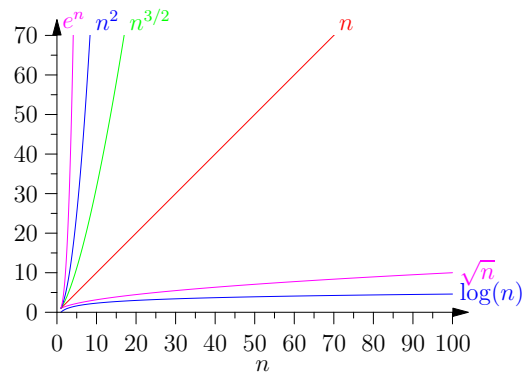


Lesson 30: Understand Time Complexity



Theta, Big-O, little-o, Big-Omega, little-omega

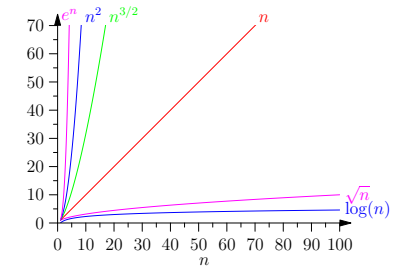
Recap

- We have seen many algorithms taking times of order 1, $\log(n)$, n , $n \log(n)$, n^2 , etc.
- Sometimes these are worst time, average time or best time results.
- We have lots of different notations, e.g. $O(1)$, $\Theta(\log(n))$, $\Omega(n^2)$, etc.
- What does it all mean?

1. Time Complexity Classes

- Theta— Θ
- Big O
- Little o
- Big Omega— Ω
- Little omega— ω

2. Computing Time Complexity



Complexity Class Sets

- The correct way to think about complexity classes is in terms of sets.
- Suppose we have an algorithm which takes an input of size n and computes an output in $f(n)$ operations.
- E.g. $f(n) = 4n^2 + 2n + 3$.
- We can partition all run times into sets by considering only the leading order term and ignoring the constant term.
- We denote these sets by $\Theta(g(n))$.
 - ★ $4n^2 + 2n + 3 \in \Theta(n^2)$
 - ★ $5n \log(n) + 3n + 2 \in \Theta(n \log(n))$

Defining $\Theta(g(n))$

- A function $f(n) \in \Theta(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \quad 0 < c < \infty$$

- E.g.

$$\lim_{n \rightarrow \infty} \frac{4n^2 + 2n + 3}{n^2} = 4$$

$$\lim_{n \rightarrow \infty} \frac{5n \log(n) + 3n + 2}{n \log(n)} = 5$$

Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n \log(n)$ algorithm will always run faster than an order n^2 algorithm even when they are run on different machines
- The constant is important in practice (if there are two algorithms A and B that are both $n \log(n)$, but algorithm A runs twice as fast as algorithm B , which one should you use?)
- Nevertheless, ignoring the constant is often essential to make analysis of algorithms doable

Ordering Complexity Classes

- We can define the relation $\Theta(f(n)) < \Theta(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

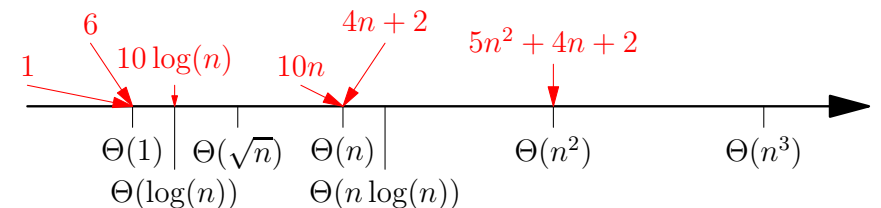
- Informally if algorithm A has time complexity $\Theta(f(n))$ and algorithm B has time complexity $\Theta(g(n))$ then if $\Theta(f(n)) < \Theta(g(n))$ algorithm A is faster for sufficiently large n
- The relation defines a **complete ordering**

The Complexity Line

- We can order all complexity classes. E.g.

$$\Theta(1) < \Theta(\log(n)) < \Theta(\sqrt{n}) < \Theta(n) < \Theta(n^2)$$

- We can depict this as a complexity line



- The line is dense (i.e. there are an uncountable infinity of complexity classes)

