CS 10: Problem solving via Object Oriented Programming

Runtime complexity

Part of the slides based on Dr. Kate Salesin slides

Main goals

- Characterize runtime complexity
  - Work for memory too
- Compare list implementations

Agenda

1. Run-time complexity
2. Asymptotic notation
3. List analysis

Find an element in the list (unsorted)

27	11	3	42	18	7	15

Find an element in the list (unsorted)

27	11	3	42	18	7	15

Search algorithm (linear search):

```
for idx i = 0 ... n–1
    if list.get(i) equals target
        return I
return -1 (not found)
```

Best case?

Worst case?

Find an element in the list (sorted)

3	7	11	15	18	27	42

Find an element in the list (sorted)

3	7	11	15	18	27	42

Search algorithm (binary search):

```
min = 0
max = n–1
while min <= max
    idx = (min + max) / 2
    if list.get(i) equals target
        return idx
    if list.get(i) > target
        max = idx–1
    else
        min = idx +1
```

Best case?

Worst case?

Find the max element in the list (unsorted)

27	11	3	42	18	7	15

Max algorithm:

```
max_value = list.get(0)
for idx i = 1 ... n–1
    if list.get(i) > max_value
        max_value = list.get(i)
return max_value
```

Best case?

Worst case?

Find the max element in the list (sorted)

3	7	11	15	18	27	42

Max algorithm

return list.get(0)

Best case?

Worst case?

Comparing algorithms

Search algorithm (linear search):

```
for idx i = 0 ... n–1
    if list.get(i) equals target
        return I
return -1 (not found)
```

Search algorithm (binary search):

```
min = 0
max = n–1
while min <= max
        idx = (min + max) / 2
        if list.get(i) equals target
                return idx
        if list.get(i) > target
                max = idx–1
        else
                min = idx +1
```

Comparing algorithms

Search algorithm (linear search):

```
for idx i = 0 ... n–1
    if list.get(i) equals target
        return i
return -1 (not found)
```

Worst case: check every item in the list

Search algorithm (binary search):

```
min = 0
max = n–1
while min <= max
        idx = (min + max) / 2
        if list.get(i) equals target
                return idx
        if list.get(i) > target
                max = idx–1
        else
                min = idx +1
```

Worst case: check half of the items in the list (though more operations)

Comparing algorithms

Search algorithm (linear search):

```
for idx i = 0 ... n–1
    if list.get(i) equals target
        return i
return -1 (not found)
```

Best case, immediately found

Max algorithm:

```
max_value = list.get(0)
for idx i = 1 ... n–1
    if list.get(i) > max_value
        max_value = list.get(i)
return max_value
```

No best or worst case, need to check entire list

How to compare efficiency?

Ideas:

– Time tests? But depends on specific hardware...

– Count # of operations? But will vary depending on size of list, and depends on exact implementation details...

– Do we compare best case, worst case, or average case?

How to compare efficiency?

- Key idea: typically in computer science, we measure how the # of operations grows with respect to the size of the input in the worst case
- “Size” is not always the length of a list or array, could also be the # characters in a string (as in SA-3), the # nodes in a binary tree, etc.

How to count operations?

- Examples of a single “operation”:
  - Assign value to variable, x = 1
  - Access element in array, x = array[i]
  - Perform arithmetic operation, x = y + z, x++, etc.
  - Compare two values, x == y, x < y, etc.
- In general, a single CPU computation or single read/write from memory

Example: Calculate sum of numbers from 1 to n

Algorithm 1:

1+2+3+…+n

```
int sum(int n) {
    int total = 0
    for (int i = 1; i <= sum; i++) {
        total += i;
    }
    return total;
}
```

Example: Calculate sum of numbers from 1 to n

Algorithm 1:

1+2+3+…+n

```
int sum(int n) {
        int total = 0
        for (int i = 1; i <= sum; i++) {
                total += i;
        }
        return total;
}
```

operations

1

1; n+1; n

n

1

Total= 3n+4 operations

Example: Calculate sum of numbers from 1 to n

Algorithm 2:

1+(1+1)+(1+1+1)+…+(1+1+…+1)

```
int sum(int n) {
        int total = 0
        for (int i = 1; i <= sum; i++) {
                for (int j = 1; j <= i; j++) {
                        total += 1;
                }
        }
        return total;
}
```

Example: Calculate sum of numbers from 1 to n

Algorithm 2:

1+(1+1)+(1+1+1)+…+(1+1+…+1)

`int sum(int n) {`	# operations
`    int total = 0`	1
`    for (int i = 1; i <= sum; i++) {`	1; n+1; n
`        for (int j = 1; j <= i; j++) {`	n; (n+1)*(n+1)/2; n*(n+1)/2
`            total += 1;`	n*(n+1)/2
`        }`	
`    }`	
`    return total;`	1
`}`	Total= $3/2n^2+11/2n+4$ operations

Example: Calculate sum of numbers from 1 to n

Algorithm 3: n*(n+1)/2

```
int sum(int n) {
	return n * (n+1) /2;
}
```

operations

4

Total= 4 operations

Example: Calculate sum of numbers from 1 to n

Algorithm 1:

1+2+3+…+n

Algorithm 2:

1+(1+1)+(1+1+1)+…+(1+1+…+1)

Algorithm 3:

n*(n+1)/2

3n+4

$3/2n^2+11/2n+4$

4

Example: Calculate sum of numbers from 1 to n

Algorithm 1:

$1+2+3+\ldots+n$

Algorithm 2:

$1+(1+1)+(1+1+1)+\ldots+(1+1+\ldots+1)$

Algorithm 3:

$n*(n+1)/2$

$3n+4$ | $3/2n^2+11/2n+4$ | 4

$O(n)$ | $O(n^2)$ | $O(1)$

Big O notation

- Ignore lower-order terms – the biggest term will dominate as the input size n grows asymptotically
- Ignore constant factors – depends on implementation details, not the core algorithm

Often run-time will depend on the number of elements an algorithm must process

Constant time O(1) – does not depend on number of items

e.g., return first element of a list

Linear time O(n) – directly depends on number of items

e.g., find value in a list

Polynomial time O(n^k) – depends on a polynomial function of number of items

e.g., nested loop in an image

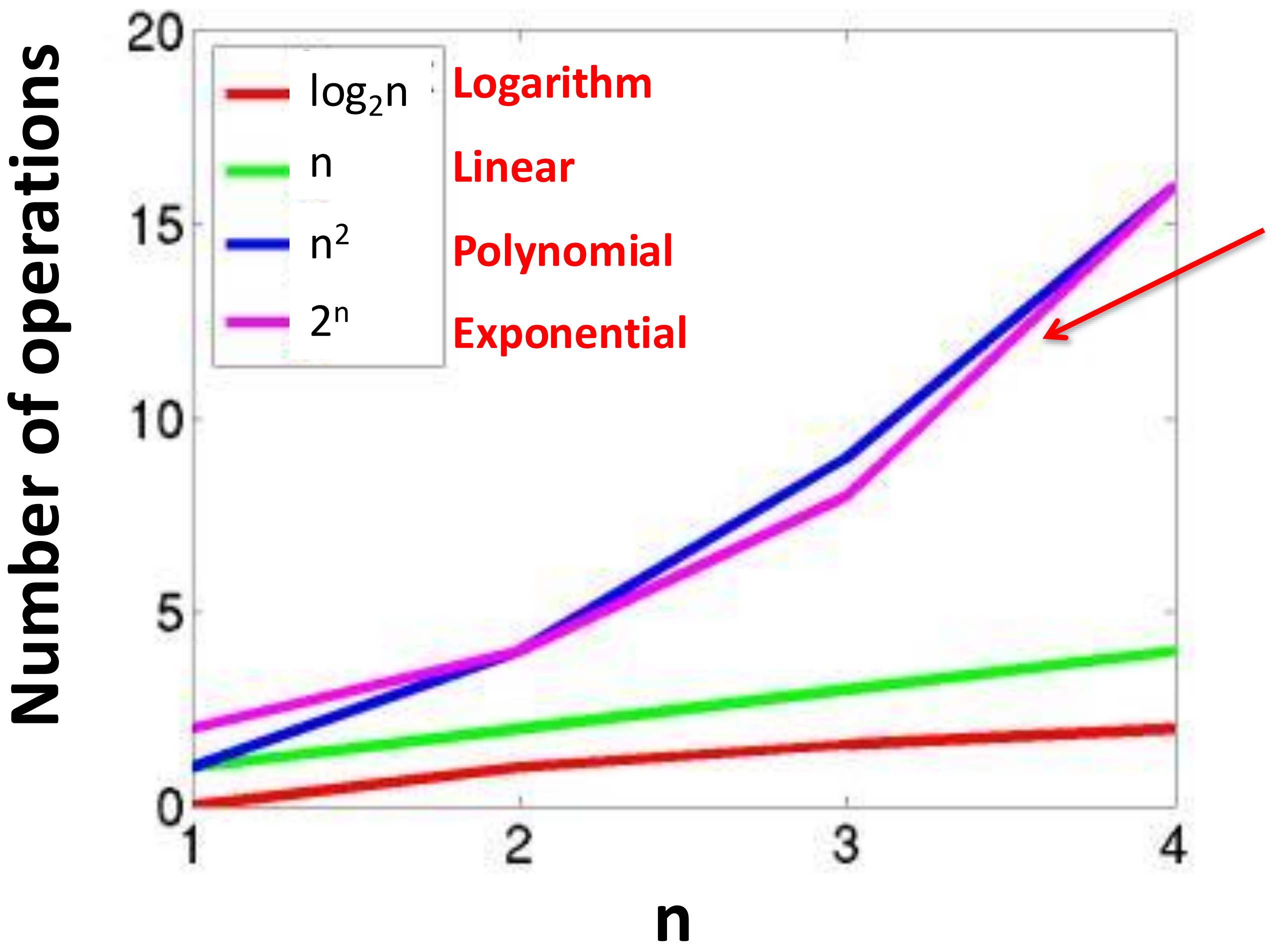
Logarithm time O(log n)– avoids operations on some items

e.g., binary search

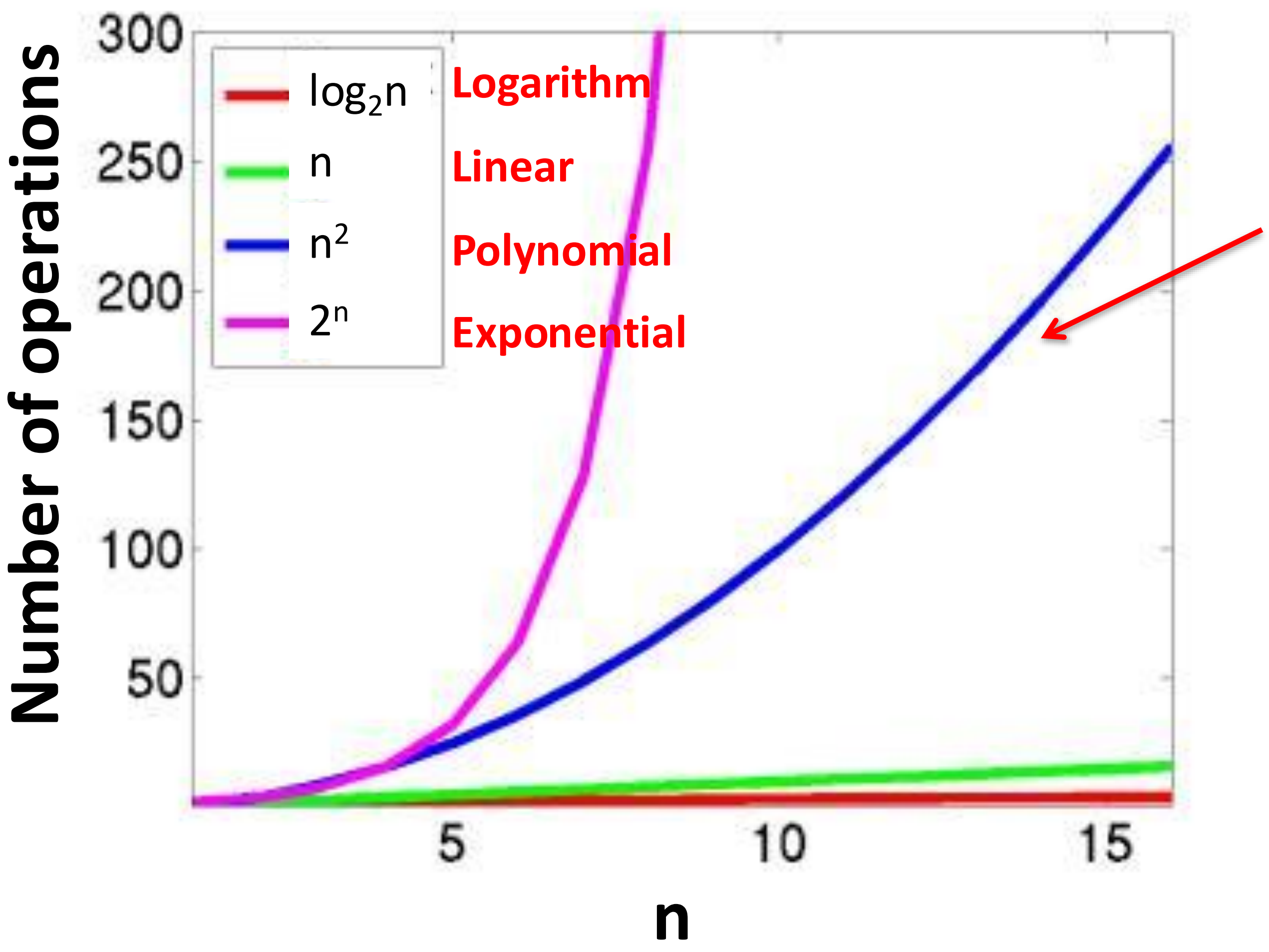
Exponential time O(2^n)– base raised to power

e.g., all possible bit combinations in n bits

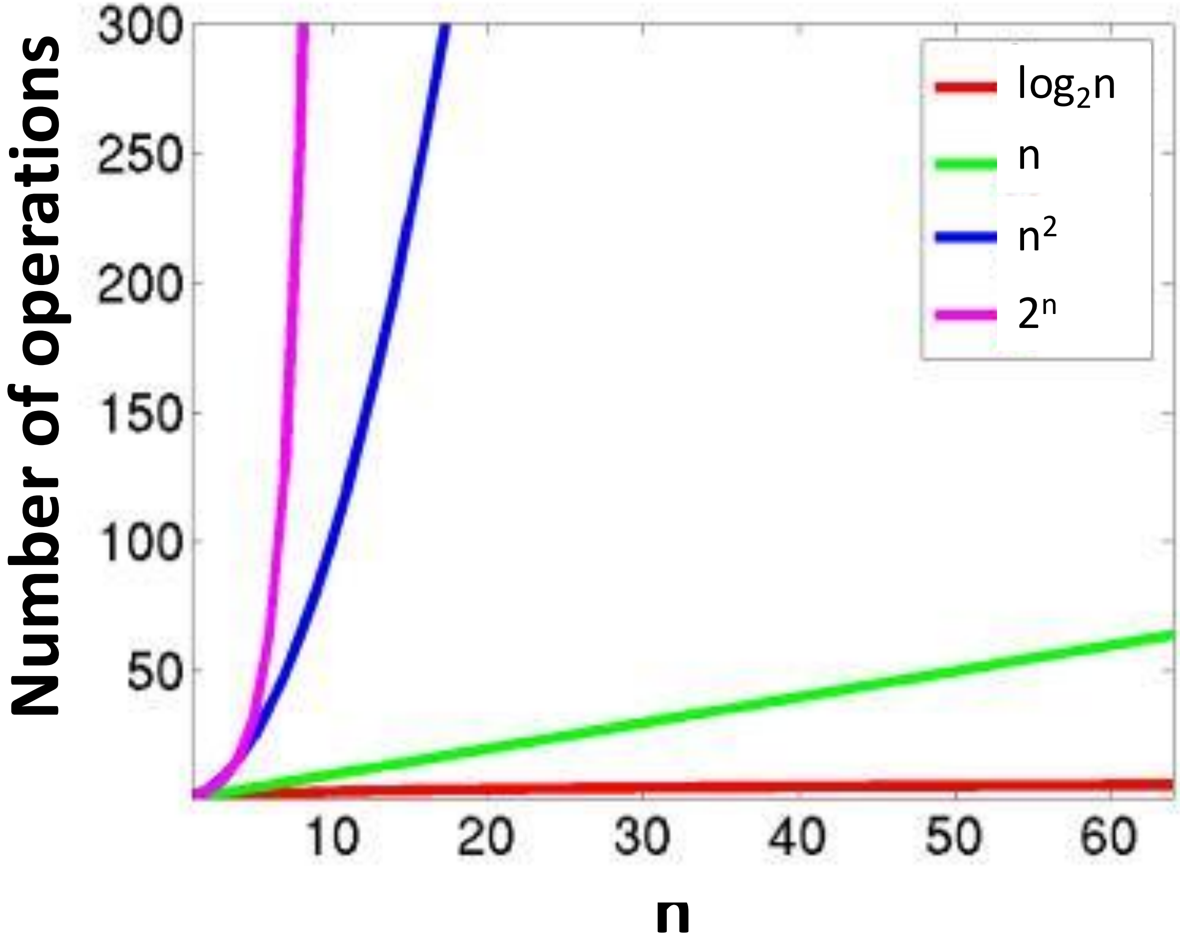
For small numbers of items, run time does not differ by much



