COMP 3760: Algorithm Analysis and Design

Lesson 2: Algorithmic Efficiency

Rob Neilson

rneilson@bcit.ca

What is an Algorithm?

From your textbook:

*An algorithm is a sequence of **unambiguous instructions** for solving a problem.*

*i.e: for obtaining a required output for any **legitimate input** in a **finite amount of time***

For example: (this is pseudo-code)

What does this algorithm do?

```
Algo: mv( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

It finds and returns the largest element in the input array A[].

It uses a variation of a 'sequential search'.

Time Efficiency

```
Algo: mv( A[0…n-1] )
    m ← A[0]
    for i ← 1 to n-1 do
        if A[i] > m
            m ← A[i]
    return m
```

Questions that we ask about an algorithm:

- *Is mv (note: poor choice of name) a time-efficient algorithm?*
  - this means – does it execute quickly for an arbitrary input?

The mv algorithm is pretty quick. It has to check each and every element to find the largest, and it checks each element exactly once.

Can we write a faster algorithm for finding the largest element in an array?

(no)

Could we write a faster algorithm if we knew the input was sorted?

(yes)

Space Efficiency

```
Algo: mv( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Another question that we might ask about an algorithm:

Is mv a space-efficient algorithm?

- this means: does it use a reasonable amount of memory or disk space?

The mv algorithm uses only one local variable (m) to store the currently largest item. Any other space (ram or disk) is just for holding the input (which we must do!) – so this is very space efficient.

Why do we care about algorithm efficiency? (1)

- consider the standard algorithm to compute the nth fibonacci number …

```
Algo: fib( n )
   if n ≤ 1
      return n
   else
      return fib( n-1 ) + fib( n-2 )
```

- (recall that the nth fibonacci number is simply the sum of the two preceding fibonacci numbers, eg: 0, 1, 1, 2, 3, 5, 8, 13 …)

- this algorithm can be easily translated into java …

```
public static int fib(int n)  {
   if (n<=1)
      return n;
   else
      return ( fib(n-1) + fib(n-2) );
}
```

How fast does it run?

Why do we care about algorithm efficiency? (2)

- new consider a better algorithm to compute the nth fibonacci number.

```
Algo: fib2( n )
   F[0] ← 0; F[1] ← 1;
   for i ← 2 to n do
      F[i] ← F[i-1] + F[i-2]
   return F[n]
```

This algo stores successive results so we don't have to recompute them …

- the new algorithm in java …

```
public static int fib2(int n) {

   int[] f = new int[n+1];

   f[0] = 0;
   f[1] = 1;
   for (int i=2; i<=n; i++)
       f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

How does it run?

What have we learned …

- simply implementing an algorithm in an obvious way does not always yield an efficient algorithm
- sometimes a well thought out algorithm runs much faster
- there can be huge variations in efficiency between different solutions to the same problem
- the actual performance depends on the size of the input being processed
- larger input sizes require more computations (obviously)

How to Determine Algorithm Efficiency? (1)

We could do it experimentally …

- however the implementation of complex algorithms is very time consuming, and you would have to implement many to determine the most efficient …
- it would be to our advantage to be able to estimate the efficiency of an algorithm ***before we have to code it*** !

How to Determine Algorithm Efficiency? (2)

Consider what we know about any given algorithm:

1. Typically, the time and space requirements of an algorithm both depend on the # input items to be processed (input size *n*)

eg: suppose you need to search for a number in an array

- if the array has 2 elements (n=2), we can perform the search with at most 2 comparisons
- if the array has 4294967296 elements ($n=2^{32}$), we require at most 2^{32} comparisons

How to Determine Algorithm Efficiency? (3)

2. The *total execution time* for any algorithm is the total number of instructions executed, multiplied by the time to execute each instruction
   - assume n=3 (array A has 3 elements for the mv algo shown below)
   - furthermore, *assume all instructions (statements) take 1 time unit to execute*

Now we ask some questions, such as ...

- *What organization of input data will result in the maximum number of instructions being executed?*
  - answer: max number of instructions when input is presented in ascending order
- *How many time units are required (ie: what is the total execution time of this algorithm for input size n=3)?*

```
1. Algo: mv( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

stmt	#times executed
1	0
2	1
3	2
4	2
5	2
6	1

this is a total of 8 time units
(# instr's X 1 time unit per instr)

Summary of Observations ...

The information on the previous 2 slides tells us ...

1. the running time (efficiency) of an algorithm is dependent on the input size

2. we can determine the running time by counting the number of operations executed during the processing if an specific input set

Dealing with input size ...

- *most* algorithms work well with small input sizes, so we tend to investigate the performance for large input sizes
- in fact, we will try to characterize the performance of an algorithm on arbitrarily large input sets (ie: how fast will it run on an input size n, where n is a large number)
- therefore it is useful to state the performance as a function of the input size
- for example, the performance of the mv algorithm could be said to be 3(n-1) + 2 (worst case)
  - *more on this later! (we will simplify it to simply be "n")*

for n=3

stmt	#times
1	0
2	1
3	2
4	2
5	2
6	1

Dealing with instruction execution time ...

