CSE 250: Midterm Review 1

Lecture 16

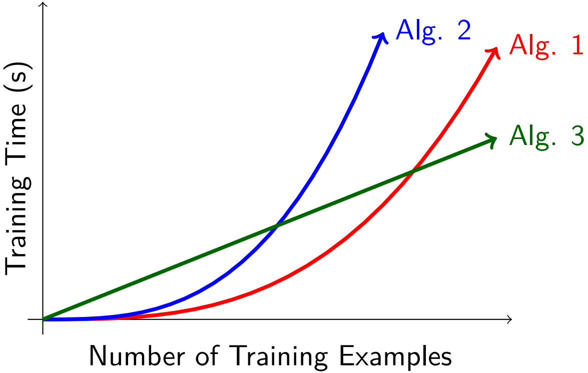
Oct 2, 2024

Exam Day

- **Do** bring...
    - Writing implement (pen or pencil)
    - One note sheet (up to $8\frac{1}{2} \times 11$ inches, double-sided)
- **Do not** bring...
    - Bag (you will be asked to leave it at the front of the room)
    - Computer/Calculator/Watch/etc...

- Wait outside before the exam starts so we can prepare.
    - You will be told when to enter.
- There will be assigned seating.
    - Seating charts will be posted on the doors and projector.
    - See the seat numbers on the chairs.

Runtime



Some Notation

- N: The input "size"
    - How many students I have to email.
    - How many streets on a map.
    - How many key/value pairs in my dictionary
- $T(N)$: The runtime of 'some' implementation of the algorithm.
    - Some... correct implementation.

We care about the "shape" of $T(N)$ when you plot it.

Class Names

$T(N) \in \ldots$

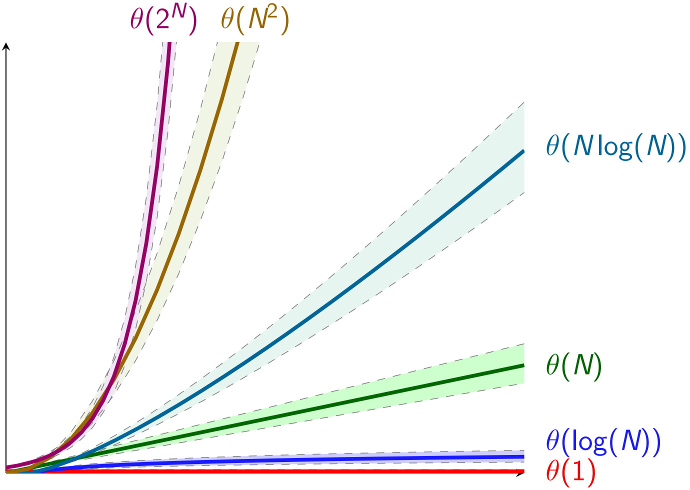
- $\ldots\theta(1)$: Constant
- $\ldots\theta(\log(N))$: Logarithmic
- $\ldots\theta(N)$: Linear
- $\ldots\theta(N\log(N))$: Log-Linear
- $\ldots\theta(N^2)$: Quadratic
- $\ldots\theta(N^k)$ (for any $k \geq 1$): Polynomial
- $\ldots\theta(2^N)$: Exponential

Complexity Bounds

f and g are in the same complexity class if:

- g is bounded from above by something f-shaped
  $g(N) \in O(f(N))$
- g is bounded from below by something f-shaped
  $g(N) \in \Omega(f(N))$

