CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Nine
19 July 2012

More on algorithmic analysis

Asymptotic performance analysis

- Asymptotic performance analysis is a coarse but useful means of describing and comparing the performance of algorithms as a function of the input size *n* when *n* gets large.
- Asymptotic analysis applies to both **time cost** and **space cost**.
- Asymptotic analysis hides details of timing (that we don't care about) due to:
  - Speed of computer.
  - Slight differences in implementation.
  - Programming language.

Mathematical formalism

- In order to justify approximating a time cost $T(n)=3n+3$ just as "$O(n)=n$", we need to define some mathematical notation:
  - We say a function $T(n)$ is big-O of another function $g(n)$ (i.e., $O(g(n))$ if there exist positive constants c and n_0 such that:
    for all $n > n_0$: $T(n) \leq c\, g(n)$

Mathematical formalism

- In order to justify approximating a time cost $T(n)=3n+3$ just as "$O(n)=n$", we need to define some mathematical notation:
    - We say a function $T(n)$ is big-O of another function $g(n)$ (i.e., $O(g(n))$ if there exist positive constants c and n_0 such that:
      for all $n > n_0$: $T(n) \leq c\, g(n)$

As long as *n* is "big enough", then *T*(*n*) will always be less than a constant multiple of *g*(*n*).

Mathematical formalism

- Example: consider $T(n)=3n-6$.
- If we pick $g(n)=n$, $n_0 = 0$ and $c = 4$, then:
- $T(n) = 3n-6 \leq 4n = c\ g(n)$ for all $n > n_0$
- Hence, we can write: "$T(n)$ is $O(g(n))$ where $g(n)=n$".
- More simply, we can write: "$T(n)$ is $O(n)$".

Mathematical formalism

- Note that, for $T(n)=3n-6$, we could also write $T(n) = O(n^2)$ because:
  - If we pick $n_0 = 10$ and $c = 1$, then:
  - $T(n) = 3n-6 \leq n^2 = c\ g(n)$ for all $n > n_0$
- The "*O*" notation gives an upper bound to the time cost T. It may not be a *tight* upper bound.

Mathematical formalism

- Note that, for $T(n)=3n-6$, we could also write $T(n) = O(n^2)$ because:
  - If we pick $n_0 = 10$ and $c = 1$, then:
  - $T(n) = 3n-6 \leq n^2 = c\, g(n)$ for all $n > n_0$
- The "*O*" notation gives an upper bound to the time cost T. It may not be a *tight* upper bound.
  - However, by convention, if we say "$T(n)$ is $O(g(n))$", then we pick $g(n)$ to be a tight bound on T. *

* This is achieved formally by also defining Ω, and θ notation.

Mathematical formalism

- Note that, for $T(n)=n^2+2n$, we could **not** write $T(n) = O(n)$ because there do **not** exist positive constants c and n_0 such that $T(n) \leq c\ g(n)$ for all $n > n_0$.

