ECE 2400 Computer Systems Programming
Fall 2021

Topic 8: Complexity Analysis

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-14-04

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

1. Analyzing Simple Algorithms

```
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;
}
```

stack

- 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	
4	
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 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)

```
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

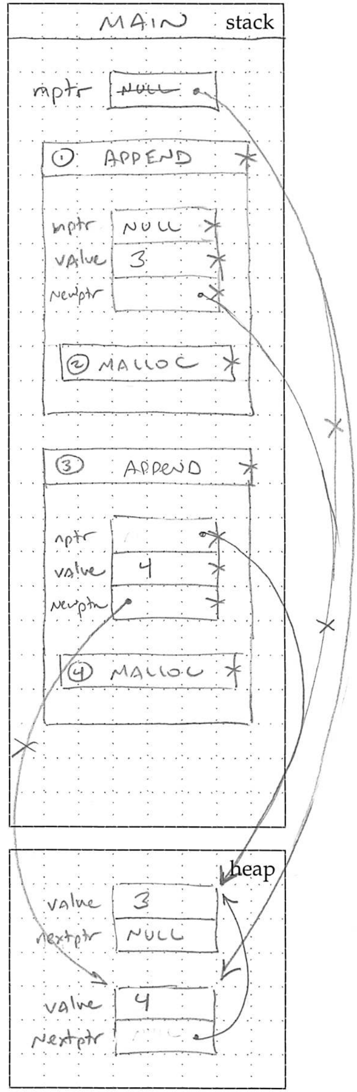
	v1		v2		v3	
x	itr	$T_{v1}(x)$	itr	$T_{v2}(x)$	itr	$T_{v3}(x)$
3						
4						
5						
6						
7						
8						
9						
10						
99						

```
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

```
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

- 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

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

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

  int* b = malloc( N*sizeof(int) );
  for ( int 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 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!
  - Auxillary 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

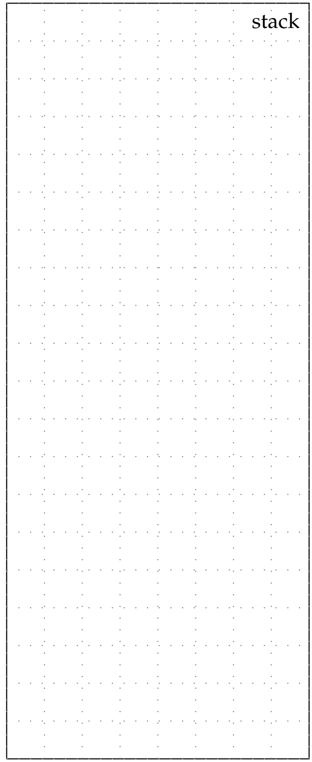
- Assume we have a sorted input array of integers
- Consider algorithms to check if a given value is in the array
- The algorithm should return 1 if value is in array, otherwise return 0

```
int 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

```
int lsearch( int* x, int n, int v )
{
  for ( int 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 b2 = lsearch( a, 8, 2 );
  int b0 = lsearch( a, 8, 0 );
  int b9 = lsearch( a, 8, 9 );
  return 0;
}
```

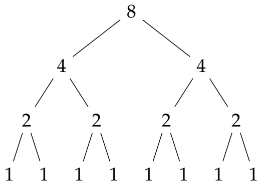


3.2. Binary Search

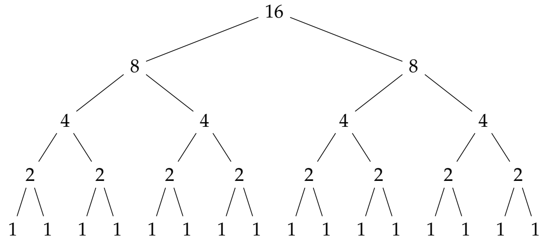
```
int bsearch_h( int* x, int lo,
               int hi, int v )
{
  int size = hi - lo;
  if ( size == 1 )
    return ( x[lo] == v );

  int mid = (lo + hi)/2;
  if ( v < x[mid] )
    return bsearch_h( x, lo, mid, v );
  else
    return bsearch_h( x, mid, hi, v );
}

int bsearch( int* x, int n, int v )
{
  return bsearch_h( x, 0, n, v );
}

int main( void )
{
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
  int b2 = bsearch( a, 8, 2 );
  int b0 = bsearch( a, 8, 0 );
  int b9 = bsearch( a, 8, 9 );
  return 0;
}
```

