Time Complexity
Loops

CSE 3318 – Algorithms and Data Structures
Alexandra Stefan

University of Texas at Arlington

CLRS - reference

- Book reference subchapters (the first number is the chapter number):
  - 1.2 Efficiency
  - Problem 1-1
  - See the pseudocode conventions in 2.1
  - In 2.2 see section “Order of growth”.

  - 2.1 covers Insertion sort and discusses detailed instruction count part of that. You can revisit this subchapter after we talk about insertion sort.
  - 2.3 we will cover later on.

(CLRS 3rd edition)

Motivation for Big-Oh Notation

- Given an algorithm, we want to find a function that describes the time *performance* of the algorithm.
- Computing the number of instructions in detail is NOT desired:
  - It is complicated and the details are not important
  - The number of machine instructions and runtime depend on factors other than the algorithm:
    - Programming language
    - Compiler optimizations
    - Performance of the computer it runs on (CPU, memory)

    (There are some details that we would actually **NOT** want this function to include, because they can make a function unnecessarily complicated.)
- When comparing two algorithms we want to see which one is better for very large data. This is called the asymptotic behavior
  - It is not important what happens for small size data.
  - Asymptotic behavior = rate of growth = order of growth
- The Big-Oh notation describes the asymptotic behavior and greatly simplifies algorithmic analysis.

Starting a business?

- Facebook: more than 2.07 billion monthly active users
- Assume:
  - If you start a business that has the potential to grow this much, and
  - Currently you have 30 users
  - You need to buy software and have 2 offers:
    - N^2
    - 1000N, also a bit more expensive
  - Switching from one software to the other later on, is undesired (disruption of service, uncertainty, increased cost …)
- Which one do you choose?

Comparing growth of functions

- Comparing **linear**, **N lg N**, and **quadratic complexity**.

N	**N lg N**	$\mathbf{N^2}$
10^6 (1 million)	≈ 20 million	10^{12} (one trillion)
10^9 (1 billion)	≈ 30 billion	10^{18} (one quintillion)
10^{12} (1 trillion)	≈ 40 trillion	10^{24} (one septillion)

- Quadratic time algorithms become impractical (too slow) much faster than linear and **N lg N** algorithms.
- Of course, what we consider "impractical" depends on the application.
  - Some applications are more tolerant of longer running times.

N	**lgN**
1000	≈ 10
$10^6 = 1000^2$	≈ 2*10
$10^9 = 1000^3$	≈ 3*10
$10^{12} = 1000^4$	≈ 4*10

Θ (Theta) made simple

- Detailed instructions counts for many algorithms are polynomial functions. To make calculations simple while still keeping relevant information we will only look at the dominant term for such functions.
- For any function we look at the 'fastest growing term' or the dominant term.
  - E.g. for $f(n) = 15n^3 + 7n^2 + 3n + 20$, the dominant term is $15n^3$.
  - Functions of multiple variables may have more than one dominant term!

  Consider $f(n,m) = 27n^4m + 6n^3 + 7nm^2 + 100$

  (Ask yourself: What if m is small and n is large? What if n is small and m is large?)

- Use Θ,Theta, for the dominant term but without the constant.
  - This is a oversimplification of Θ. We will study Θ formally in future lectures.
- Notation: **$f(n) = \Theta(n^2)$**
  - if the dominant term of $f(n)$ is n^2.
- Given a function, e.g. $f(n) = 15n^3 + 7n^2 + 3n + 20$, find Theta:
  - find the dominant term: $15n^3$
  - remove the constant: n^3
  - $f(n) = \Theta(n^3)$

Function Call Inside Loop

If foo(i) has Θ(i), what is Θ for foo(0)?

The time complexity for a function call is NOT 1, but the complexity derived from its code.

Assume that `void foo(int n)` has time complexity:

$T_{foo}(n) = \Theta(n)$

```
// Example 1. Independent loops =>
// T(n) = n* Θ(n) = Θ(n²)
for (i = 1; i<=n; i++)
        foo(n);
```

n

Θ(n), use n

$\Theta\ (n^2)$

```
// Example 2. Dependent loops. MUST use table method:
// => T(n) = Θ(n) + Θ(n-1) +…+ Θ(2)+ Θ(1) = Θ(n²)
for (i = 1; i<=n; i++)
        foo(i); // --> Θ(i)
```

Steps (for example 2):

- Table
- Summation (based on table)
- Closed form
- Dominant term:
- Θ (Theta): **$\Theta(n^2)$**

Table for Example 2

i	**foo(i) takes:**	**Simplify to**
1	Θ(1)	1
2	Θ(2)	2
…	…	…
i	Θ(i)	i
…	…	…
n-1	Θ (n-1)	n-1
n	Θ (n)	n

$$T(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots + i + \cdots + n$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2 + n}{2}$$

Insertion Sort Time Complexity

i	Inner loop time complexity: Best : 1	Worst: i	Average: i/2
1	1	1	1/2
2	1	2	2/2
…	…		
N-2	1	N-2	(N-2)/2
N-1	1	N-1	(N-1)/2
Total	(N-1)	[N * (N-1)]/2	[N * (N-1)]/4
Order of magnitude	**N**	**N^2**	**N^2**
Data that produces it.	Sorted	Sorted in reverse order	Random data

Insertion sort is adaptive

=> $O(N^2)$

'Total' instructions in worst case:
T(N) = (N-1) + (N-2) + … 2 + 1 =
= [N * (N-1)]/2 -> N^2 order of magnitude
Note that the N^2 came from the summation, NOT because 'there is an N in the inner loop' (NOT because N * N).

