TIME COMPLEXITY – PART 2

What Is a Good Solution?

- A program incurs a real and tangible cost.
  - Computing time
  - Memory required
  - Difficulties encountered by users
  - Consequences of incorrect actions by program
- A solution is good if …
  - The total cost incurs …
  - Over all phases of its life … is minimal

What Is a Good Solution?

- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency
- Be concerned with efficiency when
  - Developing underlying algorithm
  - Choice of objects and design of interaction between those objects

Measuring Efficiency of Algorithms

- Important because
  - Choice of algorithm has significant impact
- Examples
  - Responsive word processors
  - Grocery checkout systems
  - Automatic teller machines
  - Video machines
  - Life support systems

Measuring Efficiency of Algorithms

- ***Time complexity*** describes the amount of time an algorithm takes in terms of the amount of input.
- ***Space complexity*** describes the amount of memory (space) an algorithm takes in terms of the amount of input.
- For both measures, we are interested in the algorithm's ***asymptotic*** complexity.
- This asks: when *n* (number of input items) goes to infinity, what happens to the algorithm's performance?
- Analysis of algorithms
  - The area of computer science that provides tools for contrasting efficiency of different algorithms.
  - Comparison of algorithms should focus on significant differences in efficiency.
  - We consider comparisons of *algorithms*, not programs

Asymptotic Notations

- There are three asymptotic notations:

Big Omega Ω - It represents the lower bound. Therefore, it does not help much.

Example: Ω (N) means it takes **at least** N steps.

Big Theta θ – It represent the lower bound and the upper bound of an algorithm. It is hard to compute.

Example: θ(N) means it takes **at least** N and **at most** N steps.

Big Oh O – It represents the upper bound on a function for larger input to that function. Big-oh is the most useful because represents the **worst-case behavior**. So, it guarantees that the program will terminate within a certain time period.

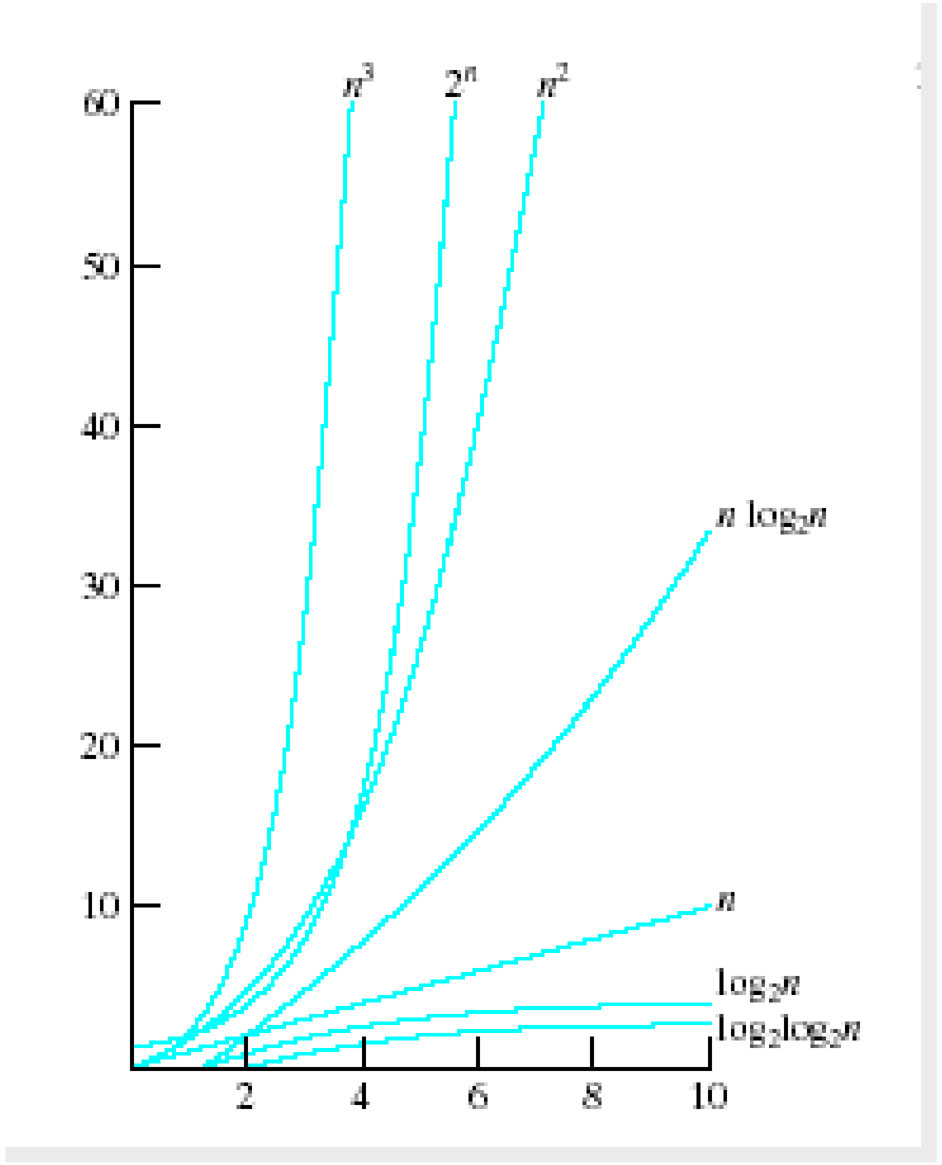
Example: O(N) means it takes **at most** N steps

