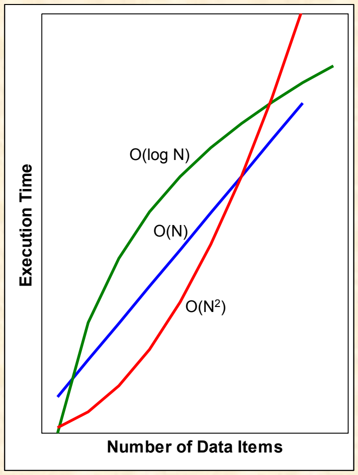
Big-O Performance Analysis

Execution Time Factors

- Computer:
  - CPU speed, amount of memory, etc.
- Compiler:
  - Efficiency of code generation.
- Data:
  - Number of items to be processed.
  - Initial ordering (e.g., random, sorted, reversed)
- Algorithm:
  - E.g., linear vs. binary search.

2

Are Algorithms Important?



- The fastest algorithm for 100 items may <u>not</u> be the fastest for 10,000 items!
- Algorithm choice is more important than any other factor!

3

Counting the instructions

```
public void SelectionSort ( int [ ] num ){
    int i, j, first, temp;
    for ( i = num.length - 1; i > 0; i - - )
    {
        first = 0;   //initialize to subscript of first element
        for(j = 1; j <= i; j ++)   //locate smallest element between positions 1 and i.
        {
i times      if( num[ j ] < num[ first ] )                                              n times
               first = j;
        }
        temp = num[ first ];   //swap smallest found with element in position i.
1 time  num[ first ] = num[ i ];
        num[ i ] = temp;
    }
}
```

$4*(n-1) + 2*(n-1) + 4 * (n-2) + 2 * (n-2)+ \quad 4 + 2*1 =$
$4(n-1) + 2((n-1)+(n-2)+(n-3)\ldots 1) = 4(n-1) * 2\ n(n-1)/2$
$=4(n-1) + n^2 - n= n^2 + 3n - 4$

4

What is Big-O?

- Big-O characterizes algorithm performance.
- Big-O describes how execution time grows as the number of data items increase.
- Big-O is a function with parameter N, where N represents the number of items.

5

Predicting Execution Time

- If a program takes 10ms to process one item, how long will it take for 1000 items?
- (time for 1 item) x (Big-O( ) time complexity of *N* items)

$\log_{10} N$	3 x 10ms	.03 sec
N	10^3 x 10ms	10 sec
$N \log_{10} N$	10^3 x 3 x 10ms	30 sec
N^2	10^6 x 10ms	16 min
N^3	10^9 x 10ms	12 days

6

Complexity

•In general, we are not so much interested in the time and space complexity for small inputs.

•For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with n = 10, it is gigantic for $n = 2^{30}$.

7

Complexity

•For example, let us assume two algorithms A and B that solve the same class of problems.

•The time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.

•For n = 10, A requires 50,000 steps, but B only 3, so B seems to be superior to A.

•For n = 1000, however, A requires 5,000,000 steps, while B requires $2.5 \cdot 10^{41}$ steps.

8

Complexity

•This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.

•So what is important is the **growth** of the complexity functions.

•The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

9

Complexity

- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

10

The Growth of Functions

•The growth of functions is usually described using the **big-O notation**.

•Definition: Let f and g be functions from the integers or the real numbers to the real numbers.
•We say that f(x) is O(g(x)) if there are constants C and k such that

•$|f(x)| \leq C|g(x)|$

•whenever $x > k$.

11

The Growth of Functions

•When we analyze the growth of **complexity functions**, f(x) and g(x) are always positive.

•Therefore, we can simplify the big-O requirement to

•$f(x) \leq C \cdot g(x)$ whenever $x > k$.

•If we want to show that f(x) is O(g(x)), we only need to find **one** pair (C, k) (which is never unique).

12

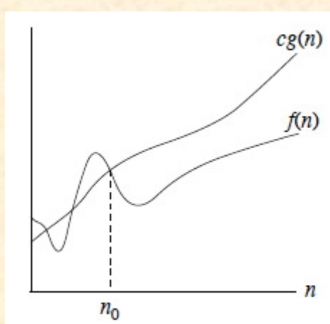
The Growth of Functions

- The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(x) for large x.
- This boundary is specified by a function g(x) that is usually much **simpler** than f(x).
- We accept the constant C in the requirement
- $f(x) \leq C \cdot g(x)$ whenever $x > k$,
- because **C does not grow with x.**
- We are only interested in large x, so it is OK if $f(x) > C \cdot g(x)$ for $x \leq k$.

13

What is Big-O

$f(n) = O(g(n))$ iff $\exists$ positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ $\forall$ $n \geq n_0$.



14

Big-O Example

$$f(x) = 6x^4 - 2x^3 + 5$$

Prove f(x)=O(n^4)

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\le 6x^4 + |2x^3| + 5 \\ &\le 6x^4 + 2x^4 + 5x^4 \\ &= 13x^4 \end{aligned}$$

15

The Growth of Functions

•Example:

•Show that f(x) = $x^2 + 2x + 1$ is O(x^2).

