ECE 2400 Computer Systems Programming
Fall 2020

Topic 8: Complexity Analysis

School of Electrical and Computer Engineering
Cornell University

revision: 2020-10-12-00-58

1 Analyzing Simple Algorithms 3

2 Analyzing Simple Data Structures 7

3 Analyzing Algorithms and Data Structures 9
   3.1. Linear Search 10
   3.2. Binary Search 11
   3.3. Comparing Linear vs. Binary Search 14

4 Time and Space Complexity 15
   4.1. Six-Step Process for Complexity Analysis 22

5 Comparing Lists and Vectors 23

6 Art, Principle, and Practice 25
Topic 8: Complexity Analysis
1. Analyzing Simple Algorithms

```c
int mul( int x, int y )
{
    int z = 0;
    for ( int i=0; i<y; i=i+1 ) {
        z = z + x;
    }
    return z;
}
```

```c
int main()
{
    int a = mul(2,3);
    int b = mul(2,4);
    return 0;
}
```

• What is the execution time of this algorithm for specific values of \( y \)?
  – Let \( T(y) \) be execution time for \( y \)

• What units to use for execution time?
  – Number of seconds
  – Number of machine instructions
  – Number of X’s in our state diagram

<table>
<thead>
<tr>
<th>( y )</th>
<th>( T(y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
1. Analyzing Simple Algorithms

```c
int mul( int x, int y )
{
    int z = 0;
    for ( int i = 0; i < y; i = i + 1 ) {
        z = z + x;
    }
    return z;
}
```

- Can we derive a generalized equation for $T(y)$ for `mul` algorithm?
  - Units are the number of $X$’s in our state diagram

- Is the number of $X$’s in our state diagram is a good choice for the units of execution time?
  - Depends on code formatting
  - Complex work in a single line (line 4)
  - Arithmetic work on some lines (line 5)
  - Hardly any work on some lines (line 6)

- We will use the number of critical operations for the units of execution time
  - Choice involves the art of computer systems programming
  - Number of multiplication, division, or remainder operations
  - Number of critical comparisons
  - Number of iterations of a critical loop
  - Number of calls to a critical function

- Can we derive a generalized equation for $T(y)$ for `mul` algorithm?
  - Units are the number of add (+) operations
The following three implementations implement a function to determine if the given number \( x \) is prime (assume \( x > 2 \))

1. Analyzing Simple Algorithms

```c
int is_prime_v1( int x )
{
    int i = 2;
    int ans = 1;
    while ( i < x ) {
        if ( x % i == 0 )
            ans = 0;
        i = i + 1;
    }
    return ans;
}

int is_prime_v2( int x )
{
    int y = x / 2;
    int i = 2;
    int ans = 1;
    while ( i <= y ) {
        if ( x % i == 0 )
            ans = 0;
        i = i + 1;
    }
    return ans;
}

int is_prime_v3( int x )
{
    int i = 2;
    int ans = 1;
    while ( i * i <= x ) {
        if ( x % i == 0 )
            ans = 0;
        i = i + 1;
    }
    return ans;
}
```

Fill in table then derive generalized equations for \( T_{v1}(x) \), \( T_{v2}(x) \), \( T_{v3}(x) \)

- The “itr” column is the number of iterations of the while loop
- \( T(x) \) is measured in mul/div/rem operations

<table>
<thead>
<tr>
<th>( x )</th>
<th>( v1 )</th>
<th>( v2 )</th>
<th>( v3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr</td>
<td>( T_{v1}(x) )</td>
<td>tr</td>
<td>( T_{v2}(x) )</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Analyzing Simple Algorithms

```c
int is_prime_v4( int x )
{
    int i = 2;
    while ( i < x ){
        if ( x % i == 0 )
            return 0;
        i = i + 1;
    }
    return 1;
}
```