Complexity Dependent on Inputs

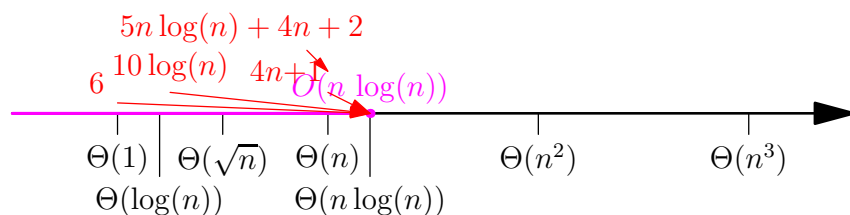
- The run time of many algorithms depends on the input
- In this case we can define different time complexities
 - ★ Worst case time complexity (the longest time an algorithm will take)
 - ★ Average complexity (the expected time averaged over all possible inputs)
 - ★ Best case time complexity (the shortest time an algorithm will take)—usually not very interesting
- Every algorithm will have a Θ complexity class for the worst, average and best time complexity

Unknown Time Complexity

- Algorithms are often rather complicated and knowing the exact time complexity (for either worst, average or best cases) might not be known
- In reality it will have some run time (e.g. $f(n) = 3n^2 \log(n) + 2n^2 - n + 3$) and will belong to a Θ time complexity set (e.g. $\Theta(n^2 \log(n))$) but we might not be able to calculate it
- However, we can usually bound the run times of algorithms

Big-O

- Big-O is an upper bound on the time complexity
- If an algorithm is $O(g(n))$ then its time complexity is no more than $\Theta(g(n))$



- I.e. $\Theta(f(n)) \leq \Theta(g(n))$ implies $f(n) \in O(g(n))$

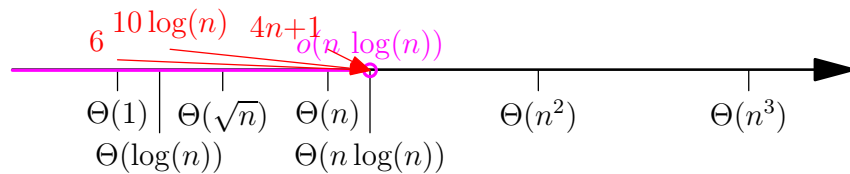
Upper Bounding Time Complexity

- Consider a program


```
// define stuff
for(int i=0; i<n; i++) {
    // do something
    if (/* some condition */) {
        for (int j=0; j<n; j++) {
            // do other stuff
        }
    }
}
// clean up
```
- If the `if` statements is never true this is a $\Theta(n)$ algorithm if it is always true it is a $\Theta(n^2)$ algorithm
- If we don't know then we can at least say that the run time is in $O(n^2)$ —we assume the worst, but the worst may never happen

Little-o

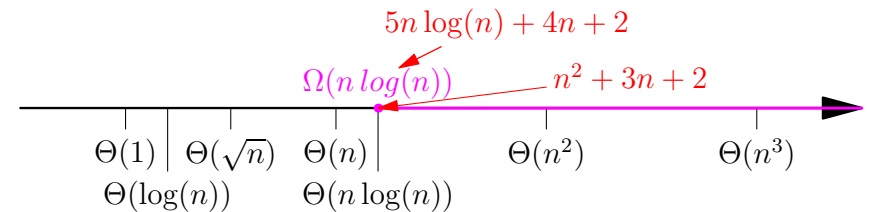
- Sometimes we want to say that the time complexity for an algorithm $\Theta(f(n))$ is **strictly less** than a known time complexity $\Theta(g(n))$



- I.e. $\Theta(f(n)) < \Theta(g(n))$ implies $f(n) \in o(g(n))$

Lower Bounds— Ω

- It is often easy to obtain a lower bound on an algorithm
- I.e. $\Theta(f(n)) \geq \Theta(g(n))$ implies $f(n) \in \Omega(g(n))$



Lower Bounding Time Complexity

- Returning to the program

```
// define stuff
for(int i=0; i<n; i++) {
    // do something
    if (/* some condition */) {
        for (int j=0; j<n; j++) {
            // do other stuff
        }
    }
}
// clean up
```