Complexity Classes

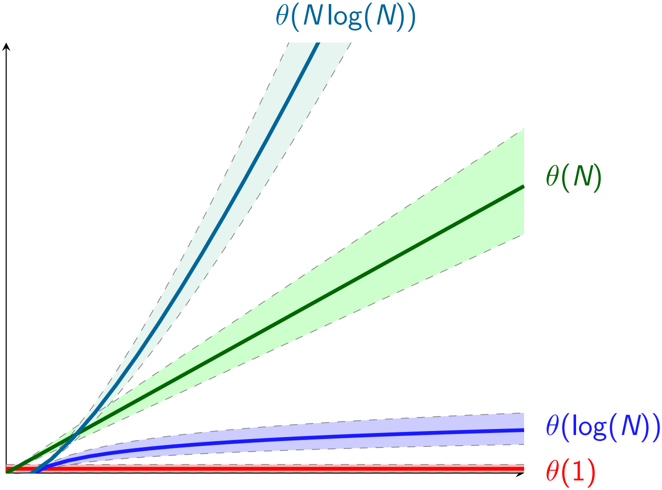


Complexity Bounds

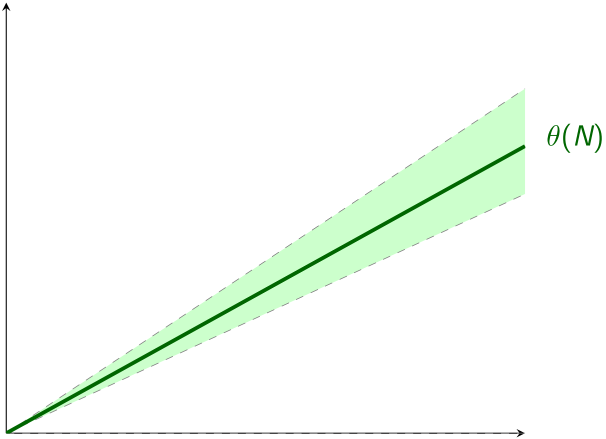
- $O(f(N))$ includes:
  - All functions in $\theta(f(N))$
  - All functions in 'smaller' complexity classes
- $\Omega(f(N))$ includes:
  - All functions in $\theta(f(N))$
  - All functions in 'bigger' complexity classes

$$O(f(N)) \cap \Omega(f(N)) = \theta(f(N))$$

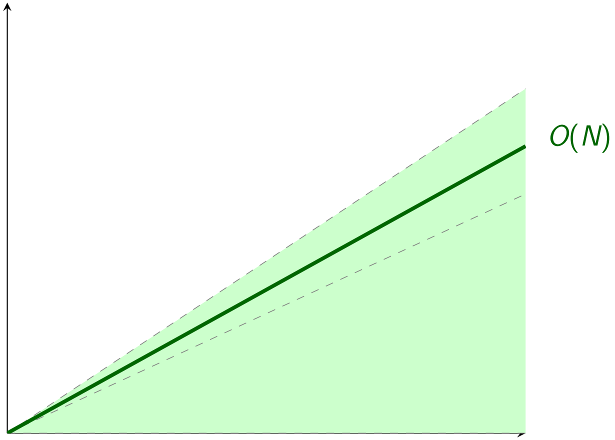
Complexity Bounds



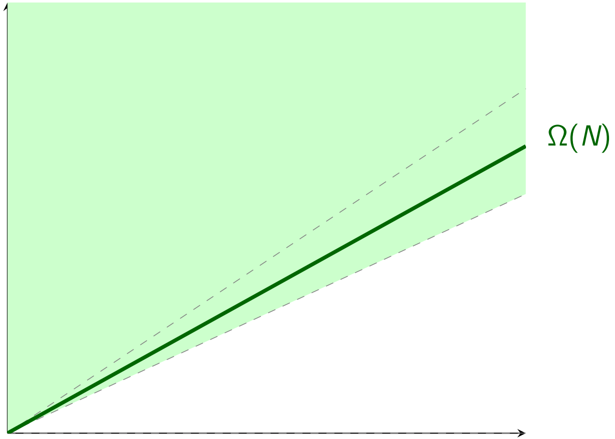
Complexity Bounds



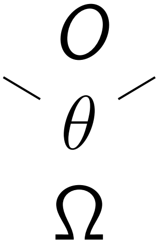
Complexity Bounds



Complexity Bounds



Rules of Thumb



O

θ

Ω

© Aleksandra Patrzalek, 2012

Complexity Bounds

$g(N) \in O(f(N))$ (f is an upper bound for g) if and only if:

- You can pick an N_0
- You can pick a c
- For all $N > N_0$: $g(N) \leq c \cdot f(N)$

$g(N) \in \Omega(f(N))$ (f is a lower bound for g) if and only if:

- You can pick an N_0
- You can pick a c
- For all $N > N_0$: $g(N) \geq c \cdot f(N)$

$g(N) \in \theta(f(N))$ if and only if:

- $g(N) \in \Omega(f(N))$
- $g(N) \in O(f(N))$

Rules of Thumb

$$F(N) = f_1(N) + f_2(N) + \ldots + f_k(N)$$

What complexity class is $F(N)$ in?

