Announcements

Upcoming Deadlines:

- Project 2 Phase 2 due March 5th.
- Hope it's been fun and not too stressful.

Lab:

- This week will be working on project 2 (free points).
- Labs next will be project 2 demos.
  - We will ask you to provide a gradescope link to the submission you want to demo.
  - If submitted after the project 2 phase 2 deadline, we'll deduct late points (but only from the demo part).

Time to start studying for midterm2:

- The lecture guides are pretty good from this point on, lots of problems.

CS61B

Lecture 19: Big O / Omega, Amortized Analysis

- Big O
- Big O vs. Big Theta
- Big Omega
- Amortized Analysis
- Extra: Empirical Analysis and Complexity Theory Preview

Big O Notation

Big Theta

We used Big Theta to describe the order of growth of a function.

function R(N)	order of growth
$N^3 + 3N^4$	$\Theta(N^4)$
$1/N + N^3$	$\Theta(N^3)$
$1/N + 5$	$\Theta(1)$
$Ne^N + N$	$\Theta(Ne^N)$
$40 \sin(N) + 4N^2$	$\Theta(N^2)$

We also used Big Theta to describe the rate of growth of the runtime of a piece of code.

- Example: binary search on N items has worst case runtime of $\Theta(\log N)$.

Big O

Whereas Big Theta can informally be thought of as something like “equals”, Big O can be thought of as “less than or equal”.

Example, the following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in O(N^4)$
- $N^3 + 3N^4 \in O(N^6)$
- $N^3 + 3N^4 \in O(N!)$
- $N^3 + 3N^4 \in O(N^{N!})$

Big Theta: Formal Definition (Visualization)

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Example: $40 \sin(N) + 4N^2 \in \Theta(N^2)$

- $R(N) = 40 \sin(N) + 4N^2$
- $f(N) = N^2$
- k1 = 3
- k2 = 5

Big O: Formal Definition (Visualization)

$$R(N) \in O(f(N))$$

means there exists positive constants k_2 such that:

$$R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Example: $40 \sin(N) + 4N^2 \in O(N^4)$

- $R(N) = 40 \sin(N) + 4N^2$
- $f(N) = N^4$
- $k2 = 1$

Big Theta vs. Big O

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$

Runtime Analysis Subtleties

Dup3 Runtime, A Trick Question: http://yellkey.com/table

Let R(N) be the runtime of the code below as a function of N.

- What is the order of growth of R(N)?
  - A. $R(N) \in \Theta(1)$
  - B. $R(N) \in \Theta(N)$
  - C. $R(N) \in \Theta(N^2)$
  - D. Something else.

```
public boolean dup3(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Dup3 Runtime, A Trick Question

Let R(N) be the runtime of the code below as a function of N.

- What is the order of growth of R(N)?
  - A. **$R(N) \in \Theta(1)$**
  - B. $R(N) \in \Theta(N)$
  - C. $R(N) \in \Theta(N^2)$
  - D. Something else.

```
public boolean dup3(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Dup4 Runtime, A Trick Question: http://yellkey.com/firm

Let R(N) be the runtime of the code below as a function of N.

- What is the order of growth of R(N)?
  - A. $R(N) \in \Theta(1)$
  - B. $R(N) \in \Theta(N)$
  - C. $R(N) \in \Theta(N^2)$
  - D. Something else.

```
public boolean dup4(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = i + 1; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Dup4 Runtime, A Trick Question

Let R(N) be the runtime of the code below as a function of N.

- What is the order of growth of R(N)?
  - A. $R(N) \in \Theta(1)$
  - B. $R(N) \in \Theta(N)$
  - C. $R(N) \in \Theta(N^2)$
  - **D. Something else (depends on the input).**

```
public boolean dup4(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = i + 1; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

The Limitations of Big Theta

Big Theta expresses exact order of growth for runtime in terms of N.

- If runtime depends on more factors than just N, may need different Big Theta for every interesting condition.

Example, using Big Theta requires us to qualify what we're talking about exactly:

- The best case runtime of dup4 is $\Theta(1)$.
- The worst case runtime of dup4 is $\Theta(N^2)$.

Or we can just use big O and avoid qualifying our statement at all.

- The runtime of dup4 is $O(N^2)$.

Question: http://yellkey.com/herself

Which statement gives you more information about a hotel?

A. The most expensive room in the hotel is $639 per night.
B. Every room in the hotel is less than or equal to $639 per night.

Question: Link TBA

Which statement gives you more information about a hotel?

A. The most expensive room in the hotel is $639 per night.

B. Every room in the hotel is less than or equal to $639 per night.

Most expensive room: $639/nt

THE RITZ-CARLTON
LAKE TAHOE

All rooms <= $639/nt

THE RITZ-CARLTON
LAKE TAHOE

(A nice place to stay!)

Question: http://yellkey.com/contain

Which statement gives you more information about the runtime of a piece of code?

A. The worst case runtime is $\Theta(N^2)$.
B. The runtime is $O(N^2)$.

Question: Link TBA

Which statement gives you more information about the runtime of a piece of code?

A. The worst case runtime is $\Theta(N^2)$.

B. The runtime is $O(N^2)$.

Runtime is $\Theta(N^2)$ in the worst case.

```
public boolean dup4(int[] a) {
      int N = a.length;
      for (int i = 0; i < N; i += 1) {
      for (int j = i + 1; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true; ...
```

Runtime is $O(N^2)$

```
public static void printLength(int[] a) {
    System.out.println(a.length);
}
```

```
public boolean dup4(int[] a) {
      int N = a.length;
      for (int i = 0; i < N; i += 1) {
      for (int j = i + 1; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true; ...
```

Big Theta vs. Big O

In the real world, Big O is often used where Big Theta would be more informative. Example: saying “mergesort is O(N log N)”.

- True statement, but not as strong as “mergesort is Θ(N log N)”.

Similar to how “They ran a mile in 3 minutes and 35 seconds” is more informative than “They ran a mile in less than 6 minutes”.

Despite lower precision, Big O is used way more often in conversation.

