CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101

Instructor: Xiao Qin

Lecture 2: Complexity Analysis

What is an algorithm?

- Ideas behind computer programs
- Solves a general, well-specified problem
- Stays the same no matter
  - Which kind of hardware it is running on
  - Which programming language it is written in
- Specifies
  - The set of instances (input)
  - The desired properties of the output

Important Properties of Algorithms

- Correct
  - *always* returns the desired output for all legal instances of the problem.
- Efficient
  - Measured in terms of time or space
  - Time tends to be more important
  - The running time analysis allows us to improve our algorithms

Expressing Algorithms

More easily expressed

More precise

- English description
- Pseudocode
- High-level programming language

Pseudocode

- A shorthand for specifying algorithms
- Leaves out the implementation details
- Focus on the essence of the algorithm

```
Algorithm ArrayMax(A,n)
        Input: An array A storing n>=1 integers
        Output: The maximum element in A.

        currentMax ←– A[0]
        for i←–1 to n-1 do
                if currentMax < A[i] then
                currentMax ←–A[i]
        return currentMax
```

Analysis of Algorithms

- Why analyze algorithms?
  - evaluate algorithm performance
  - compare different algorithms
- Analyze what about them?
  - running time, memory usage
  - worst-case and “typical” case
- Analysis of algorithms compare algorithms and not programs

Analysis of Algorithms (cont.)

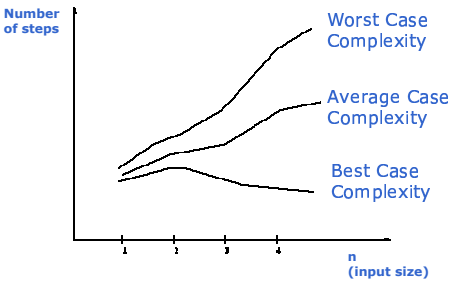
- If each line takes constant time the whole algorithm will take constant time, right?

  Wrong
- The majority of algorithms varies its number of steps based on the size of instance
- The efficiency of an algorithm is always stated as a function of the problem size
  - We generally use the variable N to represent the problem size

Algorithm Complexity

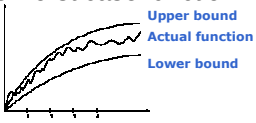
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Best, Worst, and Average Case Complexity

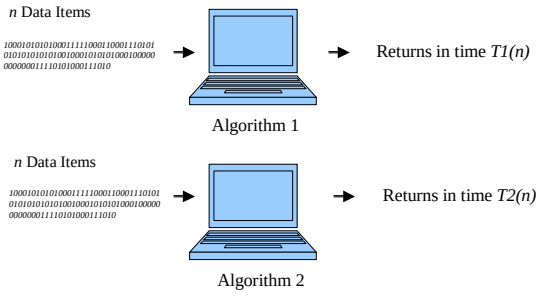


Doing the Analysis

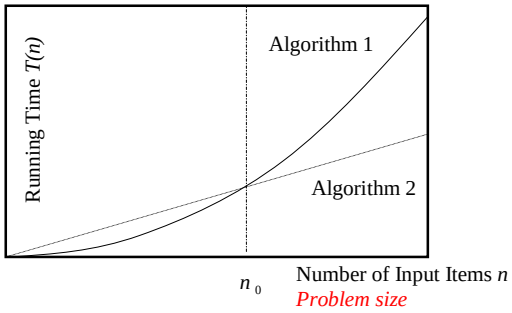
- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: try to find upper and lower bounds of the worst case function.



Running Time Analysis



Analysis of Running Time



Comparing Algorithms

- Establish a relative order among different algorithms, in terms of their relative rates of growth.
- The rates of growth are expressed as functions, which are generally in terms of the number of inputs *n*.

Asymptotic Analysis

- Asymptotic analysis of an algorithm describes the relative efficiency of an algorithm as n get very large.
- When you're dealing with small input size, most of algorithms will do
- When the input size is very large things change
- Eg: For very large *n*, algorithm 1 grows faster than algorithm 2.