Big Oh Notation

- f(n) is usually simple:

  n, n^2, n^3, …
  2n
  1, $\log_2 n$
  n $\log_2 n$
  $\log_2 \log_2 n$

- Note graph of common computing times



Case Analysis and Big O Notation

- **Worst-case analysis**
  - An input that results in the longest execution times
- **Best-case analysis**
  - An input that results in the shortest execution time.
- **Average-case analysis**
  - The average amount of time among all possible inputs of the same size.
- Average-case analysis is ideal, but difficult to perform, because for man problems it is hard to determine the relative probabilities and distribution of various input instances.
- Worst-case analysis is easy to perform so the analysis is generally conducted for the worst case.

Examples: Determining Big-O

- Repetition
- Sequence
- Selection
- Logarithm

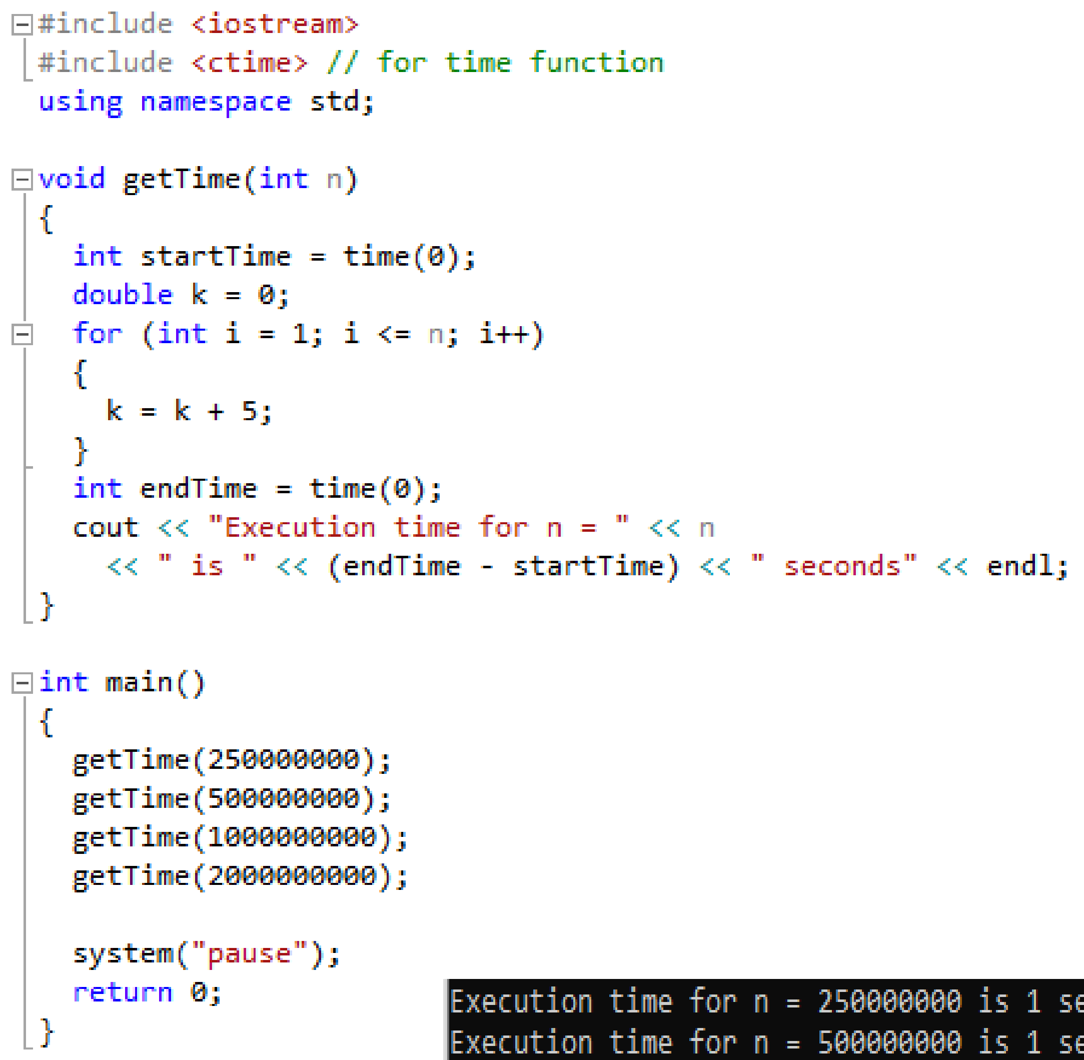
Repetition: Simple Loops

executed
n times

```
for (i = 1; i <= n; i++)
 {
   k = k + 5;
}
```

Time Complexity

n is the upper limit and the loop will run for *n* times in worst case.
So, it has a time complexity of T(n) = **O(n)**

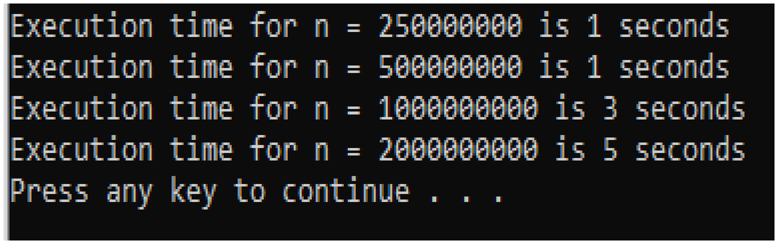


```cpp
#include <iostream>
#include <ctime> // for time function
using namespace std;

void getTime(int n)
{
  int startTime = time(0);
  double k = 0;
  for (int i = 1; i <= n; i++)
  {
    k = k + 5;
  }
  int endTime = time(0);
  cout << "Execution time for n = " << n
    << " is " << (endTime - startTime) << " seconds" << endl;
}

int main()
{
  getTime(250000000);
  getTime(500000000);
  getTime(1000000000);
  getTime(2000000000);

  system("pause");
  return 0;
}
```



```