stack

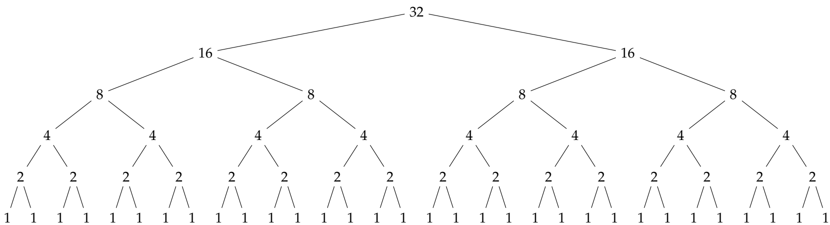
Annotating call tree with execution time and stack space usage



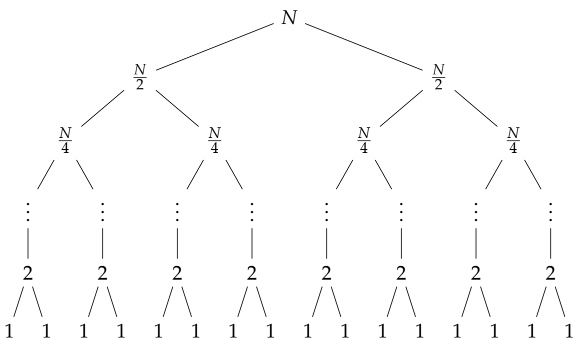
$$N = 8 = 2^3$$
$$k = 3$$



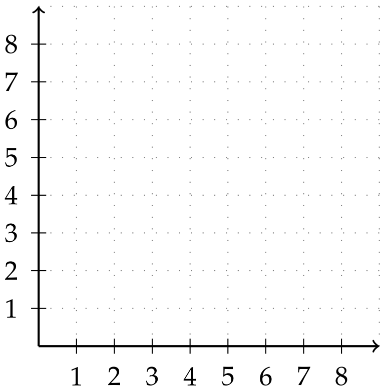
$$N = 16 = 2^4$$
$$k = 4$$



$$N = 32 = 2^5$$
$$k = 5$$



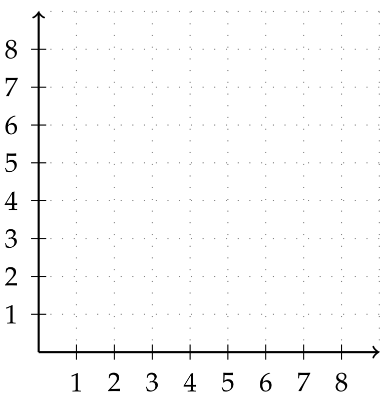
3.3. Comparing Linear vs. Binary Search



Linear: $T_w(N) = N$
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

- 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



Linear: $T_w(N) = N$
Binary: $T_w(N) = 2\ \log_2(N) + 2$

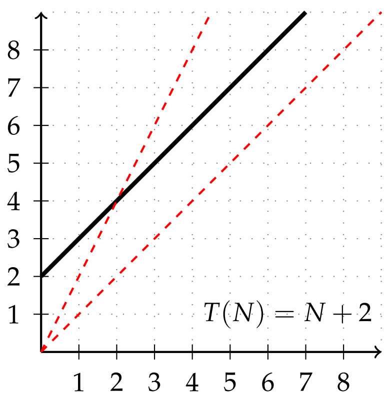
```
int bsearch_h( int* x, int lo,
               int hi, int v )
{
  int size = hi - lo;
  if ( size == 1 ) {
    // add extra 1 to trailing term?
    return ( x[lo] == v );
  }

  // add extra 2 to constant factor?
  int mid = (lo + hi)/2;
  if ( v < x[mid] )
    return bsearch_h( x, lo, mid, v );
  else
    return bsearch_h( x, mid, hi, v );
}
```