$f_1(N) + f_2(N)$ is in the greater of $\theta(f_1(N))$ and $\theta(f_2(N))$.

$F(N)$ is in the greatest of any $\theta(f_i(N))$

We say the biggest f_i is the dominant term.

Multi-Class Functions

$$T(N) = \begin{cases} \theta(1) & \textbf{if } N \text{ is even} \\ \theta(N) & \textbf{if } N \text{ is odd} \end{cases}$$

What is the complexity class of $T(N)$?

- $T(N) \in O(N)$ is a **tight** bound.
- $T(N) \in \Omega(1)$ is a **tight** bound.

Multi-Class Functions

$$T(N) = \begin{cases} \theta(1) & \textbf{if } N \text{ is even} \\ \theta(N) & \textbf{if } N \text{ is odd} \end{cases}$$

What is the complexity class of $T(N)$?

- $T(N) \in O(N)$ is a **tight** bound.
- $T(N) \in \Omega(1)$ is a **tight** bound.

If the tight Big-O and Big-Ω bounds are different,
the function is not in ANY complexity class.
(Big-Theta doesn't exist).

Does Big-Theta Exist?

$N + 2N^2$ belongs to one complexity class. $(\theta(N^2))$

$5N + 10N^2 + 2^N$ belongs to one complexity class $(\theta(2^N))$

$\begin{cases} 2^N & \textbf{if } \texttt{rand()} > 0.5 \\ N & \textbf{otherwise} \end{cases}$ does **not** belong to one complexity class.

Does Big-Theta Exist?

$N + 2N^2$ belongs to one complexity class. $(\theta(N^2))$

$5N + 10N^2 + 2^N$ belongs to one complexity class $(\theta(2^N))$

$\begin{cases} 2^N & \textbf{if } \texttt{rand()} > 0.5 \\ N & \textbf{otherwise} \end{cases}$ does **not** belong to one complexity class.

- **Usually $\theta(f_1(N) + f_2(N) + \ldots)$ is based on the dominant term**
- **If you see cases (i.e., '{'), it's probably multi-class.**

Multi-Class Functions

If…

- $g(N) \in O(f(N))$ is a **tight** upper bound
- $g(N) \in \Omega(f'(N))$ is a **tight** lower bound
- $f'(N) \notin \theta(f(N))$

… then there is no θ bound for $g(N)$ (g is multi class)

Remember: Addition does not make a function multi-class.

(A tight $\Omega(f(N))$ is the dominant (biggest) term being summed)

Rules of Thumb

- **Lines of Code**: Add Complexities
- **Loops**: Multiply Complexity by the Loop Count
- **If/Then**: Cases block ‘$\{$’

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  int N = data.size();
  for(int i = N - 2; i >= 0; i--)
  {
    for(int j = i; j <= N - 1; j++)
    {
      if(data.get(j+1) < data.get(j))
      {
        int temp = data.get(j);
        data.set(j, data.get(j+1));
        data.set(j+1, temp);
      }
    }
  }
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  $O(N^3)$
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  int[] array = data.toArray()
  bubblesort(array) // Use the array implementation
  data.clear()
  data.addAll(Arrays.toList(array))
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  $O(N)$
  $O(N^2)$
  $O(N)$
  $O(N)$
}
```

Abstract Data Types

Abstract Data Type defines...

- Domain: What kind of data is stored? (e.g., elements, key/value pairs)
- Constraints: How are items related? (e.g., ordered keys)
- Operations: How can the data be accessed/modified (e.g., 'i'th item)

Like a Java `interface`[1]

---

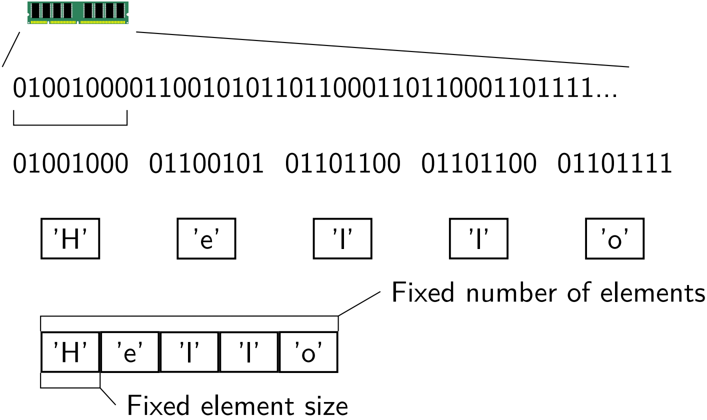
[1] The term `interface` is not quite the same as ADT; The interface only formalizes the permitted operations.

The Sequence ADT

