Algorithm Analysis

Outline

- Algorithm Analysis
- Time Complexity
- Asymptotic Notations: Big-O notation
- Determine Time Complexity using big-O Notations
- Comparing Time Complexity: Linear Search vs Binary Search

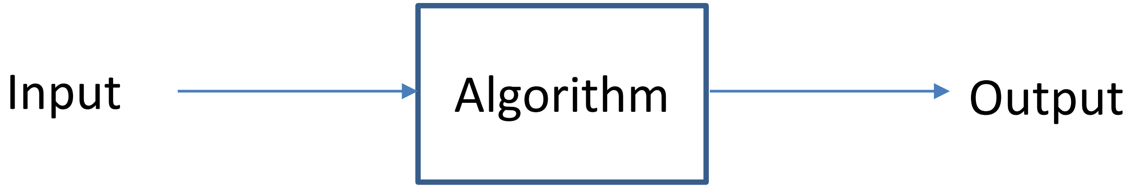
Learning Outcomes

By the end of the chapter, you should be able to

- Determine the worst-case running time of an algorithm using big-O notation.
- Compare the worst-case running time of different algorithms using big-O notation.

Algorithm

- Any well-defined **computational procedure** that transforms some **inputs** into some **outputs**.



Examples of Algorithms

- Searching
  - Input: A sequence of n numbers $\{a_1, a_2, \dots, a_n\}$ and a number k
  - Output: true if k is found in the sequence and false otherwise
  - For example,
    - Input: $\{31, 41, 59, 26, 41, 58\}$ and 31
    - Output: True

- Sorting
  - Input: A sequence of n numbers $\{a_1, a_2, \dots, a_n\}$
  - Output: A permutation $\{a'_1, a'_2, \dots, a'_n\}$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$
  - For example,
    - Input: $\{31, 41, 59, 26, 41, 58\}$
    - Output: $\{26, 31, 41, 41, 58, 59\}$

Algorithm Analysis

- Correctness analysis
  - Produce correct output for every input
- Complexity analysis
  - It describe the efficiency of an algorithm uses the computational resources (e.g., CPU time, memory and disk usage) for execution.
    - Space Complexity:
      - The amount of memory used
    - Time Complexity:
      - The amount of running time used

Time Complexity

- The actual running time of an algorithm depends on a lot of factors such as processor speed, operating system and programming language etc.

- The running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.

- The running time of an algorithm is proportional to the number of “basic operations” that it executes.

- Time complexity of an algorithm can be calculated by finding number of basic operations that it executes.

Example: Time Complexity

- Calculate the running time in terms of number of basic operations for the following algorithm.

Algorithm 1: Compute the average value in an $\underline{n\text{-element array } a}$.

$sum \leftarrow 0$
for i **from** 0 to $n - 1$
  $sum \leftarrow sum + a[i]$
$average \leftarrow \dfrac{sum}{n}$

No.	Operations
1	Assignment
	Loop n times
$2n$	Assignment and addition, each of n times
2	Assignment and division

$$T(n) = 1 + 2n + 2 = 2n + 3$$

Example: Time Complexity

- Calculate the running time in terms of number of basic operations for the following algorithm.

Algorithm 2: Compute the average value in an $\underline{n \times n}$ matrix $\underline{a}$.

$sum \leftarrow 0$
for i **from** 0 to $n-1$
    for j **from** 0 to $n-1$
        $sum \leftarrow sum + a[i][j]$
$average \leftarrow \dfrac{sum}{n}$

No.	Operations
1	Assignment
	Outer loop n times
	Inner loop n times
$2n^2$	Assignment and addition, each of n^2 times
2	Assignment and division

