Lec_34.md

Lecture 34

Dec. 03/2020

Announcements

Quiz 7

- Monday, December 7th, 2020
- 25 Minutes in duration
- Take at any time in the day
  - Don't start later than 12:35
- Covers all the material
  - Up to Lab 4

Complexity Analysis

Recall that **execution time** of a program depends on

1. Size of the input `n`
2. Input itself (data in the array w.r.t t)

- `if(a[i]<t) count = count+1;` only runs depending on values of `a[i]` and `t`

3. Hardware/Compiler optimization
4. Steps run by program (algorithm)

Define a Step

A **step** is an instruction/operation that takes a *constant* amount of time

- **step** runtime is independent of the size of the problem (e.g. `n`)

Execution Time

We've reduced the measurement of execution time from a unit in seconds

- To a unit of **step**'s

Example 1

```
count = 0;
for(int i = 0;i < n;i++){
  count = count + 1;
}
cout << count;
```

Notes:

1. This loop has no dependence on input data

- So `Tbest = Tworst = Tavg`

Example 2

```
int count = 0;
for(i = 0;i < n;i++){
  if(a[i]<t) count = count+1;
}
cout << count;
```

Notes:

1. Let `c` be the steps inside the for loop
2. Let `d` be the steps outside the for loop
3. Best Case

- `Tbest = c*n+d`
  - Only one step inside the for loop (if)

4. Worst Case

- `Tworst = (c+1)*n+d`
  - Two steps inside the for loop (if and assign)

5. Average Case

- `Tavg = (c+0.5)*n+d`
  - Mean between best case scenario and worst case scenario

Time Complexity

If `n` is small

- 5,10,100

Then the difference between magnitudes of different time complexity orders is small anyways

- For case `1000*n`: `1000*50 uS = 0.05 S`
- For case `2^n`: `2^50 uS = 0.25 S`

But *only* when `n` is reaally large do the differences between algorithm runtime start to show

- Look at how a function runtime *grows* as the value `n` is scaled up
- For case `1000*n`: `1000*1000 uS = 1 S`
- For case `2^n`: `2^1000 uS = 3.4*10^286 centuries`

So look at how algorithm runtime changes based on size of input `n` - since large `n`'s are the only place differences in speed actually show

Asymptotic Behaviour

The execution time `T(n)` is said to be `O(g(n))`, denoted as `T(n)=O(g(n))` if there exists a constant `c` and a value of `n=n_0` such that

- `T(n) <= c*g(n)`
- `n >= n_0`

Big-O Notation

`O(g(n))`, or **Big-O Notation** represents the *order* of a function's runtime

- Can separate classes into **complexity classes**

Common categories:

1. `O(1)`

- This is rare
- Same runtime regardless of how large the size of the input is

2. `O(log_n)`

- Great algorithms

3. `O(n)`

- Linear algorithms

4. `O(nlogn)`

- Somewhere between linear and n-squared

5. `O(n^2)`

- Polynomial growth

6. `O(n^3)`
7. `O(2^n)`

Two more just for fun:

8. `O(n!)`

- The dreaded "factorial time"

9. `O(n^n)`

- There is something really wrong if your algorithm is here

Big O Notation Examples

```
for (i=0 ; i < n ; ++i) {
  O(1)
}
```

Notes:

1. The `O(1)` just represents that each step in the for loop runs in constant time

- Think about an assign statement, like `int x = 0;`

2. This is `O(n)` time

```
for (i=0 ; i < n ; ++i) {
  for (j=0 ; j < n ; ++j) {
    O(1)
  }
}
```

Notes:

1. This is `O(n^2)`

- Two nested for loops
  - The inside for loop runs `n` times for each run of the outer loop
  - Therefore time complexity should be of order `n*n`

```
for (i=0 ; i < n ; ++i) {
  c[i][j] = 0;
  for (j=0 ; j < n ; ++j) {
    for (k=0; k < n ; ++k) {
      c[i][j] = c[i][j] + b[i][k] * a[k][j];
    }
  }
}
```

Notes:

1. This is `O(n^3)`

- Three nested for loops
- Best, worst, and average is `O(n^3)`

```
for (i=0 ; i < n ; ++i) {
  for (j=0 ; j < i ; ++j) {
    O(1)
  }
}
```

Notes:

1. Notice the second for loop has condition `j < i`

- `i` is not a constant
  - `i` depends on `n`, the size of the problem

2. So the execution time of the outer loop is `0*c+1*c+2*c+3*c+4*c+5*c+...+(n-1)*c`

- Where `c` is a constant
- Look up triangular sum
- `(n)*(n-1)/2` runtime
  - Which is `O(n^2)`

3. Order `O(n^2)`

```
j = n;
do {
  O(1)
  j = j/2; // integer division
} while (j > 0)
```

1. This code involves repeated division by 2

- Thats sort of similar to `log_n`
- In fact, this code is of order `O(log_n)`