```
public interface Sequence<E>
{
  public E get(int idx);
  public void set(int idx, E value);
  public int size();
  public Iterator<E> iterator();
}
```

`E` is the type of thing in the Sequence.

CSE 220 Crossover



0100100001100101011011000110110001101111...

01001000 01100101 01101100 01101100 01101111

'H' 'e' 'l' 'l' 'o'

Fixed number of elements

'H' 'e' 'l' 'l' 'o'

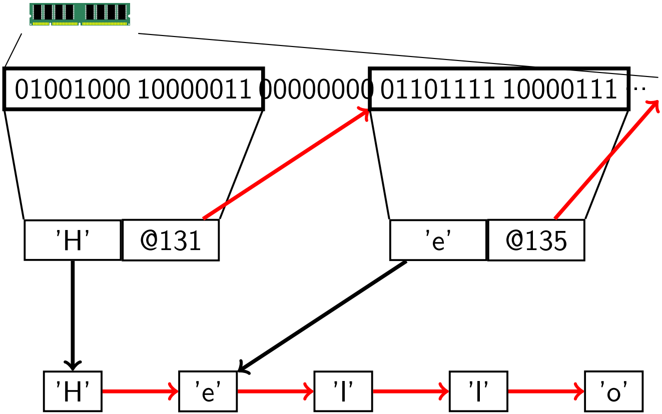
Fixed element size

OpenClipArt: `https://freesvg.org/random-access-computer-memory-ram-vector-image`

Array

- `public E get(int idx)`
    - Return bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$
    - $\theta(1)$ (if we treat bPE as a constant)
- `public void set(int idx, E value)`
    - Update bytes $\texttt{bPE} \times \mathit{idx}$ to $\texttt{bPE} \times (\mathit{idx} + 1) - 1$
    - $\theta(1)$ (if we treat bPE as a constant)
- `public int size()`
    - Return `size`
    - $\theta(1)$

CSE 220 Crossover 2: List Harder



OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

LinkedList

- `public E get(int idx)`
    - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
    - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
    - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
    - $\theta(idx)$, $O(N)$
- `public int size()`
    - Start at `head`, and move to the `next` element until you reach the end. Return the number of steps taken.
    - $\theta(N)$

Linked Lists' `size`

Can we do better?

Store `size`

```
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now? ($\theta(1)$)
- How expensive is it to maintain `size`? (Extra $\theta(1)$ work on insert/remove).

Store `size`

```
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now?
  ($\theta(1)$)
- How expensive is it to maintain `size`?
  (Extra $\theta(1)$ work on insert/remove).

Storing redundant information can reduce complexity.

Enumeration

```
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  while(node.isPresent())
  {
    int value = node.get().value;
    total += value;
    node = node.get().next;
  }
  return total;
}
```

Enumeration

This code is specialized for `LinkedLists`

- We can't re-use it for an `ArrayList`.
- If we change `LinkedList`, the code breaks.

How do we get code that is both fast and general?

- We need a way to represent a reference to the `idx`'th element of a list.

ListIterator

```
public interface ListIterator<E>
{
  public boolean hasNext();
  public E next();
  public boolean hasPrevious();
  public E previous();
  public void add(E value);
  public void set(E value);
  public void remove();
}
```

Linked Lists

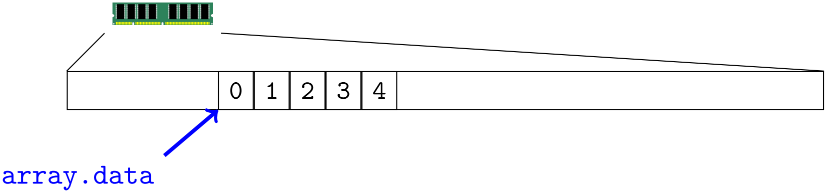
Access list element by index: $O(N)$

Access list element by reference (iterator): $O(1)$

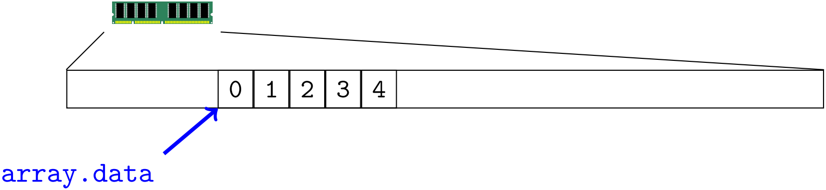
The List ADT