Different asymptotic costs

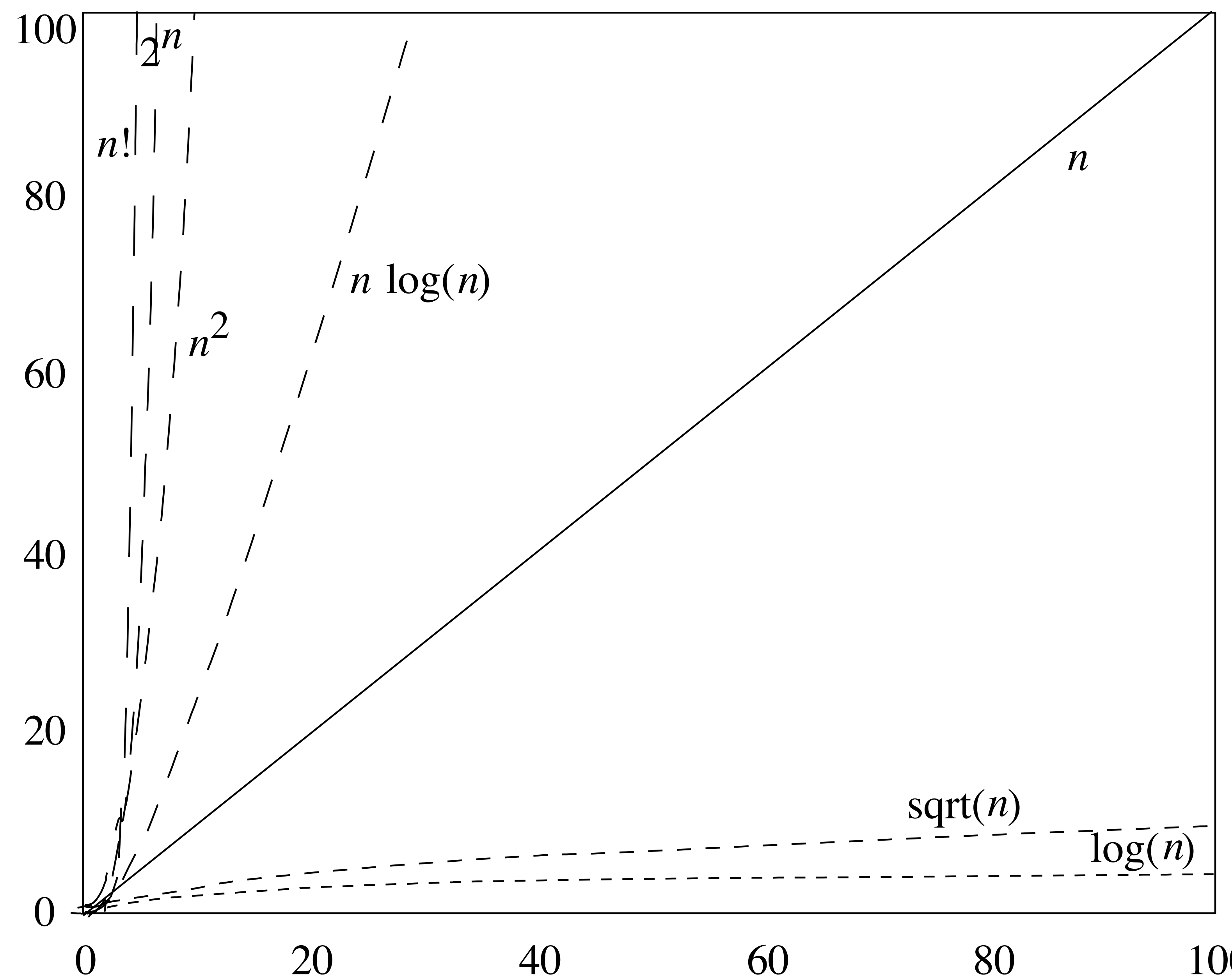


Figure 5.3 Long-range trends of common curves. Compare with Figure 5.2.

from Bailey (2007)

Exercises

- $T(n) = 2n^3 + 2n^4 - 3$
- $T(n) = 3n^2 - 3n + 17$
- $T(n) = 2 \log n$
- $T(n) = 3 \log n + 5n$

Exercises

- $T(n) = 2n^3 + 2n^4 - 3 = O(n^4)$
- $T(n) = 3n^2 - 3n + 17$
- $T(n) = 2 \log n$
- $T(n) = 3 \log n + 5n$

Exercises

- $T(n) = 2n^3 + 2n^4 - 3 = O(n^4)$
- $T(n) = 3n^2 - 3n + 17 = O(n^2)$
- $T(n) = 2 \log n$
- $T(n) = 3 \log n + 5n$

Exercises

- $T(n) = 2n^3 + 2n^4 - 3 = O(n^4)$
- $T(n) = 3n^2 - 3n + 17 = O(n^2)$
- $T(n) = 2 \log n = O(\log n)$
- $T(n) = 3 \log n + 5n$

Exercises

- $T(n) = 2n^3 + 2n^4 - 3 = O(n^4)$
- $T(n) = 3n^2 - 3n + 17 = O(n^2)$
- $T(n) = 2 \log n = O(\log n)$
- $T(n) = 3 \log n + 5n = O(n)$

Properties of asymptotic notation

- If $T(n) = U(n) + V(n)$,
  and if both $U(n) = O(g(n))$ and $V(n)=O(g(n))$,
  then $T(n) = O(g(n))$.
  - In other words, the sum of two functions that are both $O(g(n))$ is also $O(g(n))$.