As *n* grows, number of operations between different algorithms begins to diverge



Even with only 60 items, there is a large difference in number of operations



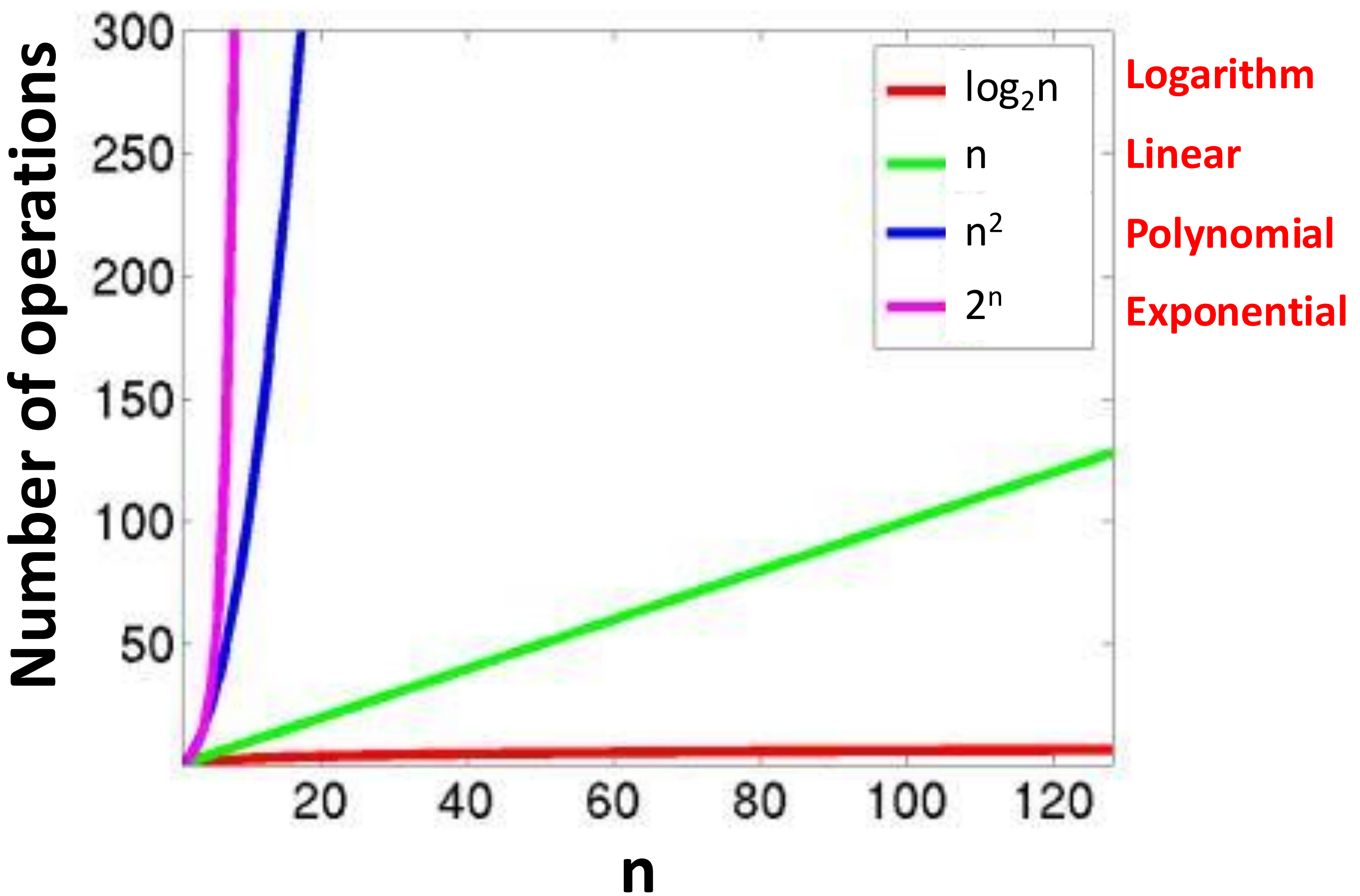
Logarithm

Linear

Polynomial

Exponential

Eventually, even with speedy computers, some algorithms become impractical



Sometimes complexity can hurt us, sometimes it can help us

Hurts us
Can't brute force chess algorithm 2^n

Helps us
Can't crack password algorithm 2^n

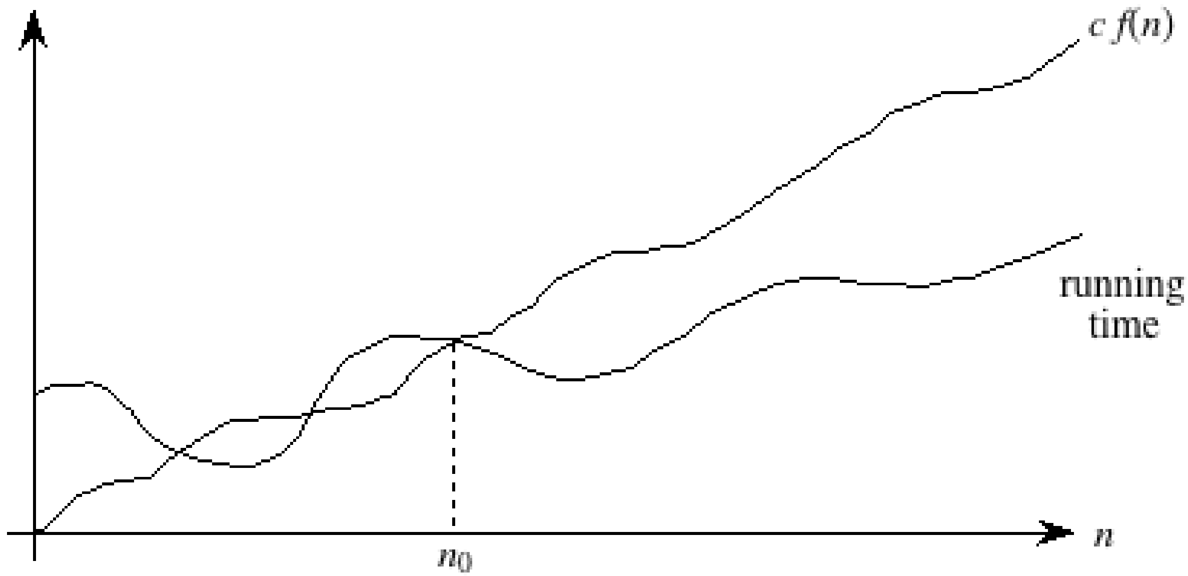
Images: thechessstore.com; studyoffice.org

Agenda

1. Run-time complexity
2. Asymptotic notation
3. List analysis

We can extend Big Oh to any, not necessarily linear, function

O gives an asymptotic <u>upper</u> bounds

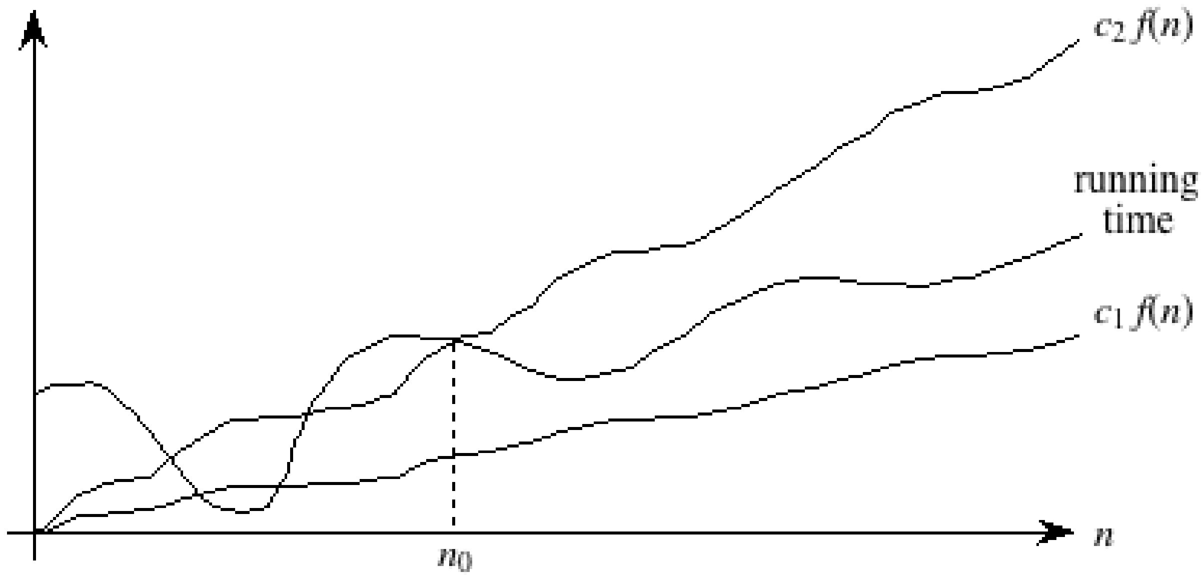


Run-time complexity is O(*f(n)*) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at ***most*** *cf(n)*, upper bound
- O(*f(n)*) is the ***worst*** case performance for large n, but actual performance could be better

Run time can also be Ω (Big Omega), where run time grows at least as fast

Ω gives an asymptotic <u>lower</u> bounds



Run-time complexity is $\Omega(f(n))$ if there exists constants n_0 and c_1 such that:

- $\forall n \geq n_0$
- run time of size n is at *<u>least</u>* $c_1 f(n)$, lower bound
- $\Omega(n)$ is the *<u>best</u>* case performance for large n, but actual performance can be worse

Comparison

Example: find *<u>specific</u>* item in a list

Example: find *<u>largest</u>* item in a list

Comparison

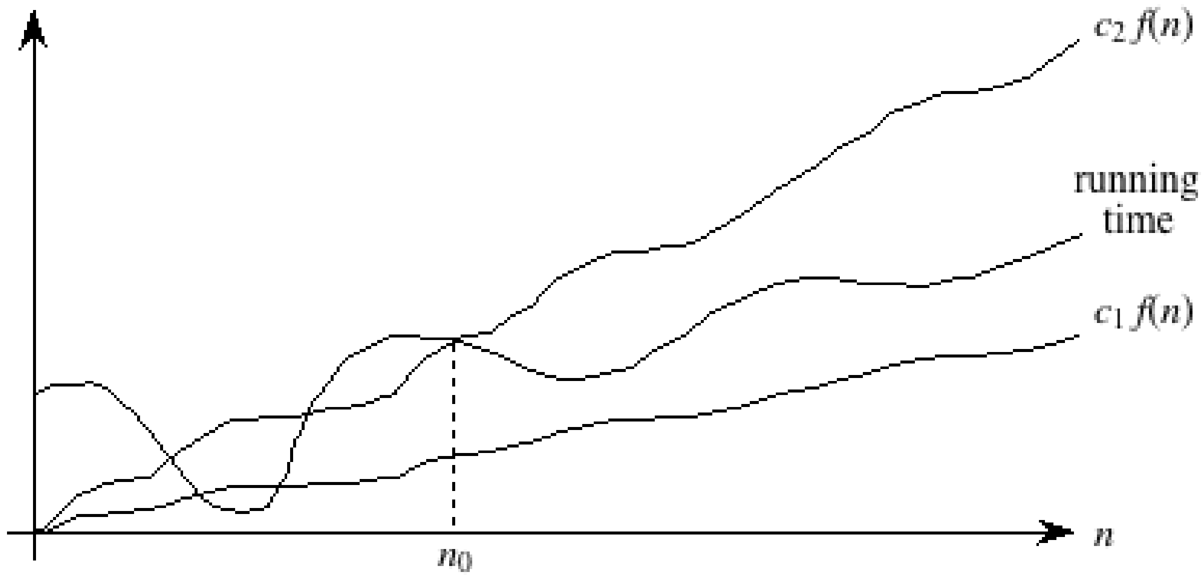
Example: find *specific* item in a list
- **Might find item on first try**
- **Might not find it at all (must check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

Example: find *largest* item in a list
- **Must check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

We use Θ (Big Theta) for tight bounds when we can define O and Ω

Θ gives an asymptotic tight bounds

If an algorithm is O(f(n)) and Ω(f(n)), then it is Θ(f(n))



Run-time complexity is $\Theta(f(n))$ if there exists constants n_0 and c_1 and c_2 such that:

- $\forall n \geq n_0$
- run time of size n is at <u>*least*</u> $c_1 f(n)$ and at <u>most</u> $c_2 f(n)$
- $\Theta(n)$ gives a tight bound, which means run time will be within a constant factor
- Generally we will use either O or Θ

Comparison: which has a tight bound?

Example: find _specific**_ item in a list**

- **Might find item on first try**
- **Might not find it at all (must check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

Example: find _largest**_ item in a list**

- **Must check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

Comparison: which has a tight bound?

