

# ECE 2400 Computer Systems Programming Spring 2025

## Topic 8: Complexity Analysis

School of Electrical and Computer Engineering, Cornell University

revision: 2026-02-25-11-47

<b>1</b>	<b>Analyzing Simple Algorithms</b>	<b>2</b>
<b>2</b>	<b>Analyzing Simple Data Structures</b>	<b>5</b>
<b>3</b>	<b>Analyzing Algorithms and Data Structures</b>	<b>7</b>
3.1.	Linear Search . . . . .	8
3.2.	Binary Search . . . . .	9
3.3.	Comparing Linear vs. Binary Search . . . . .	11
<b>4</b>	<b>Time and Space Complexity</b>	<b>12</b>
4.1.	Six-Step Process for Complexity Analysis . . . . .	19
<b>5</b>	<b>Comparing Lists and Vectors</b>	<b>20</b>
<b>6</b>	<b>Art, Principle, and Practice</b>	<b>22</b>

# 1. Analyzing Simple Algorithms

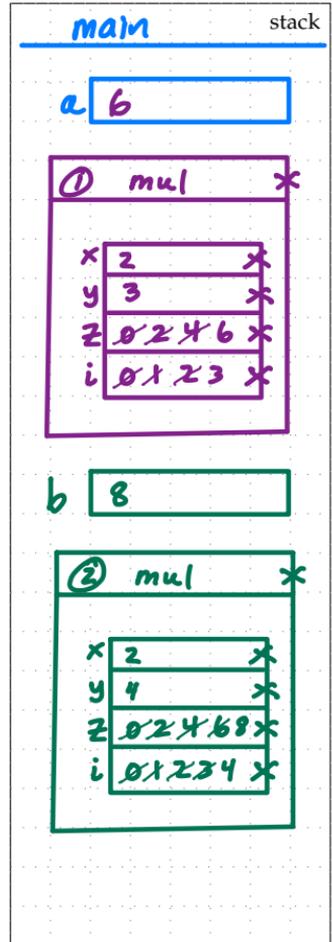
```

01 int mul( int x, int y )
02 {
03     int z = 0;
04     for ( int i=0; i<y; i=i+1 ) {
05         z = z + x;
06     }
07     return z;
08 }
09
10 int main()
11 {
12     int a = mul(2,3); ①
13     int b = mul(2,4); ②
14     return 0;
15 }

```

- 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

$y$	$T(y)$
3	14
4	17
5	
6	
7	



```
1  int mul( int x, int y )
2  {
3      int z = 0;
4      for ( int i = 0; i < y; i = i + 1 ) {
5          z = z + x;
6      }
7      return z;
8  }
```

- 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 critical multiplication, division, or remainder operations
  - Number of critical comparisons, swaps, node accesses, array accesses
  - Number of critical loop iterations
  - Number of critical function calls
  