See the Khan Academy for a discussion on the use of O(N2): https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort

Insertion Sort – Time Complexity Worksheet

- Assume all instructions have cost 1.
- If interested, see book for analysis using instruction cost.

See TedEd video

```
void insertion_sort(int A[],int N){
  int i,k,key;
  for (i=1; i<N; i++)
  key = A[i];
  // insert A[i] in the
  // sorted sequence A[0…i-1]
  k = i-1;
  while (k>=0) and (A[k]>key)
     A[k+1] = A[k];
     k = k–1;
  }
  A[k+1] = key;
}
```

i	Inner loop iterations: Best : 1	Worst: i	Average: i/2
1			
2			
…			
N-2			
N-1			

At most:

At least:

Review these summations

Useful processing of summation techniques (for Θ or dominant term calculations)

Note that some of these will NOT compute the EXACT solution for the summation

Independent case (term in summation does not have the variable of the summation).

$$\sum_{k=1}^{N} S = S + S + S + \cdots . + S = NS \quad \left(= S\sum_{k=1}^{N} 1 = SN\right)$$

Pull constant in front of summation: $\sum_{k=1}^{N}(Sk) = S\sum_{k=1}^{N} k = S\frac{N(N+1)}{2} = \Theta(SN^2)$

Break summation in two summations

$$\sum_{k=1}^{N}(kS + k^2) = \sum_{k=1}^{N} kS + \sum_{k=1}^{N} k^2 = S\sum_{k=1}^{N} k + \sum_{k=1}^{N} k^2 = S\frac{N(N+1)}{2} + \frac{N(N+1)(2N+1)}{6} = \Theta(SN^2 + N^3)$$

$Drop\ lower\ order\ term\ from\ summation\ term. E.g. 10k\ is\ lower\ order\ compared\ to\ k^2:$

$$\sum_{k=1}^{N}(10k + k^2) = \sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6} = \Theta(N^3)$$

$Use\ approximation\ by\ integrals\ for\ increasing\ or\ decreasing\ f(k):$

$$\sum_{S}^{N} f(k) = \Theta\big(F(N) - F(S)\big)\ (where\ F\ is\ the\ antiderivative\ of\ f)$$

Estimate runtime

- Problem:

  The total number of instructions in a program (or a piece of code) is 10^{12} and it runs on a computer that executes 10^{9} instructions per second. How long will it take to run this program? Give the answer in seconds. If it is very large, transform it in larger units (hours, days, years).

- Summary:
  - Total instructions: 10^{12}
  - Speed: 10^{9} instructions/second
- Answer:
  - Time = (total instructions)/speed =

  (10^{12} instructions) / (10^{9} instr/sec) = 10^{3} seconds ~ 15 minutes

- Note that this computation is similar to computing the time it takes to travel a certain distance (e.g. 120miles) given the speed (e.g. 60 miles/hour).

Self study

Estimate runtime

- A slightly different way to formulate the same problem:
  - total number of instructions in a program (or a piece of code) is 10^{12} and
  - it runs on a computer that executes one instruction in one nanosecond (10^{-9} seconds)
  - How long will it take to run this program? Give the answer in seconds. If it is very large, transform it in larger units (hours, days, years)
- Summary:
  - 10^{12} total instructions
  - **10^{-9}** seconds per instruction
- Answer:
  - Time = (total instructions) ***** (seconds per instruction) =

  (10^{12} instructions)*** (10^{-9} sec/instr)** = 10^3 seconds ~ 15 minutes

Self study

Counting instructions: detailed

Answers

1 + 1 + n* (1 + 1 + body_instr_count)

false true

for (init; cond; update) // assume the condition is TRUE n times

body

// Example A. Notice the ; at the end of the for loop.

```
temp = 5; x = temp * 2;
for (i = 0; i<n; i++)    ;
```

2 + 1 + 1 + n* (1 + 1 + 0) = 4 + 2n

// Example B (source: Dr. Bob Weems) - NOT REQUIRED

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

1 + 1 + n * (1 + 1 + ___) = 2 + n * (2 + 2 + 5*p + 3*p*r) = 2 + 4*n + 5*n*p + 3*n*p*r

1 + 1 + p* (1 + 1 + 1 + ____) = 2 + p * (3 + 2 + 3*r) = 2 + 5*p + 3*p*r

1 + 1 + r* (1 + 1 + 1) = 2 + 3 * r

Counting instructions: sequential vs nested loops

Answers

___ + ___ = (2 + 3*n) + (2 + 4*p + 3*p*r)

```
// Example sequential vs nested
for (t=0; t<n; t++)
    printf("A");
for (i=0; i<p; i++) {
    for (k=0; k<r; k++)
        printf("B");
    }
```

1 + 1 + n* (1 + 1 + 1) = 2 + 3*n

1 + 1 + p* (1 + 1 + ___) = 2 + p * (2 + 2 + 3*r) = 2 + 4*p + 3*p*r

1 + 1 + r* (1 + 1 + 1) = 2 + 3 * r