Example: find *specific* item in a list
- **Might find item on first try**
- **Might not find it at all (must check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

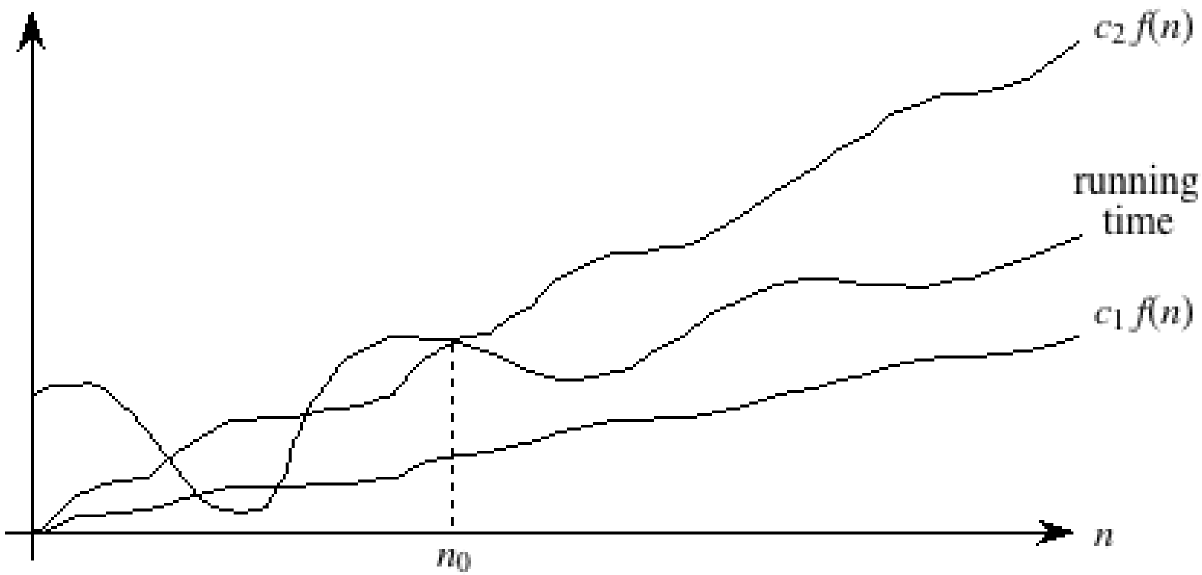
Example: find *largest* item in a list
- **Must check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

- **Worst case: must check each item, so O(n)**
- **Because Ω(n) *and* O(n) we say it is Θ(n)**

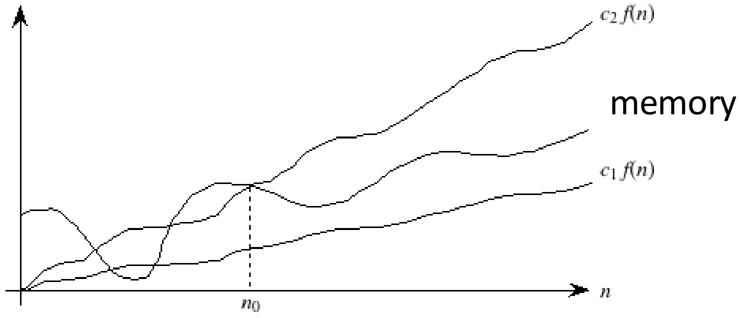
We ignore constants and low-order terms in asymptotic notation

Constants don’t matter, just adjust c_1 and c_2

Low order terms don’t matter either



These concepts are applicable for memory complexity as well



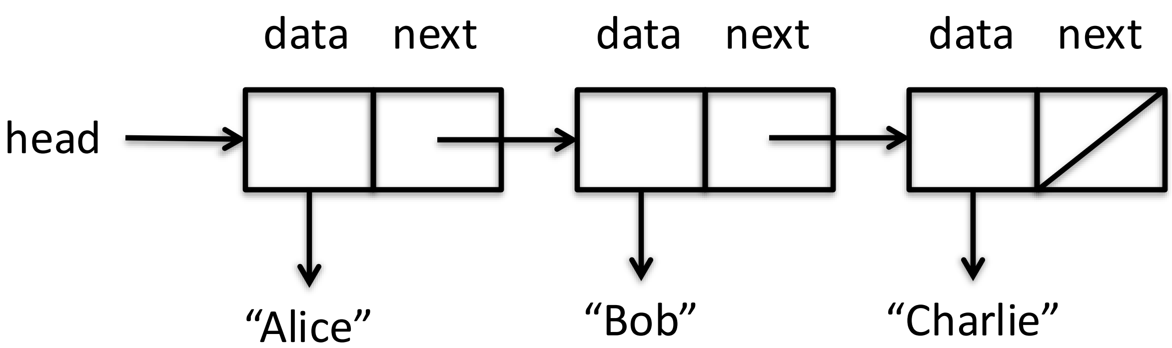
Agenda

1. Run-time complexity
2. Asymptotic notation
3. List analysis

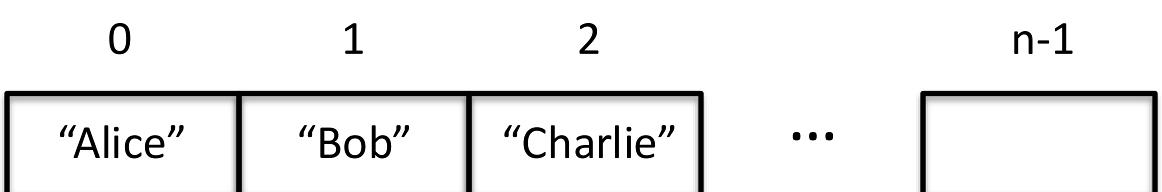
Difference between singly linked list and array

List ADT features
get()/set() element anywhere in List
add()/remove() element anywhere in List
No limit to number of elements in List

Singly linked list



Array



Growing array is *generally* preferable to linked list, except maybe growth operation

Worst case run-time complexity

	Linked list	Growing array
get(i)		
set(i,e)		
add(i,e)		
remove(i)		

Discussion

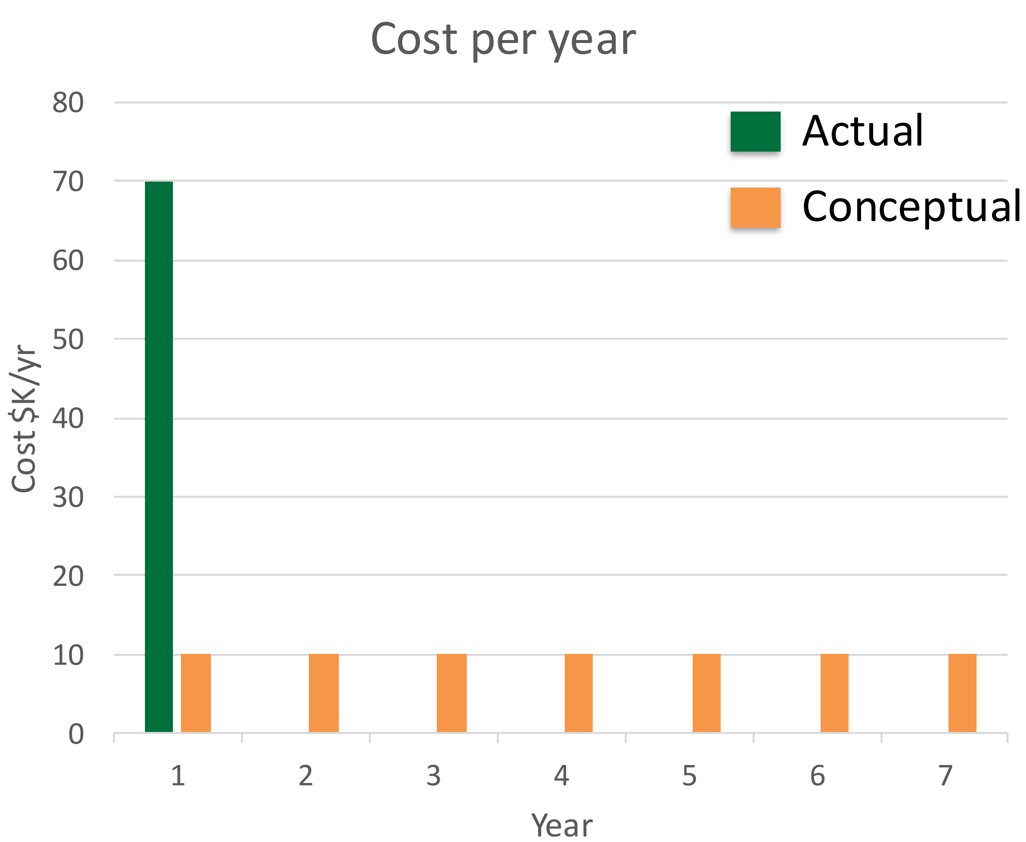
Growing array is *generally* preferable to linked list, except maybe growth operation

Worst case run-time complexity

	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	O(n) + growth
remove(i)	O(n)	O(n)

Amortization is a concept from accounting that allows us to spread costs over time

Amortized analysis

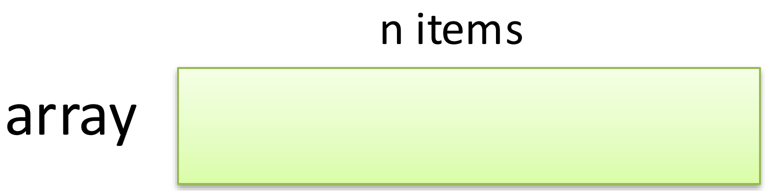


Accounting allows us to amortize costs over several years

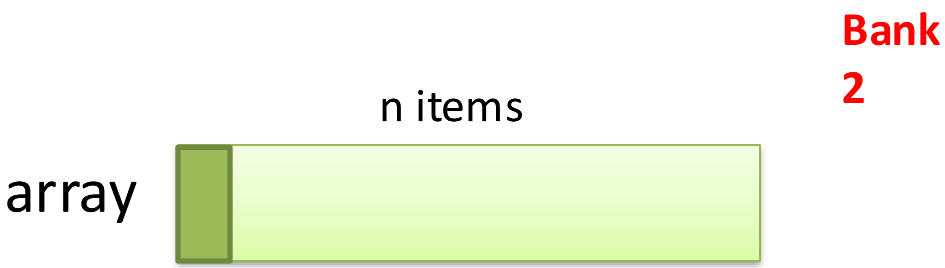
- Buy $70K truck on year 1
- Truck is good for 7 years

Amortized analysis shows growing array is actually only O(1)!

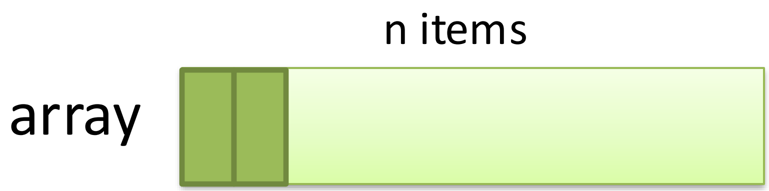
Amortized analysis



Amortized analysis shows growing array is actually only O(1)!

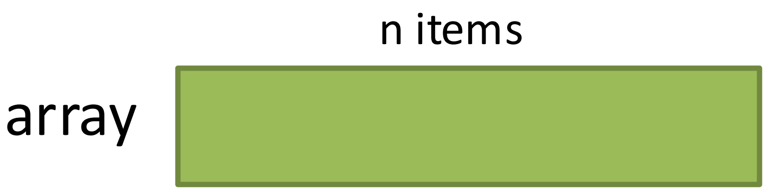


Amortized analysis shows growing array is actually only O(1)!



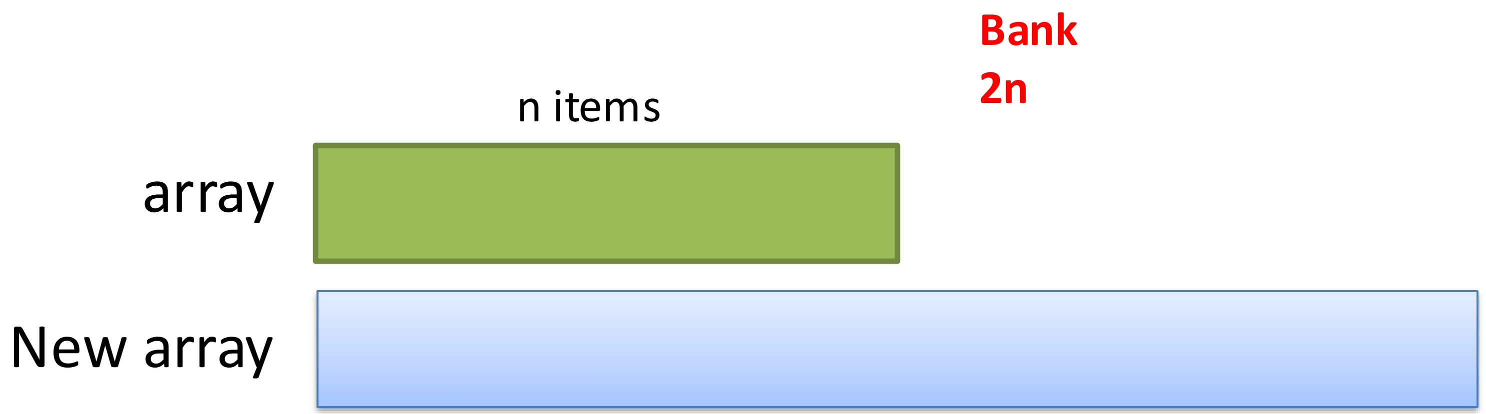
Bank
4

Amortized analysis shows growing array is actually only O(1)!

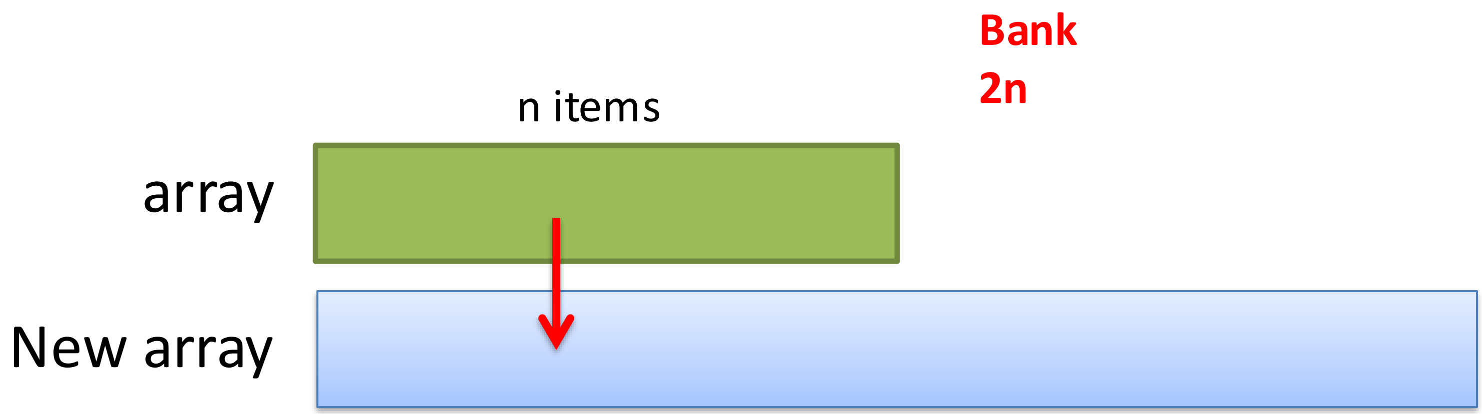


Bank
2n

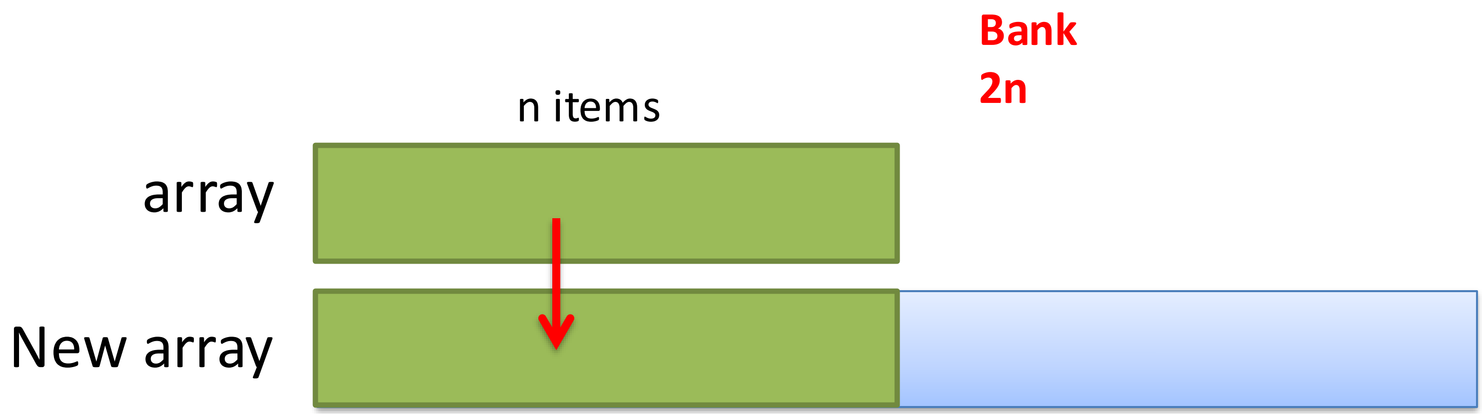
Amortized analysis shows growing array is actually only O(1)!