- Important: Big O does not mean “worst case”! Often abused to mean this.

The Usefulness of Big O

Big O is still a useful idea:

- Allows us to make simple blanket statements, e.g. can just say “binary search is O(log N)” instead of “binary search is Θ(log N) in the worst case”.
- Sometimes don’t know the exact runtime, so use O to give an upper bound.
  - Example: Runtime for finding shortest route that goes to all world cities is $O(2^N)^*$. There might be a faster way, but nobody knows one yet.
- Easier to write proofs for Big O than Big Theta, e.g. finding runtime of mergesort, you can round up the number of items to the next power of 2 (see A level study guide problems). A little beyond the scope of our course.

*: Under certain assumptions and constraints not listed.

Big Omega

Big O

Whereas Big Theta can informally be thought of as something like “equals”, Big Omega can be thought of as “greater than or equal”.

Example, the following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in \Omega(N^4)$
- $N^3 + 3N^4 \in \Omega(N^3)$
- $N^3 + 3N^4 \in \Omega(\log N)$
- $N^3 + 3N^4 \in \Omega(1)$

Big Omega: Formal Definition (Visualization)

$$R(N) \in \Omega(f(N))$$

means there exists positive constant k_1 such that:

$$k_1 \cdot f(N) \leq R(N)$$

for all values of N greater than some N_0.

i.e. very large N

Example: $40 \sin(N) + 4N^2 \in \Omega(N)$

- $R(N) = 40 \sin(N) + 4N^2$
- $f(N) = N$
- $k1 = 20$

Big Theta, Big O, and Big Omega

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N).	$\Omega(N^2)$	$N^2/2$ $2N^2$ e^N

Why Use Big Omega?

Two common uses for Big Omega:

- Very careful proofs of Big Theta runtime.
  - If R(N) = O(f(N)) and R(N) = Ω(f(N)), then R(N) = Θ(f(N)).
  - Sometimes it's easier to show O and Ω separately.
  - If you felt like our runtime proofs for Mergesort and Binary Sort were slightly shady, this is how you can make them truly robust.
  - We won't be doing this in 61B.

- Providing lower bounds for the hardness of a problem.
  - We'll do this in the last two weeks of the course.
  - Example: The time to find whether an array has duplicates is Ω(N) in the worst case for ANY algorithm (have to actually look at everything).

Amortized Analysis (Intuitive)

Grigometh's Tribute (this will relate to the class later)

Grigometh offers you the ability to appear to horses in their dreams. However, he requires you to pay tribute by placing hay in an urn, and gives you two choices:

- Choice 1: Every day, Grigometh will eat 3 pieces of hay from the urn.
- Choice 2: Grigometh will eat exponentially more hay over time as follows:
  - At the end of day 1, he will eat one piece of hay.
  - At the end of day 2, two additional pieces (totaling 3)
  - At the end of day 4, four additional pieces (totaling 7).
  - At the end of day 8, eight pieces (totaling 15).

C_i : consumption on day i

Let a_i be the amount of hay that we place in the urn on day i. For each choice, give a_i such that Grigometh is always satisfied.

Grigometh's Tribute (this will relate to the class later)

Grigometh offers you the ability to appear to horses in their dreams. However, he requires you to pay tribute by placing hay in an urn, and gives you two choices:

- Choice 1: Every day, Grigometh will eat 3 pieces of hay form the urn.
- Choice 2: Grigometh will eat exponentially more hay over time as follows:
  - At the end of day 1, he will eat one piece of hay.
  - At the end of day 2, two additional pieces (totaling 3)
  - At the end of day 4, four additional pieces (totaling 7).
  - At the end of day 8, eight pieces (totaling 15).

C_i : consumption on day i

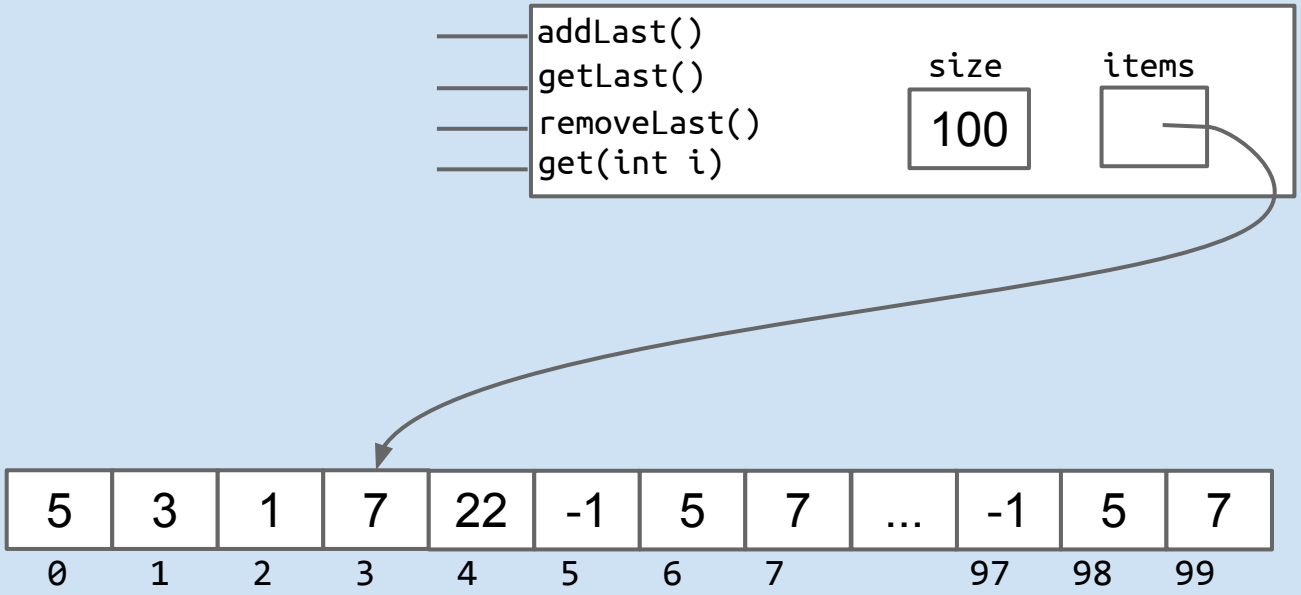
a_i : added on day i

Can satisfy EITHER of these requirements by simply adding three pieces of hay every day. The two choices are equivalent (within a constant factor).

