Computer Science 331

Introduction to Analysis of Algorithms

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #5-6

Outline

Measuring Efficiency

What sorts of measures could we use? The following are all (sometimes) important:

- **Running Time** — no one wants to wait too long for programs to execute
- **Memory Used by Data (Storage Space)** — time is (sort of) unconstrained, but any computer can run out of memory
- **Memory Used by Code** — an issue if a program is to be stored on a low-memory device (like a smart card)
- **Time to Code** —- programmers must be paid and software development usually has deadlines!

Our focus will be on *running time* and *storage space*.

How Do We Measure Efficiency?

How can we compare algorithms or programs?

1. **Run the Code and Time the Execution.**
   Problem: Execution time is influenced by many factors:
   - *Hardware* (How fast is the CPU? How many of them?)
   - *Compiler and System Software:* (Which OS?)
   - *Simultaneous User Activity:* (Potentially affected by the time of day when the program was executed)
   - *Choice of Input Data:* (Running times can vary on inputs, even inputs of the same "size")
   - *Programmer's Skill*
2. **Analyze the Code**
   Advantage: Only influenced by choice of data
   Disadvantage: Can be quite difficult!

We typically try to do *both* (analysis supported by execution timings).

What Will We Measure?

Most of the time, in this course, running time and storage space will be measured in an abstract *machine-independent* way.

Running Time:

- Number of primitive operations or “steps” (programming language statements) used
- Ignores: different costs between operations (eg. multiply vs. add)

Storage Space:

- Number of words of machine memory used, assuming each word can store the same (fixed) number of bits
- Ignores: memory hierarchy differences, eg. cache vs. main memory

How Do We Wish To Measure Resources?

We will try to measure the amount of resources (time or space) used as a function of the “input size.” (defined in various ways, depending on the type of input considered).

Example: if the input is an *array*, the appropriate measure of input size is (usually):

- array length, i.e., number of elements

Example: if the input is a single *integer*, which can be virtually as large as we want, the appropriate measure of input size is:

- the bit-length of the integer, i.e., number of bits in its binary representation

Complication: executions of a program on different inputs *with the same size* frequently have different costs!

Worst-Case Analysis

Consider the *maximal* amount of resources (such as *longest* running time) used by the algorithm, on any input of a given size

Advantages of This Type of Analysis:

- upper bound on running time (guarantee that the algorithm will not take any longer for *any* inputs of the given size)
- for some algorithms, worst-case occurs fairly often (eg. searching an array for an element not in it)

Disadvantage of This Type of Analysis:

- for some cases, the worst case rarely occurs (eg. array in reverse order is the worst case for one variation of quicksort)

Average-Case Analysis

Consider the **average** (or “expected”) amount of resources (such as **average** running time) used by the algorithm, for an input of a given size

Advantage of This Type of Analysis:

- captures resource consumption for typical inputs

Disadvantages of This Type of Analysis:

- executions on some inputs of the given size can take *much* longer than the average case
- may be difficult to determine what the average case actually is — some assumption about the distribution of the inputs is *always* needed

In some, but not all, cases, the worst-case and average-case running times (or amount of storage space used) are approximately the same.

Other Kinds of Analysis

Best-case Analysis:

- *minimal* amount of resources (such as *shortest* running time) used by the algorithm, on any input of a given size
- occasionally of interest, but usually together with other measures (eg. see whether best and worst cases running times are close)

Amortized Analysis:

- ratio of total cost of a sequence of operations to the number of operations in the sequence
- similar to average case, except that no assumptions about input distribution are required
- mostly beyond scope of course, but some results will be mentioned

Objective and Strategy

Objective: use code (or pseudocode) to estimate the *worst-case running time* of a program (or algorithm).

Useful Values:

- Worst-case running time (exact)
- Upper and lower bounds on worst-case running time (easier, often sufficient)

Strategy: consider subprograms ...

- beginning with individual statements ...
- then considering progressively larger subprograms ...
- until the whole program has been considered.

Case: Program is a Single Statement

Example: `x := 1`

Amount to charge:

- 1 unit (eg. single arithmetic/Boolean operation, comparison, or assignment)

Example: `x := y := 1`

Amount to charge:

- 2 units (one per assignment)
- be careful with compound statements
- one line does not always equal one unit!

Case: Program is a Sequence of Subprograms

Structure to Consider: S_1; S_2

Worst-Case Running Time: If

- worst-case running time of S_1 is T_1, and
- worst-case running time of S_2 is T_2,

then

- worst-case running time of entire program is *at most:* $T_1 + T_2$

Explanation (upper bound because...):

- worst-case input to S_1 may not yield a worst-case input to T_2

Case: Program is a Conditional Statement

Structure to Consider:

if c **then**
  S_1
else
  S_2
end if

Worst-Case Running Time: if

- worst-case running time to evaluate c is T_1,
- worst-case running time of S_1 is T_1, and
- worst-case running time of S_2 is T_2,

then

- worst-case running time of program is: $T_1 + \max(T_2, T_3)$

Case: Program is a Loop

Structure to Consider:

while c **do**
  S
end while

We need to know:

- the worst-case cost to evaluate c
- the worst-case cost to execute S
- the maximum number of executions of the loop body

Problem:

- it is not even clear that this will halt!

First Objective: Counting Executions of the Loop Body

Recall that a *Loop Variant* is an integer-valued function f of variables such that

- the value of f decreases by at least 1 each time loop body is executed;
- the test c is **false** if the value of f is ≤ 0

The *existence* of a loop variant implies that the loop terminates if each evaluation of c and each execution of the loop body terminates.