Execution time for n = 250000000 is 1 seconds
Execution time for n = 500000000 is 1 seconds
Execution time for n = 1000000000 is 3 seconds
Execution time for n = 2000000000 is 5 seconds
Press any key to continue . . .
```

- To see how T(n) = **O(n)** performs, we run the function `getTime()` to obtain the execution time for:

n = 25000000

n = 50000000

n = 100000000

n = 200000000

Repetition: Nested Loops

executed
***n* times**

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

inner loop
executed
n times

Time Complexity

The loop gets the multiplication of outer and inner loop.
So, it has the time complexity of $T(n) = O(n^2)$

Repetition: Nested Loops

executed
n times

```
for (i = 1; i <= n; i++)
{
  for (j = 1; j <= 20; j++)
  {
    k = k + i + j;
  }
}
```

constant time

inner loop
executed
20 times

Time Complexity

T(n) = 20 * c * n = O(n)

*Ignore multiplicative constants (e.g., 20*c)*

Selection

```
if (list.contains(e))
{
  System.out.println(e);
}
else
  for (Object t: list)
  {
    System.out.println(t);
  }
```

} Let n be list.size(). Executed n times.

Time Complexity

T(n) = test time + worst-case (if, else)
= O(n) + O(n)
= O(n)

Sequence

executed *10* times

```
for (j = 1; j <= 10; j++)
{
   k = k + 4;
}
```

executed *n* times

```
for (i = 1; i <= n; i++)
{
   for (j = 1; j <= 20; j++)
   {
     k = k + i + j;
   }
}
```

inner loop executed *20* times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

Logarithm: Analyzing Binary Search

- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value if found or the interval is empty.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