Example 1 revisited

\# operations

```
// Assume grades.length > 0
float computeAverageGrade (float[] grades) {
  float sum = 0;                                      O(1)
  for (int i = 0; i < grades.length; i++) {           O(n)
    sum += grades[i];                                 O(n)
  }

  return sum / grades.length;                         O(1)
}
```

Total:
O(n)

Using asymptotic notation, the analysis becomes much simpler.

Example 3

operations

```
int someMethod (int[] numbers) {
  int sum = 0;
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      sum += numbers[i] * numbers[j];
    }
  }
  return sum;
}
```

Example 3

operations

```
int someMethod (int[] numbers) {
  int sum = 0;
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      sum += numbers[i] * numbers[j];
    }
  }
  return sum;
}
```

Total:
$O(n^2)$

Analysis of data structures

- Let's put these ideas into practice and analyze the performance of algorithms related to **`ArrayList`**:
  - **`add(o)`**, **`get(index)`**, **`find(o)`**, and **`remove(index)`**.
- As a first step, we must decide what the "input size" means.
  - What is the "input" to these algorithms?

Analysis of data structures

- Each of the methods (algorithms) above operates on the `_underlyingStorage` *and* either `o` or `index`.
  - `o` and `index` are always length 1 -- *their size cannot grow*.
  - However, the number of data in `_underlyingStorage` (stored in `_numElements`) will grow as the user adds elements to the `ArrayList`.
- Hence, we measure asymptotic time cost as a function of *n*, the number of elements stored (`_numElements`).

Adding to back of list

- What is the time complexity of this method?

```
class ArrayList<T> {
  private T[] _underlyingStorage;
  int _numElements;
  void addToBack (T o) {
    // Assume _underlyingStorage is big enough
    _underlyingStorage[_numElements] = o;
    _numElements++;
  }
  // ...
}
```

Adding to back of list

- What is the time complexity of this method?

Note that, for this method, the worst case, average case, and best case are all the same.

```
class ArrayList<T> {
  private T[] _underlyingStorage;
  int _numElements;
  void addToBack (T o) {
    // Assume _underlyingStorage is big enough
    _underlyingStorage[_numElements] = o;
    _numElements++;
  }
  // ...
}
```

O(1) -- the number of abstract operations does not depend on `_numElements`.

Retrieving an element

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  T get (int index) {
    return _underlyingStorage[index];
  }
}
```

Retrieving an element

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  T get (int index) {
    return _underlyingStorage[index];
  }
}
```

O(1).

Adding to front of list

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  void addToFront (T o) {
    // Assume _underlyingStorage is big enough
    for (int i = 0; i < _numElements; i++) {
      _underlyingStorage[i+1] = _underlyingStorage[i];
    }
    _underlyingStorage[i] = o;
    _numElements++;
  }
}
```

Adding to front of list

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  void addToFront (T o) {
    // Assume _underlyingStorage is big enough
    for (int i = 0; i < _numElements; i++) {
      _underlyingStorage[i+1] = _underlyingStorage[i];
    }
    _underlyingStorage[i] = o;
    _numElements++;
  }
}
```

We have to move everything over by 1.

$O(n)$.

Finding an element

- What is the time complexity of this method in the *best case*? *Worst case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element

- What is the time complexity of this method in the *best case*? *Worst case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

$O(1)$ in best case; $O(n)$ in worst case.

Adding *n* elements

- Now, let's consider the time complexity of doing *many adds in sequence*, starting from an empty list:

```
void addManyToFront (T[] many) {
  for (int i = 0; i < many.length; i++) {
    addToFront(many[i]);
  }
}
```

- What is the time complexity of **`addManyToFront`** on an array of size *n*?

