1

Efficiency of Algorithms

Citius, Altius, Fortius

Determining the Efficiency of Algorithms

- Analysis of algorithms
  - Is a major field that provides tools for evaluating the efficiency of different solutions

- What is an efficient algorithm?
  - Faster is better

- How do you measure time?
  - Wall clock? Computer clock?
  - Using Less space is better

- But if you need to get data out of main memory it takes time

- Algorithm analysis should be independent of
  - Specific implementations and coding tricks (programming language, control statements)
  - Specific Computers (hardware chip, OS, clock speed)
  - Particular set of data

- But size of data should matter

Comparing Two Versions of Selection Sort

```java
public void selectionSort(int[] data) {
    int maxNum = 0;        // max integer
    int maxIndex = 0;      // index of max integer
    int i, j;
    for (j = data.length - 1; j > 0; j--) {
        maxIndex = 0;
        maxNum = data[0];
        for (i = 1; i <= j; i++)
            if (data[i] > maxNum) {
                maxNum = data[i];
                maxIndex = i;
            }
        swap(data, maxIndex, j);
    }
}

private static void swap(int[] data, int i, int j) {
    // exchanges the contents of data[i] and data[j]
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
```

Comparing Different Sorting Algorithms

```java
public void insertionSort(int[] data) {
    for (int unsorted = 1; unsorted < data.length; ++unsorted) {
        int nextItem = data[unsorted];
        int loc = unsorted;
        while (loc > 0 && data[loc-1] > nextItem) {
            // shift
            data[loc-1] = data[loc];
            loc--;
        }
        // insert
        data[loc] = nextItem;
    }
}

private static int indexOfLargest(int[] theArray, int size) {
    int indexSoFar = 0;
    for (int currIndex = 1; currIndex < size; ++currIndex)
        if (theArray[currIndex] > theArray[indexSoFar]) {
            indexSoFar = currIndex;
        }
    return indexSoFar;  // index of largest item
}
```

Both algorithms sort correctly. Is one better than the other?
Comparing Any Two Algorithms

Public void solveHanoiTowers(int n, char source, char dest, char spare) {
if (n==1) System.out.println("Move top disk from "+source+" to "+dest);
else {
solveTowers(n-1, source, spare, dest);
solveTowers(1, source, dest, spare);
solveTowers(n-1, spare, dest, source);
}
}

Algorithms solving two different problems. Is one more difficult than the other?

The Execution Time of Algorithms

• Counting an algorithm’s operations is a good way to assess its efficiency

  • An algorithm’s execution time is related to the number of operations it requires in a “worst case” scenario

  • Examples

    • Searching a linked list
      - Operations: about as many as elements
    • Selection sort
      - Operations: as we go through the array to select the next minimum, we traverse most of the array again
      - The Towers of Hanoi
        - Operations: to solve an instance of n disks, we need to solve 2 instances of n-1 disks...

    - We could also consider an “average case” scenario, but is much harder...

Single-statement Execution

• A statement that the computer can execute in one or a few (fixed number of) instructions, we count it as 1 step:
  \( O(1) \) = “order of one”

  • This includes arithmetic, logical operations, assignments, but not necessarily function calls, recursive steps, etc.

  • For example:

    ```java
    int i = 100;
    if (i < n && n%2 == 0) i = i/n;
    else { i = (i+1)/n*2; }
    ```

    Each of these statements is considered O(1). Thus the overall order of this piece of code is \( k \cdot O(1) = O(1) \) where \( k \) is the number of individual statements

Analyzing Loop Execution

• We often have to determine how often a set of statements get executed to determine the order of an algorithm

  • To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute

    ```java
    for (int i = 0; i < n; i++) {
      // some sequence of O(1) steps
    }
    ```

    The loop executes \( n \) times, and the body of the loop is \( O(1) \). Thus the overall order is \( O(n) \cdot O(1) = O(n) \)
Analyzing Nested-Loop Execution

When loops are nested, we must multiply the complexity of the outer loop by the complexity of the inner loop.

```java
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    { // some sequence of O(1) steps
        // some sequence of O(1) steps
    }
```

Both the inner and outer loops have complexity of $O(n)$. When multiplied together, the order becomes $O(n^2)$.