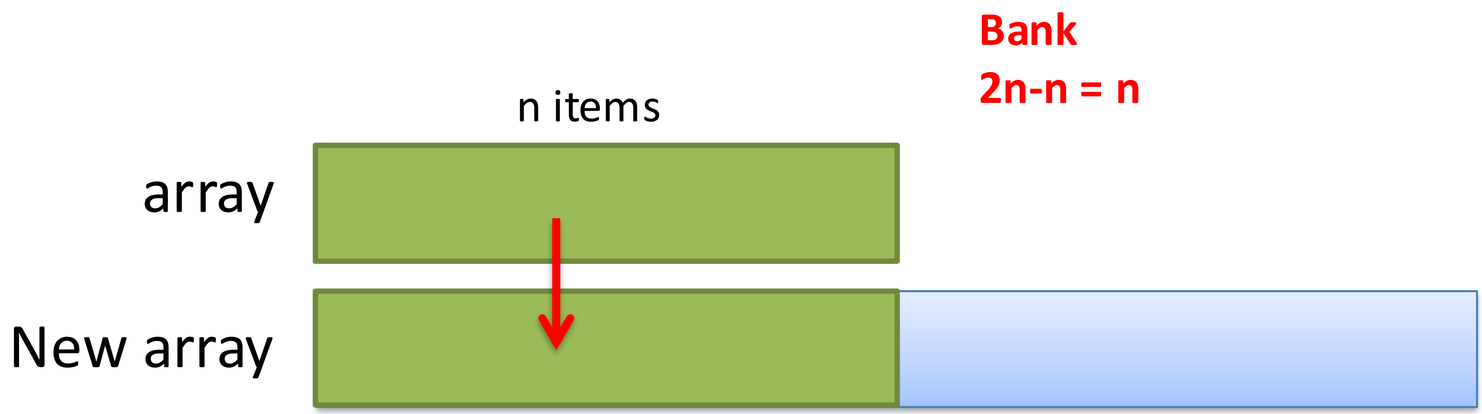
Amortized analysis shows growing array is actually only O(1)!



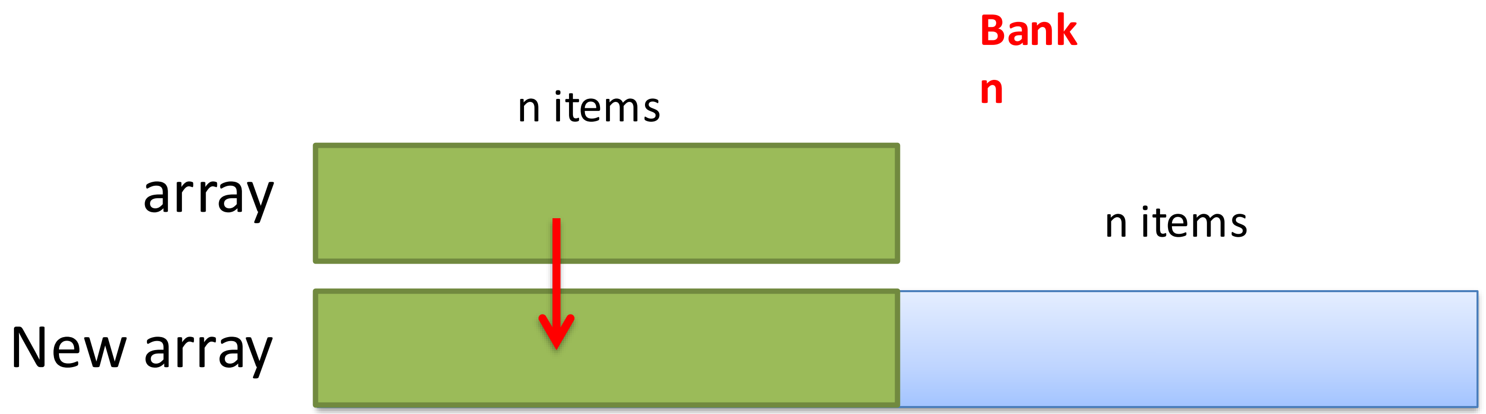
Amortized analysis shows growing array is actually only O(1)!



Amortized analysis shows growing array is actually only O(1)!



Amortized analysis shows growing array is actually only O(1)!



Growing array is *generally* preferable to linked list

Worst case run-time complexity

	Linked list	**Growing array**
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	**O(n) + O(1) = O(n)**
remove(i)	O(n)	O(n)

Summary

- Runtime and memory complexity analysis
  - Asymptotic notation
    - O(1) constant, O(log(n)) logarithm, O(n) linear, O(n^2) polynomial, O(2^n) exponential
- Runtime complexity analysis
  - Get/set O(1)
  - Add/remove O(n)
    - Amortized analysis for growth operation
- List analysis: SinglyLinkedList vs ArrayList
  - Growing array overall more efficient, unless specific assumptions on operations

Next

- Hierarchical relationships through trees

Additional Resources

RUN-TIME COMPLEXITY

How long does it take to find an item in a List?

Index	0	1	2	3	4		n-2	n-1
	Bob	Elvis	Abby	Denise	Charlie	…	Yancy	Zephyr

Assume there are *n* items in the List (index 0 … n-1)

Find index of "Paula" in List

What pseudo code would you use:

```
for i = 0 … n-1
    get item at index i
    if item is equal to search value
        return index i
return -1 (or otherwise indicate search term not in List)
```

How long to find the item? Should we time how long it takes?

Time would depend on

- Hardware
- Where Paula was located in the List

What is the best case?

What is the worst case?

What is the average case?

How long does it take to find an item in a List?

Index	0	1	2	3	4		n-2	n-1
	Bob	Elvis	Abby	Denise	Charlie	…	Yancy	Zephyr

Instead of timing execution we will ***count*** how many operations are needed in the ***worst case***

- Doesn't depend on hardware or software environment
- Could use average case, but average is hard to define sometimes because it would be based on the input's distribution
- Worst case tells us it won't take longer to execute
- Allows language-independent analysis based on number of elements

Operations to count

- Assign value to variable
- Following an object reference to heap memory
- Performing arithmetic operation (e.g., add two numbers)
- Compare two numbers (if statement)
- Access element in array
- Calling or returning from a method

Often run-time will depend on the number of elements an algorithm must process

Constant time – does not depend on number of items

- Returning the first element of a list takes a constant amount of time *irrespective* of the number of elements in the list
- Just return the first item
- No need to march down list to find the first element (*head*)
- Array *get()* implementation is also constant time (array *get()* is constant time everywhere, linked list only constant at *head*)

Linear time – directly depends on number of items

- Example: searching for a particular value stored in a list
- Start at first item, compare value with value trying to find
- Keep going until find item, or end up at end of list
- Could get lucky and find item right away, might not find it at all
- Worst case is we check all *n* items

Often run-time will depend on the number of elements an algorithm must process

Polynomial time – depends on a polynomial function of number of items

- Example: nested loop in image and graphic methods
- If changing all pixels in *n* by *n* image, must do a total of n^2 operations because inner and outer loops each run *n* times
- Normally runs slower than a constant or linear time algorithm

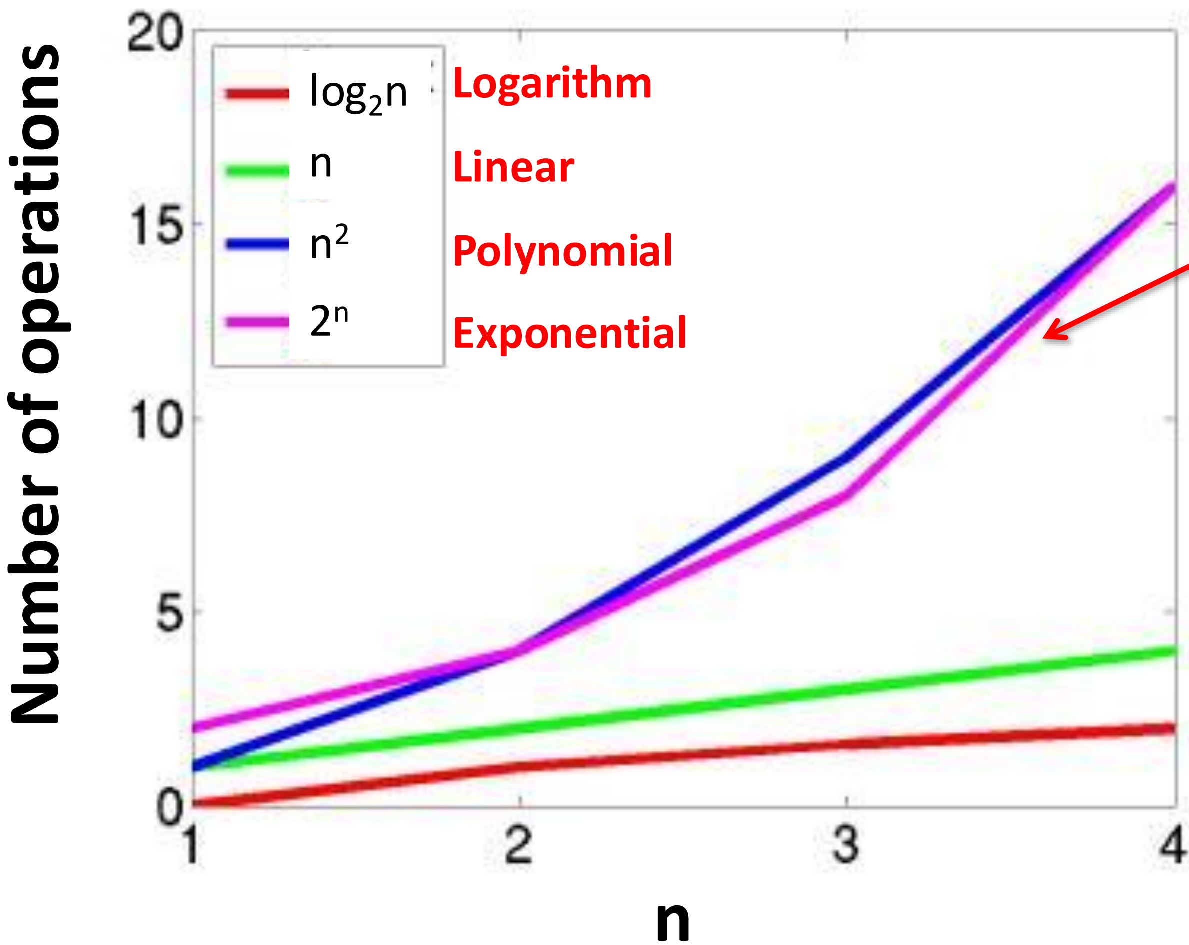
Logarithm time – avoids operations on some items

- Soon we will look at binary search
- Reduces the number of items algorithm must process (don't process all *n* items)
- Runs faster than linear or polynomial time (slower than constant)

Exponential time – base raised to power

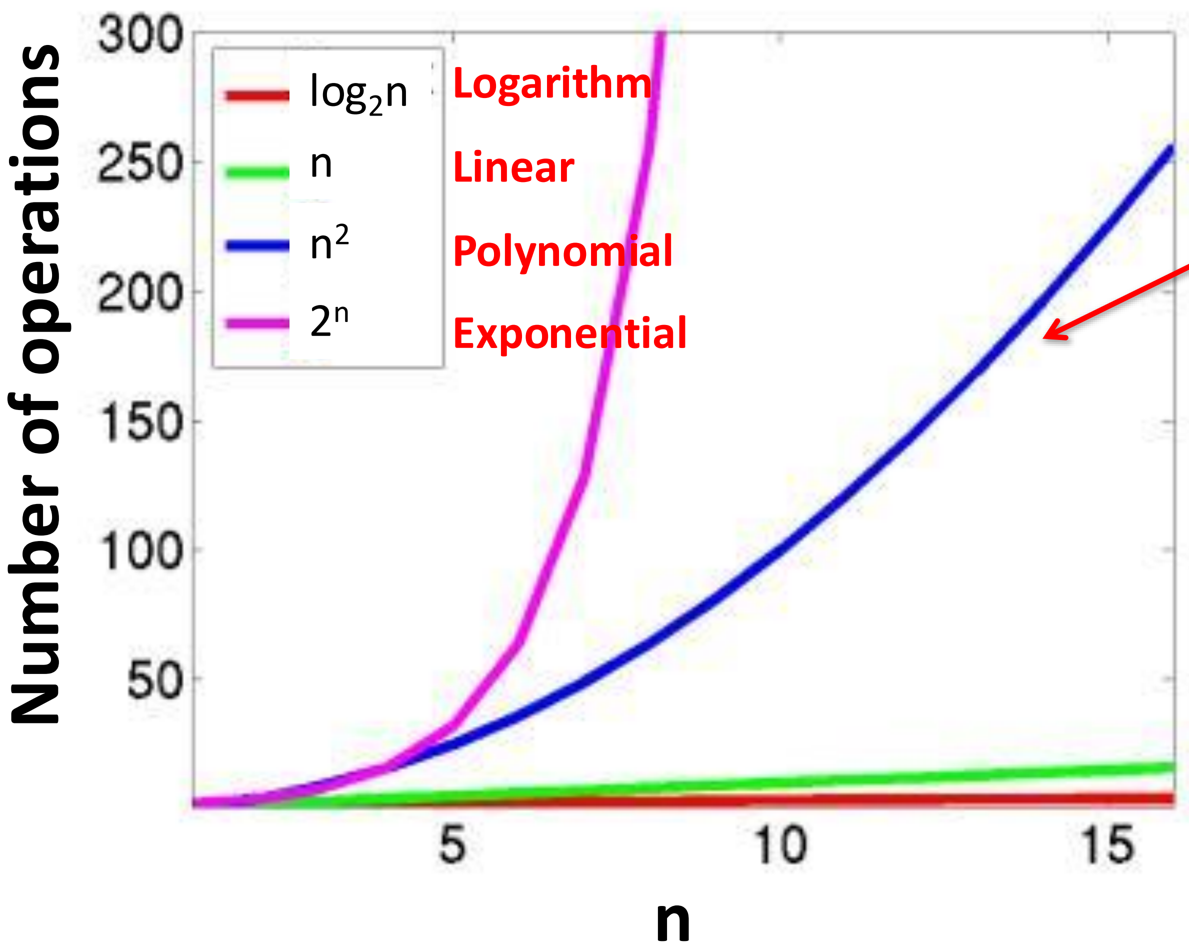
- Combination problems: all possible bit combinations in *n* bits = 2^n
- SLOW!