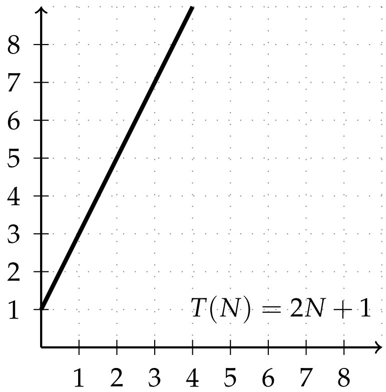
4. Time and Space Complexity

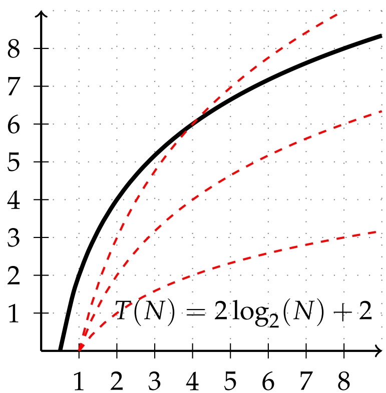
- We have been using high-level units such as the number of critical operations, stack frames, and heap variables
- We want to analyze algorithms and data-structures at an *even higher level* to broadly characterize *high-level trends* as some input variable (e.g., N) grows large
- Big-O notation is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

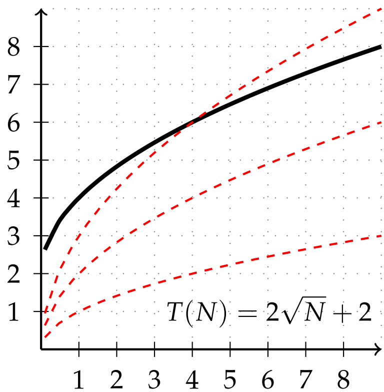
$$f(N) \text{ is } O(g(N)) \Leftrightarrow \exists N_0, c.\ \forall N > N_0.\ f(N) \leq c \cdot g(N)$$

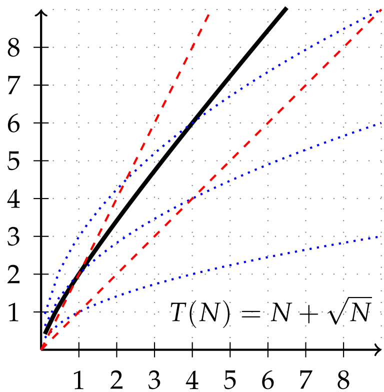
- $f(N)$ is $O(g(N))$ if there is some value N_0 and some value c such that for all N greater than N_0, $f(N) \leq c \cdot g(N)$
  - $g(N)$ can be thought of as an "upper bounding function"
  - $f(N)$ can be $T(N)$ or $S(N)$