Adding *n* elements

- To calculate the total time cost, we have to *sum* the time costs of the individual calls to `addToFront`.
  - Each call to `addToFront(o)` takes about time *i*, where *i* is the *current* size of the list. (We have to "move over" *i* elements by one step to the right.)

```
void addManyToFront (T[] many) {
  for (int i = 0; i < many.length; i++) {
    addToFront(many[i]);
  }
}
```

- Let $T(i)$ the cost of `addToFront` at iteration *i*: $T(0)=1$, $T(1)=2$, ..., $T(n-1)=n$.

Adding *n* elements

- Now we just have to add together all the *T*(*i*):

$$\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} (i+1) = \frac{n(n-1)}{2} = O(n^2)$$

- Note that we would get the same asymptotic bound even if we calculated the cost *T*(*i*) slightly differently, e.g., *T*(*i*)=3*i*+2:

$$\begin{aligned} \sum_{i=0}^{n-1} T(i) &= \sum_{i=0}^{n-1} (3i+2) \\ &= \sum_{i=0}^{n-1} 3i + \sum_{i=0}^{n-1} 2 \\ &= 3\sum_{i=0}^{n-1} i + 2n \\ &= 3\left(\frac{n(n-1)}{2}\right) + 2n \\ &= O(n^2) \end{aligned}$$

Finding an element

- What is the time complexity of this method in the *average case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element: average case

- Finding an exact formula for the *average case* performance can be tricky (if not impossible).
- In order to compute the average, or *expected*, time cost, we must know:
  - The *time cost $T(X_n)$* for a particular *input X* of size *n*.
  - The *probability* $P(X_n)$ of that input X.
  - The *expected time cost*, over all inputs X of size n, is then:

$$\text{AvgCaseTimeCost}_n = E[T(X_n)] = \sum_{X_n} P(X_n)T(X_n)$$

Finding an element: average case

- Finding an exact formula for the *average case* performance can be tricky (if not impossible).
- In order to compute the average, or *expected*, time cost, we must know:

  In this case, `x` consists of both the element `o` and the contents of `_underlyingStorage`.

  - The *time cost* $T(X_n)$ for a particular *input* X of size n.
  - The *probability* $P(X_n)$ of that input X.
  - The *expected time cost*, over all inputs X of size n, is then:

$$\text{AvgCaseTimeCost}_n = E[T(X_n)] = \sum_{X_n} P(X_n) T(X_n)$$

"*E*" for "Expectation"

Sum the time costs for all possible inputs, and weight each cost by how likely it is to occur.

Finding an element: average case

- In the `find(o)` method listed above, it is possible that the user gives us an `o` that is not contained in the list.
  - This will result in $O(n)$ time cost.
  - How "likely" is this event?
    - *We have no way of knowing* -- we could make an arbitrary assumption, but the result would be meaningless.
  - Let's *remove this case from consideration* and assume that `o` is always present in the list.
    - What is the average-case time cost *then*?

Finding an element: average case

- Even when we assume `o` is present in the list somewhere, we have no idea whether the `o` the user gives us will "tend to be at the front" or "tend to be at the back" of the list.
- However, here we can make a plausible assumption:
  - For an **`ArrayList`** of n elements, the probability that o is contained at index i is $1/n$.
    - In other words, `o` is equally likely to be in any of the "slots" of the array.

Finding an element: average case

- Given this assumption, we can finally make headway.
- Let's define *T*(*i*) to be the cost of the `find(o)` method as a function of *i*, the location in `_underlyingStorage` where `o` is located. What is *T*(*i*)?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element: average case

- Given this assumption, we can finally make headway.
- Let's define *T*(*i*) to be the cost of the `find(o)` method as a function of *i*, the location in `_underlyingStorage` where `o` is located. What is *T*(*i*)?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

$T(i)=i$

Finding an element: average case

- Now, we can re-write the expected time cost in terms of an arbitrary input X, as the expected time cost in terms of *where in the array the element* o *will be found.*

$$\begin{aligned}
\text{AvgCaseTimeCost}_n &= \sum_i P(i)T(i) && \text{Redefine } P(X_n) \text{ and } T(X_n) \text{ in terms of } P(i) \text{ and } T(i). \\
&= \sum_i \frac{1}{n} i && \text{Substitute terms.} \\
&= \frac{1}{n} \sum_i i && \text{Move } 1/n \text{ out of the summation.} \\
&= \frac{1}{n} \frac{n(n+1)}{2} && \text{Formula for arithmetic series.} \\
&= \frac{n+1}{2} && \text{The } n\text{'s cancel.} \\
&= O(n) && \text{Find asymptotic bound.}
\end{aligned}$$

Performance measurement.