Analyzing Recursive Algorithms

Determining the order of a recursive algorithm:

- determine the order of the recursion (the number of times the recursive definition is followed) and multiply it by the order of the body of the recursive method.

Example: Consider the recursive method to compute the product of integers from 1 to some $n > 1$.

```java
public int fact (int n)
{
    int result;
    if (n == 1)
        result = 1;
    else
        result = n * fact(n-1);
    return result;
}
```

The size of the problem is the number of times recursive method is invoked, the value of $n$ is decreased by 1 each time the recursive method is called. The order of the entire algorithm is $O(n)$.

Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the problem size $n$.
- Growth rate enables the comparison between algorithms.
  - Examples:
    - Searching a sorted array requires time proportional to $\log n$.
    - Searching a list requires time proportional to $n$.
    - Selection sort requires time proportional to $n^2$.
  - Notation: Big-Oh aka "order of"
    - Searching a sorted array requires time $O(\log n)$.
    - Searching a list requires time $O(n)$.
    - Selection sort requires time $O(n^2)$.
- Algorithm efficiency is typically a concern for large problems only (as $n$ grows...).

Comparison of growth-rate functions

| n | $1$ | $10$ | $100$ | $1,000$ | $10,000$ | $100,000$ | $1,000,000$
|---|---|---|---|---|---|---|---|
| $2^n$ | $1$ | $1$ | $2$ | $4$ | $16$ | $1,024$ | $1,073,741,824$
| $n$ | $1$ | $10$ | $100$ | $1,000$ | $10,000$ | $100,000$ | $1,000,000$
| $n^{\log_2 4}$ | $3$ | $64$ | $6,964$ | $9,965$ | $10^4$ | $10^{10}$ | $10^{10}$
| $n^2$ | $2$ | $10^2$ | $10^4$ | $10^6$ | $10^{12}$ | $10^{20}$ | $10^{40}$

Comparison in graphical form
Order-of-Magnitude Analysis and “Big-O”

- Definition of the order of an algorithm
  Algorithm \( A \) is order \( f(n) \), denoted \( O(f(n)) \), if constants \( k \) and \( n_0 \) exist such that \( A \) requires no more than \( k \times f(n) \) time units to solve a problem of size \( n \geq n_0 \).

- Growth-rate function
  - A mathematical function used to specify an algorithm’s order in terms of the size of the problem.

- “Big-O” notation
  - A notation that uses the capital letter \( O \) to specify an algorithm’s order.
  - Example: \( O(1) \), \( O(n \log n) \), \( O(n) \), in general, \( O(f(n)) \).

Order-of-Magnitude Analysis and “Big-O”

- Order of growth of some common functions
  \( O(1) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^4) \).

- Properties of growth-rate functions
  - Summing orders is dominated by the larger order.
    \( O(f(n)) + O(g(n)) = O(\text{max}(f(n), g(n))) \).
  - Therefore, you can ignore low-order terms.
  - Multiplying orders means multiplying terms.
    \( O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)) \).
  - Therefore, you can ignore a multiplicative constant in the high-order term.
    \( O(12n^2) = O(n^2) \).

Example: What is the running time of...

```java
public void addCD(String title, String artist, double cost, int tracks) {
    if (count == collection.length)
        increaseSize();
    collection[count] = new CD(title, artist, cost, tracks);
    totalCost += cost;
    count++;
}
```

We need to first calculate this...

```java
private void increaseSize() {
    CD[] temp = new CD[collection.length * 2];
    for (int cd = 0; cd < collection.length; cd++)
        temp[cd] = collection[cd];
    collection = temp;
}
```
Worst Case and Average Case Analyses

- An algorithm can require different times to solve different problems of the same size
  - **Worst-case analysis**
    - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
  - **Average-case analysis**
    - A determination of the average amount of time that an algorithm requires to solve problems of size n

Keeping your Perspective

- Throughout the course of an analysis, keep in mind that you are interested only in **significant differences** in efficiency
- When choosing an ADT’s implementation, consider **how frequently** particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient
- If the problem size is always small, you can probably ignore an algorithm’s efficiency
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on large problems