ECE 2400 Computer Systems Programming
Fall 2017

Topic 8: Algorithm Analysis

School of Electrical and Computer Engineering
Cornell University

revision: 2017-09-29-09-08

1. Analyzing Two Search 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

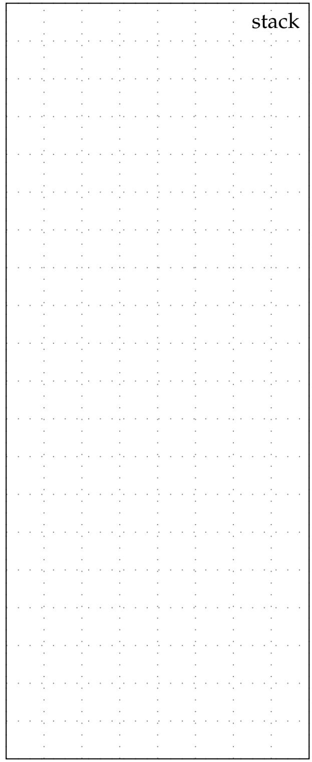
- Let N be the size of the input array
- Let T be the execution time of an algorithm
- Let S be the additional storage (space) required by the algorithm
- Our goal is to derive equations for T and S as a function of N

- Our equations can be rough estimates
- Execution time can be measured in number of C statements
- Space requirements can be measured in number of C variables

1.1. Linear Search

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

int main( void )
{
  int a[] = { 0, 2, 4, 6, 8, 10, 12, 14 };
  int find4  = find( a, 8,  4 );
  int find0  = find( a, 8,  0 );
  int find20 = find( a, 8, 20 );
  return 0;
}
```



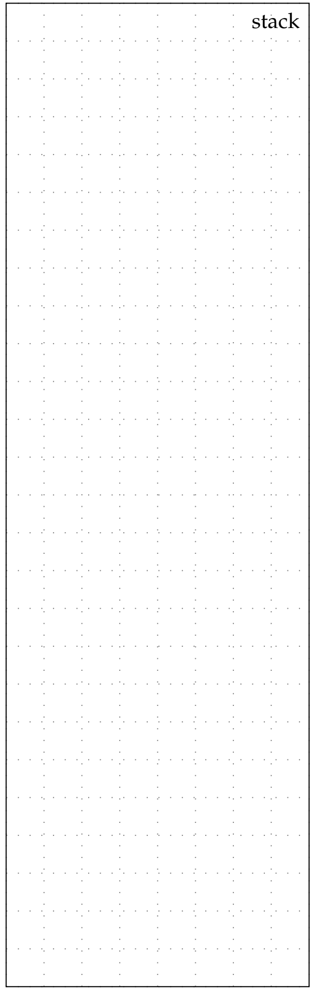
1.2. Binary Search

```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-left)/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 size, int v )
{
  return find_h( a, 0, size-1, v );
}