```
public interface List<E>
  extends Sequence<E> // Everything a sequence has, and...
{
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

Array `add(idx, value)`

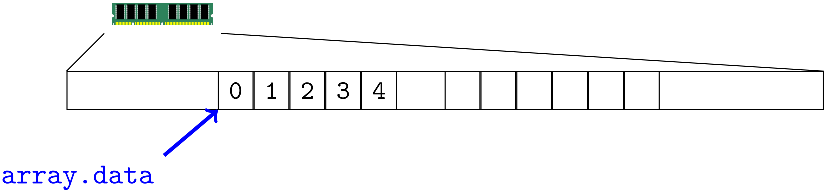


Array `add(idx, value)`



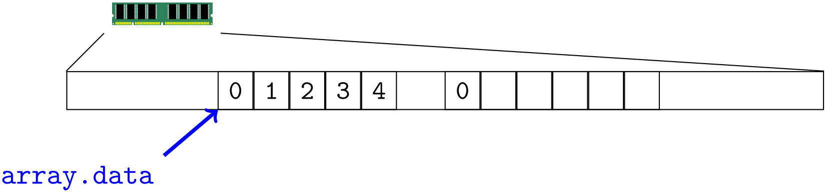
`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



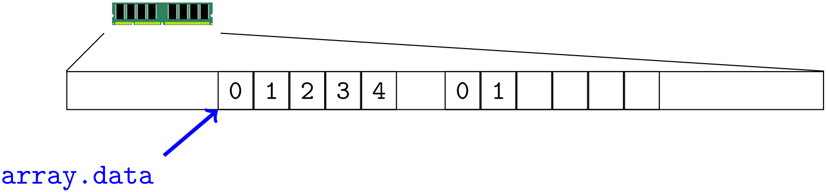
```
array.add(idx= 2, value= 5)
```

Array `add(idx, value)`



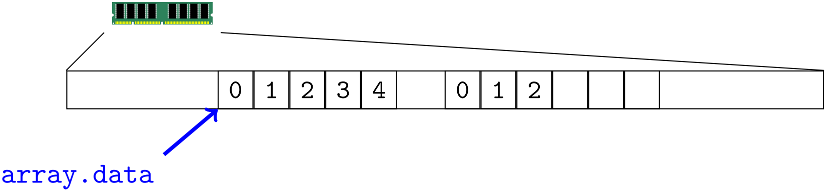
`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



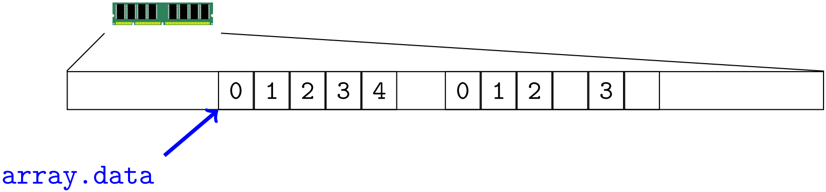
`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



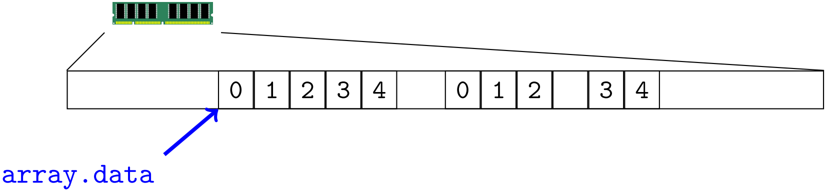
`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



