Analysis of Algorithms

Time for instructions on a computer that executes 1 billion instructions per second

n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n\log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01μs	0.003μs	0.033μs	0.1μs	1μs
20	0.02μs	0.004μs	0.086μs	0.4μs	1ms
30	0.03μs	0.005μs	0.147μs	0.9μs	1s
40	0.04μs	0.005μs	0.213μs	1.6μs	18.3min
50	0.05μs	0.006μs	0.282μs	2.5μs	13 days
100	0.10μs	0.007μs	0.664μs	10μs	$\boxed{4\times10^{13}\text{ years}}$
1000	1.00μs	0.010μs	9.966μs	1ms	
10,000	10μs	0.013μs	130μs	100ms	
100,000	0.10ms	0.017μs	1.67ms	10s	
1,000,000	1 ms	0.020μs	19.93ms	16.7m	
10,000,000	0.01s	0.023μs	0.23s	1.16 days	
100,000,000	0.10s	0.027μs	2.66s	115.7 days	

Time and space

- **Analyzing an algorithm means:**
  - developing a formula for predicting *how **fast*** an algorithm is, based on the *size of the input* (**time complexity**), and/or
  - developing a formula for predicting *how much **memory*** an algorithm requires, based on the *size of the input* (**space complexity**)
- Usually **time** is our biggest concern
  - Most algorithms require a fixed amount of space

What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be a parameter that determines the actual time (or space) required

Exact values

- It is sometimes possible, *in **assembly** language*, to compute *exact* time and space requirements
  - We know exactly how many **bytes** and how many **cycles** each machine instruction takes
  - For a problem with a known sequence of steps (factorial, fibonacci), we can determine how many instructions of each type are required
- **However, often the exact sequence of steps cannot be known in advance**
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)

Higher-level languages

- In a higher-level language, we *do not know* how long each operation takes
  - Which is faster, `x < 10` or `x <= 9` ?
  - We don't know exactly what the compiler does with this
  - The **compiler** almost certainly optimizes the test anyway (replacing the slower version with the faster one)
- In a higher-level language we *cannot* do an exact analysis
  - Our timing analyses will use *major* oversimplifications
  - Nevertheless, we can get some very useful results

Average, best, and worst cases

- Usually we would like to find the ***average*** time to perform an algorithm
- However,
  - Sometimes the “average” isn’t well defined
    - Example: Sorting an “average” array
      - Time typically depends on how out of order the array is
      - How out of order is the “average” unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the ***worst*** (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest

Constant time

- *Constant time* means there is some constant k such that this operation always takes k nanoseconds
- A statement takes constant time if:
  - It does not include a loop
  - It does not include calling a method whose time is unknown or is not a constant
- If a statement involves a choice (if or switch) among operations, each of which takes constant time, we consider the statement to take constant time
  - This is consistent with *worst-case analysis*

Example

Consider the following algorithm. (Assume that all variables are properly declared.)

	Line	# of operations
cout << "Enter two numbers";	1	1
cin >> num1 >> num2;	2	2
if (num1 >= num2)	3	1
max = num1;	4	1
else	5	
max = num2;	6	1
cout << "The maximum number is: " << max << endl;	7	3

Either Line 4 or Line 6 executes. Therefore, the total number of operations executed is 1 + 2 + 1 + 1 + 3 = 8. In this algorithm, the number of operations executed is **fixed**.

Linear time

- We may not be able to predict to the nanosecond how long a program will take, but do know *some* things about timing:

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

  - This loop takes time `k*n + c`, for some constants `k` and `c`

    `k` : How long it takes to go through the loop once
    (the time for `j = j * i`, plus loop overhead)

    `n` : The number of times through the loop
    (we can use this as the "size" of the problem)

    `c` : The time it takes to initialize the loop
  - The total time `k*n + c` is *linear in n*

Example

	Lines	# of operations
sum = 0;	1	1
num = 10;	2	1
while (num != -1)	3	1
{	4	
sum = sum + num;	5	2
num = num -1;	6	2
}	7	
cout << sum;	8	1

If the while loop executes n times, the number of operations executed is: 5n + 4.

