CSCE 222
Discrete Structures for Computing

David Kebo Hougninou

Chapter 3

Algorithms and Their Complexity
Algorithm

An **algorithm** is a finite sequence of steps that solves a problem.

**Computational complexity**

How much computing resources are needed to solve a problem?
How long (time) and how much memory (space) does it take?

We observe the behavior of algorithms as the input size **grows**.

---

Algorithm

We are interested in:

- The behavior of algorithms as the input size grows.
- Modeling a problem and an algorithm mathematically
- Analyzing the growth of functions (as argument grows without bound)
- The complexity of algorithms
Algorithms

• Algorithms can be described in English or in pseudocode.

• Pseudocode is an intermediate step between an English description of the steps and the implementation of these steps in code.

• Advantages of pseudocode:
  - Independent of the programming language
  - More general than a specific programming language

Properties of algorithms

Input: An algorithm has input values from a specified set.

Output: An algorithm produces output values from a specified set. The output values are the solution.

Definiteness: The steps of an algorithm must be properly defined.

Correctness: An algorithm should produce the correct output values for each set of input values.

Finiteness: An algorithm should produce the output after a finite number of steps for any input.

Effectiveness: It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

Generality: The algorithm should work for all problems of the desired form.
Maximum-Finding Algorithm

Find the maximum of sequence: $a_1, a_2, \ldots, a_n$:

1: max $\leftarrow a_1$
2: for $i \leftarrow 2 \ldots n$ do
3:     if max $< a_i$ then
4:         max $\leftarrow a_i$
5:     end if
6: end for
7: return max

The Searching Problem

• Input: list of elements $a_1, a_2, \ldots, a_n$ and a particular element $x$

• Output: return the index in the list where $x$ appears; if $x$ is not in the list then return 0

For now, assume all the list elements are unique.
Linear Search Algorithm

Input: $a_1, a_2, ..., a_n$ and $x$
Output: index of $x$ or 0

1: $i \leftarrow 1$
2: while $i \leq n$ and $x \neq a_i$ do
3: \hspace{1em} $i \leftarrow i + 1$
4: end while
5: if $i \leq n$ then
6: \hspace{1em} return $i$
7: \hspace{1em} else
8: \hspace{1.5em} return 0
9: end if

Linear search can be slow

- If $x$ is not in the list or is toward the end we check all (or most) elements in the list.
- We can see that the running time is proportional to the number of elements in the list.
Binary Search Example

Search for the number 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

Compare 19 with the element roughly in the middle

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

Since 19 > 10, ignore the first half of the list

... 12 13 15 16 18 19 20 22

Since 19 > 16, ignore the first half of the list

... 18 19 20 22

Found!

Binary Search Algorithm

Input: a sorted list stored in array $A[1 \ldots n]$ and $x$

Output: index of $x$ or 0

1: left $\leftarrow 1$; right $\leftarrow n$ // endpoints of the interval
2: while left $<$ right do
3:   middle $\leftarrow \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$ // approx. middle of the interval
4:   if $x > A[\text{middle}]$ then
5:     left $\leftarrow \text{middle} + 1$ // increase left end of the interval
6:   else if $x < A[\text{middle}]$ then
7:     right $\leftarrow \text{middle} - 1$ // decrease right end of the interval
8:   else if $x == A[\text{middle}]$ then
9:     return middle
10: end if
11: end while
12: return 0
Binary Search Algorithm

- Let $n$ be the size of the list
- Every iteration of the while loop reduces the size of the search interval by a factor of 2
- After about $\log_2 n$ iterations, the search interval consists of just a single list element
- The running time is proportional to the logarithm of the size of the list

Exponent vs. Logarithm

**Exponent:** how many times a number is used in a multiplication.

\[ 2 \times 2 \times 2 = 8 = 2^3 \]

**Logarithm:** how many of one number to multiply to get another number.

\[ \log_2(8) = 3 \]

\[ 2^3 = 8 \iff \log_2(8) = 3 \]
The Sorting Problem

- **Input**: list of elements $a_1, a_2, \ldots, a_n$ drawn from totally ordered set (supports $<$ operation)
- **Output**: list of elements $b_1, b_2, \ldots, b_n$ that is a rearrangement of the input list such that $b_1 < b_2 < \ldots < b_n$.
- Assume all the list elements are unique.

Bubble Sort Algorithm

Bubble sort makes **multiple passes** through a list.