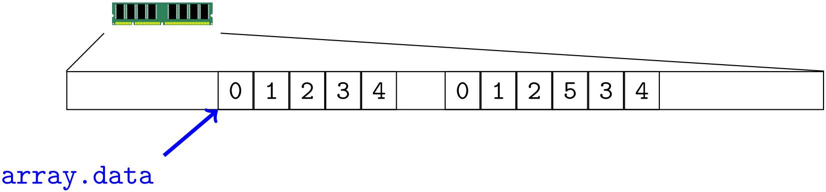
```
array.add(idx= 2, value= 5)
```

Array `add(idx, value)`



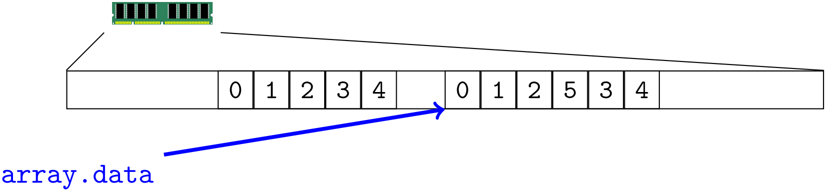
`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



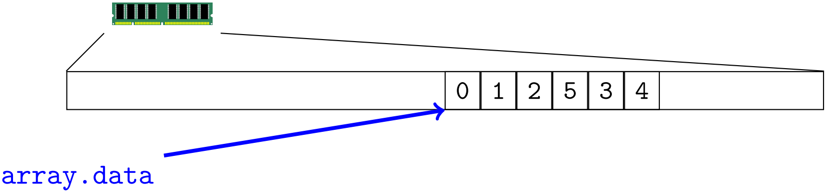
```
array.add(idx= 2, value= 5)
```

Array `add(idx, value)`



`array.add(idx= 2, value= 5)`

Array `add(idx, value)`



`array.add(idx= 2, value= 5)` ← $\theta(N)$

Idea 1

Idea: Allocate more memory than we need.

ArrayList

Start with a capacity of 2.

1. $\theta(1)$ (size now 1)
2. $\theta(1)$ (size now 2)
3. $2 \cdot \theta(1)$ (capacity now 4; size now 3)
4. $\theta(1)$ (size now 4)
5. $4 \cdot \theta(1)$ (capacity now 8; size now 5)
6. $\theta(1)$ (size now 6)
7. $\theta(1)$ (size now 7)
8. $\theta(1)$ (size now 8)
9. $8 \cdot \theta(1)$ (capacity now 16; size now 9)

...8 more operations before next $\theta(N)$

...16 more operations before next $\theta(N)$

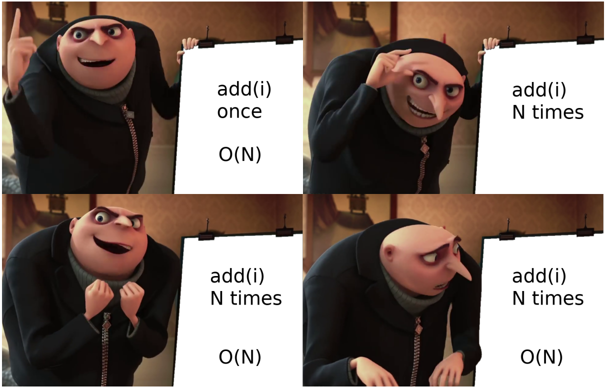
ArrayList

- 2 insertions at $\theta(1)$
- $2 \cdot \theta(1)$ plus 2 insertions at $\theta(1)$ (up to capacity of 4)
- $4 \cdot \theta(1)$ plus 4 insertions at $\theta(1)$ (up to capacity of 8)
- $8 \cdot \theta(1)$ plus 8 insertions at $\theta(1)$ (up to capacity of 16)
- $16 \cdot \theta(1)$ plus 16 insertions at $\theta(1)$ (up to capacity of 32)
- $32 \cdot \theta(1)$ plus 32 insertions at $\theta(1)$ (up to capacity of 64)
- ...

What's the pattern?
($2^i \cdot \theta(1)$ copy on the 2^i'th insertion)

For N insertions, how many copies do we perform?
($\log_2(N)$)

Huh?



Despicable Me; ©2010 Universal Pictures

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } capacity > size \\ \theta(N) & \textbf{otherwise} \end{cases}$$

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } capacity > size \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } \textit{capacity} > \textit{size} \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } \textit{capacity} > \textit{size} \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.
    - ... rarely enough (with doubling) that the expensive call amortizes over the cheap calls.

