.num \( \leftarrow \) T.num + 1
    }
```
Table Expansion

- Running time of TABLE-INSERT is proportional to the number of elementary insert operations.
- Assign a cost of 1 to each elementary insertion
- Analyze a sequence of \( n \) TABLE-INSERT operations on an initially empty table

Cost of Table Expansion

What is the cost \( c_i \) of the \( i \)-th operation?

- If there is room in the current table (or this is the first operation) \( c_i = 1 \) (only one elementary insert operation)
- If the current table is full, an expansion occurs, then the cost is \( c_i = i \).  
  1 for the elementary insertion of the new item 
  \( i-1 \) for the items that must be copied from the old table to the new table.

The Aggregate Method

Therefore the total cost of \( n \) TABLE-INSERT operations is

\[
\sum_{j=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j = n + 2n = 3n
\]

The Accounting Method

- Set a saving account with \( s_0 = 0 \) initially.
- The \( i^{th} \) operation has a budget cost of \( a_i \), which is the amortized cost of each operation.
- The account value after the \( i^{th} \) operation is
  \[
  s_i = s_{i-1} + a_i = c_i
  \]

For this example, \( a_i = 3 \).
Summary of Analysis Methods

- Worst-case analysis:
  - Pro: easy & simple to use
  - Con: worst-case might be rare
- Average-case analysis:
  - Pro: close to reality
  - Con: need probability of inputs, hard to use
- Amortized analysis:
  - Pro: tight bounds for a sequence of operations
  - Con: hard to use, a single operation may take too long.