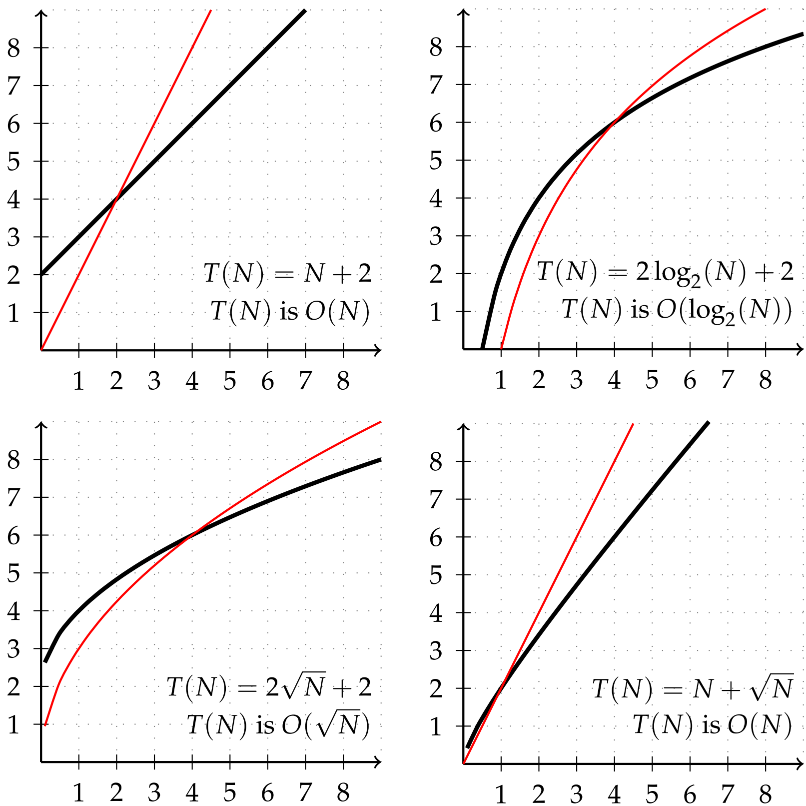

8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8
$T(N) = 2N + 1$


8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8
$T(N) = 2\log_2(N) + 2$

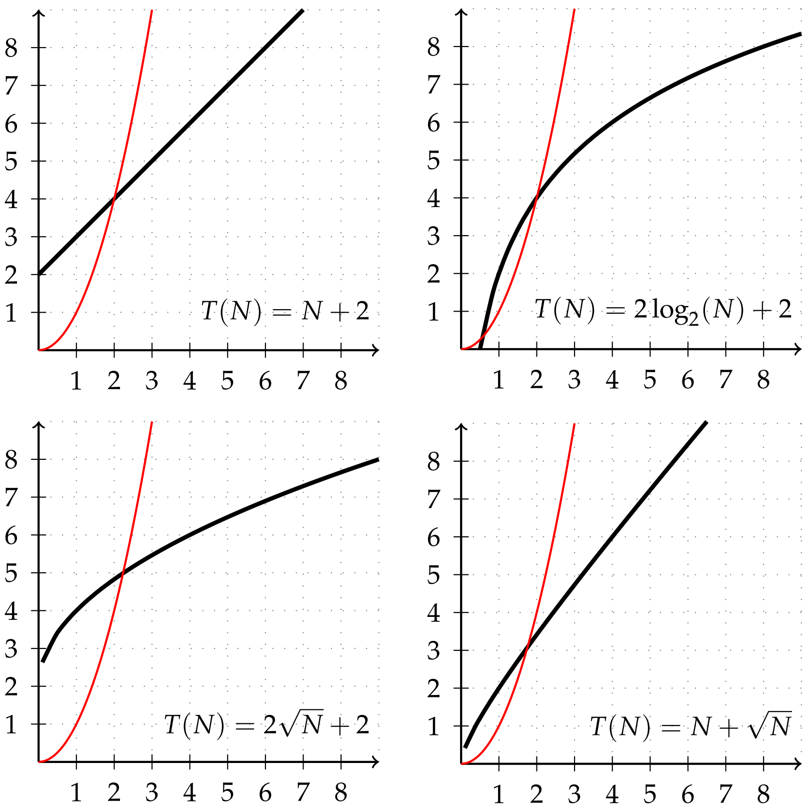

$T(N) = 2\sqrt{N} + 2$
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8


$T(N) = N + \sqrt{N}$
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

- 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 $f(N)$



- 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	$O(N^2)$
$4\ log_2(N)$	is	$O(log(N))$
$N+4\ log_2(N)$	is	$O(N)$

- 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

	Class	**N = 100 requires**
$O(1)$	Constant	1 step
$O(log(N))$	Logarithmic	6–7 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 would require many centuries and factorial time would require 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 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 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
   - critical multiplication, division, remainder operations
   - critical comparisons, swaps, array accesses, node accesses
   - critical function calls
   - iterations of a critical loop
   - 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 = 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
   - $T_8(10) = \ldots$
   - $T_{32}(99) = \ldots$
5. Generalize for any value of input variable and key parameters
   - $T_K(N) = \ldots$
6. Characterize asymptotic behavior using big-O notation
   - $T_K(N) = \ldots$ 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`

```
void slist_int_push_front( slist_int_t* this, 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
```

What is the time complexity?

What is the auxiliary heap space complexity?

Analysis of time and space complexity of `bvector_int_push_front`

```
void bvector_int_push_front( bvector_int_t* this, 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
```

What is the time complexity?