- We might not know how frequently the `if` statement is true, but we know in all cases the first `for` loop iterates over n
- Thus we know this algorithm is in $\Omega(n)$ —we assume the best, but the best may never happen

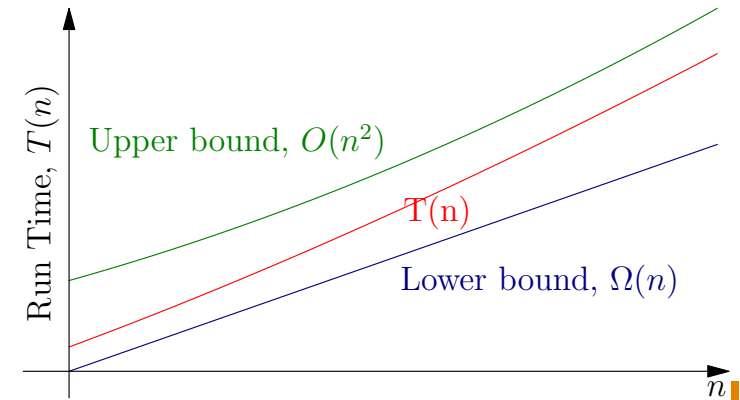
Little Omega— ω

- It is sometimes useful to talk about a strict lower bound

- I.e. $\Theta(f(n)) > \Theta(g(n))$ implies $f(n) \in \omega(g(n))$

Bounding Run Time Complexity

- When we are given an algorithm to analyse we want to compute $\Theta(n)$
- This may be difficult, however, it is often easy to find bounds



Proving Asymptotic Time Complexity

- If we know an algorithm is
 - ★ $T(n) \in O(f(n))$
 - ★ $T(n) \in \Omega(f(n))$
- Then $T(n) \in \Theta(f(n))$
- This is a common proof strategy

Meaning of Time Complexity

- Insertion sort has time complexity $\Theta(n^2)$
- Because it consists of two `for` loops
- It takes 2 seconds to sort 100 000 items
- How long does it take to sort 1 000 000 items?
- n increases by 10, time complexity increases by $10^2 = 100$
- Time taken is approximately 200 seconds or around 3.5 minutes

- When we talk about exponential time complexity we usually mean that

$$\log(T(n)) \in \Theta(n)$$

- This is true if

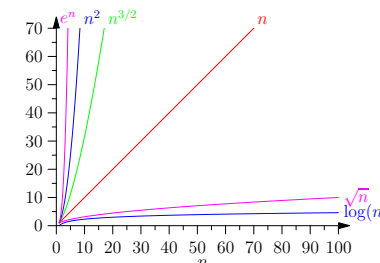
- $T(n) = 2^n$ $\log(T(n)) = n \log(2)$
- $T(n) = 6.1 e^{0.003n}$ $\log(T(n)) = 0.03n + \log(6.1)$
- $T(n) = n 10^n$ $\log(T(n)) = n \log(10) + \log(n)$

- Note that none of these are in complexity class $\Theta(e^n)$

1. Time Complexity Classes

- Theta— Θ
- Big O
- Little o
- Big Omega— Ω
- Little omega— ω

2. Computing Time Complexity



Counting For Loops

- How long does the following code take?

```
for(int i=0; i<n; i++) {
    // prepare stuff
}
for(int i=0; i<n; i++) {
    // do something
    for (int j=0; j<n; j++) {
        // do other stuff
    }
}
```

- The first for loop takes $\Theta(n)$ operations the second double for loop takes $\Theta(n^2)$
- Answer $\Theta(n^2)$

Recursion

- Determining time complexity is harder when we use recursion
- Consider Euclid's algorithm for determining the greatest common divisor

```
public static
long gcd(long m, long n)
{
    while (n!=0) {
        long rem = m%n;
        m = n;
        n = rem;
    }
    return m;
}
```

```
public static
long gcd(long m, long n)
{
    if (n==0)
        return m;
    else
        return gcd(n, m%n);
}
```