- Punchline: Grigometh's consumption per day is effectively constant.

The Mighty (?) AList

Key Idea: Use some subset of the entries of an array.



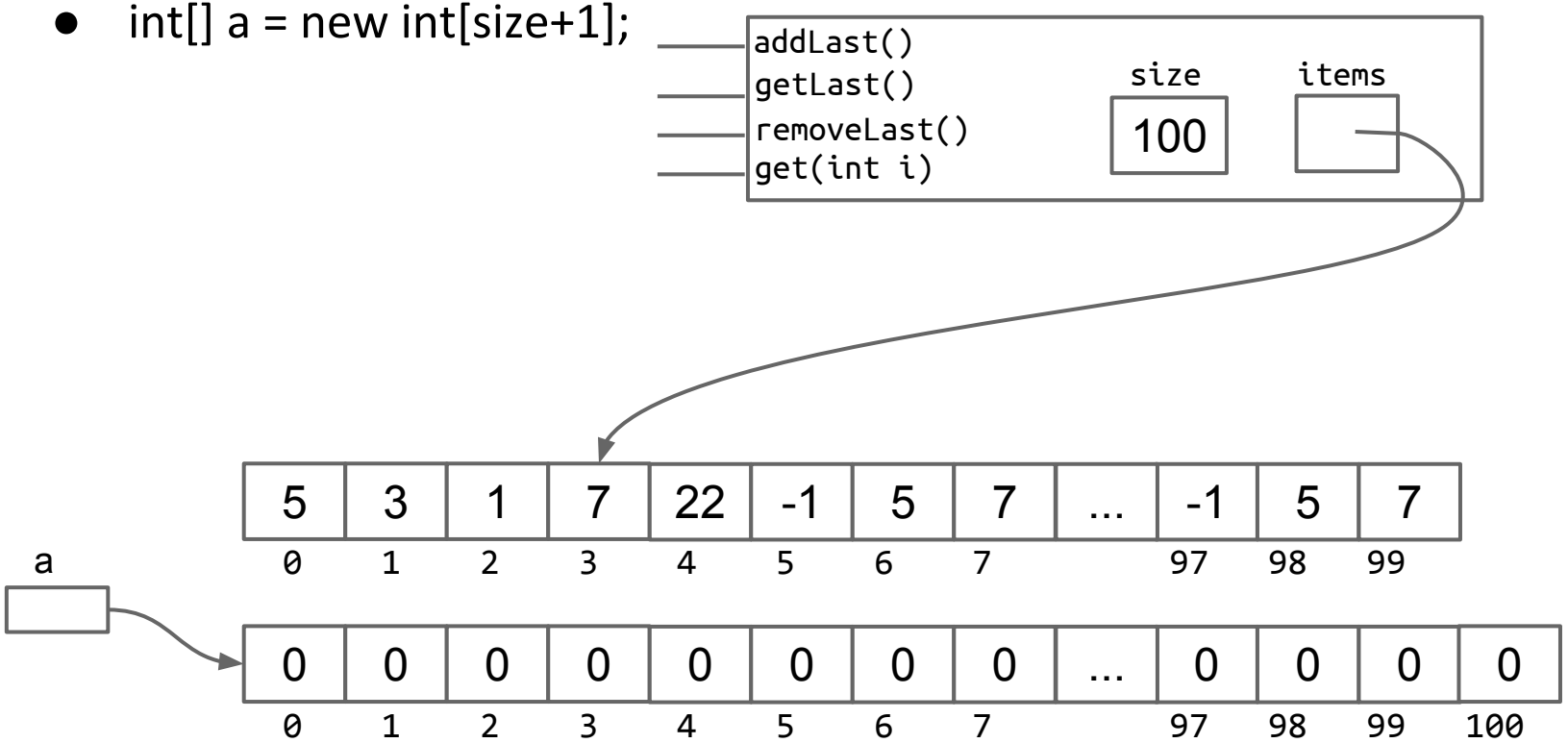
What happens if we insert into the AList above? What should we do about it?

Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];