• What if we exit the while loop early?
  – Can we derive a generalized equation for $T(x)$ in the best case?
  – Can we derive a generalized equation for $T(x)$ in the worst case?
  – Can we derive a generalized equation for $T(x)$ in the average case?
2. Analyzing Simple Data Structures

```c
typedef struct _node_t
{
    int value;
    struct _node_t* next_ptr;
} node_t;

node_t* append( node_t* n_ptr, int v )
{
    node_t* new_ptr = malloc( sizeof(node_t) );
    new_ptr->value = v;
    new_ptr->next_ptr = n_ptr;
    return new_ptr;
}

int main( void )
{
    node_t* n_ptr = NULL;
    n_ptr = append( n_ptr, 3 );
    n_ptr = append( n_ptr, 4 );
    free( n_ptr->next_ptr );
    free( n_ptr );
    return 0;
}
```

- What is the space usage of this data structure for specific values of \( N \) where \( N \) is the number of elements appended to chain of nodes?
  - Let \( S(N) \) be space usage for \( N \) elements

- What units to use for space usage?
  - Bytes on the heap or stack
  - Variables on the heap
  - Frames on the stack
2. Analyzing Simple Data Structures

- Can we derive a generalized equation for $S(N)$ for chain of nodes?
  - Units are variables on the heap
  - We care about the maximum usage not the total usage

Derive generalized equation for $S(N)$ for array of elements

- Units are the variables on the heap

```c
int main( void )
{
  int N = 1000;

  int* a = malloc( N*sizeof(int) );
  for ( size_t i = 0; i < N; i++ )
    a[i] = i;
  free(a);

  int* b = malloc( N*sizeof(int) );
  for ( size_t i = 0; i < N; i++ )
    b[i] = i;
  free(b);
  return 0;
}
```

Kinds of Heap Space Usage

- Heap space usage of the data structure itself as function of $N$
- Heap space usage for a function as a function of $N$
  - Should we include the heap space usage of the input data structure?
  - This heap space usage is always the same regardless of the function!
  - **Auxiliary heap space usage** focuses on the heap space usage that the function requires in *addition* to the heap space usage required by the data structure itself
3. Analyzing Algorithms and Data Structures

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

```c
int contains( int* x, size_t n, int v )
```

• Let $N$ be the size of the input array
• Let $T$ be the execution time measured in num of element comparisons
• Let $S$ be the stack space usage measured in number of stack frames
• Our goal is to derive equations for $T$ and $S$ as a function of $N$
3. Analyzing Algorithms and Data Structures

3.1. Linear Search

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

int main( void )
{
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int contains2 = contains( a, 8, 2 );
    int contains0 = contains( a, 8, 0 );
    int contains9 = contains( a, 8, 9 );
    return 0;
}
```
3.2. Binary Search

```c
int contains_h( int* x, size_t lo, size_t hi, int v )
{
    if ( lo == hi )
        return ( v == x[lo] );
    int mid = (lo + hi)/2;
    if ( v <= x[mid] )
        return contains_h( x, lo, mid, v );
    else
        return contains_h( x, mid+1, hi, v );
}

int contains( int* x, size_t n, int v )
{
    return contains_h( x, 0, n-1, v );
}

int main( void )
{
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int contains2 = contains( a, 8, 2 );
    int contains0 = contains( a, 8, 0 );
    int contains9 = contains( a, 8, 9 );
    return 0;
}
```

Topic 8: Complexity Analysis
Annotating call tree with execution time and stack space usage
3. Analyzing Algorithms and Data Structures

3.2. Binary Search

Algorithm: Binary Search

- **Binary Search**

1. Start at the middle of the array.
2. Compare the middle element with the target value.
3. If the target is equal to the middle element, return the middle index.
4. If the target is less than the middle element, repeat the search in the lower half of the array.
5. If the target is greater than the middle element, repeat the search in the upper half of the array.
6. Continue this process until the target value is found or the search space is exhausted.

**Example:**