Empirical performance measurement

- As an alternative to describing an algorithm's performance with a "number of abstract operations", we can also measure its time empirically using a clock.
- As illustrated last lecture, counting "abstract operations" can anyway hide real performance differences, e.g., between using `int[]` and `Integer[]`.

Empirical performance measurement

- There are also many cases where you don't know how an algorithm works internally.
    - Many programs and libraries are not open source!
        - You have to analyze an algorithm's performance as a black box.
            - "Black box" -- you can run the program but cannot see how it works internally.
- It may even be useful to *deduce* the asymptotic time cost by measuring the time cost for different input sizes.

Procedure for measuring time cost

- Let's suppose we wish to measure the time cost of algorithm *A* as a function of its input size *n*.
- We need to choose a set of values of *n* that we will test.
- If we make *n* too big, our algorithm *A* may never terminate (the input is "too big").
- If we make *n* too small, then *A* may finish so fast that the "elapsed time" is practically 0, and we won't get a reliable clock measurement.

Procedure for measuring time cost

- In practice, one "guesses" a few values for *n*, sees how fast *A* executes on them, and selects a range of values for *n*.
  - Let's define an array of different input sizes, e.g.: `int[] N = { 1000, 2000, 3000, ..., 10000 };`
- Now, for each input size `N[i]`, we want to measure *A*'s time cost.

Procedure for measuring time cost

- Procedure (draft 1):

Make sure to start and stop the clock as “tightly” as possible around the actual algorithm A.

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long startTime = getClockTime();
  A(X);  // Run algorithm A on input X of size N[i]
  final long endTime = getClockTime();

  final long elapsedTime = endTime - startTime;
  System.out.println(“Time for N[“ + i + “]: “ +
                         elapsedTime);
}
```

Procedure for measuring time cost

- The procedure would work fine if there were no variability in how long `A(X)` took to execute.
- Unfortunately, in the "real world", each measurement of the time cost of `A(X)` is corrupted by *noise*:
  - Garbage collector!
  - Other programs running.
  - Cache locality.
  - Swapping to/from disk.
  - Input/output requests from external devices.

Procedure for measuring time cost

- If we measured the time cost of **`A(X)`** based on *just one measurement*, then our estimate of the "true" time cost *of* **`A(X)`** will be very *imprecise*.
    - We might get unlucky and measure **`A(X)`** while the computer is doing a "system update".
    - If we've very unlucky, this might occur during *some* values of **`i`**, but not for others, thereby *skewing the trend* we seek to discover across the different **`N[i]`**.

Improved procedure for measuring time cost

- A much-improved procedure for measuring the time cost of A(X) is to compute the *average time across M trials*.
- Procedure (draft 2):

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long[] elapsedTimes = new long[M];
  for (int j = 0; j < M; j++) {
    final long startTime = getClockTime();
    A(X);  // Run algorithm A on input X of size N[i]
    final long endTime = getClockTime();
    elapsedTimes[j] = endTime - startTime;
  }
  final double avgElapsedTime = computeAvg(elapsedTimes);
  System.out.println("Time for N[" + i + "]: " +
                     avgElapsedTime);
}
```

Improved procedure for measuring time cost

- If the elapsed time measured in the *j*th trial is T_j, then the average over all *M* trials is:

$$\overline{T} = \frac{1}{M}\sum_{j=1}^{M} T_j$$

- We will use the *average time* "*T*-bar" as an estimate of the "true" time cost of A(X).
- The more trials *M* we use to compute the average, the more precise our estimate "*T*-bar" will be.