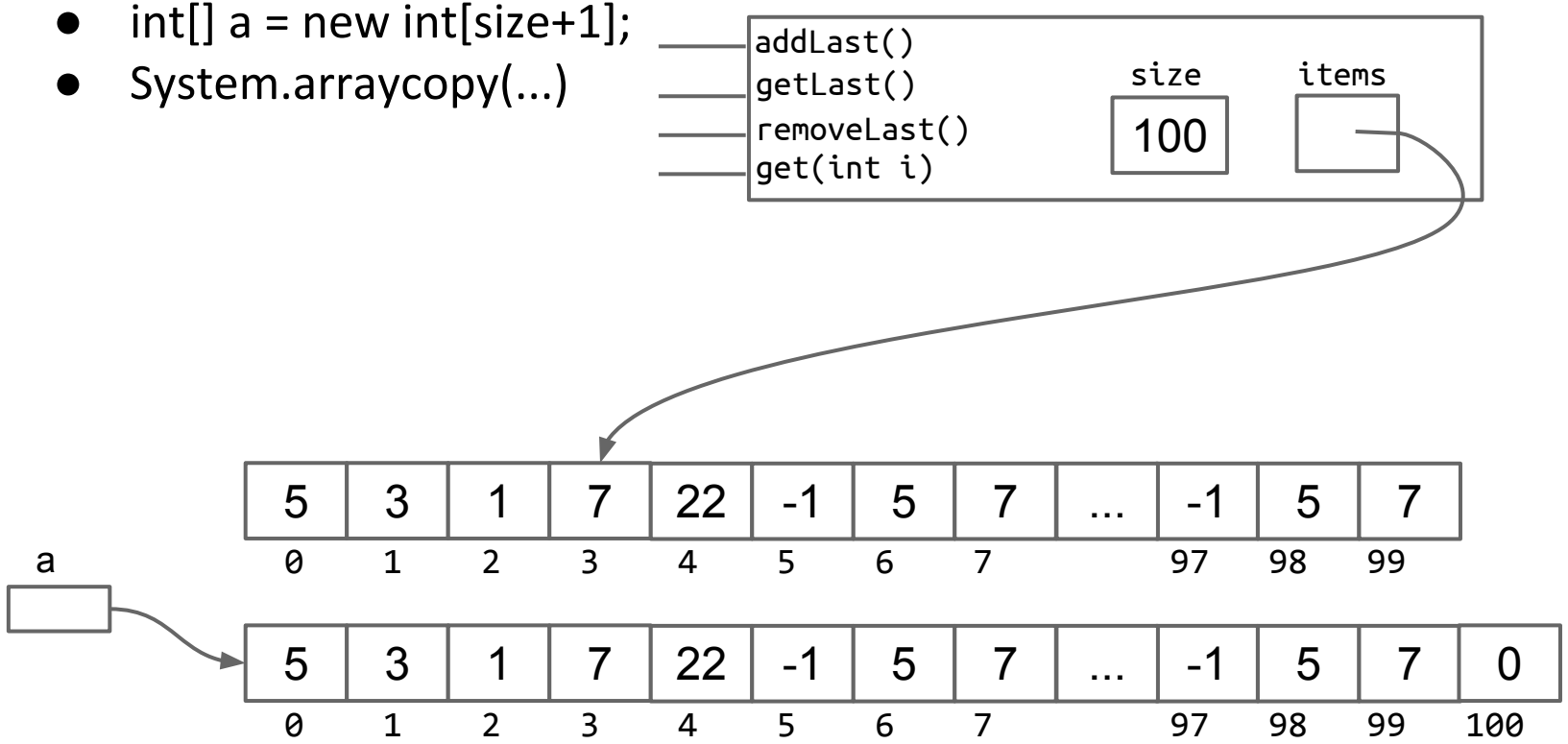


Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)

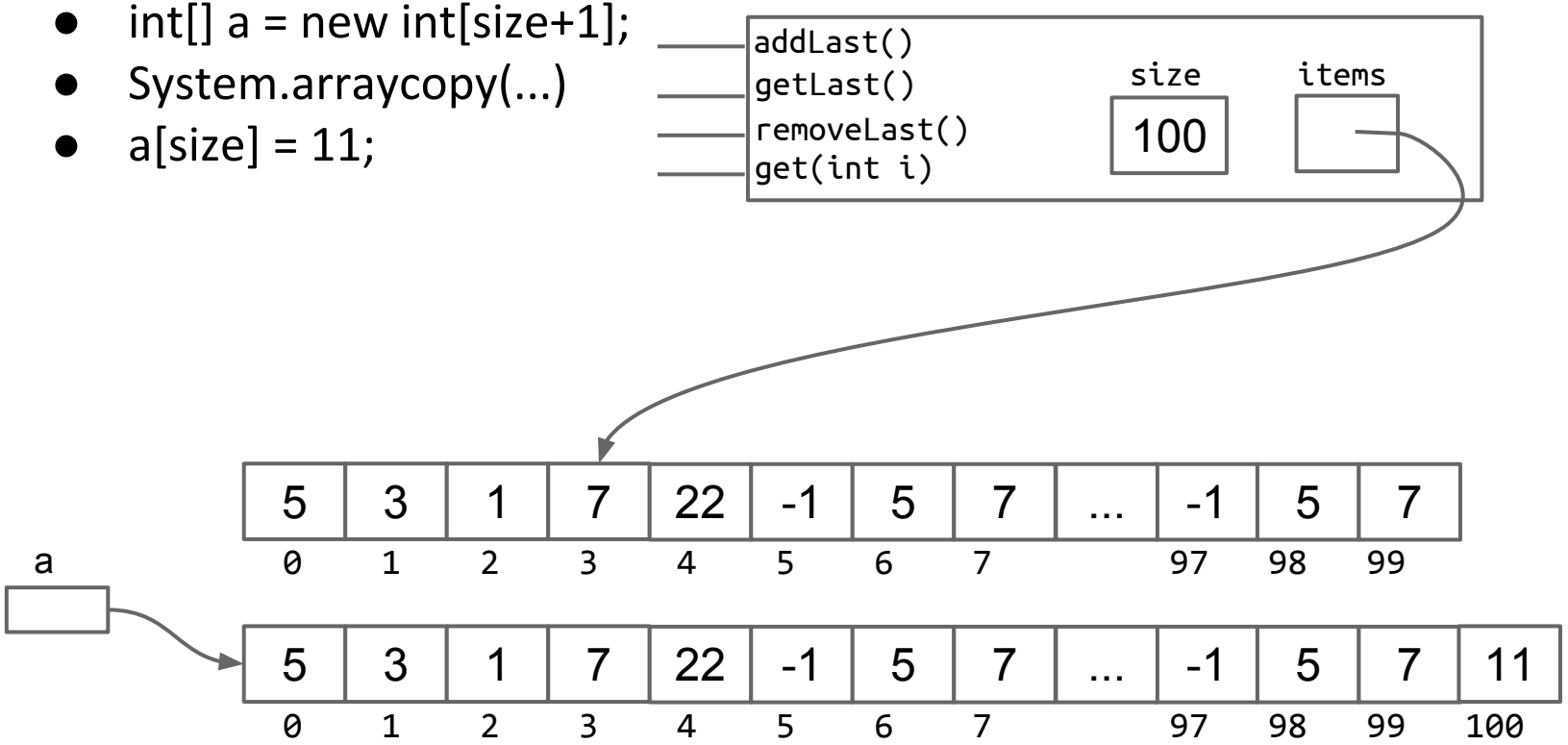


Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;