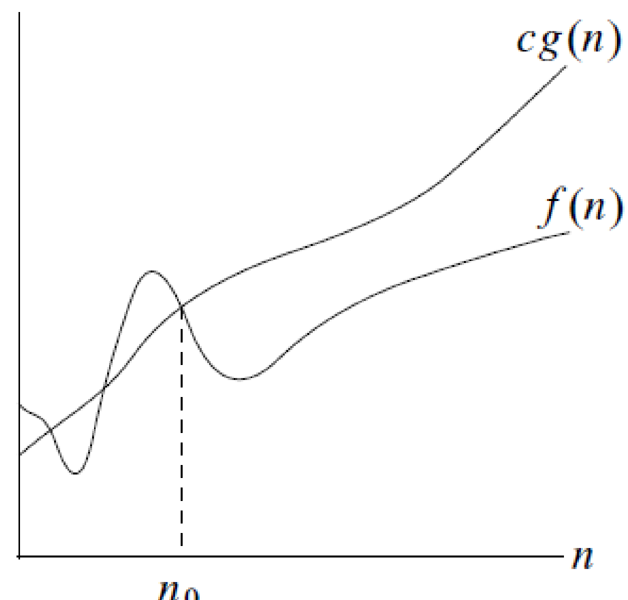
$$T(n) = 1 + 2n^2 + 2 = 2n^2 + 3$$

Asymptotic Analysis

- The running time of an algorithm depends on the size of its input.
- We are concerned with how the running time grows when the input size becomes sufficiently large.
- This can be described using *asymptotic* notations.
- Asymptotic notations are mathematical notations that describes how the value of a function $f(n)$ changes as its input argument n increases .

Big-O Notation

- O-notation gives an asymptotic upper bound.
- We denote by $f(n) = O\big(g(n)\big)$ when $\exists\, n_0 > 0, c > 0,$ such that $\forall n \geq n_0$, $0 \leq f(n) \leq cg(n)$
- We write $f(n) = O\big(g(n)\big)$ if there are positive constants n_0 and c such that when n is greater than or equal to n_0 , the value of $f(n)$ always lies on or below (smaller than or equal to) $cg(n)$.



$f(n) = O(g(n))$

Big-O Notation

- The O in the big-O notation denotes order of growth or growth rate.
- E.g., $f(n) = O(n^2)$ means
  - $f(n)$ grows in the order of n^2.
  - $f(n)$ has an order of n^2.
  - The growth rate of $f(n)$ is n^2.

Worst-Case Running Time

- The worst-case running time (i.e., longest running time) of an algorithm gives us an upper bound on the running time for any input.
- Knowing it provides a guarantee that the algorithm will never take any longer.
- Therefore, big-O notation is most suitable.
- How do we find the running time of an algorithm in big-O notation?

Dominant Term

- In the big-O notation, we only care about the dominant term.
- In other words, we only care about the term that will account for the **biggest portion** of the running time.

Dominant Term

- Assume the running time of an algorithm is
$$f(n) = n^2 + 2n + 100.$$
- We analyze both varying terms: n^2 and $2n$ separately.

n	$f(n)$	n^2	n^2 **as % of total**	$2n$	$2n$ **as % of total**
10	220	100	45.455%	20	9.091%
100	10,300	10,000	**97.087%**	200	1.942%
1,000	1,002,100	1,000,000	**99.790%**	2,000	0.2%
10,000	100,020 ,100	100,000,000	**99.980%**	20,000	0.02%
100,000	10,000,200,100	10,000,000,000	**99.99%**	200,000	0.002%

- The term n^2 dominates the rest of the terms as n increases.
- The growth rate of n^2 is much faster than $2n$.

Dominant Term

- Now let's add a **cubic** term:

$$f(n) = n^3 + n^2 + 2n + 100$$

n	f(n)	n^3	n^3 as % of total
10	1,220	1,000	81.967%
100	1,010,300	1,000,000	**97.980%**
1, 000	1,001,002,100	1,000,000,000	**99.890%**
10, 000	1,000,100,020,100	1,000,000,000,000	**99.989%**
100, 000	1,000,010,000,200,100	1,000,000,000,000,000	**99.99%**

- The term n^3 dominates the rest of the terms as n increases.
- The growth rate of n^3 is much faster than the rest.

