SCC120
Fundamentals of Computer Science

Introduction to Algorithms

The Problem of Sorting

Input

- sequence $(a_1, a_2, \ldots, a_n)$ of numbers

Output (Sorting in increasing order)

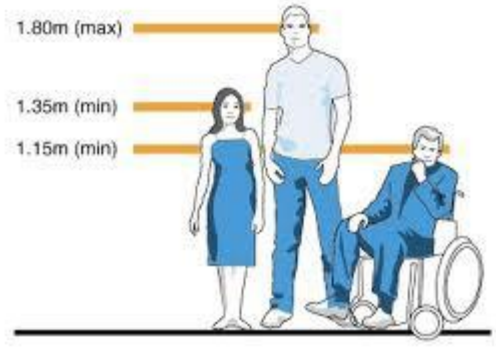
- Permutation $(a'_1, a'_2, \ldots, a'_n)$ of the sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Example

- Input: 7, -5, 2, 16, 4
- Output: -5, 2, 4, 7, 16

The Problem of Sorting

- Also applies to alphabet, size of planets, people's heights
- We will be sorting numbers, but algorithm applies to sorting other things as well



Many Sorting Algorithms

For example:

- Insertion Sort
- Merge Sort
- Quick Sort
- Shell Sort

Insertion Sort

- Go through example: 7, -5, 2, 16, 4

Insertion Sort Function

```
void insertionSort (int A[]) {
      for (int i=1; i<A.length; i++) {
            int x = A[i];
            int j;
            for (j=i-1; j>=0 && A[j]>x; j--) {
                  A[j+1] = A[j];
            }
            A[j+1] = x;
      }
}
```

Insertion Sort: Cost

The outer loop is evaluated n-1 times

How many times is the inner loop evaluated?

- Depends on the array to be sorted

Insertion Sort: Best-case Cost

Best-case: the array is already completely sorted, so no “shifting” of array elements is required

- We only test the condition of the inner loop once and the body is never executed

- Let cost of operations in outer loop be C_1
- Let cost of initialisation steps be C_2

$$T(n) = (n-1)\ C_1 + C_2$$

Insertion Sort: Worst-case Cost

Worst-case: the array is sorted in reverse order, so each item has to be moved to the front of the array

- Let cost of operations in outer loop, **neglecting the cost of the inner loop**, be C_1 (that is, the cost of the underlined operations)
- Therefore, total cost of the outer loop over n-1 iterations, **neglecting the cost of the inner loop,** is (n-1) C_1

```
void insertionSort (int A[]) {
        for (int i=1; i<A.length; i++) {
                int x = A[i];
                int j;
                for (j=i-1; j>=0 && A[j]>x; j--) {
                        A[j+1] = A[j];
                }
                A[j+1] = x;
        }
}
```

Insertion Sort Worst Case (cont.)

- Let the cost of operations of inner loop (underlined) be C_2

```
void insertionSort (int A[]) {
        for (int i=1; i<A.length; i++) {
                int x = A[i];
                int j;
                for (j=i-1; j>=0 && A[j]>x; j--) {
                        A[j+1] = A[j];
                }
                A[j+1] = x;
        }
}
```

- In first iteration of outer loop, one iteration of inner loop is executed. So cost of inner is C_2
- In second iteration of outer, two iterations of inner: $2C_2$

…

- In n-1th iteration of outer, n-1 iterations of inner: $(n-1)C_2$

Insertion Sort Worst Case (cont.)

Total worst case cost T(n)

= total cost of outer loop (neglecting inner loop) + **total cost of inner loop** + initialisation cost

$= (n-1)\, C_1 + \mathbf{C_2 + 2C_2 + \ldots + (n-1)C_2} + C_3$

$= (n-1)\, C_1 + \mathbf{(1 + 2 + \ldots + (n-1))C_2} + C_3$

$= (n-1)\, C_1 + \mathbf{0.5n(n-1)C_2} + C_3$

$= (n-1)\, C_1 + \mathbf{0.5(n^2-n)C_2} + C_3$

[By sum of arithmetic series]

Complexity is quadratic because of the n^2 term

Insertion Sort: Cost

What’s the average case of the insertion sort?