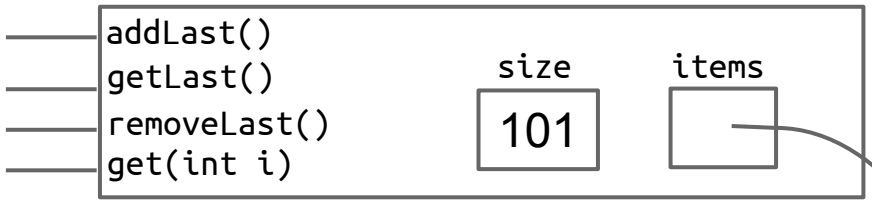


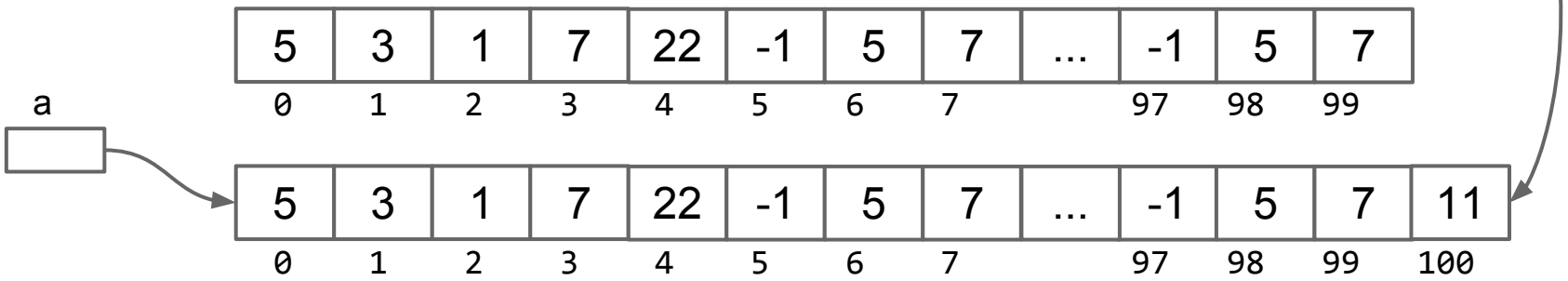
Array Resizing

size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;
- items = a; size +=1;





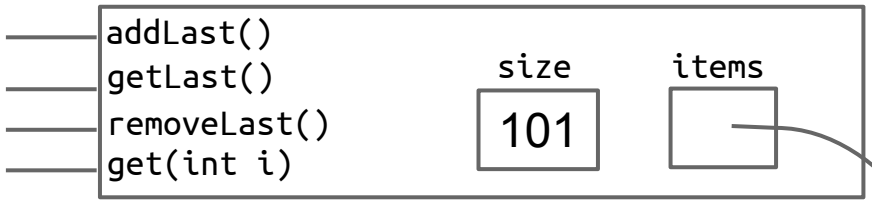
Array Resizing

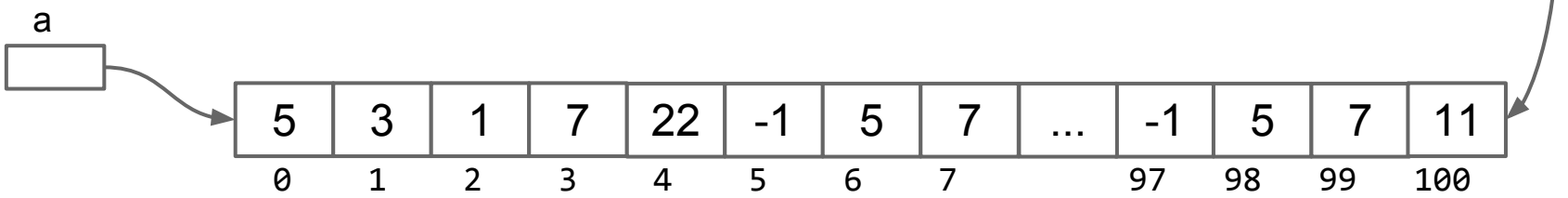
size==items.length

When the array gets too full, e.g. addLast(11), just make a new array:

- int[] a = new int[size+1];
- System.arraycopy(...)
- a[size] = 11;
- items = a; size +=1;

We call this process “resizing”





Geometric Resizing

Geometric resizing is much faster: Today we'll learn why.

```
public void addLast(int x) {
  if (size == items.length) {
     resize(size + RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

← Unusably bad.

Great performance. →

This is how the Python list is implemented.

```
public void addLast(int x) {
  if (size == items.length) {
     resize(size * RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

Geometric Array Resizing (Intuitive)

Resizes to accommodate additional entries.

`ArrayList`

- When the array inside the ArrayList is full, double in size.
- Most add operations are constant time, but some are very expensive.

```
public void add(T x) {
  if (size == items.length) {
     resize(size * 2);
  }
  items[size] = x;
  size += 1;
}
```

Given N items, cost of insert:

- Worst case: Θ(N)
- Average case: Θ(1) (unproven)