Dominant Term

- Now let's add an **exponential** term:

$$f(n) = 2^n + n^3 + n^2 + 2n + 100$$

n	$f(n)$	2^n	2^n **as % of total**
10	2,244	1,024	45.632799%
20	1,057,116	1,048,576	**99.192142%**
30	1,073,769,884	1,073,741,824	**99.997387%**
40	1,099,511,693,556	1,099,511,627,776	**99.999994%**

- The term 2^n dominates the rest of the terms as n increases.
- The growth rate of 2^n is much faster than the rest.

Dominant Term

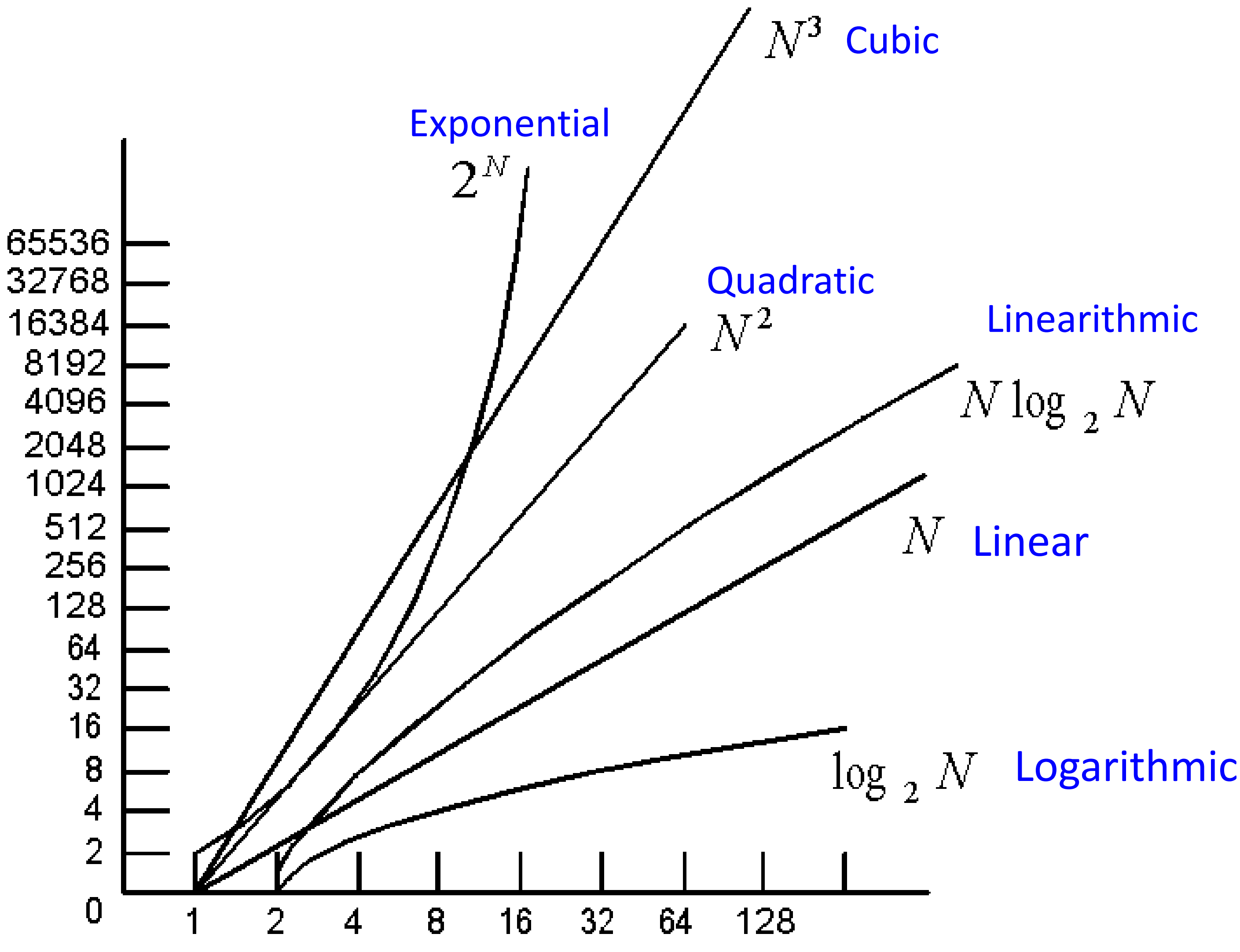
- As n gets larger, some portion of the function tends to overpower the rest.
- Lower order terms can thus be ignored because they are insignificant for large n.
  - $f(n) = n^3 + n^2 + 2n + 100 = O(n^3)$
  - $f(n) = 2^n + n^3 + n^2 + 2n + 100 = O(2^n)$
- The exact number of operations is not as important as determining the most dominant part of the function.