- Sorted array of 15 elements.
- We want to search for 40

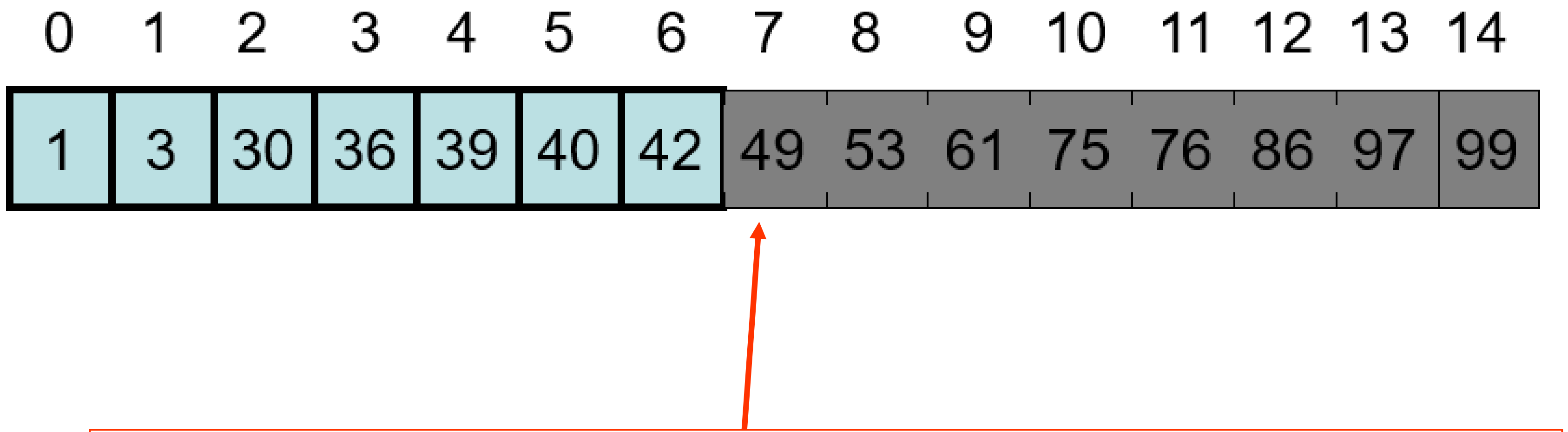
Binary Search- First Iteration

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

Look at the middle first

40 is less than 49. Since the array is sorted, we know 40 can't be in the right half.

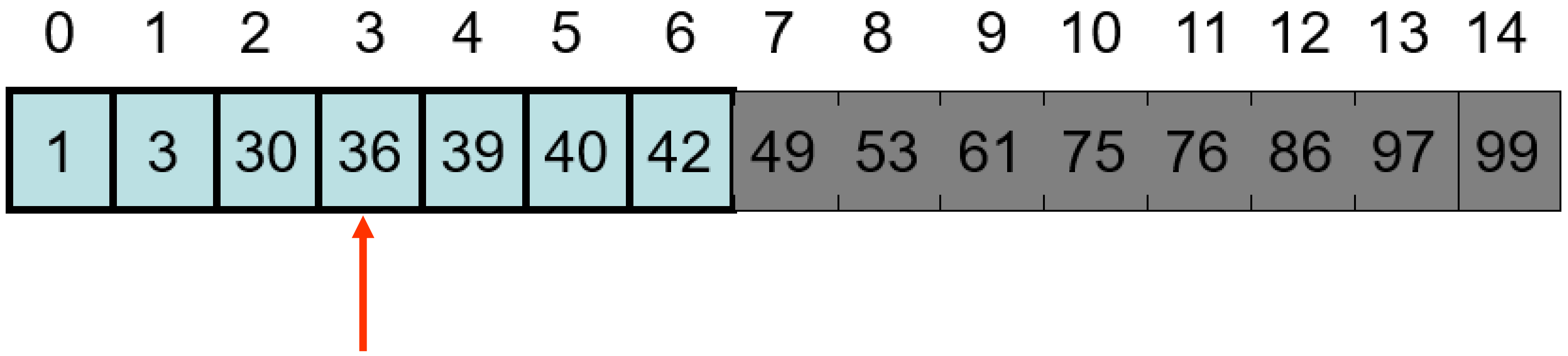
Search for 40



This would be the first iteration. The problem size is reduced by one half on the first iteration!