Cont.

n	5n+4
10	54
100	504
1000	5004
10000	50004

For very large values of n, the term 5n becomes the dominating term and the term 4 become negligible.

Constant time is (usually) better than linear time

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes `5000` time units
  - Algorithm B takes `100*n` time units
- Which is better?
  - Clearly, algorithm B is better if our problem size is small, that is, if `n < 50`
  - Algorithm A is better for larger problems, with `n > 50`
  - So B is better on small problems that are quick anyway
  - But A is better for large problems, *where it matters more*
- We usually care most about very large problems
  - But not always!

What about the constants?

- An added constant, `f(n)+c`, becomes less and less important as `n` gets larger
- A constant multiplier, `k*f(n)`, does *not* get less important, but...
  - Improving `k` gives a *linear* speedup (cutting `k` in half cuts the time required in half)
  - Improving `k` is usually accomplished by careful code optimization, not by better algorithms
  - We aren't that concerned with *only* linear speedups.
- Bottom line: ***Forget the constants!***

Simplifying the formulae

- Throwing out the constants is one of *two* things we do in analysis of algorithms
  - By throwing out constants, we simplify $12n^2 + 35$ to just n^2
- Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)
  - **We usually discard all but the *highest-order* term**
    - We simplify $n^2 + 3n + 5$ to just n^2

Big O notation

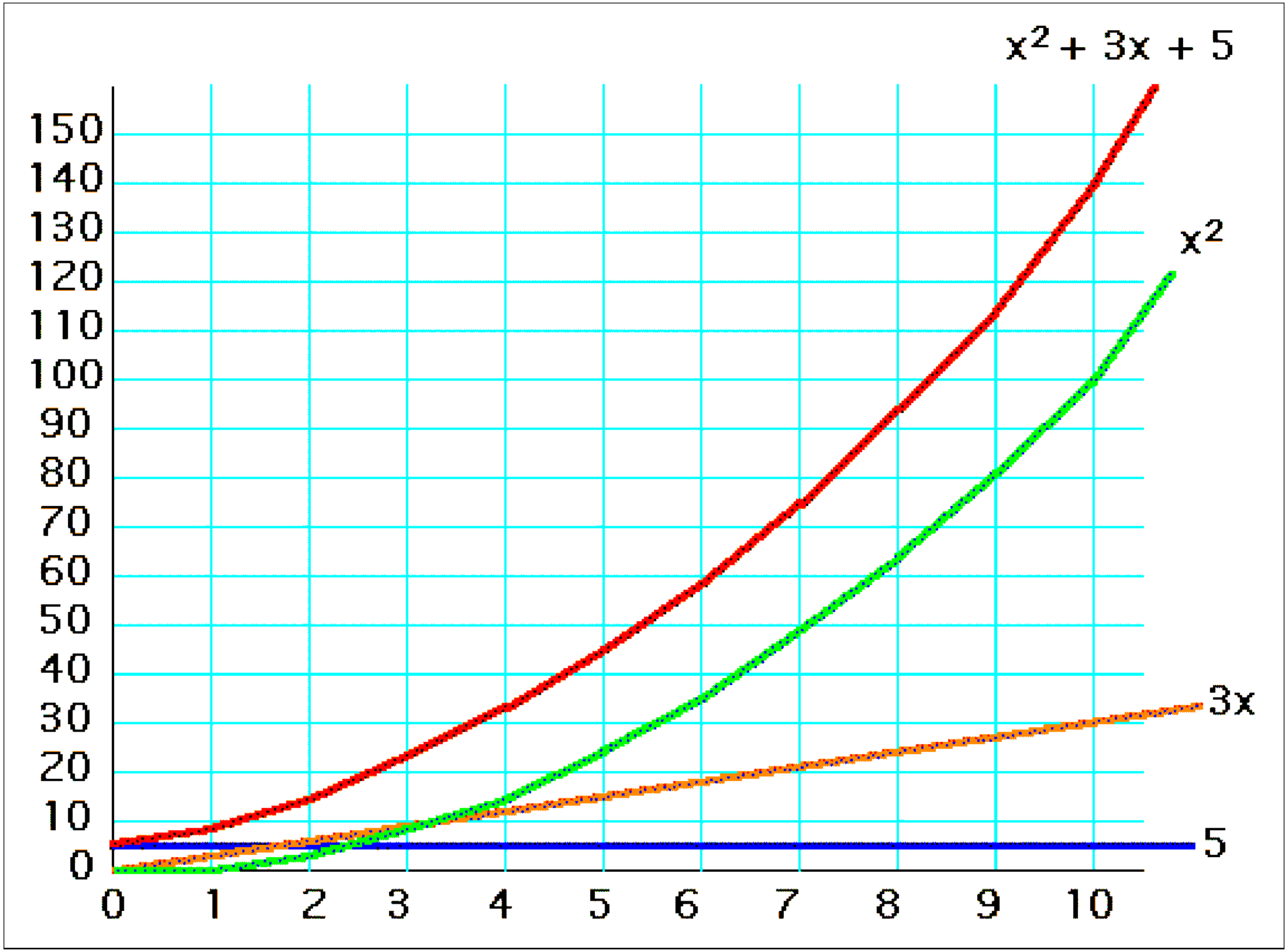
- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants
- If an algorithm takes $12n^3+4n^2+8n+35$ time, we simplify this formula to just n^3
- We say the algorithm requires $O(n^3)$ time
  - We call this Big O notation