An simple comparison

- Let's assume that you have 3 algorithms to sort a list
  - f(n) = n $\log_2$n
  - g(n) = n^2
  - h(n) = n^3
- Let's also assume that each step takes 1 microsecond (10^{-6})

n	*n log n*	*n^2*	*n^3*
10	33.2	100	1000
100	664	10000	1seg
1000	9966	1seg	16min
100000	1.7s	2.8 hours	31.7 years

Theoretical Framework

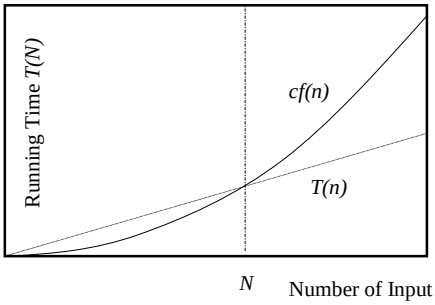
- "Big-Oh"
  - Definition:

$T(n) = O(f(n))$ if there are positive constants c and N such that $T(n) \leq c\, f(n)$ when $n \geq N$

  - This says that function *T(n)* grows at a rate no faster than *f(n);* thus *f(n)* is an upper bound on *T(n).*

Big-Oh Upper Bound

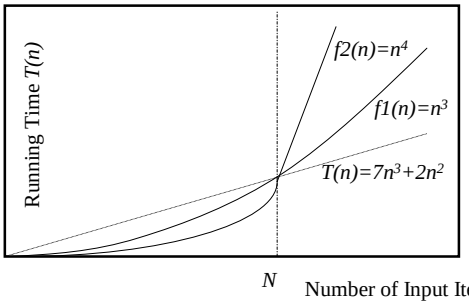
$T(n) = O(f(n))$ if there are positive constants c and N such that $T(n) \leq c\, f(n)$ when $n \geq N$



An Example

- Prove that $7n^3 + 2n^2 = O(n^3)$
  - Since $7n^3 + 2n^2 < 7n^3 + 2n^3 = 9n^3 (\text{for } n \geq 1)$
  - Then $7n^3 + 2n^2 = O(n^3)$ with $c = 9$ and $N = 1$
- Similarly, we can prove that $7n^3 + 2n^2 = O(n^4)$
  The first bound is tighter.

Tighter Upper Bound



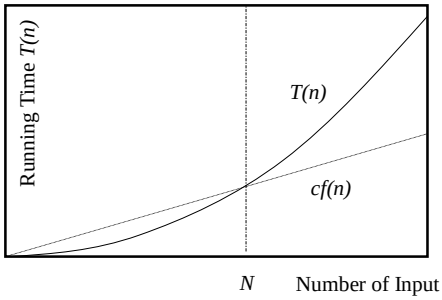
Big-Omega

Definition:

$T(n) = \Omega(f(n))$ if there are positive constants c and N such that $T(n) \geq c\, f(n)$ when $n \geq N$

– This says that function *T(n)* grows at a rate no slower than *f(n);* thus *f(n)* is a lower bound on *T(n).*

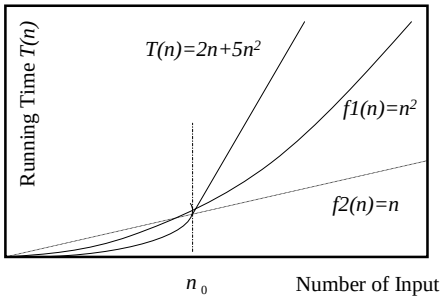
Big Omega Lower Bound



Big Omega Example

- Prove that $2n+5n^2=\Omega(n^2)$
  - Since $2n+5n^2>5n^2>1n^2(\text{ for } n\geq 1)$
  - Then $2n+5n^2=\Omega(n^2)$ *with* $c=1$ *and* $N=1$
- Similarly, we can prove that $2n+5n^2=\Omega(n)$
- The above bound is tighter.