Dominant Term

- How do we find which is the dominant term?
- We can refer to the growth rates (or order) of some common functions.

Growth rate	Name
$O(1)$	Constant
$O(\log_2 N)$	Logarithmic
$O(N)$	Linear (directly proportional to N)
$O(N \log_2 N)$	Linearithmic (proportional to $N \log N$)
$O(N^2)$	Quadratic (proportional to square of N)
$O(N^3)$	Cubic (proportional to cube of N)
$O(N^k)$ k is a constant	Polynomial (proportional to N to the power of k)
$O(a^N)(a > 1)$ a is a constant	Exponential (proportional to a to the power of N)

Common Growth Rates



Common Big-*O*h Expressions

Expression	*Name*
$O(1)$	*Constant*
$O(\log N)$	Logarithmic
$O(N)$	Linear
$O(N \log N)$	Linearithmic
$O(N^2)$	Quadratic
$O(N^3)$	Cubic
$O(N^k)$	Polynomial
$O(2^N)$	Exponential

- Higher growth rate
- Higher time complexity
- Less efficient
- Slower running time

Finding Big-O Expressions

1. Determine running time
   - $n^2 + (n \log_2 n) + 3n$
2. Drop all but the most significant terms
   - $O(n^2 + n\log_2 n + 3n) \Rightarrow O(n^2)$
   - $O(n \log_2 n + 3n) \Rightarrow O(n \log_2 n)$
3. Drop constant coefficients
   - $O(3n) \Rightarrow O(n)$
   - $O(10) \Rightarrow O(1)$

$$\boxed{\begin{array}{l} f(n) = O(3n) \\ \Rightarrow\ f(n) \le c\ 3n = c'n \\ \Rightarrow\ f(n) = O(n) \end{array}}$$

Example:
Determine the big-O Notation

- Determine the big-O notation of the following functions.
  - $f(n) = n + 1$
  - $f(n) = 2n + 1$
  - $f(n) = n \log_2 n + 2n^3 + 10$
  - $f(n) = n + \log_2 2^n + 2$

Determine the Worst-Case Running Time of an Algorithm using Big-O Notations

- Sequence of statements
- Conditional statements
- Loops
- Statements with function calls

Sequence of Statements

```
statement 1;
statement 2;
...
statement k;
```

```
Total time =

  T(statement 1)
+ T(statement 2)
+  ...
+ T(statement k)
```

- For example, If each statement is $O(1)$, then the total time is also constant: $O(1)$.

Conditional Statements

```
if (condition) {
   sequence of statements 1
}
else {
   sequence of statements 2
}
```

```
Total time = max(T(sequence 1), T(sequence 2))
```

- For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$, the worst-case time for the whole if-else statement would be $O(N)$.

Loops

```
for (i = 0; i < N; ++i) {
      sequence of statements
}
```

```
Total time = N×T(statements)
```

- For example, if the sequence of statements is $O(1)$, then the total time is $O(N)$.