Every pair of elements that are found to be out of order are interchanged.
Bubble Sort

e.g. Steps of bubble sort with: \{3, 2, 4, 1, 5\}

- After the 1\textsuperscript{st} pass the largest element is in the correct position
- After the 2\textsuperscript{nd} pass, the 2\textsuperscript{nd} largest element is in the correct position
- After each subsequent pass, an additional element is put in the correct position

Bubble Sort Algorithm

\textbf{Input:} array \texttt{A[1 ..n]} of elements

\textbf{Output:} array \texttt{A[1 ..n]} of sorted elements

1: for \texttt{i:=1} to \texttt{n-1}
2: for \texttt{j:=1} to \texttt{n-i}
3: if \texttt{a}_{\texttt{j}} > \texttt{a}_{\texttt{j+1}} then interchange \texttt{a}_{\texttt{j}} and \texttt{a}_{\texttt{j+1}}
4: end for
5: end for

\{a\textsubscript{1},..., a\textsubscript{n} is now in increasing order\}
Insertion Sort Algorithm

Assume the input list is provided in an array.

We repeatedly increase the prefix of the array that is in sorted order.

1. Take the next element $x$ from the not-yet-sorted suffix of the array
2. Find the correct location for $x$ in the sorted prefix of the array
3. Shift to make room for $x$ in the sorted prefix and insert $x$ in its place.

Sort: 7 3 1 2 4 8

#1  | 7 | 3 | 1 | 2 | 4 | 8
#2  | 7 | 3 | 1 | 2 | 4 | 8
#3  | 3 | 7 | 1 | 2 | 4 | 8
#4  | 3 | 1 | 7 | 2 | 4 | 8
#5  | 1 | 3 | 7 | 2 | 4 | 8
#6  | 1 | 3 | 2 | 7 | 4 | 8
#7  | 1 | 2 | 3 | 7 | 4 | 8
#8  | 1 | 2 | 3 | 4 | 7 | 8
#9  | 1 | 2 | 3 | 4 | 7 | 8
#10 | 1 | 2 | 3 | 4 | 7 | 8
Input: array $A[1 ..n]$ of elements
Output: array $A[1 ..n]$ of sorted elements

1: for $j ← 2...n$ do // $j$ is index where unsorted suffix starts
3:     $i ← 1$
4:     while ($m > A[i]$) // Use linear search to find $m$’s place
5:         $i ← i+1$ // $i$ is the index where $m$ should go
6:     end while
7:     for $k ← j...i+1$ do
8:         $A[k] := A[k - 1]$ // shift places to make room for $m$
9:     end for
10:    $A[i] ← m$ // Insert $m$ in its place
11: end for
12: return $A$

Insertion Sort Algorithm

- The runtime depends on the initial order of the array, and how much shifting is required
- Soon we will learn techniques for showing that the runtime is proportional to $n^2$ in the worst case.
Estimating Algorithm Runtime

How to describe the runtime?
As a function of the input size

Different inputs, even of the same length, might take different amounts of time

The runtime depends on:
1. The programming language
2. The compiler
3. The hardware
4. The level of multiprogramming

Asymptotic Analysis

\[ f(x) = \frac{1}{x} \]
Asymptotic Analysis

Informally, measure time as a function of input size:
1. ignore multiplicative constants
2. ignore lower order terms

Focus on the behavior as the input size grows to infinity.

As the input size increases, the impact of multiplicative constants and lower order terms becomes negligible!

Big-O Notation O()

Big-O is how to do the asymptotic analysis.

Definition: Let f and g be functions.

f(x) is O(g(x)) if there are constants C and k such that:

\[ |f(x)| \leq C |g(x)| \text{ whenever } x > k \]

This is read as “f(x) is big-O of g(x)”
Big-O Estimates for Polynomials

Let
\[ f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]
where \( a_0, a_1, \ldots, a_n \) are real numbers with \( a_n \neq 0 \). The leading term \( a_n x^n \) of a polynomial dominates the growth.

Therefore \( f(x) \) is \( O(x^n) \)

Big-O Example

Show that \( f(x) = x^2 + 2x + 1 \) is \( O(x^2) \)

Find a positive \( C \) and \( k \) such that:
\[ x^2 + 2x + 1 \leq Cx^2 \text{ for all } x > k \]
- \( x^2 \) increases faster than \( 2x + 1 \)
- Let’s try using four \( x^2 \) terms, i.e. \( C = 4 \)
- How big must \( x \) be to ensure that \( x^2 + 2x + 1 \leq 4x^2 \)? \( k = 1 \)
Big-O Example

- Show that $n^2$ is not $O(n)$.
- We have to show that all pairs of $C$ and $k$ fail.
- Let’s try a proof by contradiction:
  - Suppose there is $C$ and $k$ such that $n^2 \leq C \cdot n$ for all $n > k$.
- Divide both sides by $n$ to get: $n \leq C$ for all $n > k$.
- $C$ is a constant and eventually some value of $n$ exceeds $C$
- This is a contradiction!