Improved procedure for measuring time cost

- Alternatively, we can start/stop the clock just *once*.
- Procedure (draft 2b):

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long startTime = getClockTime();
  for (int j = 0; j < M; j++) {
    A(X);  // Run algorithm A on input X of size N[i]
  }
  final long endTime = getClockTime();

  final double avgElapsedTime = (endTime - startTime) / M;
  System.out.println("Time for N[" + i + "]: " +
                     avgElapsedTime);
}
```

Quantifying uncertainty

- A key issue in any experiment is to *quantify the uncertainty* of all measurements.
- Example:
  - We are attempting to estimate the "true" time cost of A(X) by averaging together the results of many trials.
  - After computing "T-bar", how far from the "true" time cost of A(X) was our estimate?

Quantifying uncertainty

- A key issue in any experiment is to *quantify the uncertainty* of all measurements.
- Example:
  - We are attempting to estimate the "true" time cost of A(X) by averaging together the results of many trials.
  - After computing "T-bar", how far from the "true" time cost of A(X) was our estimate?
    - In order to compute this, we would have to know what the true time cost is -- and that's what we're trying to estimate!
    - We must find another way to quantify uncertainty...

Standard error versus standard deviation

- Some of you may already be familiar with the *standard deviation*:

$$\sigma = \sqrt{\frac{1}{M}\sum_{j=1}^{M}(T_j - \overline{T})^2}$$

- The standard deviation measures how "varied" the individual measurements T_j are.
  - The standard deviation gives a sense of "how much noise there is."
  - However, in most cases, we are less interested in characterizing the *noise*, and more interested in measuring the *true time cost* of **`A(X)`** itself.
    - For this, we want the *standard error*.

Quantifying your uncertainty

- In statistics, the uncertainty associated with a measurement (e.g., the time cost of A(X)) is typically quantified using the *standard error*:

Standard deviation

$$\mathrm{StdErr} = \frac{\sigma}{\sqrt{M}} \quad \text{where} \quad \sigma = \sqrt{\frac{1}{M}\sum_{j=1}^{M}(T_j - \overline{T})^2}$$

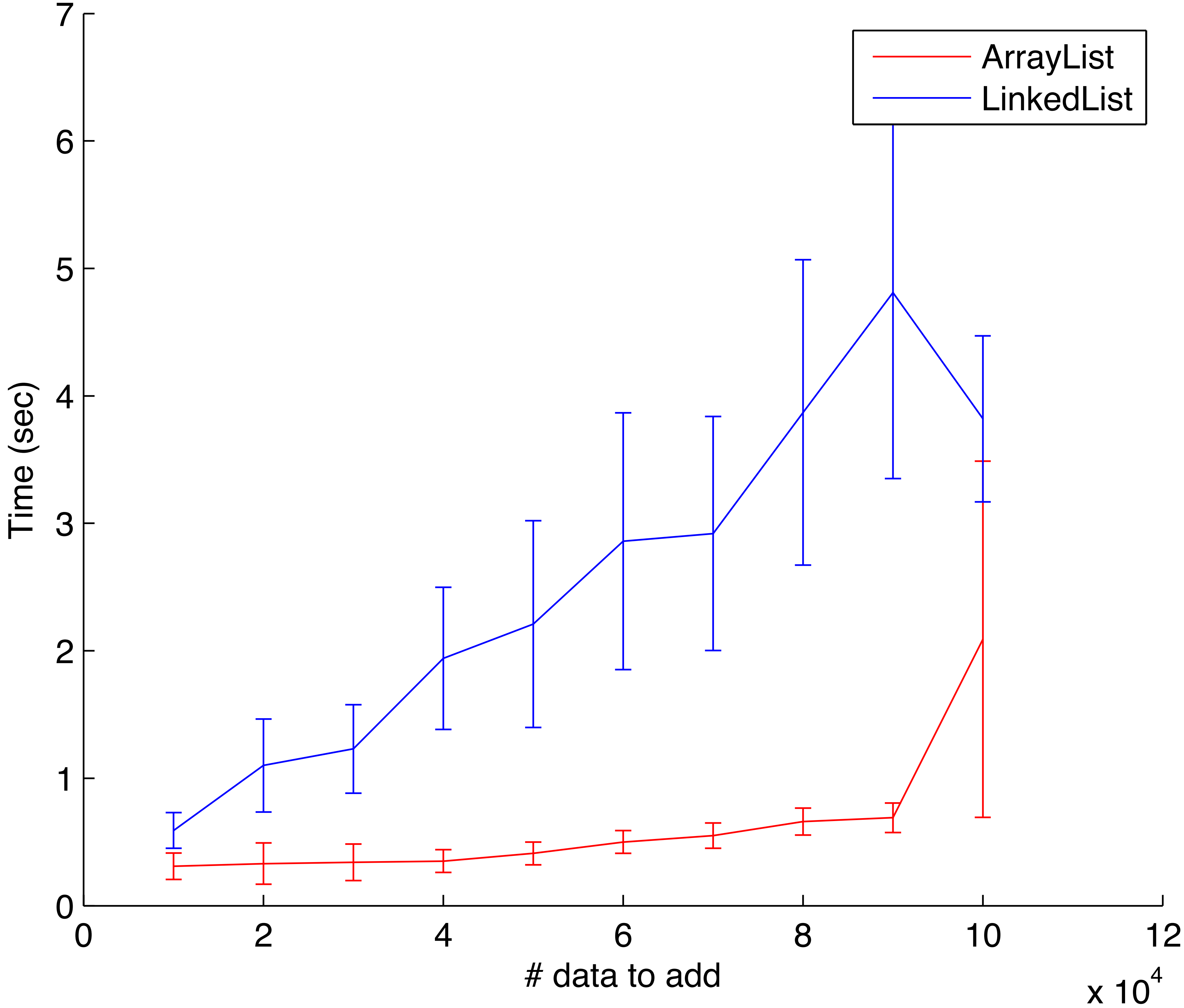
where "T-bar" is the average (computed on earlier slide).

- Notice: as *M* grows larger, the StdErr becomes smaller.

Error bars

- The standard error is often used to compute *error bars* on graphs to indicate how reliable they are.