LinkedList vs ArrayList

```
for(i = 0; i < N; i++)
{
  list.add(i);
}
```

	`LinkedList`	`ArrayList`
`add(i)` once	$O(1)$	$O(N)$
`add(i)` N times	$O(N)$	$O(N)$

LinkedList vs ArrayList

```
for(i = 0; i < N; i++)
{
  list.add(i);
}
```

	`LinkedList`	`ArrayList`
`add(i)` once	$O(1)$	$O(N)$
`add(i)` N times	$O(N)$	$O(N)$

`ArrayList.add(i)` behaves like it's $O(1)$, but only when it's in a loop.

Amortized Runtime

- The tight upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

- The tight upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

- The tight upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

- The tight upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

- The tight upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

- The tight unqualified upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

If $T(N)$ runs in amortized $O(f(N))$, then:

$$\sum_{i=0}^{N} T(N) = \quad N \cdot O(f(N)) = \quad O(N \cdot f(N))$$

Even if $T(N) \not\in O(f(N))$

Amortized Runtime

- **Unqualified Bounds**: Always true (no qualifiers)
- **Amortized Bounds**: Only valid in $\sum_{i=0}^{N} T(i)$
    - One call may be expensive, many calls average out cheap

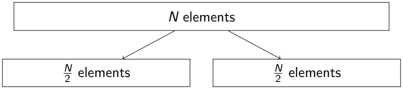
List Runtimes

Op	Array	ArrayList	Linked List (by idx)	Linked List (by iter)
`get(i)`	$\theta(1)$	$\theta(1)$	$\theta(i)$, $O(N)$	$\theta(1)$
`set(i,v)`	$\theta(1)$	$\theta(1)$	$\theta(i)$, $O(N)$	$\theta(1)$
`add(v)`	$\theta(N)$	Amm. $\theta(1)$	$\theta(1)$	$\theta(1)$
`add(i,v)`	$\theta(N)$	$\theta(N)$	$\theta(i)$, $O(N)$	$\theta(1)$
`remove(i)`	$\theta(N)$	$\theta(N)$	$\theta(i)$, $O(N)$	$\theta(1)$

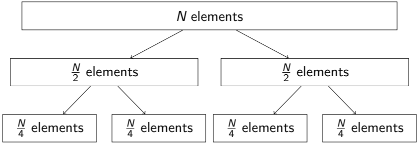
Merge Sort

N elements

Merge Sort



Merge Sort



N elements

$\frac{N}{2}$ elements

$\frac{N}{2}$ elements

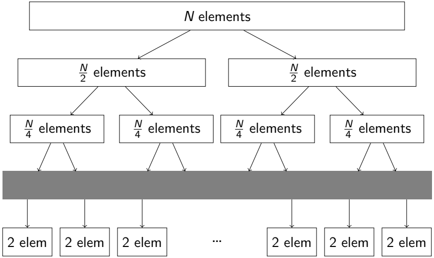
$\frac{N}{4}$ elements

$\frac{N}{4}$ elements

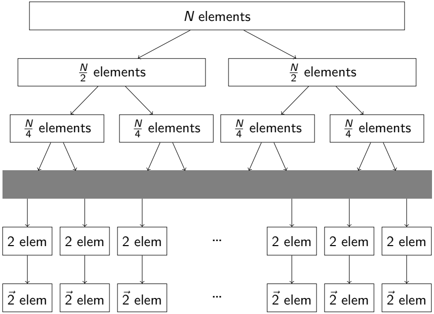
$\frac{N}{4}$ elements

$\frac{N}{4}$ elements

Merge Sort


N elements
$\frac{N}{2}$ elements
$\frac{N}{2}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
2 elem
2 elem
2 elem
...
2 elem
2 elem
2 elem

Merge Sort



N elements

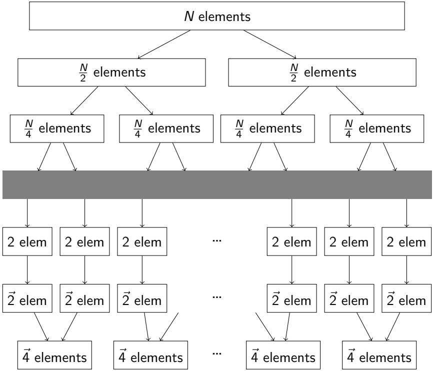
$\frac{N}{2}$ elements $\frac{N}{2}$ elements