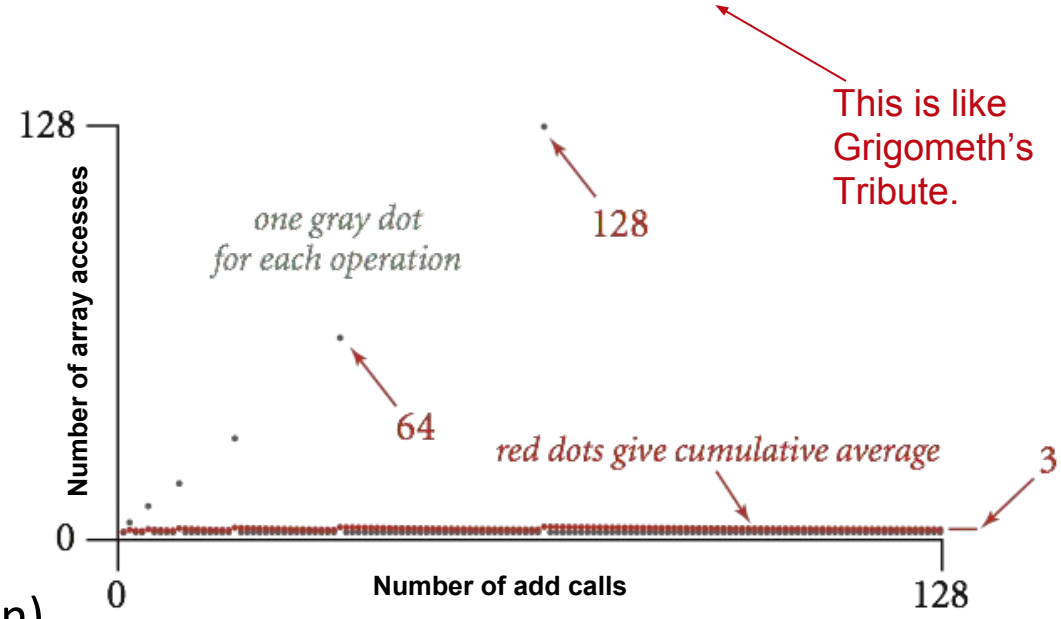


Figure from Algorithms, Sedgewick and Wayne

Amortized Analysis (Rigorous)

More Rigorous Amortized Analysis

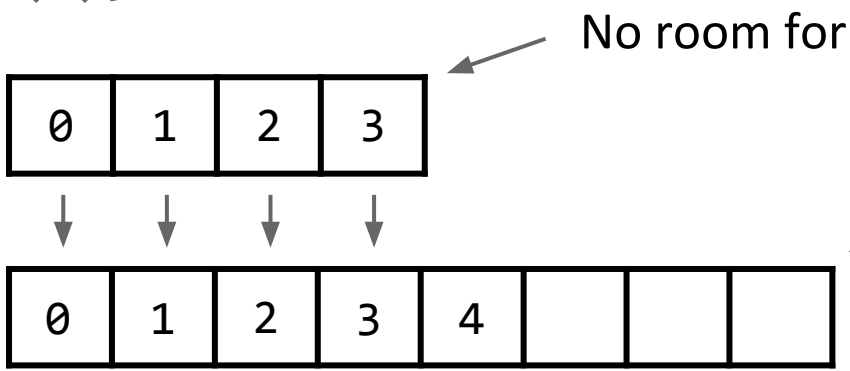
Next, we'll do a more rigorous amortized analysis where:

- We pick a cost model.
- We compute the amortized (a.k.a. average) cost of the ith insertion.
- We show that this cost is bounded above by a constant.
  - Similar to Grigometh's Tribute where the amount of hay we needed to provide was bounded above by 3.

Cost Model: Array accesses

Consider the cost (in array accesses) for the 5th insert into an ArrayList:

```
ArrayList<Integer> x = new ArrayList<>();
...
x.add(4);
```



Create 8 element array.

Copy 4 values:

- 4 array **reads**
- 4 array **writes**

Write the new value (4)

Total array accesses: 9

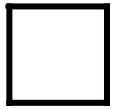
Other models possible, e.g.

- Can also count array creation.
- Can also count filling in of default values.

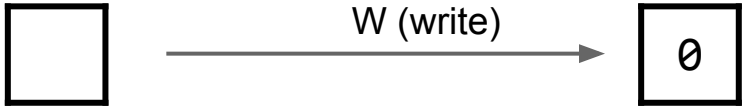
Analyses under these models yield same results.

Cost of The First Five Operations

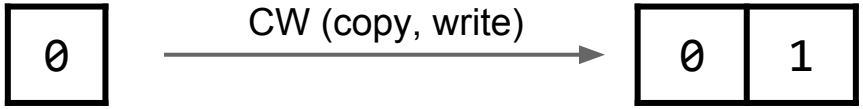
```
ArrayList<Integer> x = new ArrayList<Integer>(1);
```



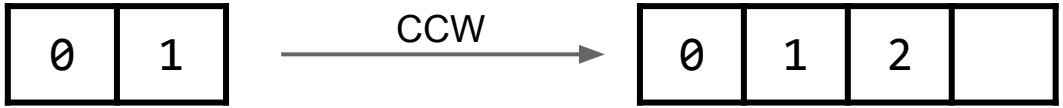
```
x.add(0);
```



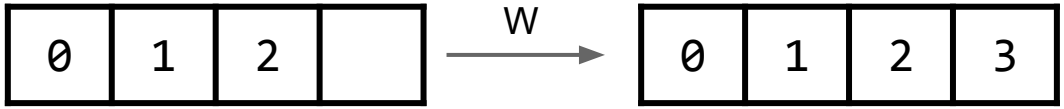

```
x.add(1);
```



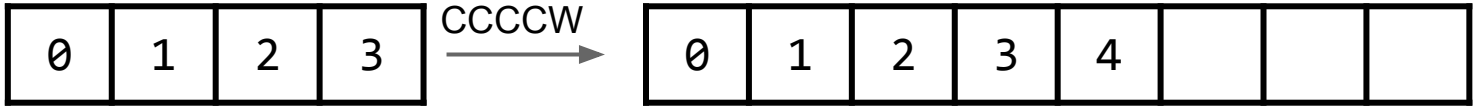

```
x.add(2);
```



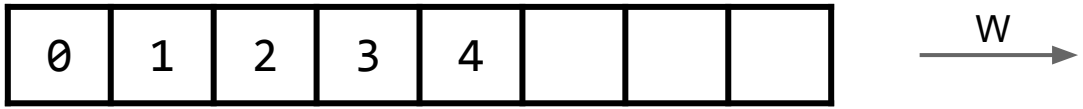

```
x.add(3);
```




```
x.add(4);
```




```
x.add(5);
```



Amortization of Runtime (starting from size 1)

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a[i] = cost (write cost)	1	1	1	1	1	1	1	1	1	1	1	1	1	1
resize cost(copy cost)	0	2	4	0	8	0	0	0	16	0	0	0	0	0
total cost for insert #	1	3	5	1	9	1	1	1	17	1	1	1	1	1
cumulative cost	1	4	9	10	19	20	21	22	39	40	41	42	43	44

Even though some elements cost linear time $\Theta(N)$, average cost of each insert is $\Theta(1)$.