•For x > 1 we have:

•$x^2 + 2x + 1 \le x^2 + 2x^2 + x^2$

•$\Rightarrow x^2 + 2x + 1 \le 4x^2$

•Therefore, for C = 4 and k = 1:

•f(x) $\le Cx^2$ whenever x > k.

•$\Rightarrow$ f(x) is O(x^2).

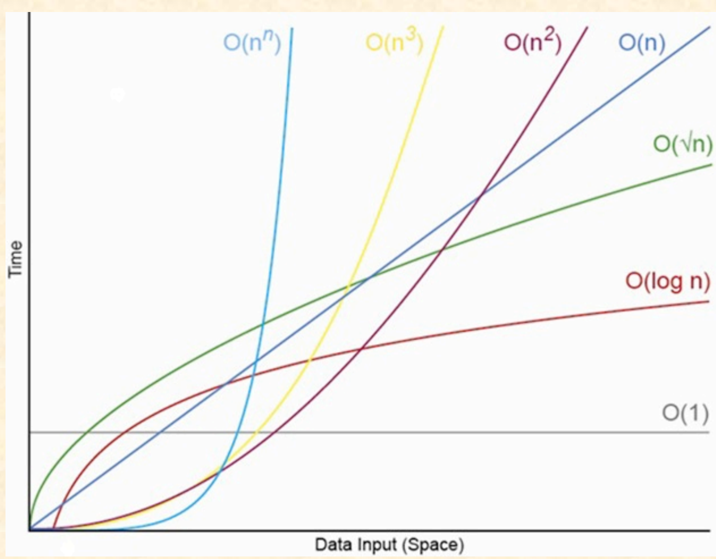
16

Common Growth Rates

Big-O Characterization		Example
O(1)	*constant*	Adding to the front of a linked list
O(log *N*)	*log*	Binary search
O(*N*)	*linear*	Linear search
O(*N* log *N*)	*n-log-n*	Binary merge sort
$O(N^2)$	*quadratic*	Bubble Sort
$O(N^3)$	*cubic*	Simultaneous linear equations
$O(2^N)$	*exponential*	The Towers of Hanoi problem

17

Common Growth Rates



18

The Growth of Functions

•Question: If f(x) is $O(x^2)$, is it also $O(x^3)$?

•**Yes.** x^3 grows faster than x^2, so x^3 grows also faster than f(x).

•Therefore, we always have to find the **smallest** simple function g(x) for which f(x) is O(g(x)).

19

The Growth of Functions

•"Popular" functions g(n) are
•n log n, 1, 2^n, n^2, n!, n, n^3, log n

•Listed from slowest to fastest growth:

- 1
- log n
- n
- n log n
- n^2
- n^3
- 2^n
- n!

20

The Growth of Functions

- A problem that can be solved with polynomial worst-case complexity is called **tractable**.
- Problems of higher complexity are called **intractable.**
- Problems that no algorithm can solve are called **unsolvable**.

21

Determining Big-O: Repetition

executed n times

```
for (i = 1; i <= n; i++)
{
    m = m + 2 ;    <------  constant time
}
```

Total time = (a constant c) * n = cn = **O(*N*)**

Ignore multiplicative constants (e.g., "c").

22

Determining Big-O: Repetition

outer loop executed *n* times

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

inner loop executed *n* times

constant time

Total time = c * n * n * = cn^2 = **O(N^2)**

23

Determining Big-O: Repetition

outer loop executed *n* times

```
for (i = 1; i <= n; i++)
{
   for (j = 1; j <= 100; j++)
   {
      k = k+1 ;
   }
}
```

inner loop executed *100* times

constant time

Total time = c * 100 * n * = 100cn = **O(N)**

24

Determining Big-O: Sequence

constant time (c_0) →

```
x = x +1;
for (i=1; i<=n; i++)
{
    m = m + 2;
}
for (i=1; i<=n; i++)
{
    for (j=1; j<=n; j++)
    {
        k = k+1;
    }
}
```

constant time (c_1) → m = m + 2;

executed *n* times

outer loop executed *n* times

inner loop executed *n* times

constant time (c_2)

Total time = $c_0 + c_1 n + c_2 n^2$ = **O(*N*²)**

Only dominant term is used

25

Determining Big-O: Selection

test + worst-case(then, else)

test: constant (c_0) →

```
if (depth( ) != otherStack.depth( ) )
{
  return false;
}
else
{
  for (int n = 0; n < depth( ); n++)
  {
    if (!list[n].equals(otherStack.list[n]))
      return false;
  }
}
```

then part: constant (c_1)

else part: $(c_2 + c_3) * n$

another if : test (c_2) + then (c_3) →

Total time = c_0 + Worst-Case(c_1, $(c_2 + c_3) * n$) = **O(*N*)**
Total time = c_0 + Worst-Case(then, else)
Total time = c_0 + Worst-Case(c_1, else)

26