SCC120
Asymptotic Efficiencies

Overview

- Why Big O notation?
- $O(1)$, $O(\log N)$, $O(N)$, $O(N^2)$, $O(2^N)$
- Properties of Big O
- Exercise/example

Overview

- Why Big O notation?
- $O(1)$, $O(\log N)$, $O(N)$, $O(N^2)$, $O(2^N)$
- Properties of Big O
- Exercise/example

Cases

- Consider an algorithm A with input of size n
- Worst case for A
  - A particular input of size n that produces the longest running time
  - Insertion sort: array sorted in reverse order
- Average case for A
  - The average runtime over all inputs of size n, assuming some probability distribution over the inputs
- Best case for A
  - A particular input of size n that produces the shortest running time
  - Insertion sort: array in sorted order

Why Count Steps?

- Example, $T(n) = 5n^2 + 5$
- Gives a logical idea of an algorithm's runtime in terms of input size
- Independent of machines (and machine-related constants), operating systems, and programming languages

Big O notation

- An even more abstract idea
  - If $T(n) = 5n^2 + 5$, then $T(n)$ is $O(n^2)$
  - If $T(n) = 100n^2 + n + 5$, then $T(n)$ is $O(n^2)$
  - Different $T(n)$ but same Big O characterization
- Big O captures **asymptotic** efficiency of algorithms
  - Running time with size of input *in the limit*
    - As input increases without bound
  - Captures *order of growth*

O Meaning

- If an algorithm's runtime is O(f(n)), then it means that the algorithm's runtime grows as fast as f(n) in the limit
  - If runtime is $O(n^2)$, then runtime grows *as fast as* n^2 in the limit

O(1)

Constant time:

O(1) describes an algorithm that will always execute in the same time regardless of input size

Example: accessing any element in array/string

```
bool isFirstElementNull(char str[])
{
	if (str[0] == null)
		return true;
	return false;
}
```

O(1)

- Another example we did before:
  Compute average of array of 5 integers
  $T(N) = C_1 \times 5 + C_2$ and this is O(1)

- Exact number does not matter, as long as it is constant (with respect to input size)
- For $T(N) = C_1 \times 500 + C_2$, this is still O(1)

- In general, $T(N) = k$ (where k is constant) is O(1)

O(log N)

Logarithmic time (highly efficient):

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size

Example:

```
int count = 0;
while (N > 1) {
      count++;
      N = N/2;
}
```

O(log N)

- Code is $T(N) = C_1 \times \log_2 N + C_2$ and this is O(log N)

- For this code:

```
int count = 0;
while (N > 1) {
      count++;
      N = N/10;
}
```

- This is $T(N) = C_1 \times \log_{10} N + C_2$ and is also O(log N), even though base 10

O(N)

Linear time:

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the input size

Example:

Search for integer within array

We did this before and we called it linear search

O(N)

- Average of N integers: $T(N) = C_1 \times N + C_2$ and this is O(N)
- Find minimum of N integers:
  $T(N) = C_1 \times N + C_2$
  and this is O(N)

- In general, traversing N integers or objects or elements in some way is O(N)
- If N is large, the constants are not significant, so that's why "doubling N doubles the time taken"

$O(N^2)$ and $O(N^3)$

Quadratic Time:

An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size

Cubic Time:

An algorithm is said to run in cubic time if its time execution is proportional to the cube of the input size

$O(N^2)$ and $O(N^3)$

```
int total = 0;
for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
            total += arr[i][j];
```

- Code is $T(N) = C_1N^2 + C_2N + C_3$ and this is $O(N^2)$
- In general, $O(N^c)$ where c>=1 is polynomial-time

$O(2^N)$

Exponential Time (highly inefficient):

An algorithm is said to run in exponential time if its time execution is exponential with respect to its input size

Example:

```
int up_bound = (int) pow(2.0, N);
for (int i=0; i<up_bound; i++)
      print i;
```