For small numbers of items, run time does not differ by much



Notice Exponential and Polynomial cross each other a few times early on

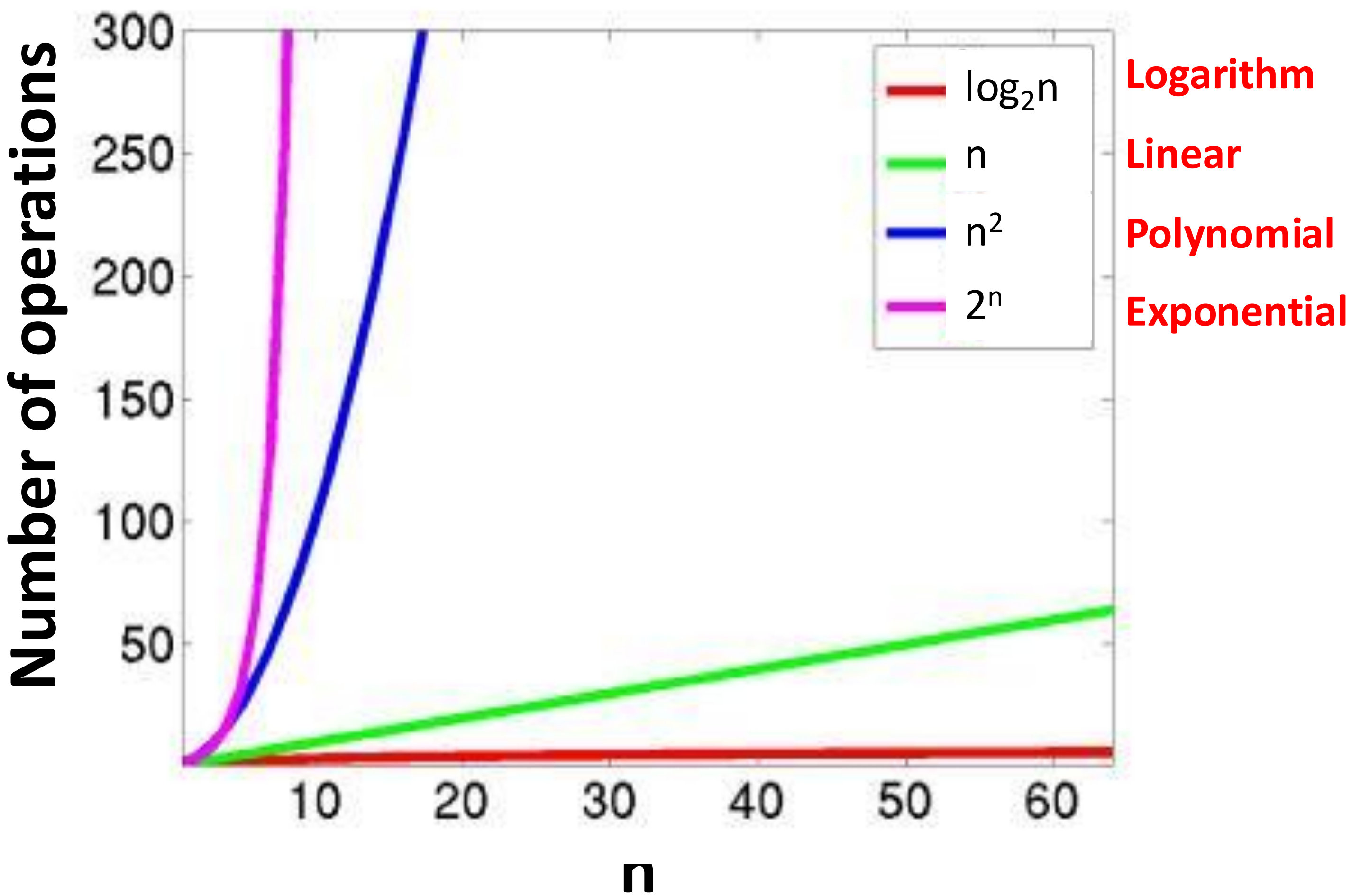
As *n* grows, number of operations between different algorithms begins to diverge



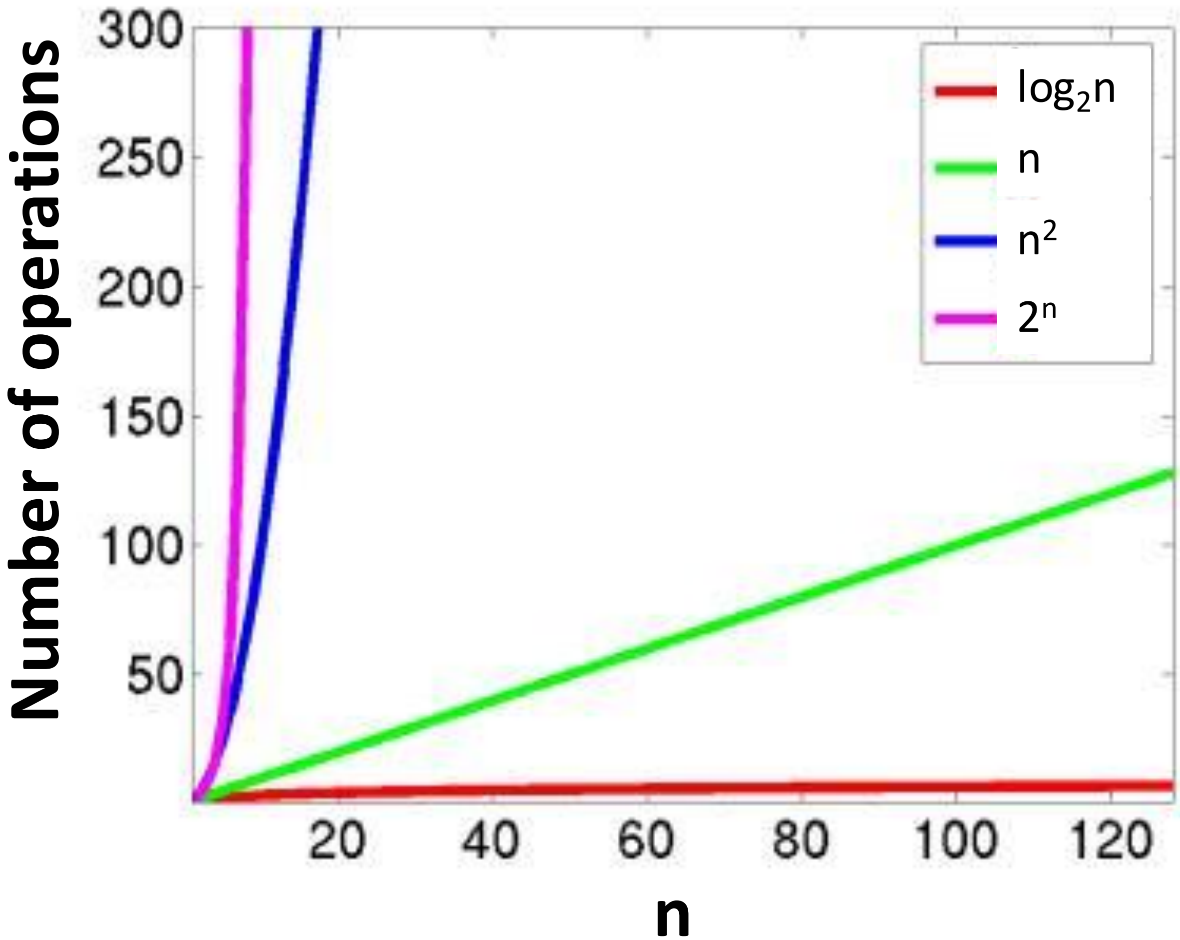
After n=4 Exponential is always greater than Polynomial

We will use that soon to define n_0 (standby for more info)

Even with only 60 items, there is a large difference in number of operations



Eventually, even with speedy computers, some algorithms become impractical



Logarithm

Linear

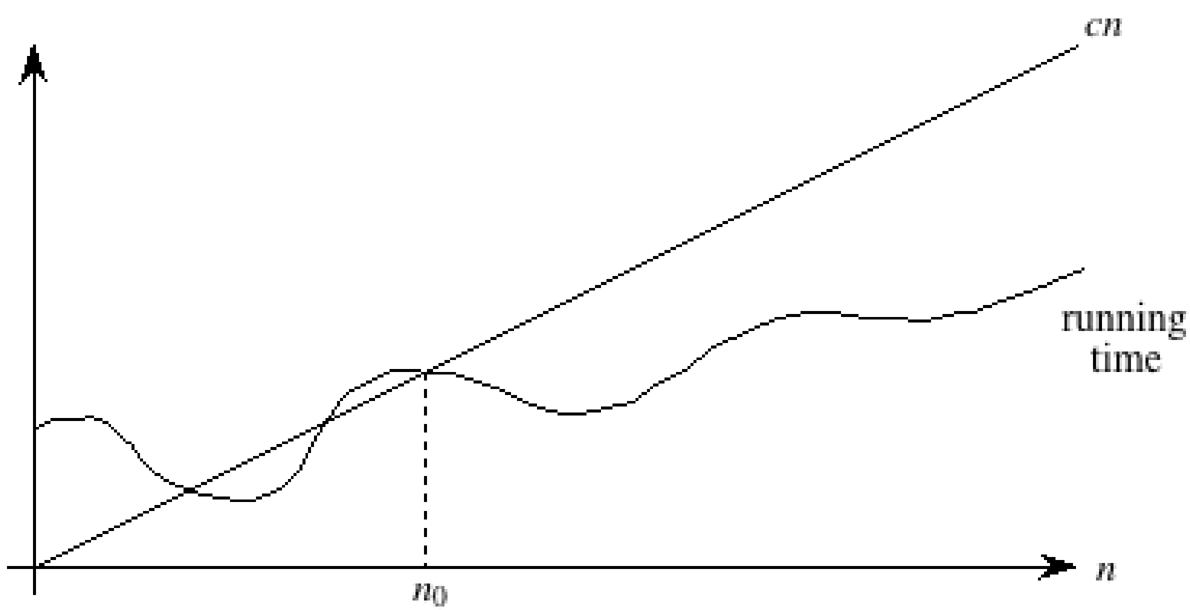
Polynomial

Exponential

ASYMPTOTIC NOTATION

Computer scientists describe upper bounds on orders of growth with "Big Oh" notation

O gives an asymptotic <u>upper</u> bounds



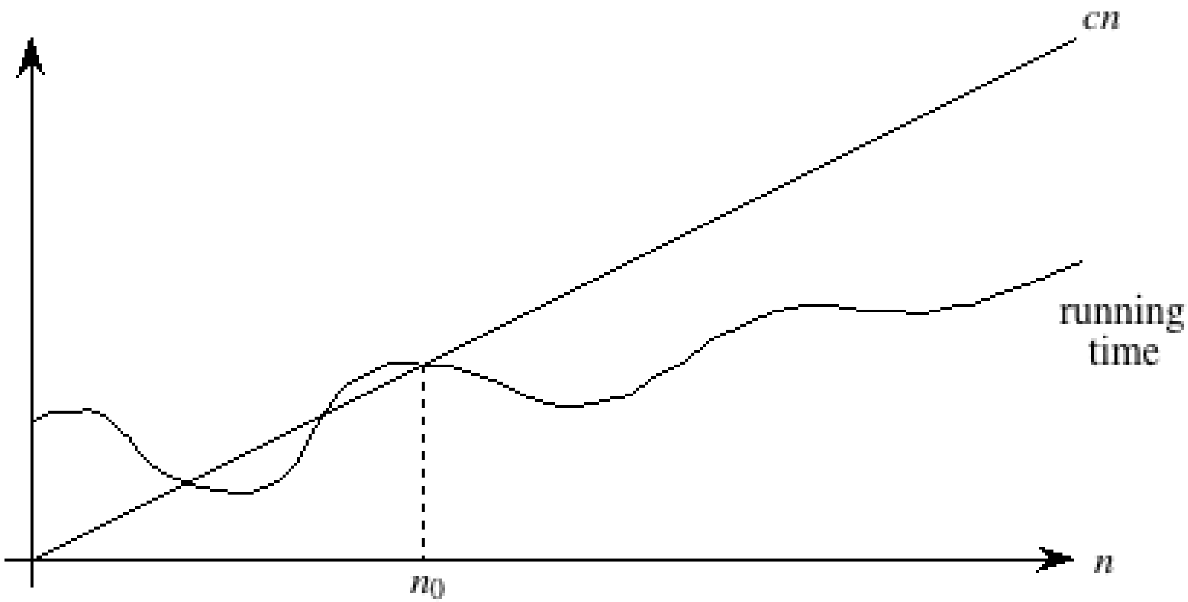
Run-time complexity is O(n) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at <u>*most*</u> cn, upper bound

Computer scientists describe upper bounds on orders of growth with "Big Oh" notation

O gives an asymptotic <u>upper</u> bounds

"Big Oh of n", and "Oh of n", and "order n" all mean the same thing!



Example: find *<u>specific</u>* item in a list

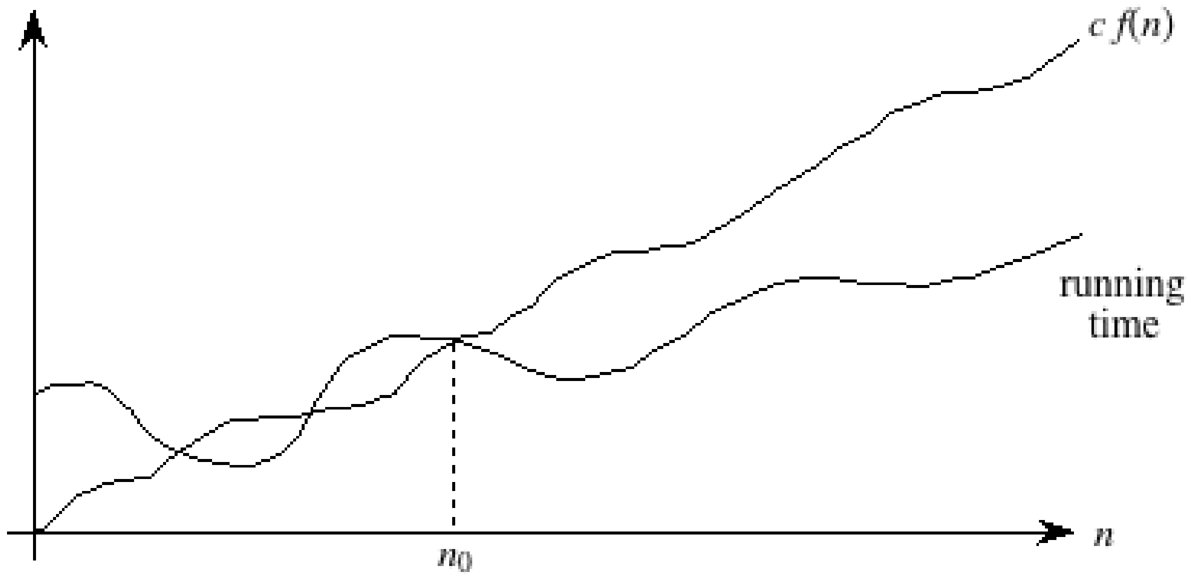
- **Might find item on first try**
- **Might not find it at all (must check all *n* items in list)**
- **Worst case (upper bound) is O(*n*)**

Run-time complexity is O(n) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at *<u>most</u>* cn, upper bound
- O(n) is the *<u>worst</u>* case performance for large n, but actual performance could be better
- O(n) is said to be "linear" time
- O(1) means constant time

We can extend Big Oh to any, not necessarily linear, function

O gives an asymptotic upper bounds

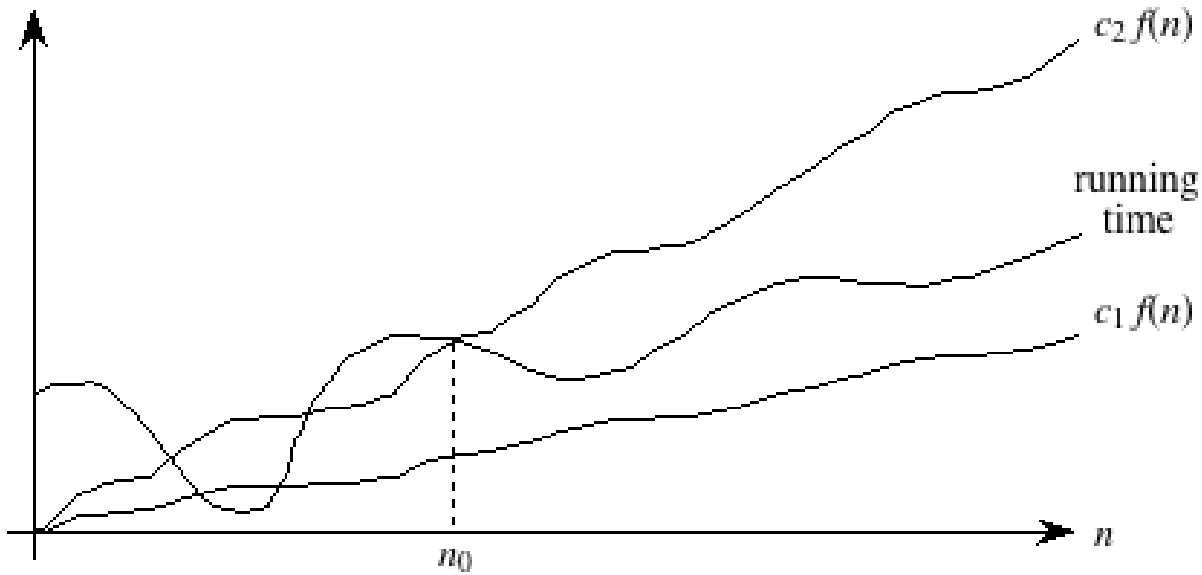


Run-time complexity is O(*f(n)*) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at *most* *cf(n)*, upper bound
- O(*f(n)*) is the *worst* case performance for large n, but actual performance could be better
- *f(n)* can be a non-linear function such as n^2 or *log(n)*
- In that case O(n^2) *or* O(*log n*)

