# Topic 8: Complexity Analysis

School of Electrical and Computer Engineering  
Cornell University

revision: 2018-10-01-08-37

## 1 Analyzing Algorithms

1.1. Linear Search ......................................................... 3
1.2. Binary Search ....................................................... 4
1.3. Comparing Linear vs. Binary Search ............................. 7

## 2 Analyzing Data Structures

2.1. Chain of Nodes ....................................................... 9
2.2. Array ................................................................. 9
2.3. Comparing a Chain of Nodes vs. an Array ....................... 10

## 3 Time and Space Complexity

3.1. Six-Step Process for Complexity Analysis ..................... 14

## 4 Comparing List and Vector CDTs


1. Analyzing Algorithms

- Assume we have a sorted input array of integers
- Consider algorithms to find if a given value is in the array
- The algorithm should return 1 if value is in array, otherwise return 0

- Let \( N \) be the size of the input array
- Let \( T \) be the execution time of an algorithm
  - Assume \( T \) is measured in number of C statements
- Let \( S \) be the additional storage (space) required by the algorithm
  - Assume \( S \) is measured in number of C variables
- Our goal is to derive equations for \( T \) and \( S \) as a function of \( N \)
1.1. Linear Search

```c
int find( int* a, size_t n, int v ) {
    for ( size_t i = 0; i < n; i++ ) {
        if ( a[i] == v )
            return 1;
        // else if ( a[i] > v )
        //     return 0;
    }
    return 0;
}

int main( void ) {
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int find4 = find( a, 8, 2 );
    int find0 = find( a, 8, 0 );
    int find20 = find( a, 8, 9 );
    return 0;
}
```
1.2. Binary Search

```c
int find_h( int* a, size_t left, size_t right, int v )
{
    if ( a[left] == v ) return 1;
    if ( a[right] == v ) return 1;
    if ( (right-left) == 1 ) return 0;

    int middle = (left + right)/2;
    if ( a[middle] > v )
        return find_h( a, left, middle, v );
    else
        return find_h( a, middle, right, v );
}

int find( int* a, size_t n, int v )
{
    return find_h( a, 0, n-1, v );
}

int main( void )
{
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int find4 = find( a, 8, 2 );
    int find0 = find( a, 8, 0 );
    int find20 = find( a, 8, 9 );
    return 0;
}
```
Annotating call tree with execution time
Annotating call tree with space requirements
1.3. Comparing Linear vs. Binary Search

<table>
<thead>
<tr>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_b(N) = 5$</td>
<td>$T_b(N) = 7$</td>
</tr>
<tr>
<td>$T_w(N) = 4N + 3$</td>
<td>$T_w(N) = 10 \log_2(N) + 9$</td>
</tr>
</tbody>
</table>
### Linear Search

- $S_b(N) = 4$
- $S_w(N) = 4$

### Binary Search

- $S_b(N) = 8$
- $S_w(N) = 5 \log_2(N) + 4$
2. Analyzing Data Structures

• Assume we want to store a series of integers
• Let $N$ be the number of integers
• Let $S$ be the space required by the data structure
• Assume $S$ is measured in number of C “primitive” variables
• Our goal is to derive an equation for $S$ as a function of $N$

2.1. Chain of Nodes

2.2. Array
2.3. Comparing a Chain of Nodes vs. an Array

<table>
<thead>
<tr>
<th>Chain of Nodes</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(N) = 2N$</td>
<td>$S(N) = N$</td>
</tr>
</tbody>
</table>

Diagram showing the comparison between Chain of Nodes and Array, with $S(N)$ values plotted against $N$.
3. Time and Space Complexity

- We want to characterize algorithms at a high-level to compare and contrast the execution time and space usage of two algorithms or data structures as some input variable (e.g., \(N\)) grows large.

- We want to gloss over low-level details:
  - Absolute time of each C statement
  - Absolute storage requirements for each C variable

- Big-Oh notation captures the asymptotic behavior of a function:

  \[
  f(x) \text{ is } O(g(x)) \iff \exists x_0, c. \forall x > x_0. f(x) \leq c \cdot g(x)
  \]

- Formally: \(f(x) \text{ is } O(g(x))\) if there is some value \(x_0\) and some value \(c\) such that for all \(x\) greater than \(x_0\), \(f(x) \leq c \cdot g(x)\).