Can we justify Big O notation?

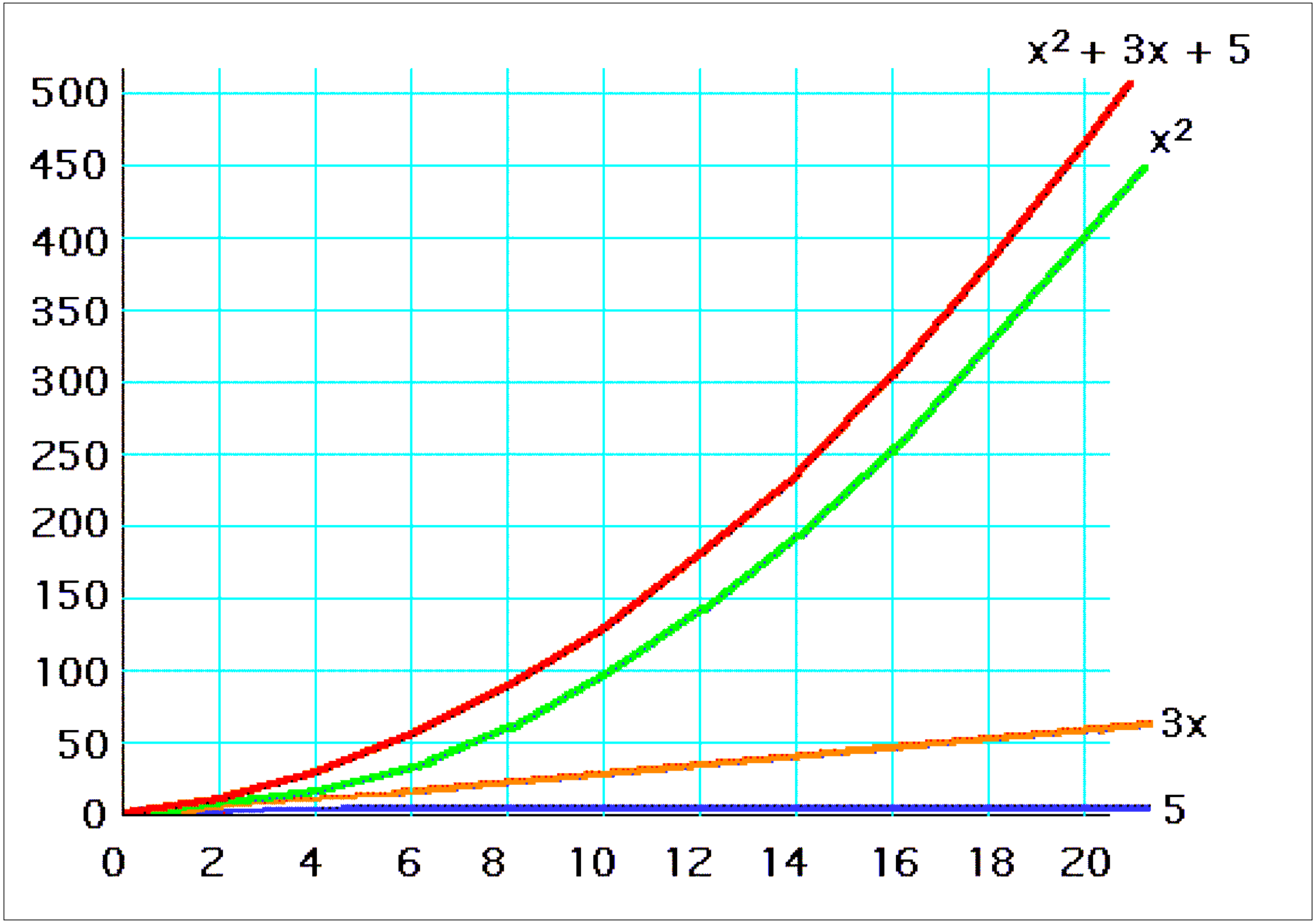
- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - **For sufficiently large problem sizes, the highest-order term swamps all the rest!**
- Consider $R = x^2 + 3x + 5$ as x varies:

$x = 0$	$x^2 = 0$	$3x = 0$	$5 = 5$	$R = 5$
$x = 10$	$x^2 = 100$	$3x = 30$	$5 = 5$	$R = 135$
$x = 100$	$x^2 = 10000$	$3x = 300$	$5 = 5$	$R = 10,305$
$x = 1000$	$x^2 = 1000000$	$3x = 3000$	$5 = 5$	$R = 1,003,005$
$x = 10,000$	$x^2 = 10^8$	$3x = 3*10^4$	$5 = 5$	$R = 100,030,005$
$x = 100,000$	$x^2 = 10^{10}$	$3x = 3*10^5$	$5 = 5$	$R = 10,000,300,005$

$y = x^2 + 3x + 5$, for $x=1..10$



$y = x^2 + 3x + 5$, for x=1..20



Common time complexities

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(n^k)$ polynomial time
- $O(2^n)$ exponential time

WORSE

NP-complete problem

- Hard problem:
  - Most problems discussed are efficient (poly time)
  - An interesting set of hard problems: NP-complete.
- Why interesting:
  - Not known whether efficient algorithms exist for them.
  - If exist for one, then exist for all.
  - A small change may cause big change.
- Why important:
  - Arise surprisingly often in real world.
  - **Not waste time on trying to find an efficient algorithm to get best solution, instead find approximate or near-optimal solution.**
- **Example**: traveling-salesman problem. Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Bjarne Stroustrup

- A Danish computer scientist, most notable for the creation and development of C++
- "I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone".

Example 1

Find the running time, worst time, complexity, or Big-Oh analysis for the following code

```
for (i = 0; i < n; i++)
   for (j = 0; j < n;  j++)
      cin >> A[i][j];
```

Number of times cin >> A[i][j]; executed: n^2
Complexity: $O(n^2)$

Example 2

```
for (i = 0; i <= n; i++)
 for (j = 0; j <= i; j++)
    A[i][j] = 0;
```

$O(n^2)$

Example 3

```
for (i = 0; i < n; i++)
   {
    for (j = 0; j < n;  j++)
            A[i][j] = j*2;
    for (k = 0; k <2* n;  j++)
           A[i][k] = k *3;
    }
```

$O(n^2)$

Example 4

```
for (i = 0; i < n; i++)
  {
    for (j = 0; j < n;  j++)
          A[i][j]= i *j;
    for (k = 0; k <2* n;  k++)
          for (m = 0; m <2* n;  m++)
                sum = sum +1;
  }
```

$O(n^3)$

Example 5

```
void main()
{
        int i,j, tofind, A[100], n = 100;
        for (j = 0; j < n; j++)
                A[j] = j * 2;
        i = 0;
        cin >> tofind;
        while (A[i] != tofind) i++;
        if (i > n)
          cout << "not found";
        else
            cout << "found";
}
```

O(n)

Summary

- http://bigocheatsheet.com/