- 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 int is_prime_v1( int x ) 1 int is_prime_v2( int x ) 1 int is_prime_v3( int x )
2 { 2 { 2 {
3     int i = 2; 3     int y = x / 2; 3     int i = 2;
4     int ans = 1; 4     int i = 2; 4     int ans = 1;
5     while ( i < x ) { 5     int ans = 1; 5     while ( i * i <= x ) {
6         if ( x % i == 0 ) 6     while ( i <= y ) { 6         if ( x % i == 0 )
7             ans = 0; 7         if ( x % i == 0 ) 7             ans = 0;
8             i = i + 1; 8             ans = 0; 8             i = i + 1;
9     } 9     i = i + 1; 9     }
10    return ans; 10 } 10    return ans;
11 } 11    return ans; 11 }
12 }
```

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

	v1		v2		v3	
$x$	itr	$T_{v1}(x)$	itr	$T_{v2}(x)$	itr	$T_{v3}(x)$
3						
4						
5						
6						
7						
8						
9						
10						
99						

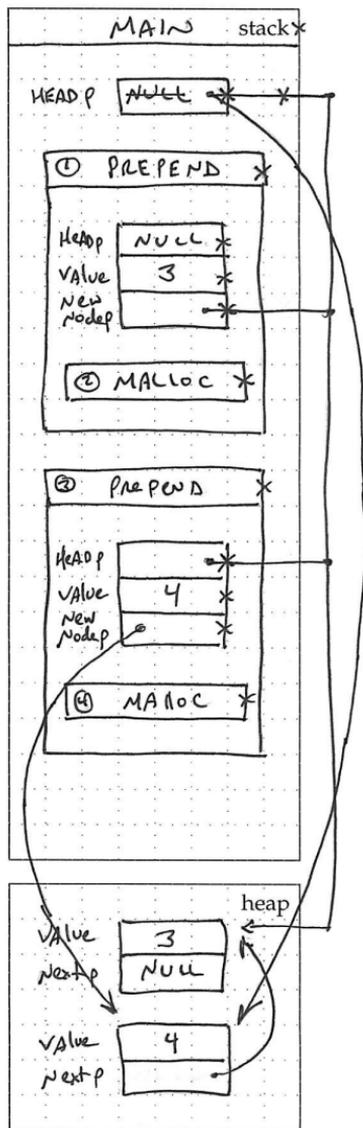
## 2. Analyzing Simple Data Structures

```

1 typedef struct _node_t
2 {
3     int         value;
4     struct _node_t* next_ptr;
5 }
6 node_t;
7
8 node_t* prepend( node_t* n_ptr, int v )
9 {
10    node_t* new_ptr;
11    new_ptr = malloc( sizeof(node_t) );
12
13    new_ptr->value    = v;
14    new_ptr->next_ptr = n_ptr;
15    return new_ptr;
16 }
17
18 int main( void )
19 {
20    node_t* n_ptr = NULL;
21    n_ptr = prepend( n_ptr, 3 );
22    n_ptr = prepend( n_ptr, 4 );
23    free( n_ptr->next_ptr );
24    free( n_ptr );
25    return 0;
26 }

```

- What is the space usage of this data structure for specific values of  $N$  where  $N$  is the number of elements prepended 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



- 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

```
1 int main( void )
2 {
3     int N = 1000;
4
5     int* a = malloc( N*sizeof(int) );
6     for ( int i = 0; i < N; i++ )
7         a[i] = i;
8     free(a);
9
10    int* b = malloc( N*sizeof(int) );
11    for ( int i = 0; i < N; i++ )
12        b[i] = i;
13    free(b);
14
15    return 0;
16 }
```

### Kinds of Heap Space Usage

- Heap space usage of the data structure itself as function of  $N$
- Heap space usage for an algorithm as a function of  $N$ 
  - Should we include the heap space usage of an 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 algorithm 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 returns true if value is in array, otherwise return false

```
bool search( int* x, int 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  $N$

### 3.1. Linear Search

```

1 #include <stdbool.h>
2
3 // searching for value v in array x
4 bool lsearch( int* x, int n, int v )
5 {
6     for ( int i = 0; i < n; i++ ) {
7         if ( x[i] == v )
8             return true;
9         // possible optimization if array x is sorted, but also 2nd cmp
10        // else if ( x[i] > v )
11        //     return 0;
12    }
13    return false;
14 }
15
16 int main( void )
17 {
18     int a[] = { 0, 10, 20, 30, 40, 50, 60, 70 };
19     int b = lsearch( a, 8, 20 );
20     return 0;
21 }

```

Fill in table then derive generalized equations for  $T_k(N)$  and  $S_k(N)$

- Let  $k$  be the array index of value  $v$  in array  $x$
- Execution time in units of element comparisons (i.e., `==` on line 7)
- Space usage in units of stack frames