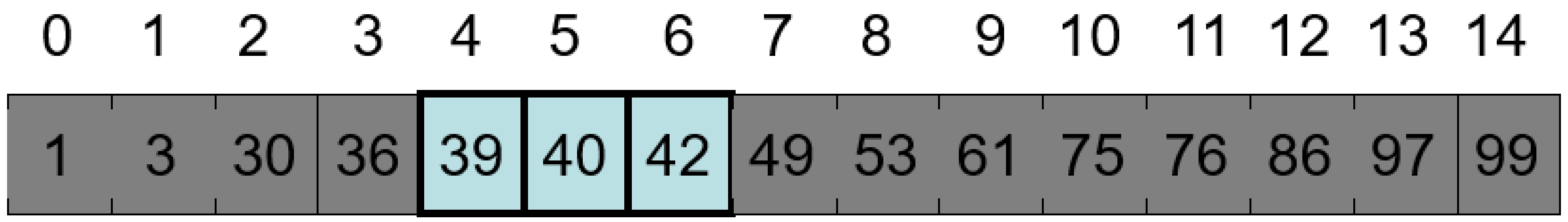
Search for 40

Binary Search- Second Iteration



- Look at the middle of the half that remains; search for 40
- 40 > 36 => we know that 40 cannot be on the left side.
- We've reduced the remaining problem size by one half on the second iteration!

Binary Search- Third Iteration



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

- Look at the middle of what remains; search for 40
- 40 == 40. We found it in just 3 iterations! (It would have taken 4 at the most.)

Binary Search (cont.)

```
int first = 0;
int last = size – 1;
int middle;
bool found = false;

while ( first <= last && !found ) {
        middle = ( first + last ) >> 1;
        if ( itemToFind == A[ middle ] )
                found = true;
        else if ( itemToFind < A[ middle ] )
                last = middle – 1;
        else // itemToFind > A[ middle ]
                first = middle + 1;
        }
```

Binary Search – Calculate Time Complexity

- The iteration in Binary Search terminates after k iterations. In the above example, it terminates after 4 iterations, so k = 4
- At each iteration, the array is divided by haft. So the length of array at any iteration is n
- At iteration 1, length of array = n
- At iteration 2, length of array = n/2
- At iteration 3, length of array = (n/2) / 2 = n / 2^2
- At iteration 4, length of array = (n/4)/2 = n/ 2^3
- Therefore after iteration k, iteration of array = n / 2^k
- After k divisions, the length of array becomes 1
- Therefore, length of array = n / 2^k = 1 => n = 2^k ➔ k = log n ➔ **O(log n)**

Logarithmic Time

Ignoring constants and smaller terms, the complexity of the binary search algorithm is *O(logn)*. An algorithm with the *O(logn)* time complexity is called a *logarithmic algorithm*. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

Ignoring Multiplicative Constants

- The linear search algorithm requires *n* comparisons in the worst-case and *n/2* comparisons in the average-case.
- Using the Big *O* notation, both cases require *O(n)* time. The multiplicative constant (1/2) can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.
- The growth rate for n/2 or 100n is the same as n, i.e., O(n) = O(n/2) = O(100n).

Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of *n* elements. If *n* is 2, it takes one comparison to find the maximum number. If *n* is 3, it takes two comparisons to find the maximum number. In general, it takes *n-1* times of comparisons to find maximum number in a list of *n* elements.

- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As *n* grows larger, the *n* part in the expression *n-1* dominates the complexity.

- The Big *O* notation allows you to ignore the non-dominating part (e.g., -1 in the expression *n-1*) and highlight the important part (e.g., *n* in the expression *n-1*). So, the complexity of this algorithm is *O(n)*.

Analyzing Selection Sort

- Selection Sort is to find the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number.
- The number of comparisons is *n-1* for the first iteration, *n-2* for the second iteration, and so on. Let *T(n)* denote the complexity for selection sort and *c* denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c \ldots + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

Ignoring constants and smaller terms, the complexity of the selection sort algorithm is $O(n^2)$.

Quadratic Time ($O(n^2)$)

- An algorithm with the **$O(n^2)$** time complexity is called a ***quadratic algorithm***.
- The quadratic algorithm grows quickly as the problem size increases.
- If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