Run time can also be Ω (Big Omega), where run time grows at least as fast

Ω gives an asymptotic <u>lower</u> bounds



Example: find *<u>largest</u>* item in a list

- **Must check each *n* items**
- **Largest item could be at end of list, can't stop early**
- **Can't do better than Ω (*n*)**

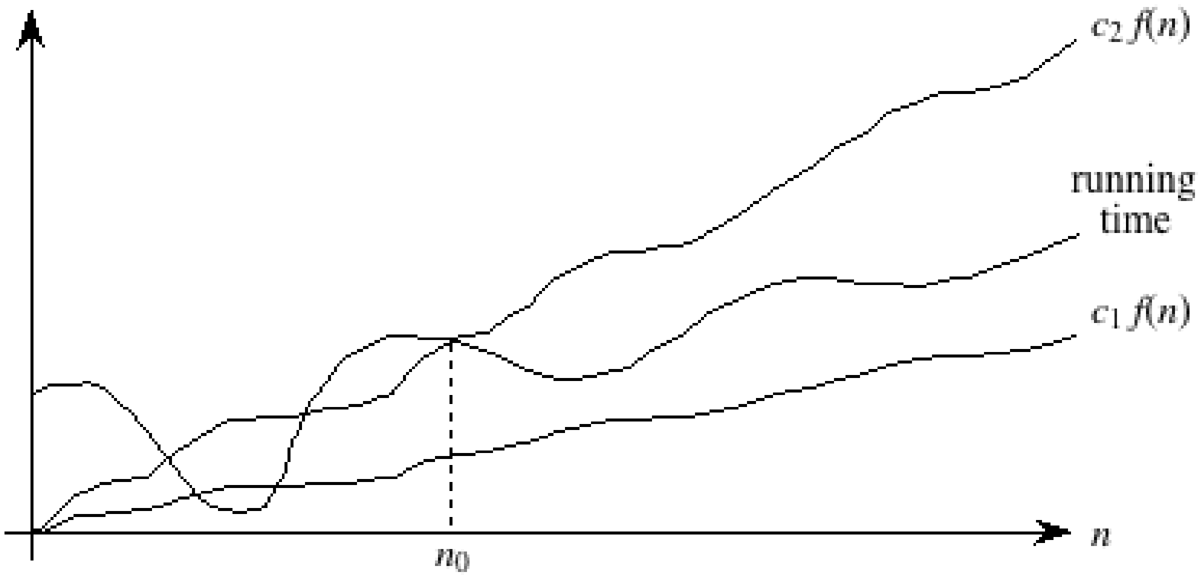
Run-time complexity is $\Omega(f(n))$ if there exists constants n_0 and c_1 such that:

- $\forall n \geq n_0$
- run time of size n is at *<u>least</u>* $c_1 f(n)$, lower bound
- $\Omega(n)$ is the *<u>best</u>* case performance for large n, but actual performance can be worse

We use Θ (Big Theta) for tight bounds when we can define O and Ω

Θ gives an asymptotic <u>tight</u> bounds

We can also apply these concepts to how much memory an algorithm uses (not just run-time complexity)



Example: find *<u>largest</u>* item in a list

- **Best case: already seen it is Ω(n)**
- **Worst case: must check each item, so O(n)**
- **Because Ω(n) *<u>and</u>* O(n) we say it is Θ(n)**

Run-time complexity is Θ(*f(n)*) if there exists constants n_0 and c_1 and c_2 such that:

- $\forall n \geq n_0$
- run time of size *n* is at *<u>least</u>* $c_1 f(n)$ and at <u>most</u> $c_2 f(n)$
- Θ(*n*) gives a tight bound, which means run time will be within a constant factor
- Generally we will use either O or Θ
- **O, Ω, Θ called asymptotic notation**

We ignore constants and low-order terms in asymptotic notation

Constants don't matter, just adjust c_1 and c_2

- Constant multiplicative factors are absorbed into c_1 (and c_2)
- Example: $1000n^2$ is O(n^2) because we can choose c_1 to be 1000 (remember bounded by $c_1 n$)
- Do care in practice – if an operation takes a constant time, O(1), but more than 24 hours to complete, can't run it everyday

Low order terms don't matter either

- If $n^2+1000n$, then choose $c_1 = 1$, so now $n^2 + 1000n \geq c_1 n^2$
- Now must find c_2 such that $n^2 + 1000n \leq c_2 n^2$
- Subtract n^2 from both sides and get $1000n \leq c_2 n^2 - n^2 = (c_2-1)n^2$
- Divide both sides by (c_2-1)n gives $1000/(c_2-1) \leq n$
- Pick $c_2 = 2$ and $n_0 = 1000$, then $\forall n \geq n_0$, $1000 \leq n$
- So, $n^2 + 1000n \leq c_2 n^2$, try with $n=1000$ get $n^2 + 1000^2 = 2*n^2$
- **In practice, we simply ignore constants and low order terms**

How to write them

Constant time	Linear time	Polynomial time
$O(1)$	$O(n)$	$O(n^2)$

DESCRIPTION OF PROS AND CONS

At first arrays seem to be a poor choice to implement the List ADT

List ADT features	Linked List	Array
get()/set() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Contiguous block of memory • Random access aspect of arrays makes *get()/set()* easy and fast
add()/remove() element anywhere in List	• Start at head and march down to index in list • Slow to find element, but fast once there	• Fast to find element, but slow once there • Have to make (or fill) hole by copying over
No limit to number of elements in List	• Built in feature of how linked lists work • Just create a new element and splice it in	• Arrays declared of fixed size

ArrTest.java

ANNOTATED SLIDES

Random access aspect of arrays makes it easy to get or set any element

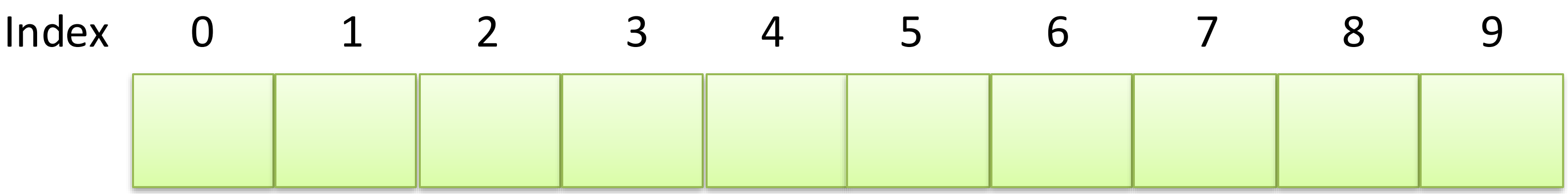
```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}

```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

- **Array reserves a contiguous block of memory**
- **Big enough to hold specified number of elements (10 here) times size of each element (4 bytes for integers) = 40 bytes**
- **Indices are 0…9**

Random access aspect of arrays makes it easy to get or set any element




```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2							

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}

```

Problems @ Javadoc Declaration Console ☒ Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

No need to march down list to get or set element

To find element:

- **Start at base address of array (this is where “*numbers*” array points)**
- **Element at index *idx* is at address:**
  ***base addr* + *idx**size(element)**

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2							

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

No need to march down list to get or set element

To find element:

- **Start at base address of array (this is where "*numbers*" array points)**
- **Element at index *idx* is at address:**
  base addr* + *idxsize(element)**
- **Index 2 at *base addr + 2*4* bytes**
- **Time to access element is constant anywhere in array (just simple math operation to calculate any index)**
- **With linked list have to march down list, takes longer to find elements at end**

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

Problems @ Javadoc Declaration Console ☒ Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}
```

What values will a, b and c have?

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

Random access aspect of arrays makes it easy to get or set any element

Index	0	1	2	3	4	5	6	7	8	9
			2			10				

```java
public class ArrTest {

    public static void main(String[] args) {
        //declare array
        int[] numbers = new int[10]; //indices 0..9

        //set some elements
        numbers[2] = 2;
        numbers[5] = 10;

        //get some elements
        int a = numbers[2];
        int b = numbers[5];
        int c = numbers[1]; //we did not set this
        System.out.println("a="+a+" b="+b+" c="+c);
    }
}

```

What values will a, b and c have?

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> ArrTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec 31, 2017, 6:

```
a=2 b=10 c=0
```

EXAMPLE OF INSERTION IN ARRAYLIST

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	0	0	0	0

14

Insert 14 at index 2

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	0	0	0	0

14

Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)

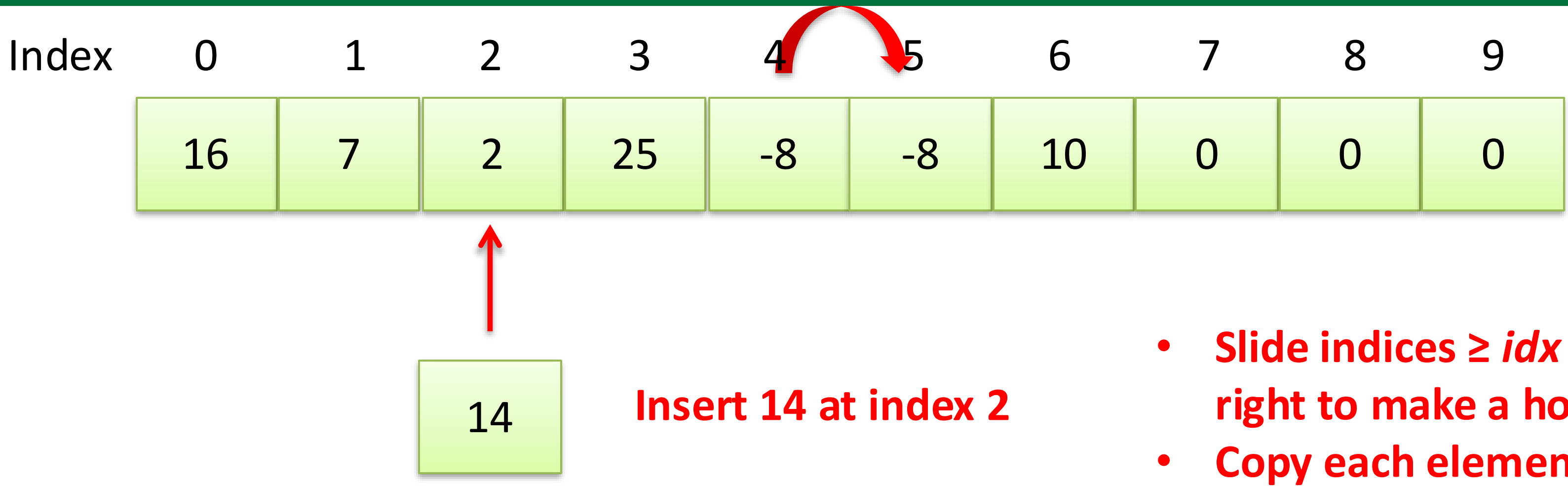
Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	-8	10	10	0	0	0

14

Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)



- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	25	25	-8	10	0	0	0

14

Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	2	25	-8	10	0	0	0

14

Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	2	2	25	-8	10	0	0	0

14

Insert 14 at index 2

- **Slide indices ≥ *idx* to the right to make a hole**
- **Copy each element to next index**

Copy new element into index

Because arrays are a contiguous block of memory, hard to insert (except at end)

Index	0	1	2	3	4	5	6	7	8	9
	16	7	14	2	25	-8	10	0	0	0

- **Works, but takes a lot of time (said to be “expensive”)**
- **Especially expensive with respect to time if the array is large and we insert at the front**
- **Linked list is slow to find the right place (have to march down list starting from head), but fast to insert, just update two pointers and you’re done**
- **Linked list is fast, however, if only dealing with head**
- **With arrays, easy to find right place, but slow afterward due to copying to make a hole**

EXAMPLE OF GROWING ARRAYLIST

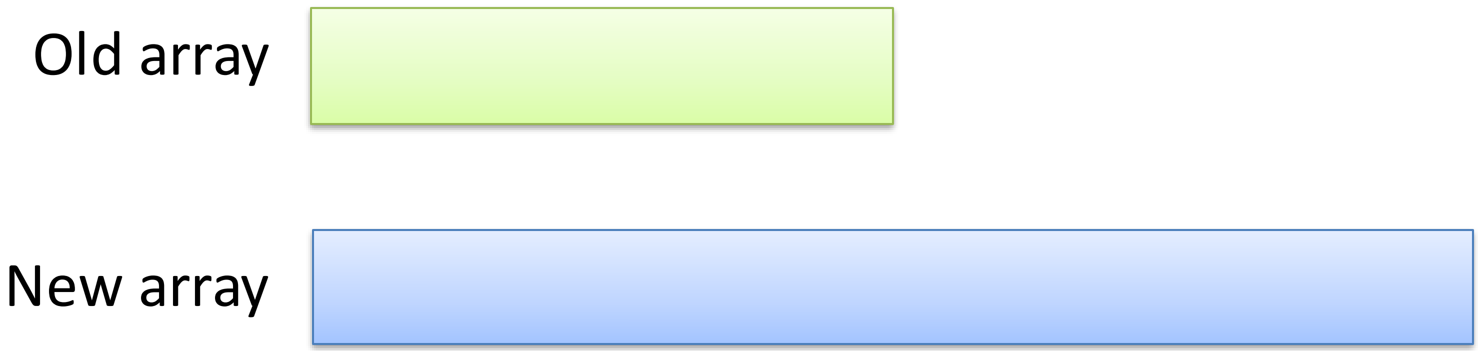
Arrays are of fixed size, but List ADT allows for growth

Index	0	1	2	3	4	5	6	7	8	9
	16	7	14	2	25	-8	10	52	-19	6

What do we do when the array is full, but we want to add more elements?

Answer: create another, larger array, and copy elements from old array into new array

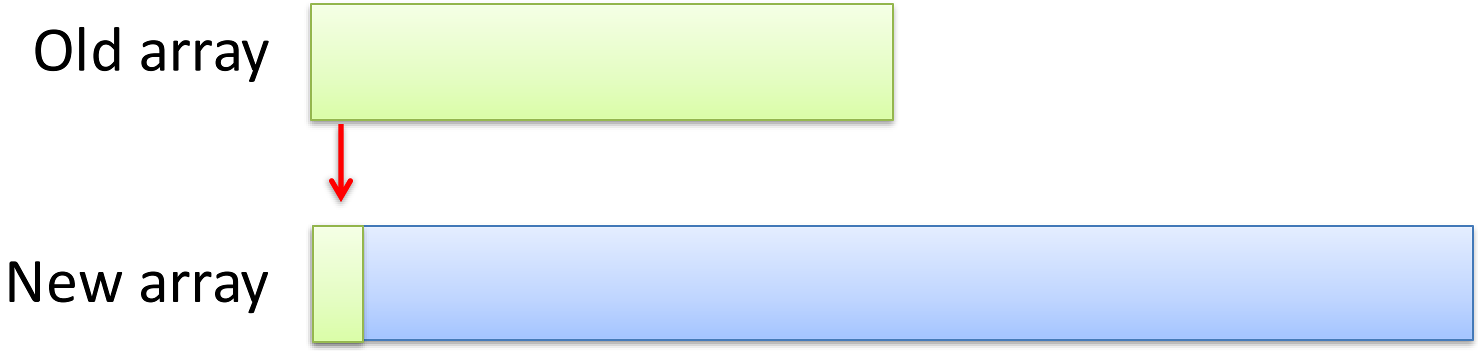
Arrays are of fixed size, but List ADT allows for growth



Grow array

1. **Make new array, say 2 times larger than old array**

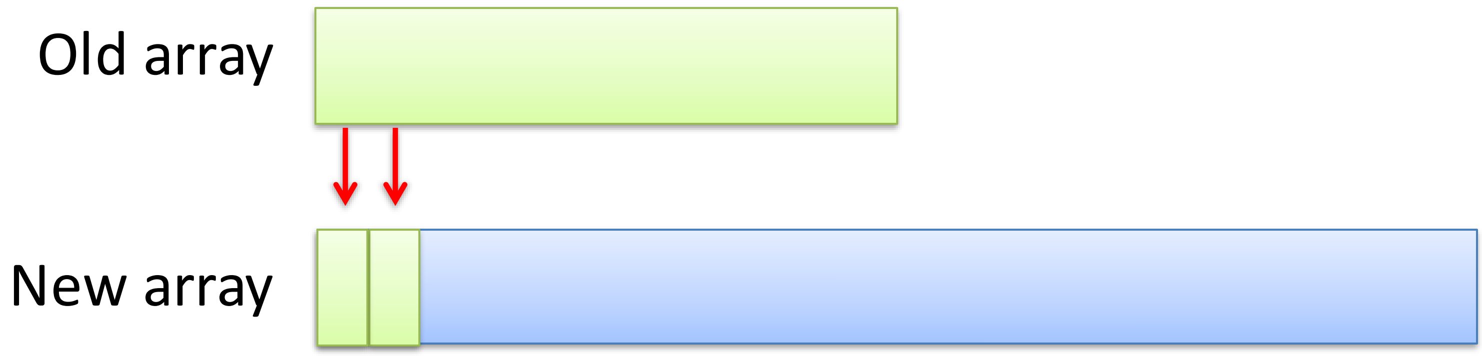
Arrays are of fixed size, but List ADT allows for growth



Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth



Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

Grow array

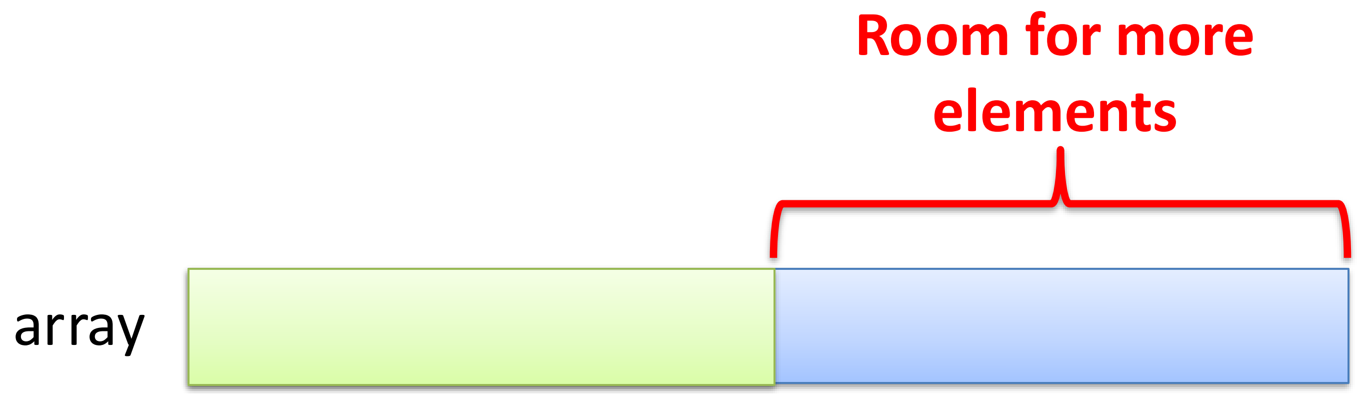
1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth

Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**

Arrays are of fixed size, but List ADT allows for growth



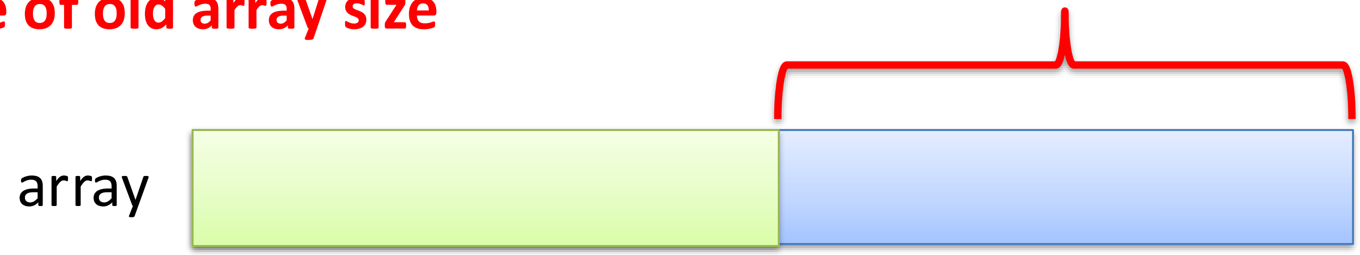
Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**
3. **Set instance variable to point at new array (old array will be garbage collected)**

Arrays are of fixed size, but List ADT allows for growth

Growing is expensive operation, but we don't have to do it frequently if new array size is multiple of old array size

Room for more elements



Grow array

1. **Make new array, say 2 times larger than old array**
2. **Copy elements one at a time from old array to new**
3. **Set instance variable to point at new array (old array will be garbage collected)**

GrowingArray.java

ANNOTATED SLIDES

GrowingArray.java: implements List ADT using an array instead of a linked list