$v$	$k$	$T(8)$	$S(8)$
0	0		
10	1		
20	2		
70	7		
99	n/a		

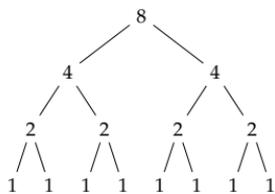
## 3.2. Binary Search

```

1 #include <stdbool.h>
2
3 bool bsearch_h( int* x, int first, int last, int v )
4 {
5     int size = last - first;
6     if ( size == 1 )
7         return ( x[first] == v );
8
9     int mid = (first + last)/2;
10    if ( v < x[mid] )
11        return bsearch_h( x, first, mid, v );
12    else
13        return bsearch_h( x, mid, last, v );
14 }
15
16 bool bsearch( int* x, int n, int v )
17 {
18     return bsearch_h( x, 0, n, v );
19 }
20
21 int main( void )
22 {
23     int a[] = { 0, 10, 20, 30, 40, 50, 60, 70 };
24     int b = bsearch( a, 8, 20 );
25     return 0;
26 }

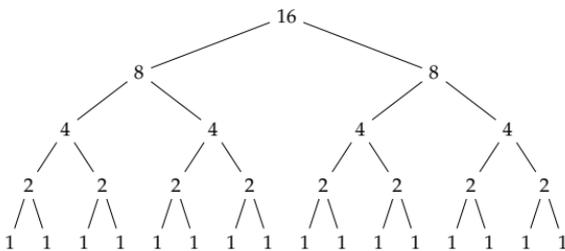
```

$v$	$k$	$T(8)$	$T(16)$
0	0		
10	1		
20	2		
70	7		
99	n/a		



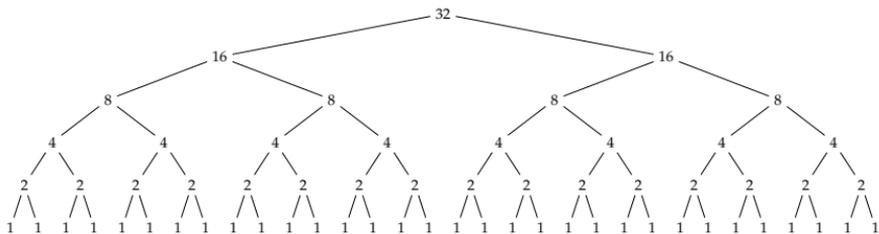
$$N = 8 = 2^3$$

$$r = 3$$



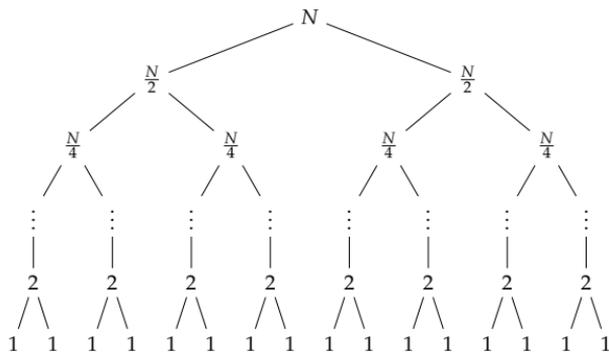
$$N = 16 = 2^4$$

$$r = 4$$

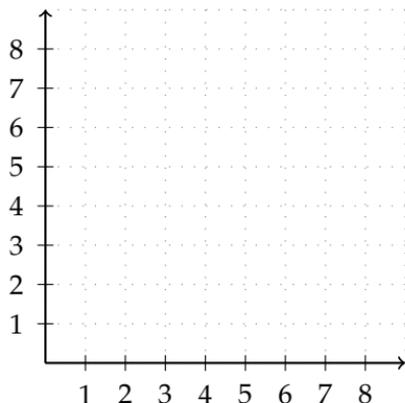


$$N = 32 = 2^5$$

$$r = 5$$



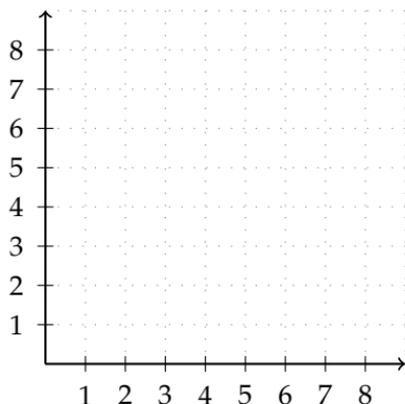
### 3.3. Comparing Linear vs. Binary Search