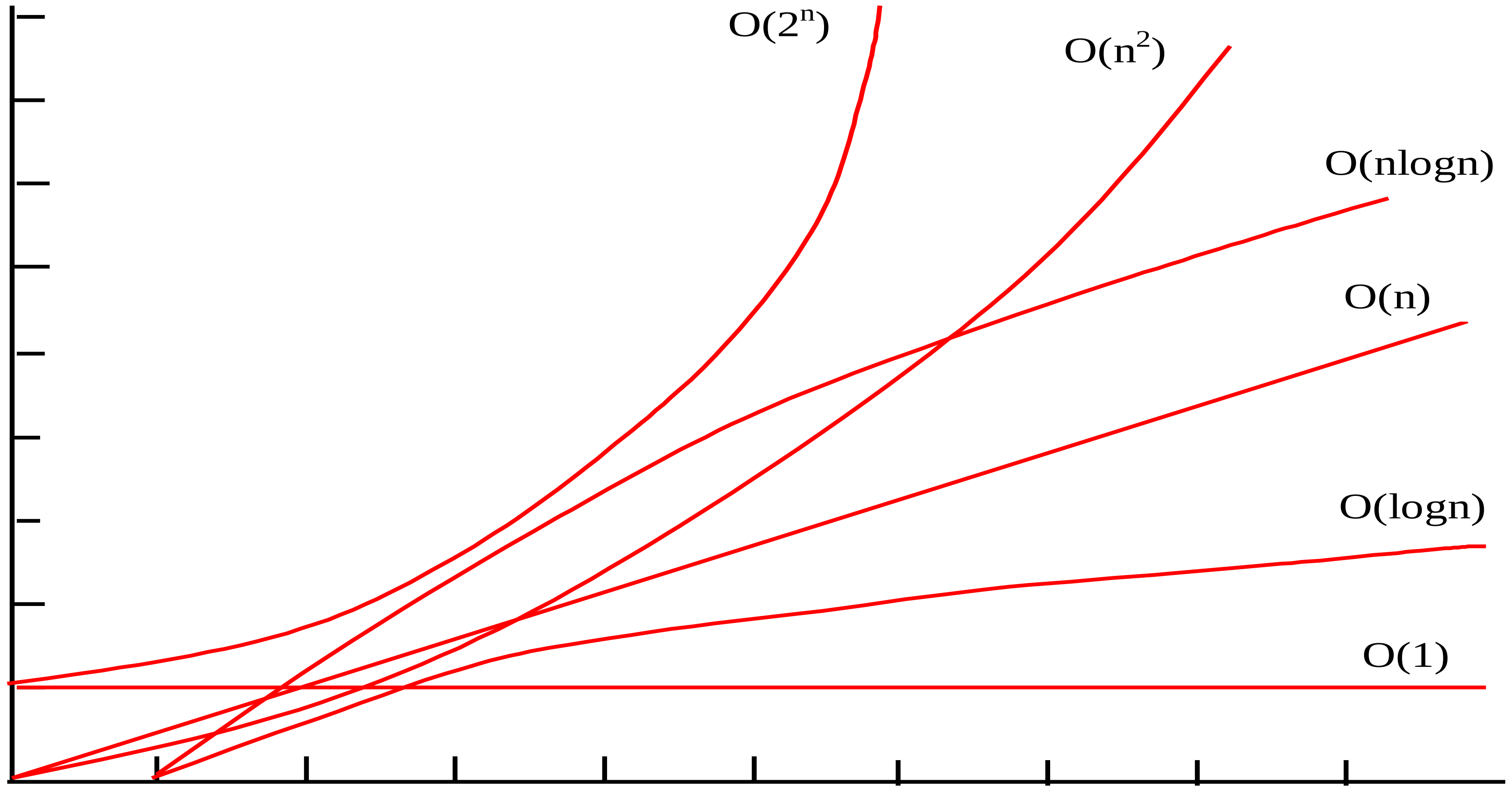
Comparing Common Growth Functions

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

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n\log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

Comparing Common Growth Functions

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



Common Recurrence Relations

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

Example: Phone Book Search

- Goal: Given a name, find the matching phone number in the phone book
  - Algorithm 1: Linear search through the phone book until the name is found
  - Best case: O(1) (it’s the first name in the book)
  - Worst case: O(*n*) (it’s the final name)
  - Average case: The name is near the middle, requiring *n*/2 steps, which is O(*n*)

Example: Phone Book Search (cont.)

Algorithm 2: Since the phone book is sorted, we can use a more efficient search

1) Check the name in the middle of the book
2) If the target name is less than the middle name, search the first half of the book
3) If the target name is greater, search the last half
4) Continue until the name is found

Example: Phone Book Search (cont.)

Algorithm 2 Characteristics:

- Each step reduces the search space by half
- Best case: O(1) (we find the name immediately)
- Worst case: $O(\log_2 n)$ (we find the name after cutting the space in half several times)
- Average case: $O(\log_2 n)$ (it takes a few steps to find the name)

Example: Phone Book Search (cont.)

Which algorithm is better?

- For very small *n*, algorithm 1 may be faster
- For target names in the very beginning of the phone book, algorithm 1 can be faster
- Algorithm 2 will be faster in every other case
- Success of algorithm 2 relies the fact that the phone book is sorted

Data structures matter!