$O(2^N)$

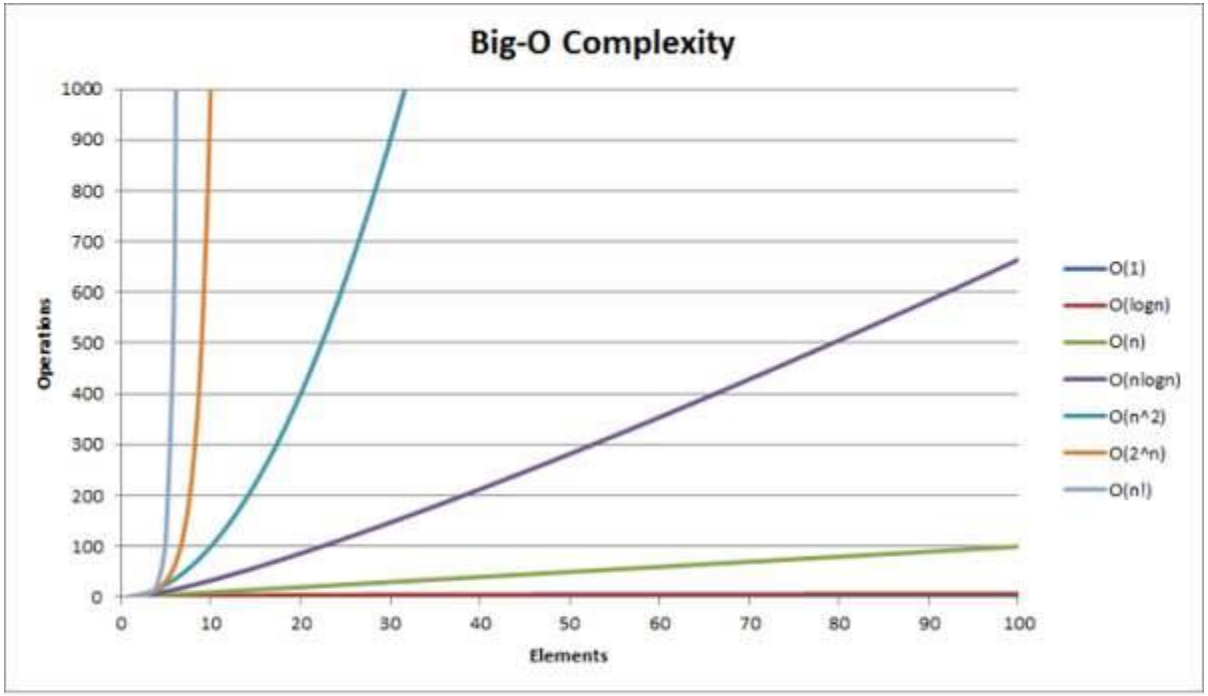
- Code is $T(N) = C_1 \times 2^N + C_2$ and this is $O(2^N)$

$O(2^N)$

- An interesting example: given N bits, list all possible of binary numbers
- There are 2^N such numbers

- $O(3^N)$ and $O(10^N)$ are also exponential-time
- In general, $O(c^N)$ where $c>1$ is exponential-time

Running Time Graphs



Running Times Table

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Big O notation

$O(1) < O(\log N) < O(N) < O(N^2) < O(N^3) < O(2^N)$

- The difference between these can be large!

Overview

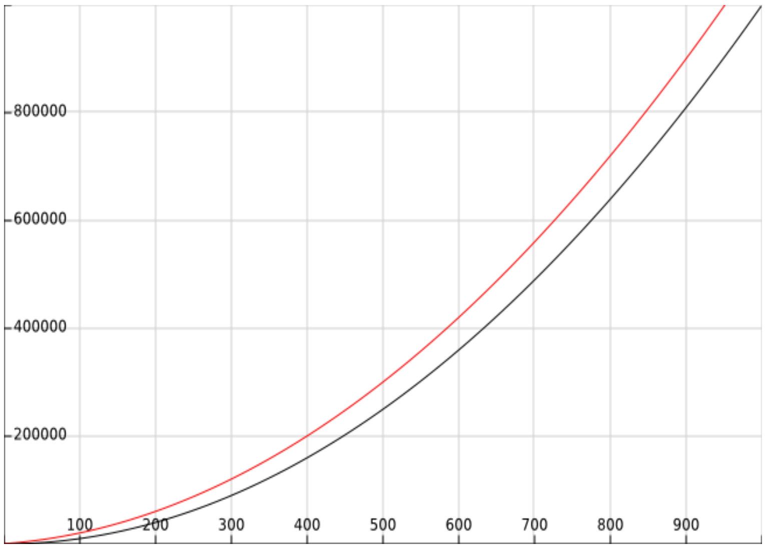
- Why Big O notation?
- $O(1)$, $O(\log N)$, $O(N)$, $O(N^2)$, $O(2^N)$
- $\boxed{\text{Properties of Big O}}$
- Exercise/example