- "Amortized" total cost seems to be about 44/14 array accesses / item.

Amortization of Runtime (starting from size 1)

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a[i] = cost (write cost)	1	1	1	1	1	1	1	1	1	1	1	1	1	1
resize cost(copy cost)	0	2	4	0	8	0	0	0	16	0	0	0	0	0
total cost for insert #	1	3	5	1	9	1	1	1	17	1	1	1	1	1
cumulative cost	1	4	9	10	19	20	21	22	39	40	41	42	43	44

Even though some elements cost linear time $\Theta(N)$, average cost of each insert is $\Theta(1)$.

- "Amortized" total cost seems to be about (44 / 14) = 3.14 accesses/item.
- How do we prove that the amortized (a.k.a. average) cost is constant?

Potentials and Amortized Cost Bounds

For operation i, choose an arbitrary "amortized cost" a_i. This cost may be more or less than the "actual cost" C_i of that operation.

- Let Φ_i be the potential at time i. The potential represents the cumulative difference between arbitrary amortized costs and actual costs over time.
  - $\Phi_{i+1} = \Phi_i + a_i - C_i$
- If we select a_i such that Φ_i is never negative, then amortized cost is an upper bound on actual cost.
- Using Grigometh's Tribute (choice 2) as an example, we see that using a constant "amortized cost" is sufficient to keep Grigometh fed ($\Phi_i > 0$).

actual cost, ci	1	2	0	4	0	0	0	0	8	0	0	0	0	0
amortized cost, ai	3	3	3	3	3	3	3	3	3	3	3	3	3	3
change in potential	2	1	3	-1	3	3	3	3	-5	3	3	3	3	3
potential Φi	2	3	6	5	8	11	14	17	12	15	18	21	24	27

Amortization of Runtime

withdrawal

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
total cost, ci	1	3	5	1	9	1	1	1	17	1	1	1	1	1
amortized cost, ai	5	5	5	5	5	5	5	5	5	5	5	5	5	5
change in potential	4	2	0	4	-4	4	4	4	-12					
potential Φi	0 4	6	6	10	6	10	14	18	6					

deposit

we pick whatever we want

total holdings

Goals for ArrayList: $a_i \in \Theta(1)$ and $\Phi_i \geq 0$ for all i.

- Cost for operations is 1 for non-powers of 2, and 2i+1 for powers of 2.
- For high cost ops, we'll need ~2i+1 in the bank. Have previous i/2 operations to reach this balance (e.g. deposits 5, 6, 7 and 8 to cover #8).

Amortization of Runtime

withdrawal

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
total cost, ci	1	3	5	1	9	1	1	1	17	1	1	1	1	1
amortized cost, ai	5	5	5	5	5	5	5	5	5	5	5	5	5	5
change in potential	4	2	0	4	-4	4	4	4	-12	4	4	4	4	4
potential Φi 0	4	6	6	10	6	10	14	18	6	10	14	18	22	26

deposit

we pick whatever we want

total holdings

Goals for ArrayList: $a_i \in \Theta(1)$ and $\Phi_i \geq 0$ for all i.

- Cost for operations is 1 for non-powers of 2, and 2i+1 for powers of 2.
- For high cost ops, we'll need ~2i+1 in the bank. Have previous i/2 operations to reach this balance (e.g. deposits 5, 6, 7 and 8 to cover #8).

Amortization of Runtime

withdrawal

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
total cost, ci	1	3	5	1	9	1	1	1	17	1	1	1	1	1
amortized cost, ai	5	5	5	5	5	5	5	5	5	5	5	5	5	5
change in potential	4	2	0	4	-4	4	4	4	-12	4	4	4	4	4
potential Φi 0	4	6	6	10	6	10	14	18	6	10	14	18	22	26

deposit

total holdings

The punchline (see CS170 for full rigor):

- On average, each op takes constant time. Arrays make good lists.
- Rigorously show by overestimating constant time of each operation, and proving that resulting potential is never < 0.

Summary

Big O and Big Omega are complementary concepts to Big Theta.

- Big Omega: Bounded below.
- Big O: Bounded above.
- Big Theta: Bounded above AND below.
- Can think of Big Theta as “equals”, of Big O as “less than or equals”, and Big Omega as “greater than or equals.”
- Common conceptual errors:
  - Big O does NOT mean worst case.
  - Big Omega does NOT mean best case.

Amortized Analysis: Provides a way to prove the average cost of operations.

- Key idea: Choosing a_i so that Φ_i stays positive.

Empirical Analysis (Extra)

In prior semesters, this topic was required, but was made optional in 2018.

Tilde Notation

Function f(n) ~ g(n) iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

That is, tilde notation ignores lower order terms, but takes into account constant factors of highest order terms.

Examples:

g(n)	Family members: f(n) ~ g(n)	Non-family members
$8n^2$	$8n^2 + 4n + 2$ $8n^2 + 2\ \log(3n) + \text{sqrt}(n)$ $8n^2$	$7n^2 + 3\ \log(n)$ $8n^{2.4} + 4n$
42*n*log(n) + 1.5*n	42*n*log(n) + n - log(n) 42*n*log(n)	24*n*log(n) 42*n*log(n) + n^2

Empirical Analysis

3-Sum. Given N distinct integers, how many triples sum to zero?

- Example: `int a[] = {30, -40, -20, -10, 40, 0, 10, 5};`

a[i]	a[j]	a[k]	sum
30	-40	10	0
30	-20	-10	0
-40	40	0	0
-10	0	10	0

Context: Related to problems in computational geometry,

i.e. Given a set of lines in the plane, are there three that meet in a point?

The Naive 3-SUM Solution

```
public static int count(int[] a) {
   int N = a.length;
   int count = 0;
   for (int i = 0; i < N; i += 1)
     for (int j = i + 1; j < N; j += 1)
       for (int k = j + 1; k < N; k += 1)
         if (a[i] + a[j] + a[k] == 0)
           count += 1;
   return count;
}
```

As a function of N, what is the order of growth of the runtime of the code above?