- Informally: \(f(x) \text{ is } O(g(x))\) if \(g(x)\) captures the “most significant trend” of \(f(x)\) as \(x\) becomes large.
3. Time and Space Complexity

**Big-Oh Examples**

<table>
<thead>
<tr>
<th>$f(x)$</th>
<th>is $O(g(x))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$2N$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$2N + 3$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$4N^2$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>$4N^2 + 2N + 3$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>$4 \log_2(N)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>$N + 4 \log_2(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

- Constant factors do not matter in big-oh notation
- Non-leading terms do not matter in big-oh notation
- What matters is the general trend as $N$ becomes large

**Big-Oh Classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>$N = 100$ requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant Time</td>
</tr>
<tr>
<td>$O(\log_2(N))$</td>
<td>Logarithmic Time</td>
</tr>
<tr>
<td>$O(N)$</td>
<td>Linear Time</td>
</tr>
<tr>
<td>$O(N \cdot \log_2(N))$</td>
<td>Linearithmic Time</td>
</tr>
<tr>
<td>$O(N^2)$</td>
<td>Quadratic Time</td>
</tr>
<tr>
<td>$O(N^3)$</td>
<td>Cubic Time</td>
</tr>
<tr>
<td>$O(N^c)$</td>
<td>Polynomial Time</td>
</tr>
<tr>
<td>$O(2^N)$</td>
<td>Exponential Time</td>
</tr>
<tr>
<td>$O(N!)$</td>
<td>Factorial Time</td>
</tr>
</tbody>
</table>

- Exponential and factorial time algorithms are considered intractable
- With one nanosecond steps, exponential time would require many centuries and factorial time would require the lifetime of the universe
Revisiting linear vs. binary search and chain of nodes vs. array

<table>
<thead>
<tr>
<th></th>
<th>Linear $T_w(N)$</th>
<th>Binary $T_w(N)$</th>
<th>Linear $S_w(N)$</th>
<th>Binary $S_w(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$4N + 3$</td>
<td>$10 \log_2(N) + 9$</td>
<td>$4$</td>
<td>$5 \log_2(N) + 4$</td>
</tr>
<tr>
<td>Binary</td>
<td>$O(N)$</td>
<td>$O(\log_2(N))$</td>
<td>$O(1)$</td>
<td>$O(\log_2(N))$</td>
</tr>
<tr>
<td>Linear</td>
<td>$2N$</td>
<td>$N$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Array</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

- Does this mean binary search is always faster?
- Does this mean linear search always requires less storage?
- Does this mean a chain requires the same storage as an array?
- For large $N$, but we don’t always know $x_0$
  - $T$ or $S$ can have very large constants
  - $T$ or $S$ can have non-leading terms
- This analysis is for worst case complexity
  - results can look very different for best case complexity
  - results can look very different for typical/average complexity
- If two algorithms or data structures have the same complexity, the constants and other terms are what makes the difference!
- For reasonable problem sizes and/or different input data characteristics, sometimes an algorithm with worse time (space) complexity can still be faster (smaller)
3.1. Six-Step Process for Complexity Analysis

1. **Choose** units for execution time or space usage
   - C statements
   - pseudocode lines
   - machine instructions
   - comparisons
   - multiplications
   - variables
   - bytes

2. **Choose** input variables
   - N, K, M, ...

3. **Choose** best case, worst case, typical case
   - explain what is meant by these terms!
   - worst case values in array, random search value
   - ... N copies of the same value in array
   - random values in array, worst case choice
   - ... always search for non-present value

4. **Analyze** for at least two concrete values of input variables
   - T(10) = ...
   - T(99) = ...

5. **Generalize** for any value of input variables
   - T(N) = ...

6. **Characterize** asymptotic behavior
   - T(N) = ... which is O(1)
4. Comparing List and Vector CDTs

• The list and vector CDTs ...  
  – ... have similar interfaces, but  
  – ... very different execution times, and  
  – ... very different space usage.

Analysis of time and space complexity of list_int_push_front

1. void list_int_push_front( list_int_t* this, int v )
2. allocate new node
3. set new node’s value to v
4. set new node’s next ptr to head ptr
5. set head ptr to point to new node

What is the time complexity for list_int_push_front?

What is the space complexity for list_int_push_front?

Analysis of time and space complexity of vector_int_push_front

1. void vector_int_push_front( vector_int_t* this, int v )
2. set prev value to v
3. for i in 0 to vector’s size (inclusive)
4. set temp value to vector’s data[i]
5. set vector’s data[i] to prev value
6. set prev value to temp value
7. set vector’s size to size + 1

What is the time complexity for vector_int_push_front?

What is the space complexity for vector_int_push_front?
### 4. Comparing List and Vector CDTs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>List</td>
<td>Vector</td>
</tr>
<tr>
<td>push_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push_back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Does this mean `list_int_find` and `vector_int_find` will have the same execution time?
- What about the space complexity of the data structure itself?
- If two algorithms or data structures have the same complexity, the constants and other terms are what makes the difference!
- This analysis is for worst case time complexity
  - results can look very different for best case complexity
  - results can look very different for typical/average complexity
- In computer systems programming, we care about time and space complexity, but we also care about absolute execution time and absolute space usage on a variety of inputs