Nested Loops

```
for (i = 0; i < N; ++i) {
   for (j = 0; j < M; ++j) {
      sequence of statements
   }
}
```

```
Total time = N×M×T(statements)
```

- For example, if the sequence of statements in the nested for loop is $O(1)$, then the total time is $O(NM)$.

Function Calls

```
f(n); // Assume O(1)

for (j = 0; j < N; ++j)
  f(j);
```

Total time = $N\ O(1)$
$= O(N)$

```
g(n); // Assume O(n)

for (j = 0; j < N; ++j)
  g(N);
```

Total time = $N\ O(N)$
$= O(N^2)$

Example

```
void Op(int M){
  for (int i=0;i<M; ++i){
    //a single operation
  }
}
```

```
void Case1(void){
   Op(5);
}
```

```
void Case2(void){
   Op(N);
   Op(500);
}
```

Example

```
void Op(int M){
  for (int i=0;i<M; ++i){
    //a single operation
  }
}
```

```
void Case3(void){
   for (int i=0;i<5;++i)
      Op(1);
}
```

```
void Case4(void){
   for (int i=0;i<N;++i)
      Op(1);
   Op(N);
}
```

```
void Case5(void){
   for (int i=0; i<N; ++i)
      Op(N);
}
```

Example

```
void Op(int M){
  for (int i=0;i<M; ++i){
    //a single operation
  }
}
```

```
void Case6(void){
   for (int i=0;i<10;++i)
     for (int j=0;j<N;++j)
       Op(N);
}
```

```
void Case7(void){
   for (int i=0;i<N; ++i)
     for (int j=0;j<N; ++j)
       Op(N);
}
```

```
void Case8(void){
  for (int i=0;i<N;++i)
   for (int j=i;j<N;++j)
     Op(1); }
```

Example

```
void Op(int M){
  for (int i=0;i<M; ++i){
    //a single operation
  }
}
```

```
void Case9(void){
   if (/* condition */)
     Op(N);
   else
     Op(500); }
```

```
void Case10(void){
  for(int i=1;i<=N;i*=2)
     Op(1);
}
```

```
void Case11(void){
  for(int i=1;i<=N;i*=2)
     Op(N);
}
```

Compare the Time Complexity: Linear Search vs Binary Search

- Linear Search

for i **from** 0 **to** $N-1$
    if $a[i] = x$, **then**
        return i
return -1

- Worse Case:
  - When the value to search is the last element or is not in the array.
  - Time complexity: $O(N)$

7	2	4	6	10	1	9	8	3	5

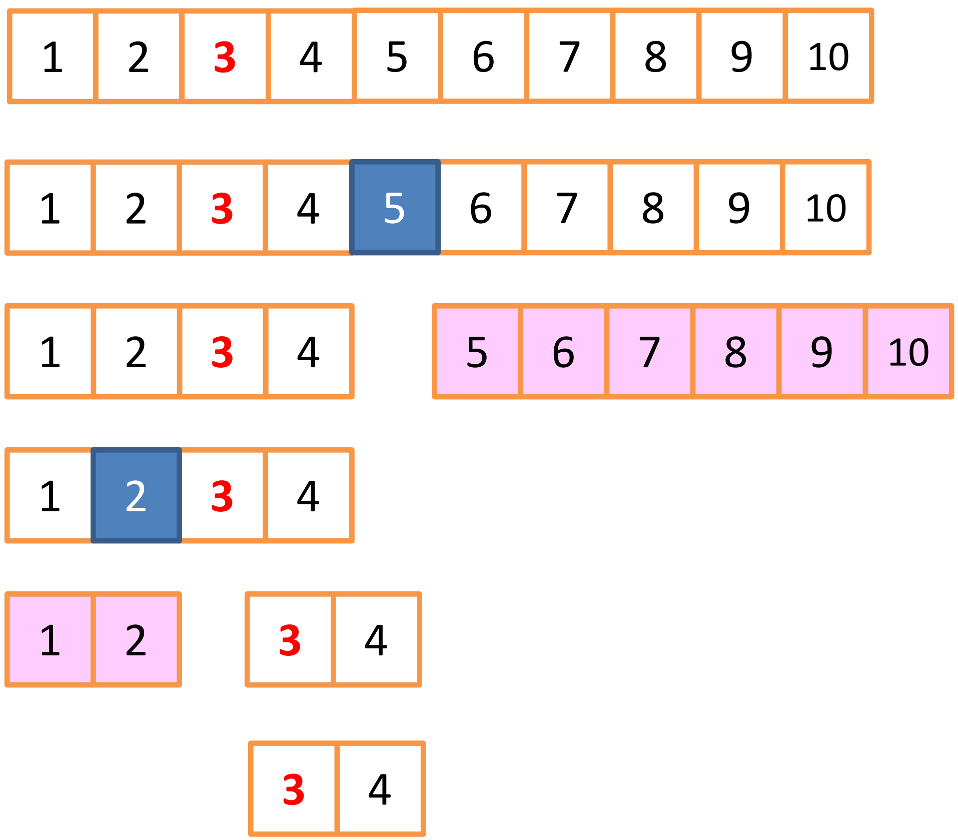
Compare the Time Complexity: Linear Search vs Binary Search