- This doesn't even look like a recursion

Example of gcd

- Example of Euclid's algorithm $\text{gcd}(1989, 1590)$ ■
- Sequence of remainders is 399, 393, 6, 3, 0 ■
- The greatest common divisor is 3 ■
- How long does it take compute $\text{gcd}(n, m)$ with $n > m$ ■
- This is subtle as could depend in a complex way on the pair n and m ■

Recursive Formulae

- An observation which makes the analysis relatively simple is that the remainder is reduced by at least 2 after two iterations ■
- To prove
 - ★ Using the recursion (assuming $m, n < 0$)
$$\text{gcd}(m, n) = \text{gcd}(n, \text{rem}(m, n)) = \text{gcd}(\text{rem}(m, n), \text{rem}(n, \text{rem}(m, n)))$$
 - ★ The proof follows by showing that $\text{rem}(n, \text{rem}(m, n)) < n/2$ ■
- Thus $T(n) < T(n/2) + 2$ ■

Solving Recursions

- To show that $T(n) \in O(\log(n))$ we observe
 - ★ Note that $T(1) = 1$ ■
$$T(n) < T(2^{-1}n) + 2 < T(2^{-2}n) + 4 < \dots < T(2^{-t}n) + 2t$$
 - ★ Choose $t = \lceil \log_2(n) \rceil$
 - ★ then $2^{-t}n = 2^{-\lceil \log_2(n) \rceil}n \leq 2^{-\log_2(n)}n = \frac{n}{n} = 1$ ■
 - ★ Thus $T(2^{-t}n) < T(1) = 1$ ■
 - ★ $T(n) < 1 + 2t = 1 + 2\lceil \log_2(n) \rceil \in O(\log(n))$ ■
- A huge calculation shows the the average number of iterations is about $(12 \log(2) \log(n))/\pi^2 + 1.47$ ■

Probability of Relative Primes

- Consider the following program to compute the probability of relative primes for all numbers up to n

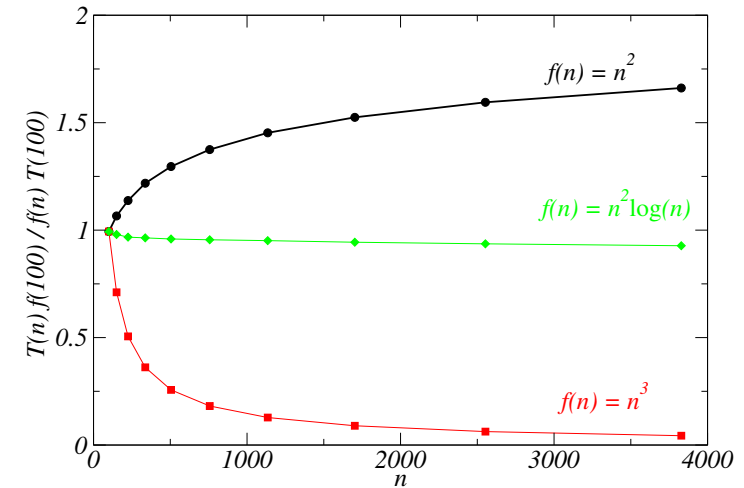
```
public static double probRelPrime(n)
{
    int rel=0, tot=0;
    for(int i=1; i<=n; i++)
        for(int j=i+1; j<=n; j++) {
            tot++;
            if (gcd(i, j)==1)
                rel++;
        }
    return (double) rel / tot;
} ■
```
- What is the time complexity? ■

Time Complexity

- Program involves two nested loops of size $O(n)$ ■
- Then we need to calculate $\gcd(i, j)$ at each iteration■
- Time complexity is $n \times n \times \log(n) = n^2 \log(n)$ ■
- How could we provide empirical support for this calculation?■

Testing Hypothesis

- We can test our hypothesis by scaling the run time by the complexity



Conclusions

- You should understand the difference between Θ , O , ω and Ω ■
- You need to be able to compute time complexity by loop counting■
- To compute time complexity for recursive functions you need to be able to obtain recurrence equations■
- You should be able to solve simple recurrence equations and sum up simple series■
- You should be able to prove more complicated results using proof by induction■
- Thank you for attending the course■