```
public class GrowingArray<T> implements SimpleList<T>, Iterable<T> {
   private T[] array;
   private int size;     // how much of the array is actually filled up so far
   private static final int initCap = 10; // how big the array should be initially

   public GrowingArray() {
     array = (T[]) new Object[initCap];  // java generics oddness – cast array of objects
     size = 0;
   }

   /**
    * Return the number of elements in the List (they are indexed 0..size-1)
    * @return number of elements
    */
   public int size() {
     return size;
   }
```

Implements SimpleList and Iterable from last class

Array is now the data structure used to store elements in List

- **Array initially sized to 10 Objects (note the funky Java allocation syntax, must cast to array of generic type)**
- **Remember, arrays are of fixed size, but the List ADT does not specify a size**

Track size
Will increment on each *add* and decrement on each *remove*
Run-time complexity?
O(1)

GrowingArray.java: *get()/set()* are easy and fast with an array implementation

```
/**
 * Return item at index idx
 * @param idx index of item to return
 * @return item stored at index idx
 * @throws Exception invalid index
 */
public T get(int idx) throws Exception {
    if (idx >= 0 && idx < size) return array[idx];
    else throw new Exception("invalid index");
}

/**
 * Overwrite item at index idx with item parameter
 * @param idx index of item to get
 * @param item overwrite existing item at index idx with this item
 * @throws Exception invalid index
 */
public void set(int idx, T item) throws Exception {
    if (idx >= 0 && idx < size) array[idx] = item;
    else throw new Exception("invalid index");
}
```

Get and set are easy, just make sure index is valid, then return or set item

Notice: no curly braces!

Only next line in if statement

Run-time complexity?
O(1) for any index!
Just two math operations to compute memory address

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
      // Double the size of the array, to leave more space
      T[] copy = (T[]) new Object[size*2];
      // Copy it over
      for (int i=0; i<size; i++) copy[i] = array[i];
      array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}
```

array.length **is how many elements** ***array*** ***can*** **hold**

size **has how many elements** ***array*** ***does*** **hold**

add() **makes a new, larger array if needed**

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
  if (idx > size || idx < 0) throw new Exception("invalid index");
  if (size == array.length) {
    // Double the size of the array, to leave more space
    T[] copy = (T[]) new Object[size*2];
    // Copy it over
    for (int i=0; i<size; i++) copy[i] = array[i];
    array = copy;
  }
  // Shift right to make room
  for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
  array[idx] = item;
  size++;
}
```

array.length **is how many elements** ***array*** ***can*** **hold**

size **has how many elements** ***array*** ***does*** **hold**

add() **makes a new, larger array if needed**

Copy elements one at a time into new array

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
     // Double the size of the array, to leave more space
     T[] copy = (T[]) new Object[size*2];
     // Copy it over
     for (int i=0; i<size; i++) copy[i] = array[i];
     array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}
```

array.length **is how many elements** ***array*** ***<u>can</u>*** **hold**

size **has how many elements** ***array*** ***<u>does</u>*** **hold**

Update instance variable to new array

add() **makes a new, larger array if needed**

Copy elements one at a time into new array

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
     // Double the size of the array, to leave more space
     T[] copy = (T[]) new Object[size*2];
     // Copy it over
     for (int i=0; i<size; i++) copy[i] = array[i];
     array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}
```

- **Here we know we have enough room to add a new element**
- **Now do insert**
- **Start from last item and copy to one index larger**
- **Stop at index *idx***
- **Set item at *idx* to item**

GrowingArray.java: With growing trick, can implement the List interface with an array

```
public void add(int idx, T item) throws Exception {
   if (idx > size || idx < 0) throw new Exception("invalid index");
   if (size == array.length) {
     // Double the size of the array, to leave more space
     T[] copy = (T[]) new Object[size*2];
     // Copy it over
     for (int i=0; i<size; i++) copy[i] = array[i];
     array = copy;
   }
   // Shift right to make room
   for (int i=size-1; i>=idx; i--) array[i+1] = array[i];
   array[idx] = item;
   size++;
}

public void add(T item) throws Exception {
   add(size,item);
}
```

Add an item at the end is easy
Just call *add* with *size* as index

What did we call it when two methods have the same name but different variables?
Overloading

Run-time complexity
O(1)

GrowingArray.java: With growing trick, can implement the List interface with an array