$\frac{N}{4}$ elements $\frac{N}{4}$ elements $\frac{N}{4}$ elements $\frac{N}{4}$ elements

2 elem 2 elem 2 elem ... 2 elem 2 elem 2 elem

$\vec{2}$ elem $\vec{2}$ elem $\vec{2}$ elem ... $\vec{2}$ elem $\vec{2}$ elem $\vec{2}$ elem

Merge Sort



N elements

$\frac{N}{2}$ elements | $\frac{N}{2}$ elements

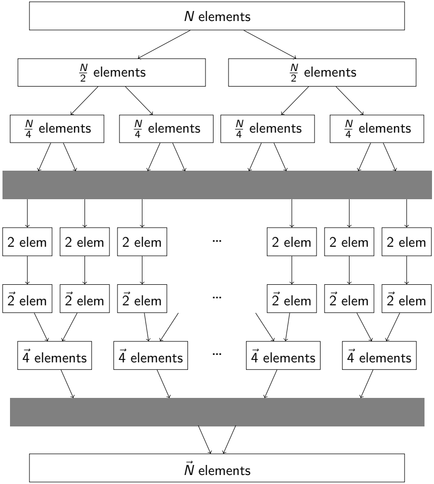
$\frac{N}{4}$ elements | $\frac{N}{4}$ elements | $\frac{N}{4}$ elements | $\frac{N}{4}$ elements

2 elem | 2 elem | 2 elem | ... | 2 elem | 2 elem | 2 elem

$\vec{2}$ elem | $\vec{2}$ elem | $\vec{2}$ elem | ... | $\vec{2}$ elem | $\vec{2}$ elem | $\vec{2}$ elem

$\vec{4}$ elements | $\vec{4}$ elements | ... | $\vec{4}$ elements | $\vec{4}$ elements

Merge Sort



N elements

$\frac{N}{2}$ elements | $\frac{N}{2}$ elements

$\frac{N}{4}$ elements | $\frac{N}{4}$ elements | $\frac{N}{4}$ elements | $\frac{N}{4}$ elements

2 elem | 2 elem | 2 elem | ... | 2 elem | 2 elem | 2 elem

$\vec{2}$ elem | $\vec{2}$ elem | $\vec{2}$ elem | ... | $\vec{2}$ elem | $\vec{2}$ elem | $\vec{2}$ elem

$\vec{4}$ elements | $\vec{4}$ elements | ... | $\vec{4}$ elements | $\vec{4}$ elements

$\vec{N}$ elements

Merge Sort

$$\left(\sum_{i=0}^{\log(N)-1} 2^i \cdot \theta(1)\right) + \left(\sum_{i=1}^{\log(N)-1} \theta(N)\right)$$

Merge Sort

$$\left(\sum_{i=0}^{\log(N)-1} 2^i \cdot \theta(1)\right) + \left(\sum_{i=1}^{\log(N)-1} \theta(N)\right)$$

$$\left(2^{\log(N)}\theta(1)\right) + (\log(N)\theta(N))$$

Merge Sort

$$\left(\sum_{i=0}^{\log(N)-1} 2^i \cdot \theta(1)\right) + \left(\sum_{i=1}^{\log(N)-1} \theta(N)\right)$$

$$\left(2^{\log(N)}\theta(1)\right) + (\log(N)\theta(N))$$

$$\theta(N) + \theta(N\log(N))$$

Merge Sort

$$\left(\sum_{i=0}^{\log(N)-1} 2^i \cdot \theta(1)\right) + \left(\sum_{i=1}^{\log(N)-1} \theta(N)\right)$$

$$\left(2^{\log(N)}\theta(1)\right) + (\log(N)\theta(N))$$

$$\theta(N) + \theta(N\log(N))$$

Merge Sort: $\theta(N\log(N))$

Merge Sort

$$\left(\sum_{i=0}^{\log(N)-1} 2^i \cdot \theta(1)\right) + \left(\sum_{i=1}^{\log(N)-1} \theta(N)\right)$$

$$\left(2^{\log(N)}\theta(1)\right) + (\log(N)\theta(N))$$

$$\theta(N) + \theta(N\log(N))$$

Merge Sort: $\theta(N\log(N))$

Bubble Sort: $\theta(N^2)$