The Naive 3-SUM Solution

```
public static int count(int[] a) {
   int N = a.length;
   int count = 0;
   for (int i = 0; i < N; i += 1)
     for (int j = i + 1; j < N; j += 1)
       for (int k = j + 1; k < N; k += 1)
         if (a[i] + a[j] + a[k] == 0)
           count += 1;
   return count;
}
```

As a function of N, what is the order of growth of the runtime of the code above?

- Looks like $\Theta(N^3)$

Timing Experiments: Empirical Analysis of ThreeSum

See ThreeSum.java

Very Crude Curve Fitting: http://shoutkey.com/all

To estimate runtime, we'll assume R(N) asymptotically approaches aN^b

Assuming $R(N) \sim aN^b$, given the data to the right, estimate b.

a. 1
b. sqrt(2)
c. 2
d. 3
e. 8

N	time
500	0.08 seconds
1000	0.70 seconds
2000	1.22 seconds
4000	5.15 seconds
8000	43.62 seconds

Very Crude Curve Fitting

To estimate runtime, we'll assume R(N) asymptotically approaches aN^b

Assuming $R(N) \sim aN^b$, given the data to the right, estimate b.

$R(8000)/R(4000) = (a\ 8000^b) \ / \ (a\ 4000^b)$

$43.62/5.15 = (8000 / 4000)^b$

$\lg(43.62/5.15) = b * \lg(8000/4000) = b$

b = 3.08

N	time
500	0.08 seocnds
1000	0.70 seconds
2000	1.22 seconds
4000	5.15 seconds
8000	43.62 seconds

Very Crude Curve Fitting

To estimate runtime, we'll assume R(N) asymptotically approaches aN^b

Assuming $R(N) \sim aN^b$, give a and b.

$R(N) \sim aN^b$

b = 3.08

43.62 = a * 8000^(3.08)

lg(43.62) = lg(a*8000) * 3.08

lg(43.62)/3.08 =

N	time
500	0.08 seocnds
1000	0.70 seconds
2000	1.22 seconds
4000	5.15 seconds
8000	43.62 seconds

Very Crude Curve Fitting

To estimate runtime, we'll assume R(N) asymptotically approaches aN^b

Assuming $R(N) \sim aN^b$, give a and b.

$R(N) \sim aN^b$

$5.15 = a(4000)^{3.08}$ a =4.14440755e-11

N	time
500	0.08 seocnds
1000	0.70 seconds
2000	1.22 seconds
4000	5.15 seconds
8000	43.62 seconds

Program Measurement is an Empirical Science

Assumed that R(N) asymptotically approaches aN^b or equivalently that $R(N) \sim aN^b$

- Data provides a hypothesis: $R(N) \sim 4.14440755e\text{-}11 * N^{3.08}$
- Can test this hypothesis:
  - Prediction: R(3000) = 2.12 seconds.
  - Actual (when I ran it at home): 2.65 seconds
  - Why the discrepancy?

Runtimes Are Not Always Predictable

If we perform our test in a slightly different manner, we observe strange results (see ThreeSumBizarre.java)

Many possible sources of runtime strangeness:

- High system load
- Branch prediction
- Caching
- Just-in-time compilation (main suspect for ThreeSumBizarre)
- Large constants for low order terms: $aN + N^2$ looks linear for large a.

Complexity Theory (Sneak Preview (Extra))

3Sum: Can We Do Better Than $\Theta(N^3)$?

```
public static int count(int[] a) {
   int N = a.length;
   int count = 0;
   for (int i = 0; i < N; i += 1)
     for (int j = i + 1; j < N; j += 1)
       for (int k = j + 1; k < N; k += 1)
         if (a[i] + a[j] + a[k] == 0)
           count += 1;
   return count;
}
```

Code above works, but is slow: $\Theta(N^3)$

- Extra problem in discussion this week: Develop an idea that can be used to solve 3Sum in $\Theta(N^2)$ time.

3Sum Theoretical Upper Bound: Can We Do Better Than $O(N^2)$?

We know that the optimal algorithm for 3Sum has a runtime that is $O(N^2)$.

- (Because either the discussion algorithm is optimal, or it is not).
- Can we do better?

Long conjectured that N^2 was as good as it gets!

- 2014: In "Threesomes, Degenerates, and Love Triangles", Gronlund and Pettie found algorithm with runtime: $O(n^2 \frac{(\log \log n)^2}{\log n})$

Ever so slightly faster than N^2 (no practical difference).

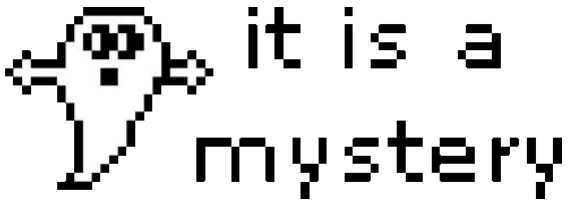
3Sum Theoretical Lower Bound: Can We Do Better Than Ω(N^2)?

Let R(N) be the runtime of the (unknown) optimal algorithm for 3Sum.

- Can we **lower bound** the runtime R(N)?
  - Does the optimal algorithm have a runtime $R(N) \in \Omega(1)$?
    - In other words: Is it at LEAST constant?
  - Does the optimal algorithm have a runtime $R(N) \in \Omega(N)$?
    - In other words: Is it at least linear?

Open theoretical question: Can we find some function f(N) that grows more quickly than N such that $R(N) \in \Omega(f(N))$, i.e. can we prove that 3Sum necessarily take more than linear time?

- Nobody knows!
- Lower bound proofs are tough (more later).



Asymptotics in Multiple Variables (Extra)

Citations

Image of Kilauea from
http://media-2.web.britannica.com/eb-media/33/91933-004-DAEEF82A.jpg

ThreeSum code (but not analysis) and ArrayList timing figure adapted from Algorithms (Sedgewick and Wayne).