```java
/**
 * Remove and return the item at index idx.  Move items left to fill hole.
 * @param idx index of item to remove
 * @return the value previously at index idx
 * @throws Exception invalid index
 */
public T remove(int idx) throws Exception {
    if (idx > size-1 || idx < 0) throw new Exception("invalid index");
    T data = array[idx];
    // Shift left to cover it over
    for (int i=idx; i<size-1; i++) array[i] = array[i+1];
    size--;
    return data;
}
```

***remove()* slides elements left one slot for index > idx**

Run-time complexity? O(n)

LIST ANALYSIS

Growing array is *<u>generally</u>* preferable to linked list, except maybe growth operation

Worst case run-time complexity

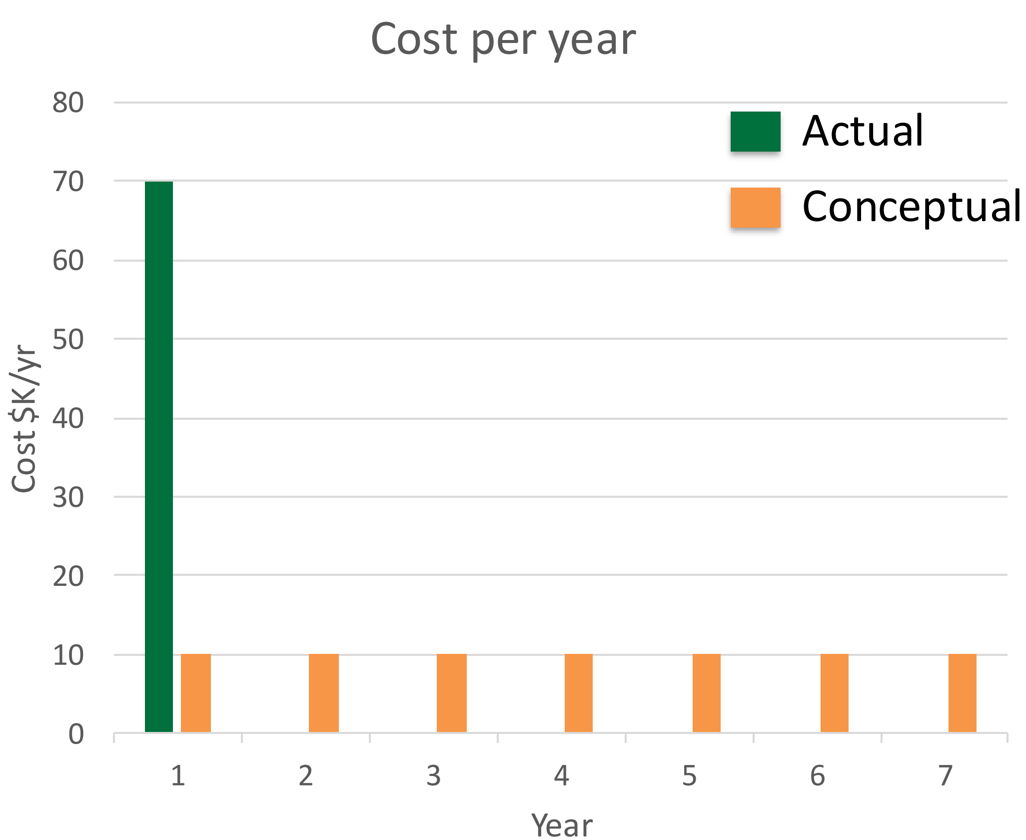
	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	O(n) + growth
remove(i)	O(n)	O(n)

- **Start at *head* and march down to find index *i***
- **Slow to get to index, O(n)**
- **Once there, operations are fast O(1)**
- **Best case: all operations on head**

- **Faster *get()/set()* than linked list**
- **Tie with linked list on *remove()***
- **Best case: all operation at tail**
- ***add()* might cause expensive growth operation**
- **How should be think about that?**

Amortization is a concept from accounting that allows us to spread costs over time

Amortized analysis

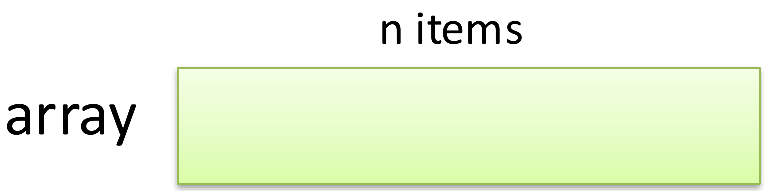


Accounting allows us to amortize costs over several years

- Buy $70K truck on year 1
- Truck is good for 7 years
- Can think of the cost as $10K/year instead of one payment of $70K on year 1
- Actually pay $70K on year 1, but this is equivalent to paying $10K/year for 7 years
- Idea is to spread the cost ("amortize" the cost) over the lifetime of the truck
- We will use this concept to "pre-pay" for expensive growth operation

Amortized analysis shows growing array is actually only O(1)!

Amortized analysis

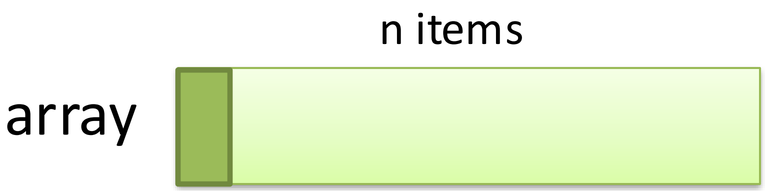


Each time add an item to array, *<u>conceptually</u>* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

Amortized analysis shows growing array is actually only O(1)!

Bank
2

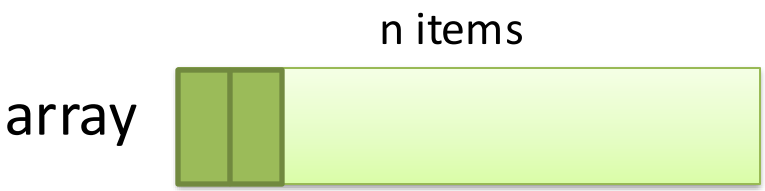


Each time add an item to array, *conceptually* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

Amortized analysis shows growing array is actually only O(1)!

Bank
4

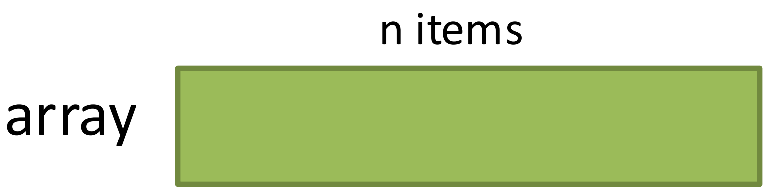


Each time add an item to array, *<u>conceptually</u>* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

Amortized analysis shows growing array is actually only O(1)!

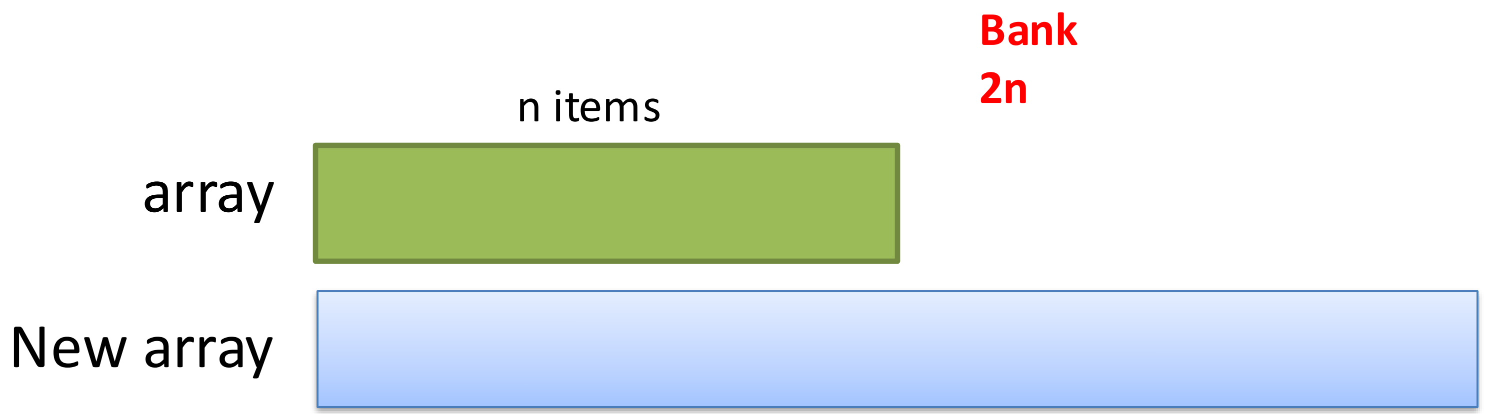
Bank
2n



Each time add an item to array, *__conceptually__* charge 3 "tokens"
- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

After *n add()* operations, array is full, but have *2n* tokens in bank

Amortized analysis shows growing array is actually only O(1)!



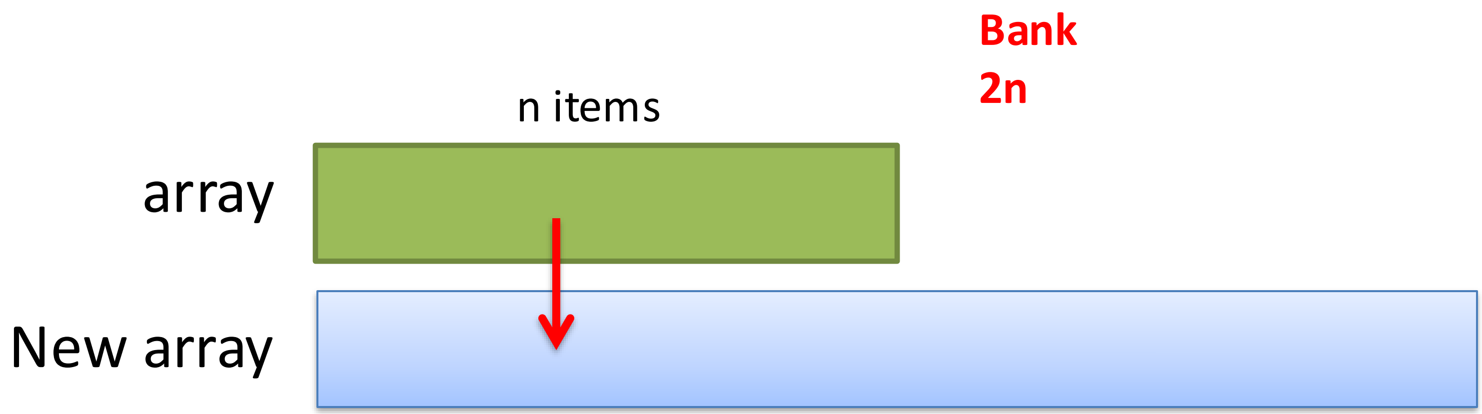
Each time add an item to array, *conceptually* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, ***conceptually*** charge 3 "tokens"

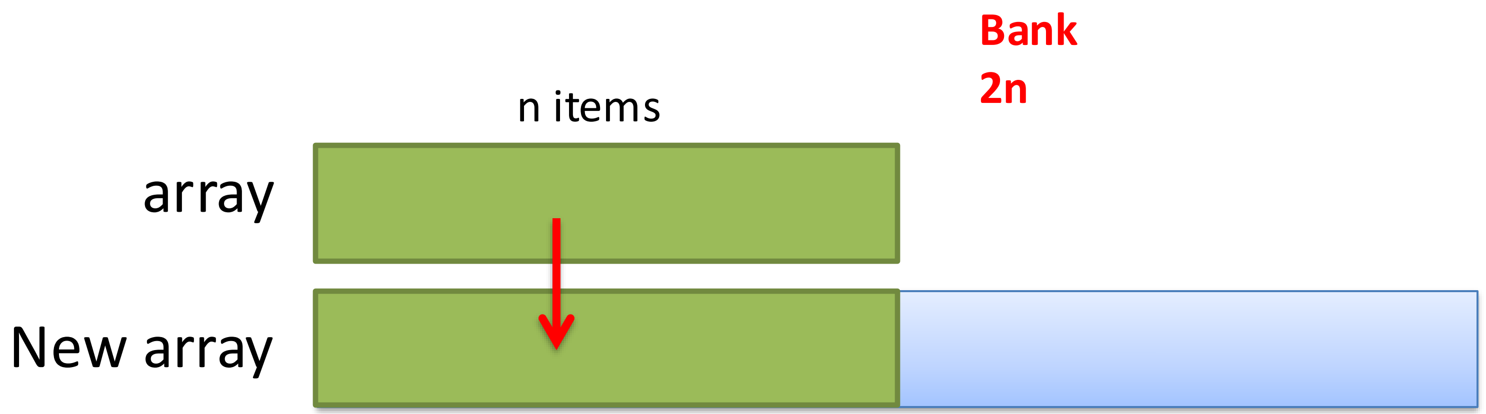
- One token pays for current add()
- Two tokens go into "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, ***conceptually*** charge 3 "tokens"

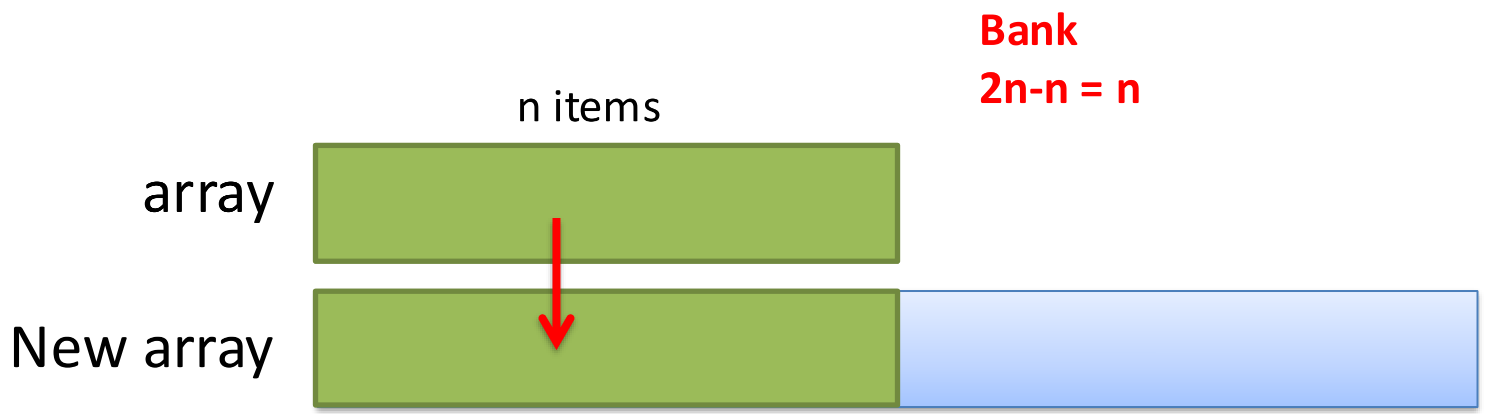
- One token pays for current add()
- Two tokens go into "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, ***conceptually*** charge 3 "tokens"

- One token pays for current add()
- Two tokens go into "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

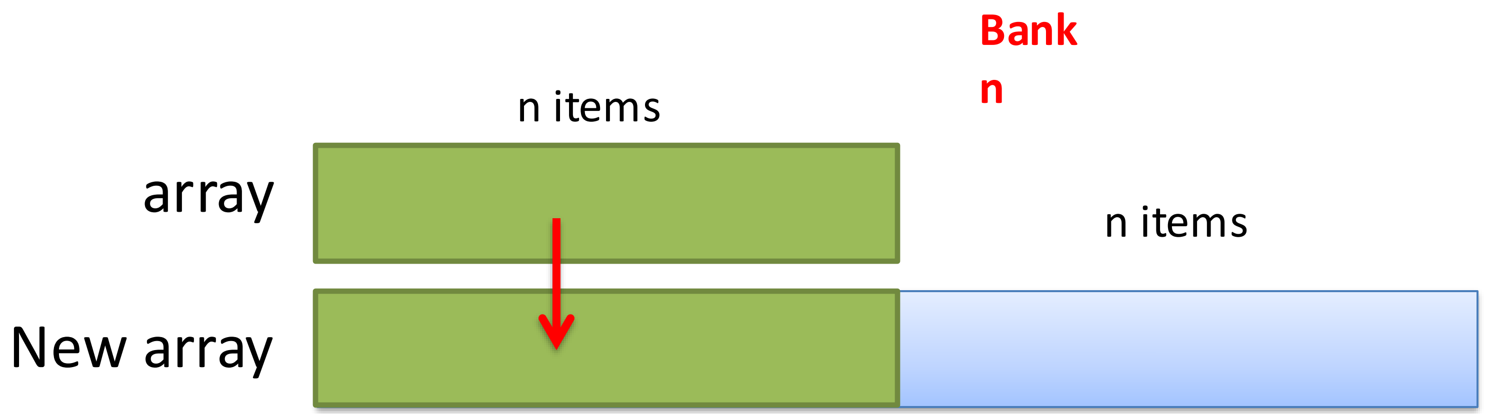
After *n add()* operations, array is full, but have *2n* tokens in bank

Allocate new 2X larger array

Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, *<u>conceptually</u>* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

After *n add()* operations, array is full, but have *2n* tokens in bank

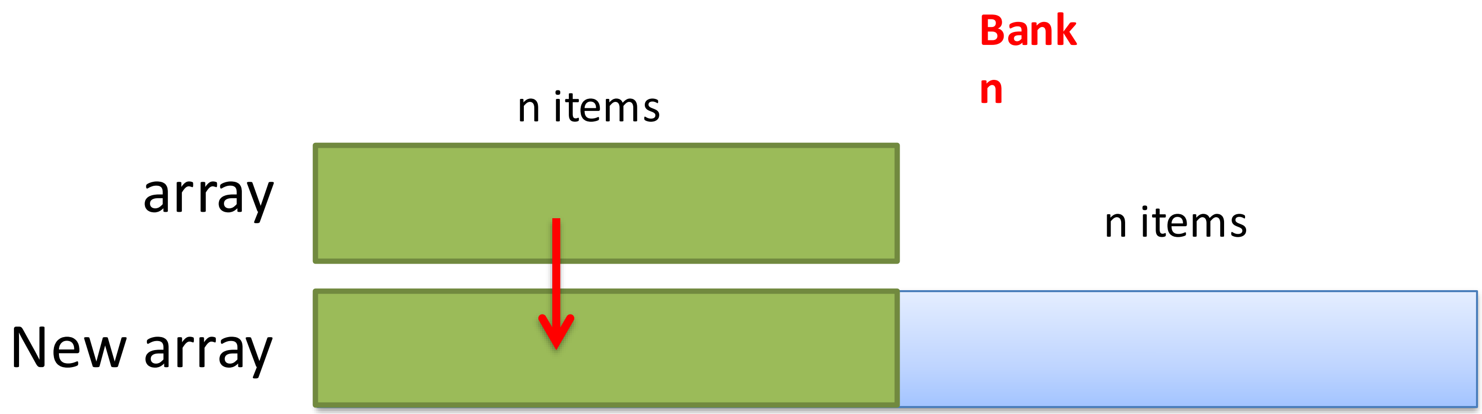
Allocate new 2X larger array

Copy elements from old array to new array

Have to copy *n* items, so charge *n* pre-paid tokens from bank

Remaining *n* items in bank "pay for" empty *n* spaces

Amortized analysis shows growing array is actually only O(1)!

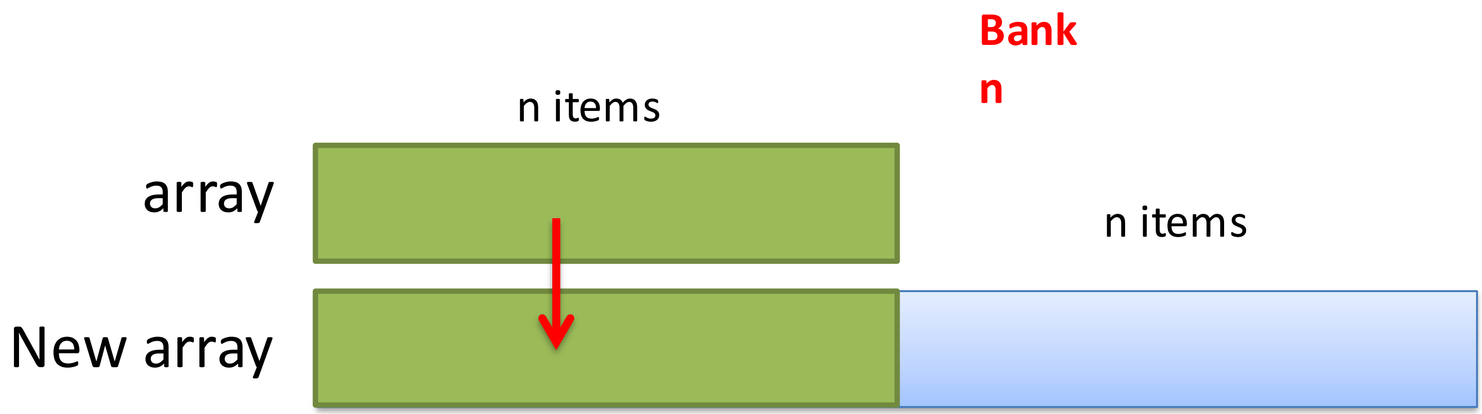


Each time add an item to array, *conceptually* charge 3 "tokens"

- **One token pays for current add()**
- **Two tokens go into "Bank"**
- **We are spread out (amortizing) the cost of the expensive, but infrequent growth operation**

After *n add()* operations, array is full, but have *2n* tokens in bank
Allocate new 2X larger array
Copy elements from old array to new array
Have to copy *n* items, so charge *n* pre-paid tokens from bank
Remaining *n* items in bank "pay for" empty *n* spaces
Charging a little extra for each *add* spreads out cost for infrequent growth operation

Amortized analysis shows growing array is actually only O(1)!



Each time add an item to array, ***conceptually*** charge 3 "tokens"

- One token pays for current add()
- Two tokens go into "Bank"
- We are spread out (amortizing) the cost of the expensive, but infrequent growth operation

After *n add()* operations, array is full, but have *2n* tokens in bank
Allocate new 2X larger array
Copy elements from old array to new array
Have to copy *n* items, so charge *n* pre-paid tokens from bank
Remaining *n* items in bank "pay for" empty *n* spaces
Charging a little extra for each *add* spreads out cost for infrequent growth operation
The charge, however, is a constant, so O(3) = O(1)

Growing array is *generally* preferable to linked list

Worst case run-time complexity

	Linked list	Growing array
get(i)	O(n)	O(1)
set(i,e)	O(n)	O(1)
add(i,e)	O(n)	**O(n) + O(1) = O(n)**
remove(i)	O(n)	O(n)

Amortized analysis shows infrequent growth operation is constant time

Pay a constant amount more on each *add()* to pay for the occasional expensive growth

- Start at *head* and march down to find index *i*
- Slow to get to index, O(n)
- Once there, operations are fast O(1)
- Best case: all operations on head

- Faster *get()/set()* than linked list
- Tie with linked list on *remove()*
- Best case: all operations on tail
- *add()* might cause expensive growth operation