- Binary Search
  - When the elements of the array is sorted, binary search can be used.
  - Divide-and-conquer strategy
    - Probe the middle element of the array
    - If the value is smaller than the middle one, discard the right part of the array.
    - Otherwise, discard the left part of the array.
    - Repeat the above until the value is found as the middle element, or the size of the array reduces to zero.

0	mid	$n-1$
$\leq a[mid]$	$a[mid]$	$\geq a[mid]$

Compare the Time Complexity: Linear Search vs Binary Search

- Binary Search Example: Search for 3 in the sorted array



mid = (0 + 9)/2 = 4

a[4] = 5 > 3
Discard the right part of array

mid = (0 + 3)/2 = 1

a[1] = 2 < 3
Discard the left part of array

mid =(2+3)/2=2
a[2] = 3
Done!

Compare the Time Complexity: Linear Search vs Binary Search

- Binary Search

0 … $mid-1$	mid	$mid+1$ … $n-1$
$\leq a[mid]$	$a[mid]$	$\geq a[mid]$

```
lower ← 0
upper ← n − 1
while lower ≤ upper
    mid ← (lower + upper)/2    // integer division
    if x = a[mid]
        return mid
    else if x < a[mid]
        upper ← mid − 1
    else  // x > a[mid]
        lower ← mid + 1

return −1
```

Compare the Time Complexity: Linear Search vs Binary Search

- Binary Search
  - Worse case
    - When the size of the array becomes 1 or the value is not in the array.

No of Divisions	Size of Array
0	N
1	$\frac{N}{2}$
2	$\frac{N}{2^2}$
3	$\frac{N}{2^3}$
k	$\frac{N}{2^k}$

$$\frac{N}{2^k} = 1 \implies N = 2^k \implies \log_2(N) = k$$

- Time complexity is $O(\log_2 N)$.

Final Notes on Asymptotic Analysis Big-O Notations

- It helps in predicting how an algorithm will perform on larger input sizes.
- It is a useful tool for comparing the efficiency of different algorithms and selecting the best one for a specific problem.
- The limitation is that it does not provide an accurate running time of an algorithm.
  - Two algorithms with the same asymptotic complexity may have different actual running times.
  - It is only valid for sufficiently large input size.

Summary

- Algorithm Analysis
- Time Complexity
- Asymptotic Notations: Big-O notation
- Determine Time Complexity using big-O Notations
- Comparing Time Complexity: Linear Search vs Binary Search

References

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms.* 3rd ed. MIT Press, 2009.
- S. N. Mohanty and P. K. Tripathy, *Data Structure and Algorithms using C++: A Practical Implementation.* John Wiley & Sons, 2021.
- L. Wittenberg, *Data Structures and Algorithms in C++: Pocket Primer*, Mercury Learning & Information, 2017.