$$\text{Linear: } T_{worst}(N) = N$$

$$\text{Binary: } T_w(N) = \log_2(N) + 1$$

$N$	Linear	Binary
$10^2$	$10^2$	7
$10^3$	$10^3$	10
$10^4$	$10^4$	14
$10^5$	$10^5$	17
$10^6$	$10^6$	20



$$\text{Linear: } T_w(N) = N$$

$$\text{Binary: } T_w(N) = 2 \log_2(N) + 2$$

```
bool bsearch_h( int* x, int first,
               int last, int v )
{
    int size = last - first;
    if ( size == 1 ) {
        // what if we need extra
        // element comparison here?
        return ( x[first] == v );
    }

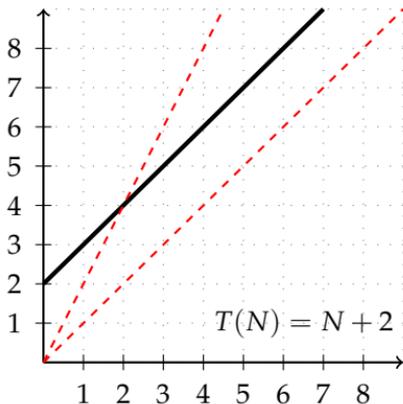
    // what if we need extra
    // element comparison here?
    int mid = (first + last)/2;
    if ( v < x[mid] )
        return bsearch_h( x, first, mid, v );
    else
        return bsearch_h( x, mid, last, v );
}
```

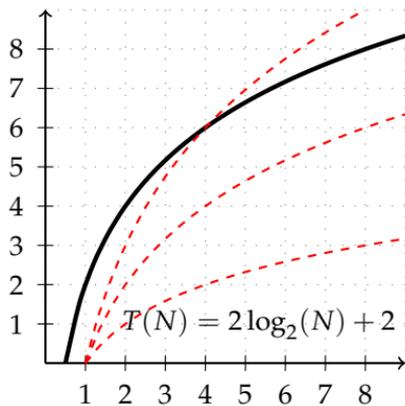
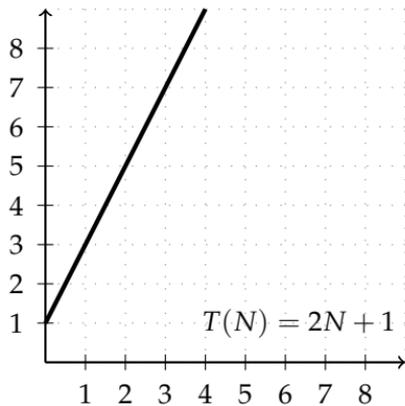
## 4. Time and Space Complexity

