Asymptotic Analysis (Big O Notation)

Motivation

We want a way to quantify the efficiency of algorithms.

Example

Problem: given an integer r, count the number of lattice points (points with integer coordinates) that have positive x and y coordinates and lie within the circle centered on the origin with radius r.

```
int ans = 0;
for(int i = 1; i <= r; i++){
    for(int j = 1; j <= r; j++){
        if(i*i + j*j <= r*r) ans++;
    }
}
```

```
int ans = 0;
for(int i = 1; i <= r; i++){
    ans += sqrt(r*r - i*i);
}
```

Why not just measure actual speed?

- You could run the program on a computer that's 20 times faster and the algorithm would appear be 20 times faster.
- What if different architectures have different speeds for computing sqrt?

Python

```python
ans = 0
for i in range(1, r+1):
    for j in range(1, r+1):
        if i*i + j*j <= r*r: ans += 1
```

```python
ans = 0
for i in range(1, r+1):
    ans += int(sqrt(r*r - i*i))
```

Problems with directly measuring speed

- Differs based on architecture
- Differs based on environment it’s run in
- Differs based on programming language
- Differs based on slight implementation details
- Hard to calculate

Another approach

```
int ans = 0;
for(int i = 1; i <= r; i++){
    for(int j = 1; j <= r; j++){
        if(i*i + j*j <= r*r) ans++;
    }
}
```

```
int ans = 0;
for(int i = 1; i <= r; i++){
    ans += sqrt(r*r - i*i);
}
```

One thing we can still say about the algorithms is that for large r, the runtime of the first algorithm is proportional to r^2 and the runtime of the second algorithm is proportional to r.

If you time how long it takes to run each with r = 1000, then multiply r by 2, you would expect that the first algorithm takes 4 times as long to run and that the second only takes 2 times as long.

Big O notation

We look at the behaviour of the runtime of the algorithm for large inputs.

We ignore all but the most significant term.

We ignore constant factors.

$7N^3 + 2N^2 + 34 \rightarrow O(N^3)$

4N + 2N*sqrt(N) + 3N*log(N) -> O(N*sqrt(N))

Also sometimes called the **computational complexity** of the algorithm.

Advantages of Big O

+ Relatively easy to compute
+ Does not change when smaller details of the algorithm change (architecture, programming language, exact implementation...)
+ Still gives a useful way to compare efficiency of different algorithms
- Is not exact. Does not include constant factor

Formal definition

Big O notation is defined for functions.

$f(x) \in O(g(x))$ only if there exist constants x_0 and c such that $f(x) <= c*g(x)$ for all $x >= x_0$.

Other related notations

Notation	Informal explanation	Formal definition
$f(x) \in o(g(x))$	f(x) grows slower than g(x)	For every constant k, there exists a constant x_0 such that $f(x) < kg(x)$ for all $x >= x_0$.
$f(x) \in O(g(x))$	f(x) grows at most as fast as g(x)	There exist constants k and x_0 such that $f(x) <= kg(x)$ for all $x >= x_0$.
$f(x) \in \Theta(g(x))$	f(x) grows at the same speed as g(x)	There exist constants k_1, k_2, and x_0 such that $k_1g(x) <= f(x) <= k_2g(x)$ for all $x >= x_0$.
$f(x) \in \Omega(g(x))$	f(x) grows at least as fast as g(x)	There exist constants k and x_0 such that $f(x) >= kg(x)$ for all $x >= x_0$.
$f(x) \in \omega(g(x))$	f(x) grows faster than g(x)	For every constant k, there exists a constant x_0 such that $f(x) > kg(x)$ for all $x >= x_0$.

Note: k must also be positive in all cases. g(x) must be positive for all $x >= x_0$. f and g must be defined on some unbounded subset of the positive real numbers.