Useful fact: number of executions of loop body is *less than or equal to* the value of f immediately before execution of the loop begins

Next Objective: Bounding Total Running Time

Suppose:

- Loop body is executed at most k times
- Worst-case cost for each evaluation of the loop test c is $\leq T_1$
- Worst-case cost for each execution of the loop body S is $\leq T_2$

Then:

- *Total* cost for *all* evaluations of test c is at most: $(k+1)T_1$
- *Total* cost for *all* executions of loop body is at most: kT_2
- Therefore, the *total* cost to execute the loop is at most: $(k+1)T_1 + kT_2$

If cost of jth iteration of S is $T_2(j)$: $\quad (k+1)T_1 + \sum_{j=0}^{k} T_2(j)$

Example

Suppose A is an integer array with length n, *key* is an integer, and the following code is executed.

$i := 0$
while $((i < n)$ and $(A[i] <> key))$ **do**
$i := i + 1$
end while

Loop Variant for this program's loop: $f(n, i) = n - i$

- i increases after each iteration, so $f(n, i)$ decreases
- $f(n, i) \leq 0$ if $i \geq n$ and the loop terminates if $i \geq n$

What about 2nd condition in test? ignore (doesn't affect worst case)

Example, Continued

Maximum number of executions of the loop body:

- $f(n, 0) = n - 0 = n$

Worst-case cost to evaluate test:

- 3 units (two comparisons, one Boolean operation), or constant c_1

Worst-case cost for an execution of the loop body:

- 2 units (one addition, one assignment), or constant c_2

Upper bound on worst-case cost to execute the loop:

- $3(n + 1) + 2n = 5n + 3$, or
- $c_1(n + 1) + 2n = d_1 n + d_2$ for constants d_1, d_2

Case: Program is a Nested Loop

Structure to Consider:

while c_1 **do**
while c_2 **do**
S
end while
end while

Method:

- compute worst-case cost of inner loop as above
- compute cost of outer loop using computed inner loop cost as the worst-case cost of the outer loop's body

An example will be covered in next week's labs.

Case: Program Calls Itself a Constant Number of Times

Example: Fibonacci Number Program

public int **fib**(int n)
if $n == 0$ **then**
return 0
else if $n == 1$ **then**
return 1
else
return $\text{fib}(n - 1) + \text{fib}(n - 2)$
end if

Objective: Writing an Expression for the Running Time

Let $T(n)$ be the number of steps used on input n. Then

$$T(n) \leq \begin{cases} 2 & \text{if } n = 0, \\ 3 & \text{if } n = 1, \\ 6 + T(n-1) + T(n-2) & \text{if } n \geq 2. \end{cases}$$

This is an example of a *recurrence relation*:

- $T(n)$ expressed using the same function T evaluated at **smaller** inputs
- Explicit (non-recursive) values of T given for small inputs n (base cases)

$T(2) \leq 6 + T(1) + T(0) = 11, \; T(3) \leq 6 + T(2) + T(1) = 20$, etc...

Analysis of Recursive Programs

The following exercises on computing bounds on $T(n)$ can be solved using *mathematical induction*.

Exercises:

1. Use the above information to prove that
$$T(n) \leq 6 \times 2^n - 4$$
for every integer $n \geq 0$.
2. Use the above information to prove that
$$T(n) \leq 6 \times \text{fib}(n+1) - 4$$
for every integer $n \geq 0$.

Bounding a recurrence using induction is one way to analyse recursive programs (you will learn others in CPSC 413).

Finding a *Lower Bound*

In order to prove that the worst-case running time of a program P is *at least* T, for input size N (for a fixed N):

- Find a valid input I of size N (where "valid" means that P's precondition is satisfied)
- Count the number of steps used by P on input I
- *If* this number is greater than or equal to T *then* you have proved what we want!

Why This Works:

- worst-case cannot be less than the running time of any particular input

Finding a *Lower* Bound, Continued

In order to prove that the worst-case running time of a program P is *at least* $T(n)$, for a function $T(n)$:

- Find a collection $I_1, I_2, I_3, I_4, \ldots$ of inputs, where I_i is a valid input of size i for all $i \geq 1$
- Show that the number of steps used by P on input I_i is greater than or equal to $T(i)$, for every integer $i \geq 1$

A Common Mistake

Some people try to prove that the worst-case running time of a program P is *at most* $T(n)$, for a function $T(n)$, by doing the following:

- They give a collection $I_1, I_2, I_3, \ldots$ of inputs, where I_i is a valid input of size i for all $i \geq 1$
- They show (generally, correctly) that the number of steps used by P on input I_i is less than or equal to $T(i)$, for every integer $i \geq 1$.
- They then conclude that the worst-case running time of P on inputs of size n is at most $T(n)$ (for all n)

Why This is Incorrect:

- does not prove that there no inputs for which the running time is larger

A Variation on This Mistake

Frequently, on tests, students do *almost* what has just been described: Before stating their conclusion, they write

> *Clearly, program P takes at least as much time on input I_i as it does on any other input of size i. ...*

... and the Comment the Marker Provides is ...

> *Incorrect — only proves a lower bound on worst-case*

... and the Mark That is Assigned is ...

> *zero*

Further Reading ...

- Textbook, Section 4.2
  - As an example there shows, we can sometimes use *summations* to find better bounds on the time used by loops
  - Also includes material we will cover in lectures next week
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

  Introduction to Algorithms, Second Edition
  - available as an ebook
  - includes *much* more material about this topic