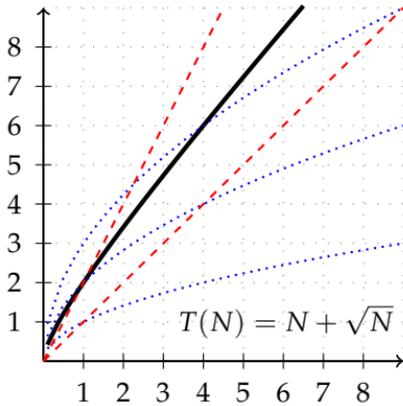
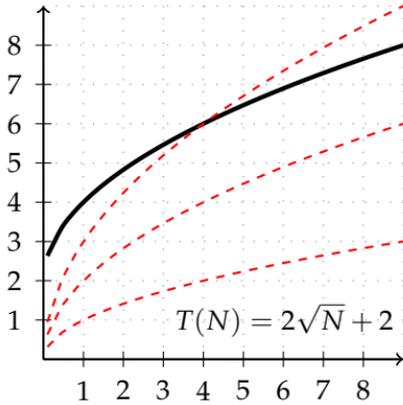
- We have been using high-level units such as the number of critical operations, stack frames, and heap variables
- We want to analyze algorithms and data-structures at an *even higher level* to broadly characterize *high-level trends* as some input variable (e.g.,  $N$ ) grows asymptotically large
- Big-O notation is a formal way to characterize *high-level trends*

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

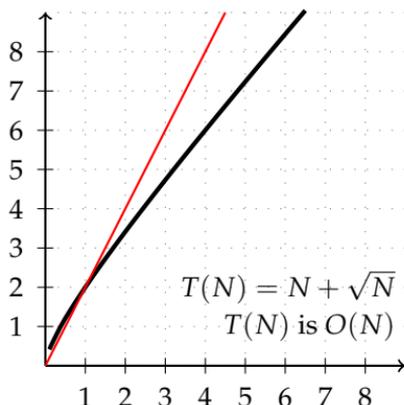
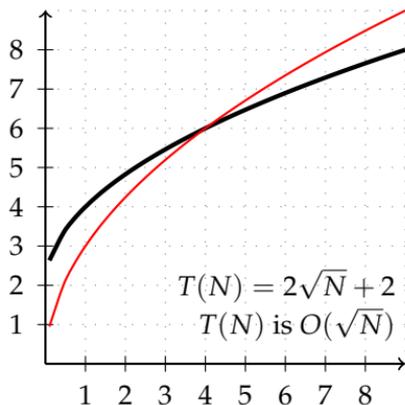
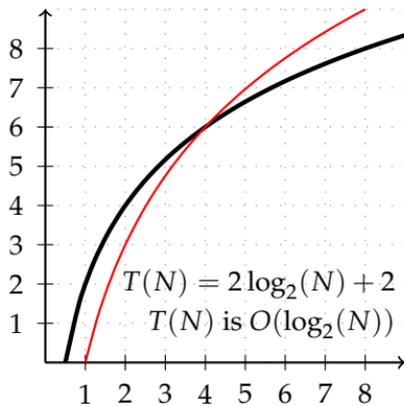
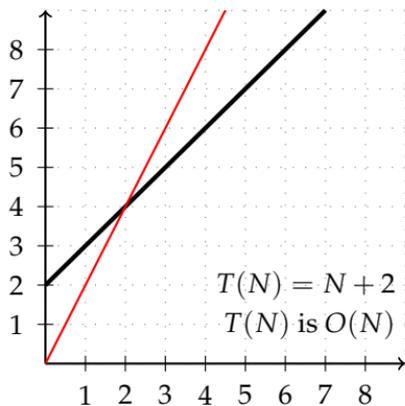
- $f(N)$  is  $O(g(N))$  if there is some value  $N_0$  and some value  $c$  such that for all  $N$  greater than  $N_0$ ,  $f(N) \leq c \cdot g(N)$ 
  - $g(N)$  can be thought of as an “upper bounding function”
  - $f(N)$  can be  $T(N)$  or  $S(N)$  (i.e., the “function of interest”)