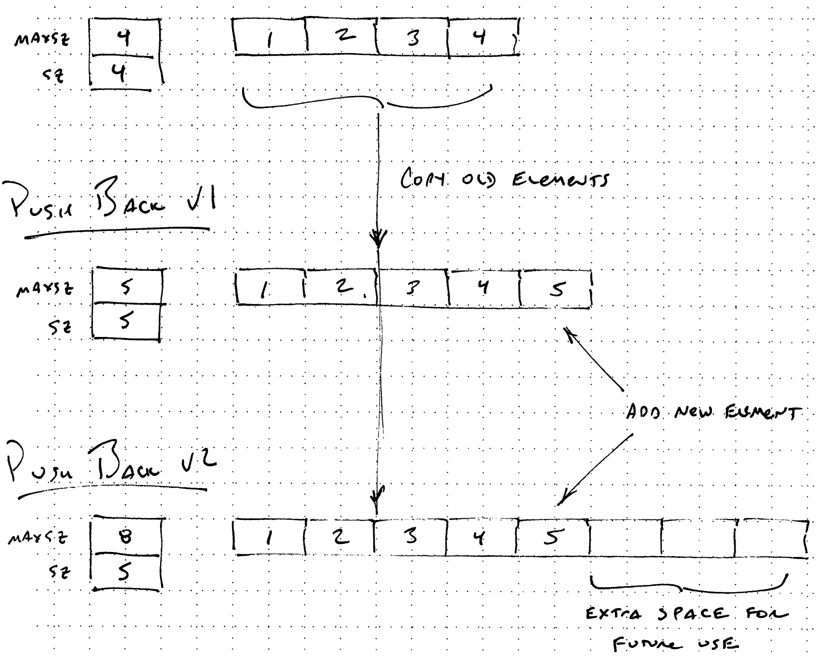
What is the auxiliary heap space complexity?

	Time Complexity		Space Complexity	
Operation	`slist`	`bvector`	`slist`	`bvector`
`push_front`				
`reverse`				
`push_back`				
`size`				
`at`				
`contains`				
`print`				

- 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

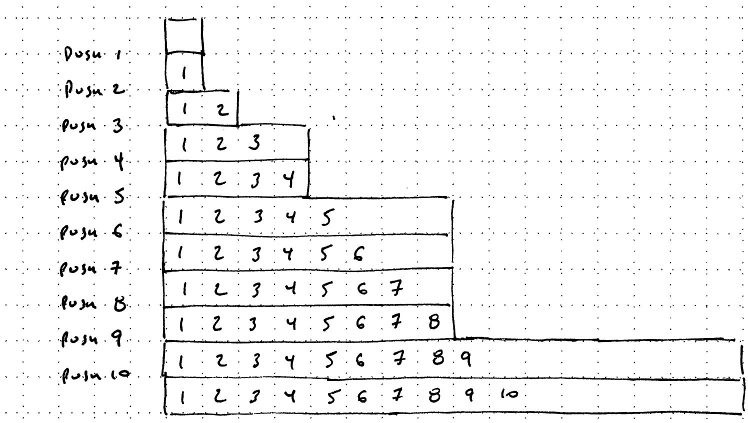
Amortized complexity analysis

- Complexity analysis in the previous table also applies to a doubly linked list and a resizable vector *except* for the `push_back` operation
- Recall two versions of `push_front` for a resizable vector discussed in the previous topic; let's consider two similar versions of `push_back`



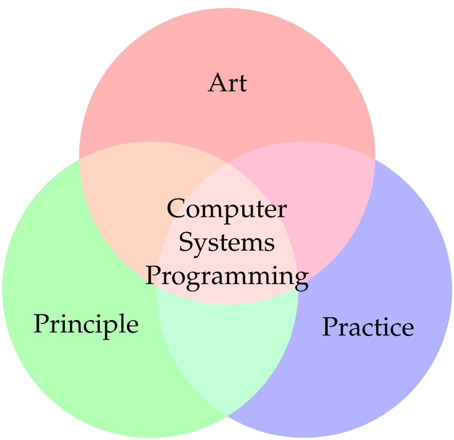
- `push_back_v1` always allocates a new array and then copies N elements so time complexity is always $O(N)$
- `push_back_v2` *sometimes* allocates a new array then copies N elements ($O(N)$), and *sometimes* just writes a single element ($O(1)$)

- So far all of our complexity analysis has been for a *single* execution of an algorithm or operation (e.g., time complexity for calling `push_back` once)
- Amortized complexity analysis considers the cost of a *sequence* of executions of an algorithm or operation (e.g., time complexity for calling `push_back` many times in a sequence)



6. Art, Principle, and Practice

- The art of computer systems programming is ...
  - developing the algorithm
  - 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
  - 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



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