- Earlier on we made an assumption: all instructions take the same amount of time.
  - *Is this a reasonable assumption?*
- We know:
  - modern computers have a fairly rich instruction set
  - different instructions take different numbers of clock cycles to run
- Answer: yes, it is fair – because of the concept of a “basic instruction”.

Basic Instructions

- Consider our mv algorithm on an *arbitrary* set of input data.
- Which instruction gets executes the most?

```
1. Algo: mv( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

	(n=3)	*(n=10)*	*(n=100)*
stmt	#times	#times	#times
1	0	0	0
2	1	1	1
3	2	9	99
4	2	9	99
5	2	9	99
6	1	1	1

- We can see that many of the statements (eg: 1,2,6) become irrelevant.
- The loop is what impacts the performance the most.
- Therefore, we define an algorithm's basic operation as the statement that gets executed most frequently. *Typically this is an instruction that is found in the innermost loop.*

Basic Operation (cont)

- we can get a good idea of a programs performance on an arbitrary sized input set by *simply counting the number of basic operations*

This is the fundamental concept we use to analyze algorithmic efficiency:

count the number of basic operations executed for an input of size n

- using this simplification, we would say that mv is an ***n-1 algorithm*** (ie: we don't count the instructions that are not basic operations)

- later on – maybe next lesson – we will learn how to organize algorithms into efficiency classes based on their basic op count
  - for example mv is **n-1**, as is fib2
    - we will say that they are both O(n) algorithms;
  - contrast this with fib (the recursive version), which has approx $\mathbf{1.6^n}$ basic ops
    - ie: it is an $O(2^n)$ algorithm

Summary (so far) ...

- to be able to compare algorithms we will make some simplifications:

  1. assume all algos are running on the same machine (so that clock speed and instruction set do not influence efficiency)

  2. assume that all instructions take the same constant amount of time (ie: 1 time unit)

  3. then, to characterize the performance of an algorithm, we:
     i. decide what the worst case input looks like
     ii. determine the “basic operation” in the algorithm
     iii. calculate how often this basic op is executed for an input set of size n

An example (identifying basic operations)

- consider this algorithm

```
1. Mystery(n)   // n > 0
2.   S ← 0
3.   for i ← 1 to n do
4.        S ← S + i * i
5.   return S
```

1. What does this algorithm do?

   - *it computes* $\sum_{i=1}^{n} i^2$

2. What is the basic operation?

   - *multiplication or addition*

3. How many times is the basic operation executed (for input size n)?

   - it goes through the loop n times, and the basic op is executed once each time through the loop.

   ... so we say it is executed: $\sum_{i=1}^{n} 1 = n$ times

Mystery algorithm (cont)

- Can you think of some way to improve this algorithm?
  - (ie: speed it up for large input sizes)

- How about re-implementing it to use the closed form equation:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

- If you did this (re-implement it), how many basic operations would the program require for an input set of size n?

  - well … if the basic operation is multiplication, it would require ***2 basic ops, no matter how large the input size is!***

  - we call this a "constant time algorithm", as it can process any set of input in a constant time

Another example (identifying basic operations)

- consider this algorithm

```
Loops(A[0..n-1])
  for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

1. What does this algorithm do?

   - *it sorts the input array A[]*

   - *this is the classic algo known as "insertion sort"*

eg: assume the input set {89,45,68,90,29,34,17}

```
89 | 45   68   90   29   34   17
45   89 | 68   90   29   34   17
45   68   89 | 90   29   34   17
45   68   89   90 | 29   34   17
29   45   68   89   90 | 34   17
29   34   45   68   89   90 | 17
17   29   34   45   68   89   90
```

Another example (identifying basic operations)

- consider this algorithm ⇒

```
1.  Loops(A[0..n-1])
2.   for i ← 1 to n-1 do
3.     v ← A[i]
4.     j ← i-1
5.     while j≥0 and A[j]>v do
6.       A[j+1] ← A[j]
7.       j ← j-1
8.     A[j+1] ← v
```

1. What does this algorithm do?

   - *it sorts the input array A[]*

   - *this is the classic algo known as "insertion sort"*

2. What is the basic operation?

   - the comparison of the current key ***A[j]>v*** in the inner most loop

3. How many times is the basic operation executed (for input size n)?

   - it goes through the outer loop (for loop) ***n-1*** times for any input set
   - in the worst case it goes through the inner loop (while loop) ***i*** times (with the worst case input) … ie: from `0 to i-1`

   - we can express this as a summation: $\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$

Another example (identifying basic operations) ... cont

- so now we need to solve this summation (ie: change it into a closed-form equation)

$$C_{ops} = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

  – we know that $\sum_{j=0}^{i-1} 1 = i$, therefore we transform the equation to...

$$C_{ops} = \sum_{i=1}^{n-1} i$$

  – and, we know that $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$

  – therefore the number of basic ops is ... $C_{ops} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2}$

for large values of n

Homework from this week's lessons

(due start of lab next week)

Reading:

- chapters 1.1, 1.2, 1.3, 1.4
- chapters 2.1, 2.2, 2.3

Homework:

- chapter 1.1, page 8, question 8
- chapter 1.2, page 18, question 9
- chapter 1.3, page 23, question 1
- chapter 1.4, page 38, question 1
- chapter 2.1, page 52, question 2
- chapter 2.3, page 67, question 4

The End