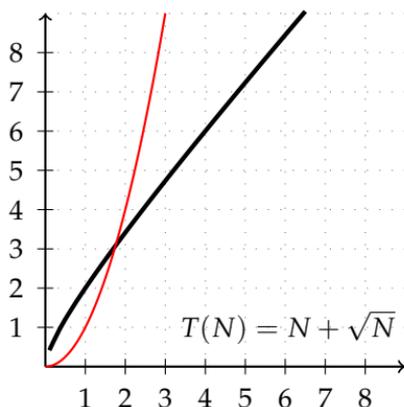
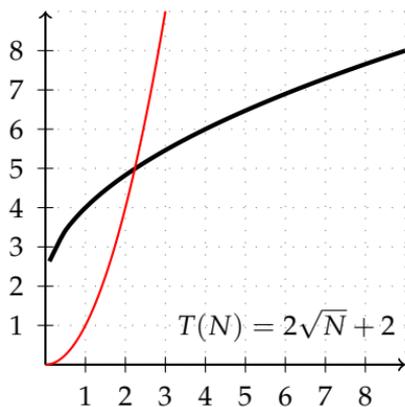
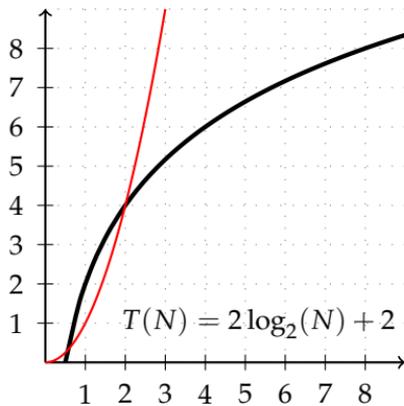
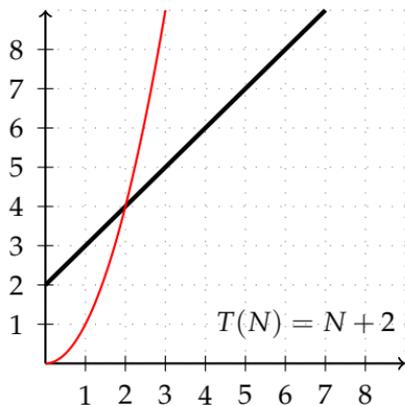




- Big-O notation captures the fastest-growing term (high-level trend) as  $N$  becomes asymptotically large
- With large enough  $c$ ,  $g(N)$  can ignore ...
  - ... constant factors in  $f(N)$
  - ... trailing terms in  $f(N)$



- 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



**Big-O Examples**

---

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

---

- Big-O notation captures the fastest-growing term (high-level trend) as  $N$  becomes asymptotically large
- Constant factors do not matter in big-O notation
- Trailing terms do not matter in big-O notation
- Base of log does not matter in big-O notation

**Big-O Classes**

---

	Class	N = 100 requires
$O(1)$	Constant	1 step
$O(\log(N))$	Logarithmic	2–7 steps
$O(\sqrt{N})$	Square Root	10 steps
$O(N^c)$ where $c < 1$	Fractional Power	
$O(N)$	Linear	100 steps
$O(N \cdot \log(N))$	Log-Linear	664 steps
$O(N^2)$	Quadratic	10K steps
$O(N^3)$	Cubic	1M steps
$O(N^c)$ where $c > 1$	Polynomial	
$O(2^N)$	Exponential	1e30 steps
$O(N!)$	Factorial	9e157 steps

---

- Exponential and factorial time algorithms are considered intractable
- With one nanosecond steps, exponential time requires many centuries and factorial time requires the lifetime of the universe

**Revisiting linear vs. binary search**

---

Linear	$T_w(N) = N$	is	$O(N)$	linear time
Binary	$T_w(N) = \log_2(N) + 1$	is	$O(\log(N))$	logarithmic time
Linear	$S_w(N) = 1$	is	$O(1)$	constant stack space
Binary	$S_w(N) = \log_2(N) + 2$	is	$O(\log(N))$	logarithmic stack space

---

- 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 trailing terms
- This analysis is for worst case complexity
  - results can look very different for best case complexity
  - results can look very different for average complexity
- For real-world problem sizes and/or different input data characteristics, sometimes an algorithm with worse time (space) complexity can still be faster (smaller)
- If two algorithms or data structures have the same complexity, then constants and trailing terms are what makes the difference!