```
1 1 1 1 1 1 1 1
8
4
2
1
1
1
1
```

- **Binary Search Trees**

```
N = 8 = 2^3
k = 3
```

```
N = 16 = 2^4
k = 4
```

```
N = 32 = 2^5
k = 5
```

**Recursive Tree Representation:**

```
\[
\begin{array}{c}
N \\
N/2 \\
N/4 \\
N/8 \\
\vdots \\
2 \\
1
\end{array}
\]
```

**Topic 8: Complexity Analysis**
3.3. Comparing Linear vs. Binary Search

Linear: $T_w(N) = N$
Binary: $T_w(N) = \log_2(N) + 1$

<table>
<thead>
<tr>
<th>$N$</th>
<th>Linear</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^2$</td>
<td>$10^2$</td>
<td>7</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$10^3$</td>
<td>10</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$10^4$</td>
<td>14</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$10^5$</td>
<td>17</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>20</td>
</tr>
</tbody>
</table>

- By measuring execution time in critical operations, we have abstracted away many real-world overheads
  - Binary search has many more C statements and function calls
  - Real-world overheads can increase the real execution time by constant factors and trailing terms

```
int contains_h( int* x, size_t lo, size_t hi, int v )
{
  if ( lo == hi ) {
    // add extra 1 to trailing term?
    return ( v == x[lo] );
  }