Big-O

Let $f(x) = a_n x^n + ... + a_1 x + a_0$. Then $f(x)$ is $O(x^n)$

Proof:
$f(x) \leq |a_n| x^n + |a_{n-1}| x^{n-1} + ... + |a_1| x + |a_0|
= x^n \left( |a_n| + \frac{|a_{n-1}|}{x} + ... + \frac{|a_1|}{x^{n-1}} + \frac{|a_0|}{x^n} \right)
\leq x^n (|a_n| + |a_{n-1}| + ... + |a_1| + |a_0|)$ as long as $x > 1$

Set $k = 1$ and $C = |a_n| + |a_{n-1}| + ... + |a_1| + |a_0|$
More Big-O examples

1. Sum of first $n$ positive integers:
   \[ 1 + 2 + 3 + \ldots + n \leq n + n + n + \ldots n \leq n^2, \text{ which is } O(n^2) \]

2. Factorial function:
   \[ n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n - 1) \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n = n^n, \text{ which is } O(n^n) \]

3. Logarithm (base 2) of factorial:
   \[ \log(n!) \leq \log(n^n) = n \log n, \text{ which is } O(n \log n) \]

Growth of Functions

Note the difference in behavior of functions as $n$ gets larger
Combinations of Functions

- If \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \) then:
  \( (f_1 + f_2)(x) \) is \( O(\max(|g_1(x)|, |g_2(x)|)) \)

- If \( f_1(x) \) and \( f_2(x) \) are both \( O(g(x)) \) then:
  \( (f_1 + f_2)(x) \) is \( O(g(x)) \)

- If \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \) then:
  \( (f_1 \cdot f_2)(x) \) is \( O(g_1(x) \cdot g_2(x)) \)

Upper and Lower Bounds

- Big-Oh gives us upper bounds on functions: if \( f(x) \) is \( O(g(x)) \),
  then the value of \( f(x) \) is at most the value of \( g(x) \) (for large enough \( x \))

- If we want to know a lower bound on a function?
  This is done with the “Big-Omega” notation!
**Big-Omega Notation \( \Omega() \)**

f(x) is **Big-Omega** of a function g(x) if there are positive constants C and k such that \( f(x) \geq C \cdot g(x) \) for all \( x > k \).

Notation: \( f(x) \in \Omega(g(x)) \) or \( f(x) = \Omega(g(x)) \).

**e.g.**

\( f(x) = 8x^3 + 5x^2 + 7 \) is \( \Omega(x^3) \). Verify with \( C=1 \) and \( k=1 \)

**note:**

\( f(x) \) is \( \Omega(g(x)) \) if and only if \( g(x) \) is \( O(f(x)) \).

---

**Big-Omega Example**

Let’s find a lower bound on the sum of the first \( n \) positive integers. Assume \( n \) is even.

\[
\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + (n-1) + n
\]

\[
> \left( \frac{n}{2} + 1 \right) + \left( \frac{n}{2} + 2 \right) + \ldots + (n-1) + n
\]

\[
> \frac{n}{2} + \frac{n}{2} + \ldots + \frac{n}{2} + \frac{n}{2}
\]

\[
= \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}
\]
Big-Theta Notation $\Theta()$

What if a function $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$?

$f(x)$ is Big-Theta of function $g(x)$ if:
- $f(x)$ is $O(g(x))$
- $f(x)$ is $\Omega(g(x))$

Notation: $f(x) \in \Theta(g(x))$ or $f(x) = \Theta(g(x))$

The behavior of $f$ is lying between an upper bound and lower bound that are asymptotically equal. This is a tight bound. Informally, $f \sim g$.

Big-Theta Example

- The sum of the first $n$ positive integers is $O(n^2)$
- The sum of the first $n$ positive integers is $\Omega(n^2)$
- Therefore, the sum of the first $n$ positive integers is $\Theta(n^2)$
The Complexity of Algorithms

- How efficient is an algorithm for a given input size?
- How much time does an algorithm use to solve a problem?
- How much computer memory does an algorithm use to solve a problem?