## 4.1. Six-Step Process for Complexity Analysis

1. **Choose** units for execution time or space usage
  - critical multiplication, division, remainder operations
  - critical comparisons, swaps, node accesses, array accesses
  - critical function calls
  - critical loop iterations
  - stack frames, heap variables
2. **Choose** input variable and key parameters
  - Let  $N$  be a variable, we want to explore how time and space grow with  $N$
  - Let  $K$  be a parameter, we want to explore the interaction between  $N$  and  $K$   
( $K$  is constant w.r.t. to  $N$ ?,  $K$  is function of  $N$ ?, optimal  $K$ ?)
  - Let  $T_K(N)$  be execution time,  $N$  is input variable,  $K$  is key parameter
  - Let  $S_K(N)$  be space usage,  $N$  is input variable,  $K$  is key parameter
3. **Choose** kind of analysis
  - worst, average, or best case *input data value* analysis
  - must explain what is meant by worst, average, or best case input data!
  - usually focus on worst or average case, occasionally best case
  - worst/best case is never  $N = \text{large number}$  or  $N = 1$
  - we want worst/average/best case *function* of  $N$ , not value of  $N$
  - worst/average/best case can involve  $K$  (e.g., worst case is when  $K = N$ )
  - amortized analysis is over a *sequence* of operations
4. **Analyze** for specific values of input variable and key parameters
  - $T_8(10) = \dots$
  - $T_{32}(99) = \dots$
5. **Generalize** for any value of input variable and key parameters
  - $T_K(N) = \dots$
6. **Characterize** asymptotic behavior using big-O notation
  - $T_K(N) = \dots$  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?

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   for i in bvector's size to 0
3     set bvector's data[i] to data[i-1]
4   set bvector's data[0] to v
5   set bvector's size to size + 1
```

What is the time complexity?

What is the auxiliary heap space complexity?

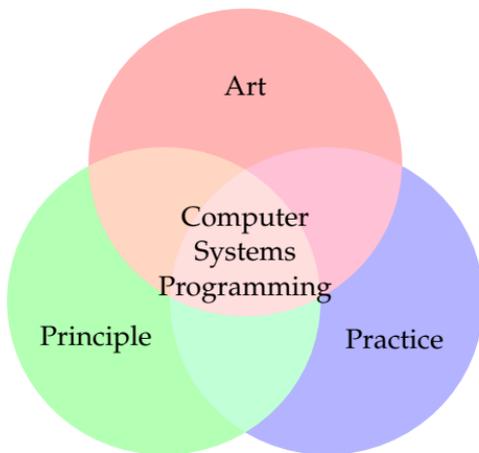
**Compare the time and space complexity of the algorithms**

<b>Operation</b>	<b>Time Complexity</b>		<b>Space Complexity</b>	
	slist	bvector	slist	bvector
construct				
destruct				
push_front				
pop_front				
push_back				
pop_back				
size				
at				
insert_idx				
remove_idx				
insert_ptr				
remove_ptr				

- What about comparing a doubly linked list or a resizable vector?
- What about the space complexity of the data structure itself?

## 6. Art, Principle, and Practice

- The **art** of computer systems programming is ...
  - designing new algorithms and/or data structures
  - choosing the right units that provide most insight
  - choosing the input variable and key parameters that provide most insight
  - choosing the kind of analysis that provides the most insight
- The **principle** of computer systems programming is ...
  - performing rigorous time and space complexity analysis
  - analyzing, generalizing, characterizing
  - considering modularity, hierarchy, encapsulation, extensibility
- The **practice** of computer systems programming is ...
  - implementing and integrating algorithms and/or data structures
  - performing real experiments to evaluate execution time and space usage
  - considering the constant factors and trailing terms



---

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

---