ECE 2574: Data Structures and Algorithms - Analysis of Algorithms

C. L. Wyatt

Today we will look at how we analyze (compare and contrast) different algorithms and how that analysis can guide our choice of algorithm in a particular case.

- Warmup
- Algorithm Efficiency
- Algorithm Growth Rates
- Example

Algorithm Efficiency

What does it mean to say algorithm A is more efficient than algorithm B?
Definition of efficient: achieving results without wasted time or effort
Time: How long does it take?
Space: How much memory does it take ?
Power: How much power does it consume ?

Consider two implementations of a function that halves an integer:

Version 1

```
int halve(int n)
{
  return n/2;
}
```

Version 2

```
int halve(int n)
{
  return (n >> 1);
}
```

Is there any significant difference?

Warmup

Considering number of copies as the operation of interest, which requires more operations on average, insertion into a linked list or insertion into an array?

- insertion into a linked list 19%
- insertion into an array 81% (Correct)

To answer the Warmup lets compare the array and linked versions of insert

Array version (simplified)

```
void ArrayList<T>::insert(size_t position, const T& item)
{
  if(size == capacity){
    // need to reallocate
  } // capacity > size and size >= 1 now

  for(std::size_t i = size-1; i > position; --i){
    data[i+1] = data[i];
  }

  data[position-1] = item;
  ++size;
}
```

To answer the Warmup lets compare the array and linked versions of insert

A Singly Linked version (simplified)

```
void LinkedList<T>::insert(size_t position, const T& item)
{
  if(position == 1){
    Node<T> *temp = new Node<T>(item);
    temp->next = head;
    head = temp;
  }
  else{
    Node<T> *loc = find(position);
    Node<T> *temp = new Node<T>(item);
    temp->next = loc->next;
    loc->next = temp;
  }
  size += 1;
}
```

Problems comparing algorithms by their implementation

First you must have implementations. Then

- How are the algorithms coded, which language, style?
- What machine is used to test ?
- What data should be used to test ?

Algorithm Analysis uses mathematical techniques to quantify complexity in terms of basic operations.

The total operations typically depend on the number of data required, n.
Basic operations are those that all languages support

- additions, subtractions, multiply, divide
- comparisons
- assignments and/or function calls

What if each operation takes a different unit of time?

Example: Let a and c be the unit of time required to perform an assignment and comparison respectively.
Suppose we do (n+1) assignments and comparisons.
Then the total (theortical) time is: (n+1)(a+c)

Example: Towers of Hanoi, number of moves.

Move N disks from peg A to peg B, using C as an intermediary, at all times disks must be ordered largest to smallest vertically.

```
     |         |         |
    <1>        |         |
   <-2->       |         |
  <--3-->      |         |
===================================
     A         B         C
```

How many moves (and thus how long) does the algorithm take to solve the towers problem ?

```
Function Towers(First, Aux, Last, n)
Input: Names of three pegs: First, Aux, Last
Output: solution to problem

  if( n == 1)
      write("Move disk 1 from peg" First "to Last)
  else
      Towers(First, Last, Aux, n-1)
      write("Move disk" n "from peg" First "to Last)
      Towers(Aux, First, Last, n-1)
  endif
endfunction
```

The recurrence relation for the number of moves when solving the towers problem.

Let t be the number of moves made to solve for n disks, then t(n) = 2*t(n-1) + 1; with I.C. t(1) = 1 for n > 0
Prove: t(n) = 2^n-1

The recurrence relation for the number of moves when solving the towers problem.

Let t be the number of moves made to solve for n disks, then t(n) = 2*t(n-1) + 1; with I.C. t(1) = 1 for n > 0
Prove: t(n) = 2ˆn-1
Base case: t(1) = 1 by definition and 2ˆ1-1 = 1
Inductive step:

- assume t(k) = 2*t(k-1) + 1 = 2ˆk-1
- then t(k+1) = 2*t(k) + 1 from recurrence relation
- = 2(2ˆk-1) + 1 from assumption
- = 2ˆk+1-2 + 1
- = 2ˆ(k+1)-1

Towers of Hanoi, number of moves.

It is convenient when analyzing algorithms to have a closed form solution to the recurrence relations.
For example we just showed that the number of moves for n disks, t(n) = 2ˆn-1 for the Towers of Hanoi solution.
Recall: How fast does this grow as a function of the number of disks?
The number of moves is exponential in the size of the problem (n disks).
If the number of moves is the operation we care about, we say that it has *exponential complexity*.

Growth rate functions

Is there much difference between the following algorithms:

- Algorithm A takes 3nˆ2 operations
- Algorithm B takes 5nˆ2 operations
- Algorithm C takes 1000n operations

Growth rate functions

What we are really interested in is general trends.
We want to know that generally

- Algorithm A and B are proportional to nˆ2
- Algorithm C is proportional to n

If an algorithm is proportional to f(n), f(n) is the *growth-rate function* for that algorithm.

Next Actions and Reminders

- Read about Big-O notation in CH pp. 294-301
- No class Friday (Fall Break)
- Midterm Exam is in-class Friday 10/20. Closed Book, closed notes, written.