int main( void )
{
  int a[] = { 0, 2, 4, 6, 8, 10, 12, 14 };
  int find4  = find( a, 8,  4 );
  int find0  = find( a, 8,  0 );
  int find20 = find( a, 8, 20 );
  return 0;
}
```



Annotating call tree with execution time

Annotating call tree with space requirements

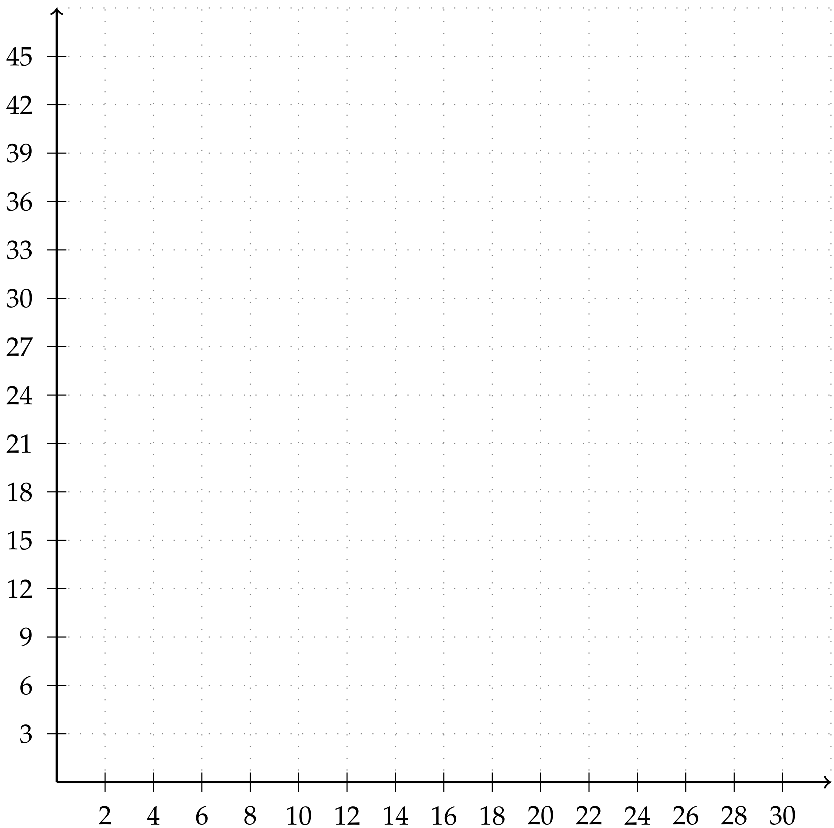
1.3. Comparing Linear vs. Binary Search

Linear Search

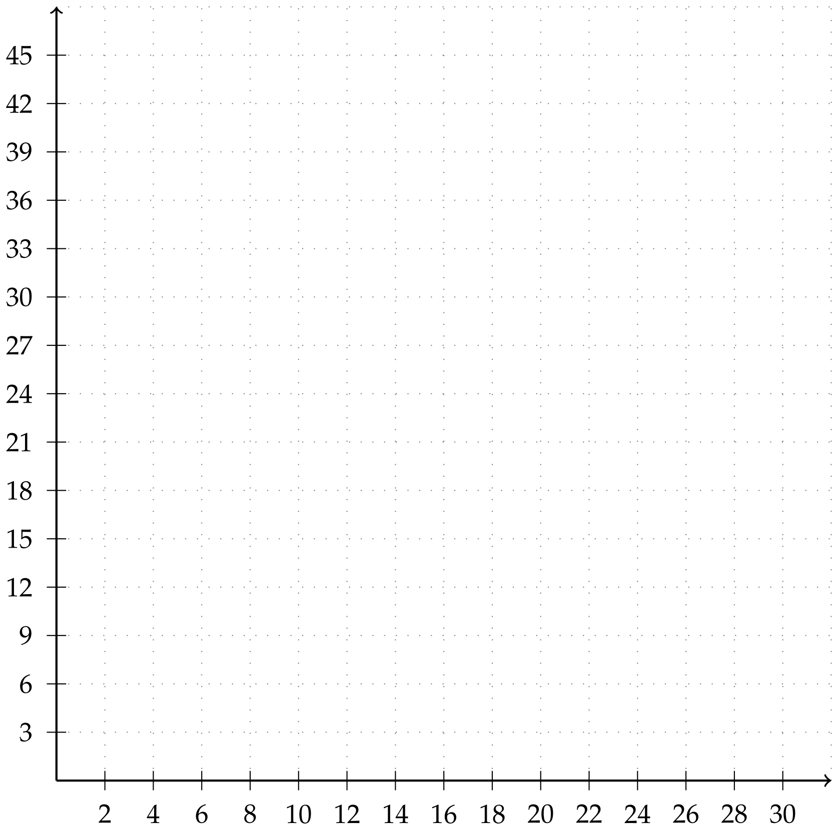
$$T_b(N) = 3$$
$$T_w(N) = 2N + 2$$

Binary Search

$$T_b(N) = 3$$
$$T_w(N) = 8\ log_2(N) + 3$$



Linear Search	Binary Search
$S_b(N) = 4$	$S_b(N) = 8$
$S_w(N) = 4$	$S_w(N) = 5\ log_2(N) + 4$

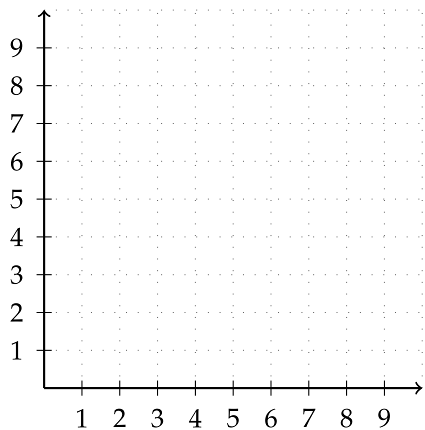


2. Time and Space Complexity

- We want a way characterize algorithms at a high-level so we can quickly compare and contrast the expected performance of two algorithms as N grows large
- We want to gloss over low-level details
  - Absolute time of each C statement
  - Absolute storage requiremententes for each C variable
- Big-Oh notation captures the asymptotic behavior of a function

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

- 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 very large



Big-Oh Examples

$f(x)$	is	$O(g(x))$
3	is	$O(1)$
$2N$	is	$O(N)$
$2N+3$	is	$O(N)$
$4N^2$	is	$O(N^2)$
$4N^2+2N+3$	is	$O(N^2)$
$4\ log_2(N)$	is	$O(log_2(N))$
$N+4\ log_2(N)$	is	$O(N)$

- 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 very large

Big-Oh Classes

	Class	**N = 100 requires**
$O(1)$	Constant Time	1 step
$O(log_2(N))$	Logarithmic Time	6–7 steps
$O(N)$	Linear Time	100 steps
$O(N \cdot log_2(N))$	Linearithmic Time	664 steps
$O(N^2)$	Quadratic Time	10K steps
$O(N^3)$	Cubic Time	1M steps
$O(N^c)$	Polynomial Time	
$O(2^N)$	Exponential Time	1e30 steps
$O(N!)$	Factorial Time	9e157 steps

- 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

Linear	$T_w(N) = 2N + 2$	is	$O(N)$	linear time
Binary	$T_w(N) = 8\ log_2(N) + 3$	is	$O(log_2(N))$	logarithmic time
Linear	$S_w(N) = 4$	is	$O(1)$	constant space
Binary	$S_w(N) = 5\ log_2(N) + 4$	is	$O(log_2(N))$	logarithmic space

- Does this mean binary search is always faster?
- Does this mean linear search always require less storage?
- For very large N, but we don't always know x_0
  - T can have very large constants
  - T can have non-leading terms
- This analysis is for worst case complexity
  - results can look very different for best case complexity (both $O(1)$)
  - results can look very different for typical/average complexity
- For reasonable problem sizes and/or different input data characteristics, sometimes an algorithm with worse time (space) complexity can still be faster (smaller)

3. Analysis of List and Vector Data Structures

Let's analyze the time and space complexity of various operations on the list and vector data structures.

Analysis of time and space complexity of `list_insert`