- When we analyze the time an algorithm uses, we study the time complexity of the algorithm.
- When we analyze the computer memory the algorithm uses, we study the space complexity of the algorithm.

The Complexity of Algorithms

- Let’s us the last two topics, Algorithms and Asymptotic Analysis, to analyze time complexity of algorithms
- We will measure time complexity as the number of operations executed

Simplifying Assumption:
All operations take the same amount of time.

Why?
Operations might vary by a constant (e.g., a division might take 4 times as long as an addition on some hardware), but we ignore constant factors.
Time Complexity
Maximum Finding Algorithm: $O(n)$

1. $\text{max} \leftarrow a_1$
2. $\text{for } i \leftarrow 2 \ldots n \text{ do}$
3. \quad if $\text{max} < a_i$ then
4. \quad \quad $\text{max} \leftarrow a_i$
5. \quad end if
6. end for
7. return $\text{max}$

1: ‘assign’ (1 op)
2: ‘add’, ‘assign’, ‘test’ n times (3n ops)
3: 1 ‘compare’ per iteration (n - 1 ops)
4: at most 1 ‘assign’ per iteration ($\leq n{-}1$ ops)

Total is at most $1 + 3n + 2(n - 1) + 1 = 5n = O(n)$

Time Complexity
Linear Search Algorithm: $O(n)$

1. $i \leftarrow 1$
2. while $i \leq n$ and $x \neq a_i$
3. \quad do $i \leftarrow i + 1$
4. end while
5. if $i \leq n$ then
6. \quad return $i$
7. else
8. \quad return 0
9. end if

1: ‘assign’ (1 op)
2: ‘$\leq$’, ‘$\neq$’, ‘and’ at most $n+1$ times ($\leq 3(n + 1)$ ops)
3: ‘add’, ‘assign’ at most $n$ times ($\leq 2n$ ops)

5-8: ‘compare’, ‘return’ (2 ops)

Total is at most $1 + 3(n + 1) + 2n + 2 = 5n + 6 = O(n)$
Behavior of Binary Search

How many iterations of the while loop are there?

\( n \): # of elements in the list

- Assume \( n \) is a power of 2, \( n = 2^k \rightarrow k = \log_2 n \)
- At each iteration, \( i \) increases or \( j \) decreases
- Iteration 1, \( j - i = \frac{2^k}{2} = 2^{k-1} \)
- Iteration 2, \( j - i = \frac{2^{k-1}}{2} = 2^{k-2} \)
- Iteration 3, \( j - i = \frac{2^{k-2}}{2} = 2^{k-3} \)
- How many iterations until \( i \geq j \) (stopping condition of while)?
- After \( k \) iterations, \( j - i = 2^0 \), so after one more \( i \geq j \)
  - Thus \( x = k + 1 = \log_2 n + 1 \)
Time Complexity
Optimized Insertion Sort $O(n^2)$

1. for $j \leftarrow 2 \ldots n$ do
2. \hspace{1em} $m \leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. \hspace{1em} while $(i > 0) \land (A[i] > m)$
5. \hspace{2em} $A[i + 1] \leftarrow A[i]$
6. \hspace{2em} $i \leftarrow i - 1$
7. \hspace{2em} end while
8. \hspace{1em} $A[i + 1] \leftarrow m$
9. end for
10. return $A$

1: test $n$ times, each test takes 2 ops ($2n$ ops)
2: 2 ops per for-loop iteration ($2(n - 1)$ ops)
3: 2 ops per for-loop iteration ($2(n - 1)$ ops)
4: test on a variable number of times, 3 ops per test.
5: 2 ops per while-loop iteration
6: 2 ops per while-loop iteration
8: 2 ops per for-loop iteration ($2n$ ops)
10: 1 op

Common Terminologies

**Constant Complexity**: $O(1)$ Running time is independent of the input size

- e.g. return the first element of a list

**Logarithmic Complexity**: $O(\log n)$

- e.g. binary search

**Linear Complexity**: $O(n)$

- e.g. linear search

**Quadratic Complexity**: $O(n^2)$

- e.g. insertion sort

**Polynomial Complexity**: $O(n^b)$ for some constant $b$

- e.g. all of the above

**Exponential Complexity**: $O(b^n)$ for some constant $b$

- e.g. generate the powerset of some input set of $n$ elements
## Sorting Algorithms Time Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$\Omega(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$\Omega(n + k)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Omega(nk)$</td>
<td>$O(nk)$</td>
</tr>
</tbody>
</table>

### Sources