ECE 2400 Computer Systems Programming
Fall 2019

Topic 8: Complexity Analysis

School of Electrical and Computer Engineering
Cornell University

revision: 2019-10-07-09-09

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 .......... 13

4 Time and Space Complexity 14
   4.1. Six-Step Process for Complexity Analysis .......... 17

5 Comparing Lists and Vectors 18

6 Art, Principle, and Practice 20
Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.
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;
}

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 the 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>
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 $T(y)$?
  - 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 multiply, division, or remainder operations
  - Number of key comparisons
  - Number of calls to a specific complex function

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

```c
1 int is_prime_v1( int x )
2 {
3    int i = 2;
4    int ans = 1;
5    while ( i < x ) {
6        if ( x % i == 0 )
7            ans = 0;
8        i = i + 1;
9    }
10   return ans;
11 }
```

```c
1 int is_prime_v2( int x )
2 {
3    int y = x / 2;
4    int i = 2;
5    int ans = 1;
6    while ( i <= y ) {
7        if ( x % i == 0 )
8            ans = 0;
9        i = i + 1;
10    }
11   return ans;
12 }
```

```c
1 int is_prime_v3( int x )
2 {
3    int i = 2;
4    int ans = 1;
5    while ( i * i <= x ) {
6        if ( x % i == 0 )
7            ans = 0;
8        i = i + 1;
9    }
10   return ans;
11 }
```

**Fill in the table with the execution time for each implementation, then derive generalized equation for \( T(x) \)**

- Critical operations are the mul/div/rem operations

<table>
<thead>
<tr>
<th>( x )</th>
<th>( T_{v1}(x) )</th>
<th>( T_{v2}(x) )</th>
<th>( T_{v3}(x) )</th>
</tr>
</thead>
<tbody>
<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_v1( 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 );
    n_ptr->next_ptr = NULL;
    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 the space usage

- What units to use for execution time?
  - Bytes on the stack and heap
  - Frames on the stack
  - Variables on the heap
2. Analyzing Simple Data Structures

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

- 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;
}
```
3. Analyzing Algorithms and Data Structures

• 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 element comparisons
• Let \( S \) be the stack storage space required by the algorithm
  • Assume \( S \) is measured in number of frames
• Our goal is to derive equations for \( T \) and \( S \) as a function of \( N \)
3.1. Linear Search

```c
int find( 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 find2 = find( a, 8, 2 );
    int find0 = find( a, 8, 0 );
    int find9 = find( a, 8, 9 );
    return 0;
}
```
3.2. Binary Search

```c
int find_h( int* x, size_t left, size_t right, int v ) {
    if ( x[left] == v ) return 1;
    if ( x[right] == v ) return 1;
    if ( (right-left) == 1 ) return 0;
    int middle = (left + right)/2;
    if ( x[middle] > v )
        return find_h( x, left, middle, v );
    else
        return find_h( x, middle, right, v );
}
int find( int* x, size_t n, int v ) {
    return find_h( x, 0, n-1, v );
}
int main( void ) {
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int find2 = find( a, 8, 2 );
    int find0 = find( a, 8, 0 );
    int find9 = find( a, 8, 9 );
    return 0;
}
```

Topic 8: Complexity Analysis
Annotating call tree with execution time and (stack) space usage
### 3.3. Comparing Linear vs. Binary Search

<table>
<thead>
<tr>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_b(N) = 1$</td>
<td>$T_b(N) = 1$</td>
</tr>
<tr>
<td>$T_w(N) = N$</td>
<td>$T_w(N) = 3 \log_2(N) + 2$</td>
</tr>
</tbody>
</table>
4. 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.
- Big-O 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: $f(x)$ 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: $f(x)$ is $O(g(x))$ if $g(x)$ captures the “most significant trend” of $f(x)$ as $x$ becomes large.
4. Time and Space Complexity

**Big-O 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-O 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

---

**Topic 8: Complexity Analysis**
4. Time and Space Complexity

Revisiting linear vs. binary search

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

<table>
<thead>
<tr>
<th></th>
<th>Linear $S_w(N) = 1$</th>
<th>is $O(1)$</th>
<th>constant space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>$S_w(N) = \log_2(N) + 1$</td>
<td>is $O(\log_2(N))$</td>
<td>logarithmic 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 $x_0$
  - $T$ or $S$ can have very large constants
  - $T$ or $S$ 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)
4.1. 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 key constant parameters and input variables
   - 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
   - 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 at least two concrete values of input variables
   - $T_8(10) = ...$
   - $T_{32}(99) = ...$

5. Generalize for any value of input variables
   - $T_K(N) = ...$

6. Characterize asymptotic behavior
   - $T_K(N) = ...$ which is $O(1)$
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 for `slist_int_push_front`?

What is the (heap) space complexity for `slist_int_push_front`?

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)
   - set temp value to vector’s data[i]
   - set vector’s data[i] to prev value
   - set prev value to temp value
   - set vector’s size to size + 1

What is the time complexity for `bvector_int_push_front`?

What is the (heap) space complexity for `bvector_int_push_front`?
### 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></td>
<td></td>
<td></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>find</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_find` and `bvector_int_find` 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 variables
  - 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

---

<table>
<thead>
<tr>
<th>time and space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical operations</td>
</tr>
<tr>
<td>stack frames</td>
</tr>
<tr>
<td>heap variables</td>
</tr>
<tr>
<td>conceptual state diagrams</td>
</tr>
<tr>
<td>machine instructions</td>
</tr>
<tr>
<td>machine memory</td>
</tr>
<tr>
<td>seconds and bytes on real machine</td>
</tr>
</tbody>
</table>