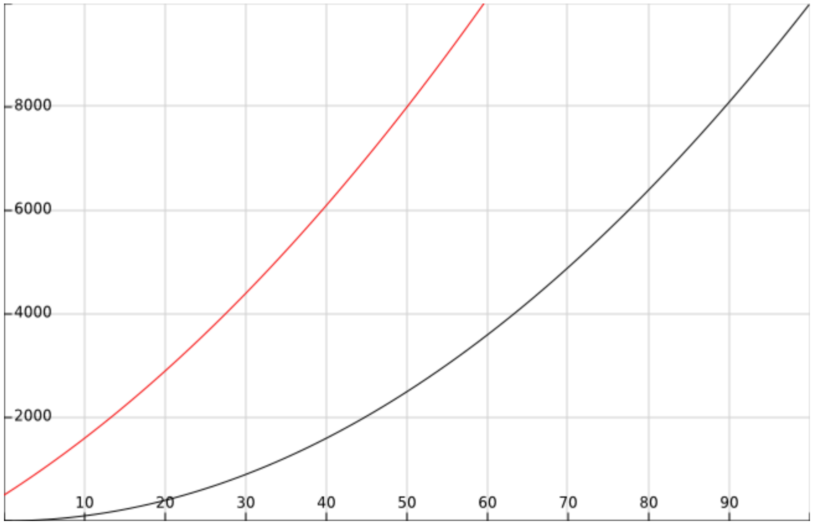
Properties of Big O

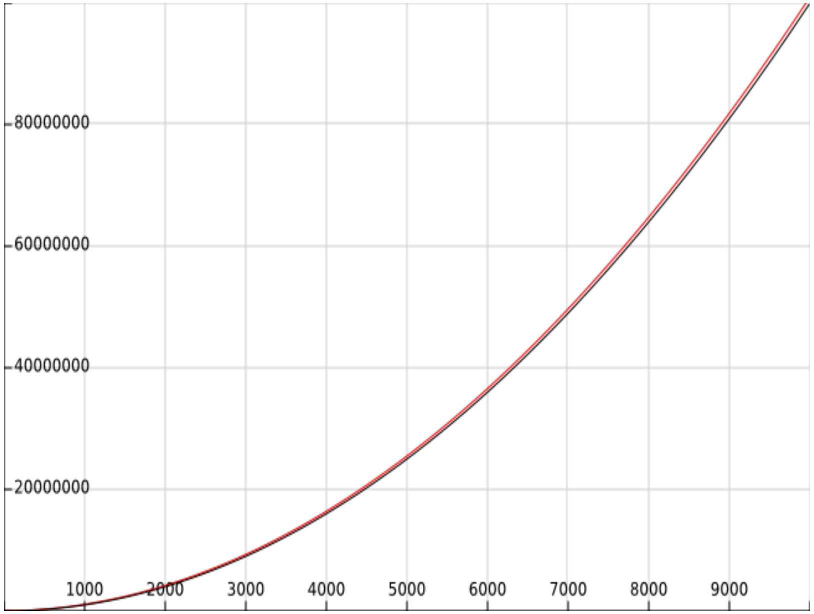
$T(n) = n^2 + 100n + 500 = O(n^2)$ (RED)

$T(n) = n^2 = O(n^2)$ (BLACK)

(In graphs, n on x axis and T(n) on y)







Properties of Big O

Any lower order terms in the function can be ignored:

$O(n^3 + n^2 + n + 5000) = O(n^3)$

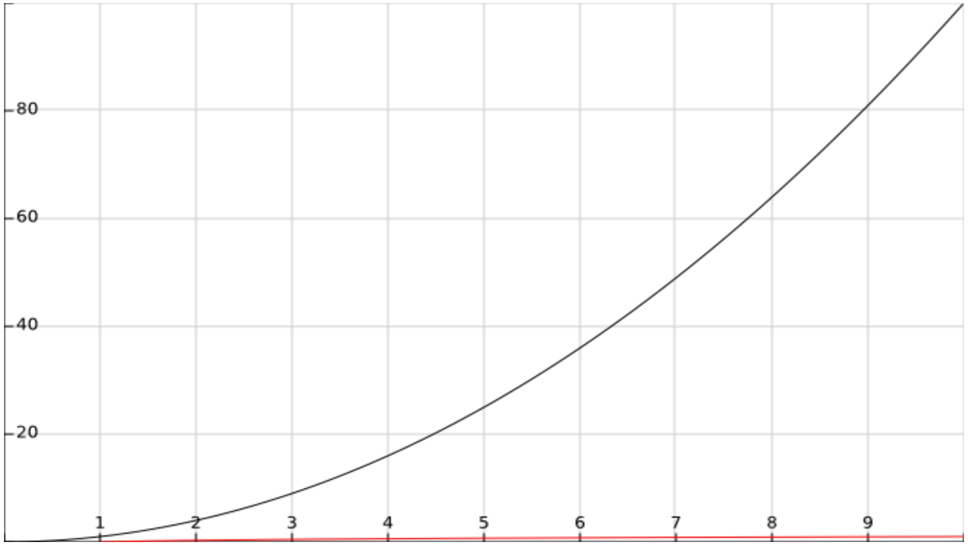
$O(n + n^2 + 5000) = O(n^2)$

$O(1500000 + n) = O(n)$

Properties of Big O

$T(n) = \log_{10}(n) = O(\log n)$ (RED)