Tighter Lower Bound



Big Theta

- Definition:
  $T(n)=\Theta(f(n))$ if and only if
  $T(n)=O(f(n))$ and $T(n)=\Omega(f(n))$
  - This says that function *T(n)* grows at the same rate as *f(n).*
- Put another way:

  $T(n)=\Theta(f(n))$ if there are positive constants c, d, and N such that $cf(n)\leq T(n)\leq d\,f(n)$ when $n\geq N$

Big Theta Example

- If two functions *f* and *g* are proportional then $f(N) = \Theta(g(n))$
- Since $log_A n = log_B n / log_B A$
  - Then: $\log_A(n) = \Theta(\log_B n)$
  - The base of the log is irrelevant.

little-oh

- Definition:

  $T(n) = o(f(n))$ if and only if

  $T(n) = O(f(n))$ and $T(n) \neq \Theta(f(n))$

  - This says that function *T(n)* grows at a rate strictly less than *f(n)*.

A Hierarchy of Growth Rates

$$c < \log n < \log^2 n < \log^k n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n! < n^n$$

General Rules

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then

(a) $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

(b) $T_1(n) * T_2(n) = O(f(n)) * O(g(n))$

Example: Algorithm A:

Step 1: Run algorithm A1 that takes $O(n^3)$ time

Step 2: Run algorithm A2 that takes $O(n^2)$ time

$T_A(n) = T_{A1}(n) + T_{A2}(n) = O(n^3) + O(n^2)$

$= \max(O(n^3), O(n^2)) = O(n^3)$

General Rules (cont.)

If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$

Example:

$T(n) = n^8 + 3n^5 + 4n^2 + 6 \quad = \Theta(n^8)$

$\log^k(n) = O(n)$ for any constant k.

An Example

Let $f(n) = n \log n$ and $g(n) = n^{1.5}$.

Which grows faster?

$nlogn$ → $log\ n$ → $log^2 n = O(n)$

$n^{1.5}$ → $n^{0.5}$ → n

$g(n)$ grows faster than $f(n)$.

Another Useful Technique

We can always determine the relative growth rates of two functions $f(n)$ and $g(n)$ by computing $L = \lim_{n\to\infty} f(n)/g(n)$, invoking L'Hopital's rule if necessary.

1) If $L = 0$, then $f(n) = o(g(n))$
2) If $L = c$, then $f(n) = \Theta(g(n))$
3) If $L = \infty$, then $g(n) = o(f(n))$

An Example

Let $f(n) = 7n^2 + n$ and $g(n) = n^2$.

Then $\lim_{N\to\infty} f(n)/g(n)$

$= \lim_{n\to\infty} 7 + 1/n$

$= 7 + \lim_{n\to\infty} 1/n = 7.$

So $f(n) = \Theta(n^2)$.

The Model

- To analyze algorithms in the formal framework, we need a model of computation.
- Our model has the standard set of simple instructions including addition, multiplication, comparison, and assignment. We assume:
  - It takes one time unit to do anything simple. This is not entirely realistic since not all operations take the same amount of time.
  - We also assume infinite memory. This won't take into account effects like page faults.

The Model (cont.)

- The most important resource to analyze is the running time.
  - We will not model the compiler and computer, although they affect the results.
  - Instead we focus primarily on the algorithm (not necessarily the program) and the input to the algorithm. Typically the size of the input (*n*) is the main consideration.

The Model (cont.)

- We define two functions, $T_{avg}(n)$ and $T_{worst}(n)$ as the average and worst-case running time of the algorithm.
- Generally the quantity required is the worst-case time, since this provides a bound for all input.
- The average running time is much harder to compute, let alone define.
  - For example, what is the "average" input to the algorithm? This is not always well defined.

A Simple Example

$$\sum_{i=1}^{n} i^2$$

Time Units to Compute

- 1 for the assignment.
- 1 assignment, *n+1* tests, and *n* increments.
- *n* loops of 3 units for an assignment, an addition, and two multiplications.
- 1 for the return statement.

Total: $1+(1+n+1+n)+3n+1 = 5n+4 = O(n)$

```
int sum (int n)
{
   int partial_sum = 0;
   int i;
   for (i = 1; i <= n; i++)
      partial_sum = partial_sum + (i * i );
   return partial_sum;
}
```

```
   i = 1;
if (i <= n) {
      partial_sum = partial_sum + (i * i );
      i++; /* i = i + 1 */
}
```

Analysis too complex

General Rules

- Loops
  - The running time of a "for" loop is at most the running time of the statements inside the "for" loop (including tests) times the number of iterations.

```
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
```

- The above example is *O(n).*

General Rules (cont.)