```
void list_push_front( list_t* list_p, int v )
  allocate new node
  set new node's value to v
  set new node's next ptr to head ptr
  set head ptr to point to new node

void list_insert( list_t* list_p, node_t* node_p, int v )
  if list is empty
    list_push_front( list_p, v )
  else
    allocate new node
    set new node's value to v
    set new node's next ptr to node's next ptr
    set node's next ptr to point to new node
```

What is the time complexity for `list_insert`?

What is the space complexity for `list_insert`?

Analysis of time and space complexity of `vector_insert`

```
void vector_push_front( vector_t* vec_p, int v )
  set prev value to v
  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

void vector_insert( vector_t* vec_p, size_t idx, int v )
  if vector is empty
    vector_push_front( vec_p, v )
  else
    set prev value to v
    for i in idx+1 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 `vector_insert`?

What is the space complexity for `vector_insert`?

Analysis of time and space complexity of `list_sorted_insert`

```
void list_sorted_insert( list_t* list_p, int v )
  set prev node ptr to head ptr
  set curr node ptr to head node's next ptr

  while curr node ptr is not NULL
    if v is less than curr node's value
      list_insert( list_p, prev node ptr, v )
      return

    set prev node ptr to curr node ptr
    set curr node ptr to curr node's next ptr
```

What is the time complexity for `list_sorted_insert`?

What is the space complexity for `list_sorted_insert`?

Analysis of time and space complexity of `vector_sorted_insert`

```
void vector_sorted_insert( vector_t* vec_p, int v )
  for i in 0 to vector's size
    if v is less than vector's data[i]
      vector_insert( vec_p, i-1, v )
      return
```

What is the time complexity for `vector_sorted_insert`?

What is the space complexity for `vector_sorted_insert`?

Analysis of time and space complexity of `list_sort_insert`

```
void list_sort( list_t* list_p )
  construct output list

  set curr node ptr to input list's head ptr
  while curr node ptr is not NULL
    list_sorted_insert( output list, curr node's value )
    set curr node ptr to curr node's next ptr

  destruct input list
  set input list's head ptr to output list's head ptr
```

What is the time complexity for `list_sort`?

What is the space complexity for `list_sort`?

Analysis of time and space complexity of `vector_sort_insert`

```
void vector_sort( vector_t* vec_p )
  construct output vector

  for i in 0 to vector's size
    vector_sorted_insert( output vector, input vector's data[i] )

  destruct input vector
  set input vectors data ptr to output list's data ptr
```

What is the time complexity for `vector_sort`?

What is the space complexity for `vector_sort`?

	Time Complexity		Space Complexity	
Operation	**List**	**Vector**	**List**	**Vector**
`insert`				
`sorted_insert`				
`sort`				
`push_front`				
`push_back`				
`find`				

- Does this mean `list_sort` and `vector_sort` will have the same execution time? absolutely not!
- If two algorithms have the same time complexity, the constants and other terms are what makes the difference!
- 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
- In computer systems programming, we care about time and space complexity, but we also care about absolute execution time and absolute space requirements on a variety of inputs