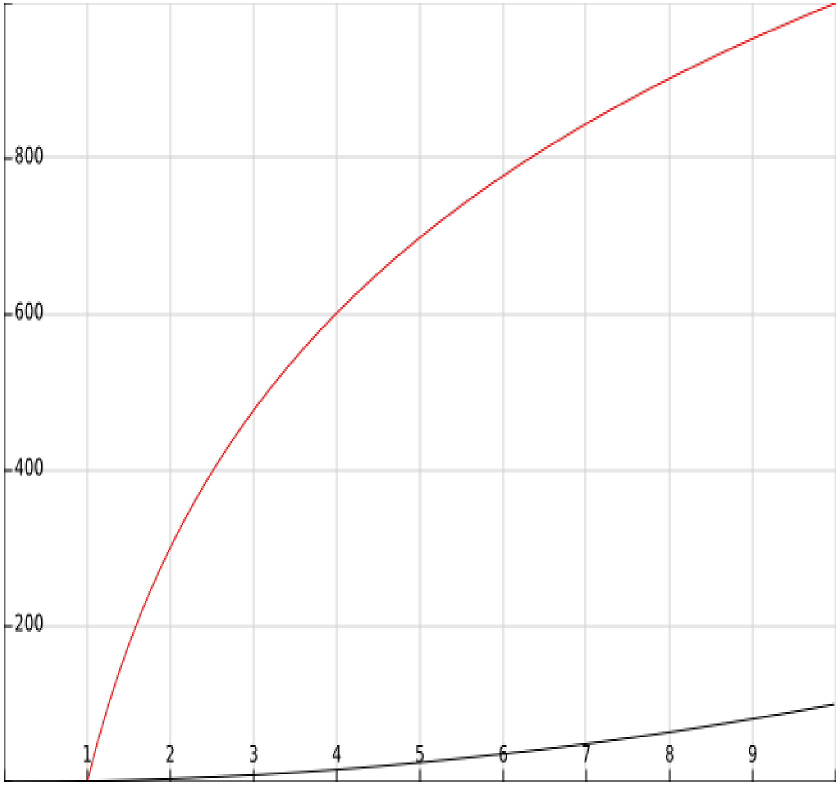
$T(n) = n^2 = O(n^2)$ (BLACK)

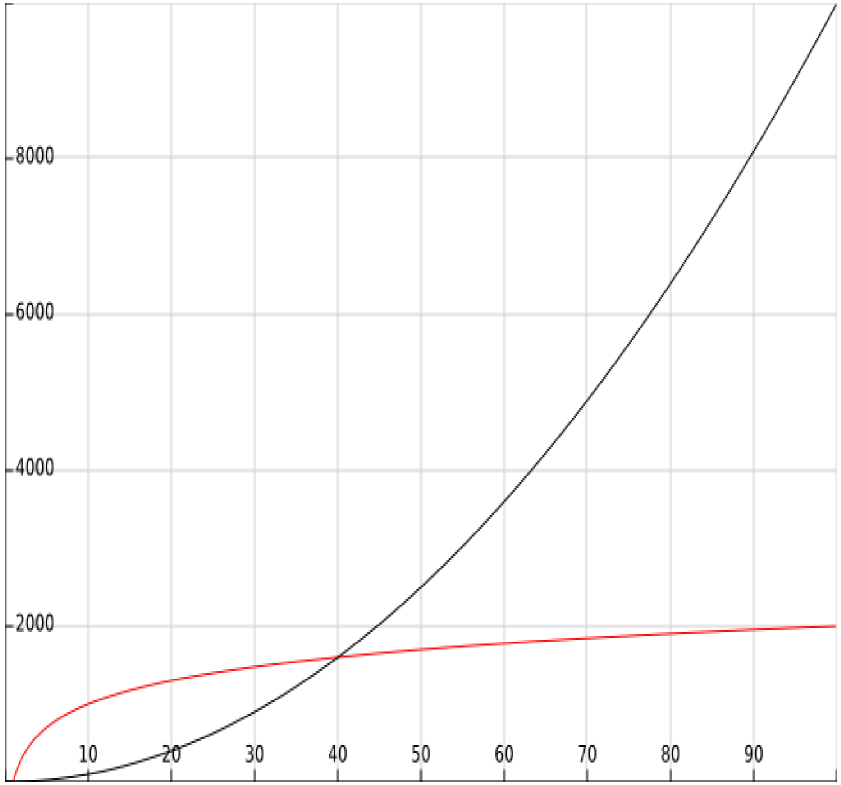


Properties of Big O

$T(n) = 1000\log_{10}(n) = O(\log n)$ (RED)