- Nested loops
  - The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
        sum = sum + i + j;
    }
}
```

3mn = O(mn)

- The above example is *O(mn).*

General Rules (cont.) A Question:

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
        for (k = 1; k <= p; k++) {
            sum = sum + i + j + k;
        }
    }
}
```

- *4pmn = O(pmn).*

General Rules (cont.)

- Consecutive statements
  - These just add, and the maximum is the one that counts.

```
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        sum = sum + i + j;
    }
}
```

← O(n)

← $O(n^2)$

- The above example is $O(n^2+n) = O(n^2)$.

General Rules (cont.)
A Question:

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        sum = sum + i + j;
    }
}
sum = sum / n;
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
for (j = 1;  j<= n; j++) {
    sum = sum + j*j;
}
```

- $n^2+1+n+n=O(n^2+2n+1) = O(n^2)$.

General Rules (Cont.)

- If (test) s1 else s2
  - The running time is never more than the running time of the test plus the larger of the running times of s1 and s2.

```
if (test == 1) {
    for (i = 1; i <= n; i++) {
        sum = sum + i;
    }
}
else for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            sum = sum + i + j;
        }
    }
```

- The running time = 1 + $max(n,n^2) = O(n^2)$.

General Rules (Cont.)

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        for (k = 1; k <= n; k++) {
            sum = sum + i + j + k;
        }
    }
}

if (test == 1) {
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            sum = sum + i;
        }
    }
}
else for (i = 1; i <= n; i++) {
        sum = sum + i + j;
    }
```

- The running time = $O(n^3) + O(n^2) = O(n^3)$.

General Rules (Cont.)

- Recursion:
  - Analyze from the inside (or deepest part) first and work outwards. If there are function calls, these must be analyzed first. This even works for recursive functions:

```
long factorial (int n) {
  if (n <= 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

Time Units to Compute
---
1 for the test.
1 for the multiplication statement.

What about the function call?

- The running time of *factorial(n) = T(n) = 2+T(n-1) = 4+T(n-2) = 6+T(n-3) = … = 2n = O(n).*

Another Recursive Example

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

Time Units to Compute
---
1 for the test.

1 for the addition.
What about the function calls?

- The running time of *fib(n) = T(n) = T(n-1) + T(n-2) + 2*. Can we estimate *T(n)* from this?

Fibonacci Analysis

Let $F(n)$ be the nth Fibonacci number. We can prove that (1) $T(n) \geq F(n)$ and (2) $F(n) \geq (3/2)^n$. Thus $T(n) \geq (3/2)^n$, which means the running time grows exponentially. This is quite bad.

Proof by Induction

- Proof by induction:
  - Show a theorem true for trivial case.
  - Assuming the theorem true up to case k-1.
  - Show true for k.
  - Thus true for all k.

Proof that $T(n) >= F(n)$

Proof by induction.

Base cases: $T(1) = 1 \geq F(1) = 1$,

$T(2) = 3 \geq F(2) = 1$.

By inductive hypothesis: $T(n\text{-}1) \geq F(n\text{-}1)$ and $T(n\text{-}2) \geq F(n\text{-}2)$. Then clearly:

$T(n\text{-}1) + T(n\text{-}2) \geq F(n\text{-}1) + F(n\text{-}2)$ and

$T(n) \geq F(n\text{-}1) + F(n\text{-}2) = F(n)$

Proof that $F(N) >= (3/2)^N$

Proof by induction.

Base cases: $F(6) = 8 \geq (3/2)^6 = 7.6$,

$F(7) = 13 \geq (3/2)^7 = 11.4$.

By inductive hypothesis: $F(n\text{-}1) \geq (n/2)^{n-1}$ and $F(n\text{-}2) \geq (3/2)^{n-2}$. Then: $F(N) = F(n\text{-}1) + F(n\text{-}2) \geq (3/2)^{n-1} + (3/2)^{n-2} =$

$(3/2)^{n-1}(1+(2/3)) = (3/2)^{n-1}(5/3) >$

$(3/2)^{n-1}(3/2) = (3/2)^n$.