  // add extra 2 to constant factor?
  int mid = (lo + hi)/2;
  if ( v < x[mid] )
    return contains_h( x, lo, mid, v );
  else
    return contains_h( x, mid+1, hi, v );
}
```
4. Time and Space Complexity

- We have been using high-level units such as the number of critical operations, stack frames, and heap variables.

- We want to analyze algorithms and data-structures at an even higher level to broadly characterize high-level trends as some input variable (e.g., \( N \)) grows large.

- Big-O notation is a formal way to characterize high-level trends:

  \[
  f(N) \text{ is } O(g(N)) \iff \exists N_0, c. \ \forall N > N_0. \ f(N) \leq c \cdot g(N)
  \]

- \( f(N) \text{ is } O(g(N)) \) if there is some value \( N_0 \) and some value \( c \) such that for all \( N \) greater than \( N_0 \), \( f(N) \leq c \cdot g(N) \)
  - \( g(N) \) can be thought of as an “upper bounding function”
  - \( f(N) \) can be \( T(N) \) or \( S(N) \)
4. Time and Space Complexity

\[ T(N) = 2N + 1 \]

\[ T(N) = 2 \log_2(N) + 2 \]
4. Time and Space Complexity

\[ T(N) = 2\sqrt{N} + 2 \]

\[ T(N) = N + \sqrt{N} \]
• Big-O notation captures the fastest-growing term (high-level trend) as $N$ becomes large

• With large enough $c$, $g(N)$ can ignore ...
  – ... constant factors in $f(N)$
  – ... non-leading terms in $f(N)$

\[
T(N) = N + 2 \\
T(N) \text{ is } O(N)
\]

\[
T(N) = 2\log_2(N) + 2 \\
T(N) \text{ is } O(\log_2(N))
\]

\[
T(N) = 2\sqrt{N} + 2 \\
T(N) \text{ is } O(\sqrt{N})
\]

\[
T(N) = N + \sqrt{N} \\
T(N) \text{ is } O(N)
\]
4. Time and Space Complexity

- Technically all four functions are $O(N^2)$
  - Choose $c = 1$ and $N_0$ is around 2

- Saying all four functions are $O(N^2)$ does not provide any insight

- We want to choose the function with the “tightest” bound which will provide the most insight for our analysis

\[
T(N) = N + 2
\]

\[
T(N) = 2\log_2(N) + 2
\]

\[
T(N) = 2\sqrt{N} + 2
\]

\[
T(N) = N + \sqrt{N}
\]
4. Time and Space Complexity

Big-O Examples

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

• Big-O notation captures the fastest-growing term (high-level trend) as \( N \) becomes large

• Constant factors do not matter in big-O notation

• Non-leading terms do not matter in big-O notation

• Base of log does not matter in big-O notation

Big-O Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>( N = 100 ) requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>Constant</td>
</tr>
<tr>
<td>( O(\log(N)) )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( O(\sqrt{N}) )</td>
<td>Square Root</td>
</tr>
<tr>
<td>( O(N^c) ) where ( c &lt; 1 )</td>
<td>Fractional Power</td>
</tr>
<tr>
<td>( O(N) )</td>
<td>Linear</td>
</tr>
<tr>
<td>( O(N \cdot \log(N)) )</td>
<td>Log-Linear</td>
</tr>
<tr>
<td>( O(N^2) )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( O(N^3) )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( O(N^c) ) where ( c &gt; 1 )</td>
<td>Polynomial</td>
</tr>
<tr>
<td>( O(2^N) )</td>
<td>Exponential</td>
</tr>
<tr>
<td>( O(N!) )</td>
<td>Factorial</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

<table>
<thead>
<tr>
<th></th>
<th>Linear $T_w(N) = N$ is $O(N)$</th>
<th>Binary $T_w(N) = \log_2(N) + 1$ is $O(\log(N))$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear $S_w(N) = 2$ is $O(1)$</td>
<td>Binary $S_w(N) = \log_2(N) + 3$ is $O(\log(N))$</td>
</tr>
<tr>
<td></td>
<td>linear time</td>
<td>logarithmic time</td>
</tr>
<tr>
<td></td>
<td>constant stack space</td>
<td>logarithmic stack space</td>
</tr>
</tbody>
</table>

- Does this mean binary search is always faster?
- Does this mean linear search always requires less storage?
- For large $N$, but we don’t always know $N_0$
  - $T(N)$ or $S(N)$ can have very large constants
  - $T(N)$ or $S(N)$ can have very large 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)
4. Time and Space Complexity

4.1. Six-Step Process for Complexity Analysis

1. **Choose** units for execution time or space usage
   - multiplication, division, remainder operations
   - critical comparisons
   - iterations of a critical loop
   - calls to a critical function
   - stack frames
   - heap variables

2. **Choose** key constants and input variable
   - K might be a key constant, explore how time and space change with K
   - N, M might be variables, explore how time and space change with N, M
   - Let $T_K(N)$ be execution time, K is key constant, N is input variable
   - Let $S_K(N)$ be space usage, K is key constant, N is input variable

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 specific values of key constants and input variable
   - $T_8(10) = ...$
   - $T_{32}(99) = ...$

5. **Generalize** for any value of key constants and input variable
   - $T_K(N) = ...$

6. **Characterize** asymptotic behavior using big-O notation
   - $T_K(N) = ...$ which is $O(1)$

Then possibly iterate! (different units? different K? best, worst, typical?)
5. Comparing Lists and Vectors

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

Analysis of time and space complexity of slist_int_push_front

1. void slist_int_push_front( slist_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?

What is the auxiliary heap space complexity?

Analysis of time and space complexity of bvector_int_push_front

1. void bvector_int_push_front( bvector_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?

What is the auxiliary heap space complexity?
5. Comparing Lists and Vectors

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>slist</td>
<td>bvector</td>
</tr>
<tr>
<td>push_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push_back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_closest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print</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 slist_int_contains and bvector_int_contains will have the same execution time?

- What about the space complexity of the data structure itself?

- 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.
6. **Art, Principle, and Practice**

- The **art** of computer systems programming is ...
  - developing the algorithm
  - choosing the right units, key constants, input variable
  - choosing best, worst, average case analysis

- The **principle** of computer systems programming is ...
  - performing rigorous time and space complexity analysis
  - analyze, generalize, characterize

- The **practice** of computer systems programming is ...
  - considering the constant factors and trailing terms
  - performing real experiments to evaluate execution time and space usage

---

**Art**

**Computer Systems Programming**

**Practice**

- time and space complexity
- critical operations
- stack frames
- heap variables
- conceptual state diagrams
- machine instructions
- machine memory
- seconds and bytes on stack and heap
- seconds and bytes on real machine