$T(n) = n^2 = O(n^2)$ (BLACK)

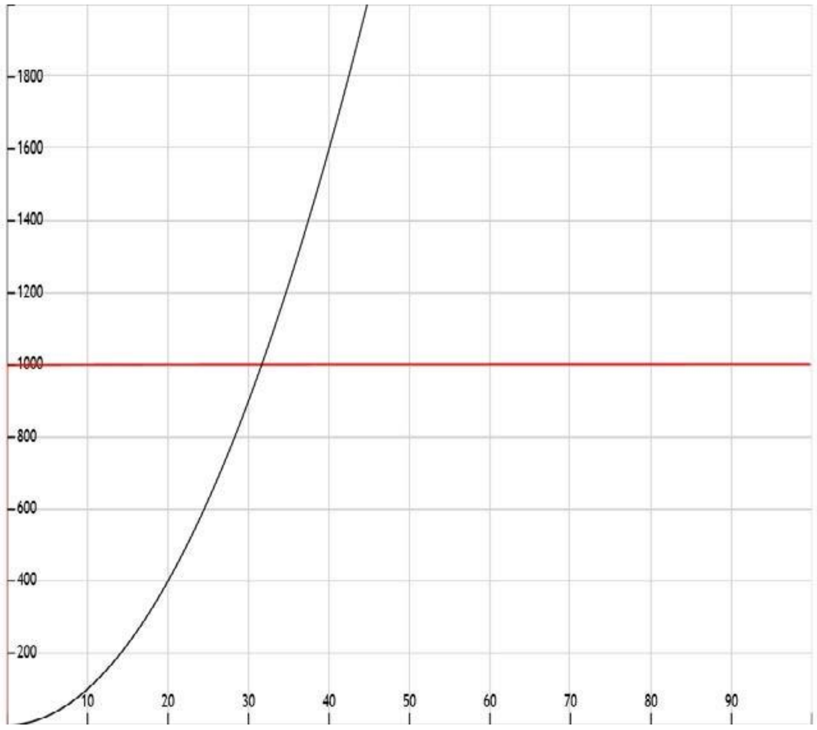




Properties of Big O

$T(n) = \log_{10}(n) + 10000 = O(\log n)$ (RED)

$T(n) = n^2 = O(n^2)$ (BLACK)



Properties of Big O

Any lower order terms in the function can be ignored:

$O(n^3 + n^2 + n + 5000) = O(n^3)$

$O(n + n^2 + 5000) = O(n^2)$

$O(1500000 + n) = O(n)$

Any constant multiplications in the function can be ignored:

$O(254n^2 + n) = O(n^2)$

$O(546n) = O(n)$

Big O's can be combined:

$O(n^2) + O(n) = O(n^2 + n) = O(n^2)$

$O(n^2) + O(n^4) = O(n^2 + n^4) = O(n^4)$

General Rules

- Multiplication by a constant (non-zero k)

  $$O(k*g) = O(g)$$

  If an algorithm (it's runtime) is $O(g)$, then running algorithm k times is also $O(g)$

- Product

  $$O(g_1) * O(g_2) = O(g_1*g_2)$$

  If algorithm a_1 is $O(g_1)$ and a_2 is $O(g_2)$, then running a_2 from inside a_1 is $O(g_1*g_2)$

- Sum

  $$O(g_1) + O(g_2) = O(g_1+g_2)$$

  If a_1 is $O(g_1)$ and a_2 is $O(g_2)$, then running a_1 and a_2 one after the other is $O(g_1+g_2)$

Nota Bene:

- O meaning earlier not entirely accurate
  - O does not mean *as fast as*; it means *not faster than*
    - O is an upper bound
  - Θ in fact captures *as fast as*
    - Θ is an *tight* (exact) bound
  - Ω (Big Omega): *grows at least as fast as*
    - Ω is a lower bound