  - Different error bars have different meanings! Some of them indicate confidence intervals, some indicate standard error, some indicate standard deviation -- it's important to know which!

Example



ArrayList
LinkedList
Time (sec)
data to add
x 10^4
0
1
2
3
4
5
6
7
0
2
4
6
8
10
12

Linear data structures: a brief review.

Linear data structures

- So far in this course we have learned the basic *linear* data structures:
  - Array list
  - Linked list
  - Stack
  - Queue
- These structures are *linear* because each element contained within them is *adjacent* to at most 2 other elements.

Linear data structures

- Linked lists and array lists provide a form of "permanent" storage of arbitrary data.
- Stacks and queues provide (typically) "temporary" storage to data that we expect to remove at some later point in time.
  - LIFO for stack, FIFO for queue.
- All these data structures provide convenient containers for storing *unrelated* data.
  - There needn't be any relationship among the individual data.

Linear data structures

- With Java generics, we gained the ability to restrict membership to an ADT to a particular class.
  - E.g., allow only `String` objects to be added to a `List12` container).
- But beyond the class of the objects, we didn't "care" about any relationships between the data.
- In particular, we didn't care whether the ADT stored the individual data in some "natural order":
  - E.g., alphabetical order for `Strings`, integer order for `Integers`.

Linear data structures

- Ignoring any relationships between data elements allowed for an ADT that was:
  - Simple to implement -- no need to *consider* order relations.
  - Flexible to use -- no need to *define* an order relation.
- However, this simplicity/flexibility comes at the cost that data retrieval is often *slower than it needs to be*.
  - By considering the natural order relations between objects, we can create data structures with superior asymptotic time costs for storage/retrieval operations.

Linear data structures: asymptotic time costs

- Let's review the "score card" of the ADTs we've covered so far.
- Let's consider three fundamental operations:
  - `void add (T o);`
  - `void remove (T o);`
  - `T find (T o);`
    Search for an element in the container that `equals o` and returns it; if no such object exists, then returns `null`.

Array-list and linked-list scorecard

	Array-list	Linked-list	
`add(o)`	$O(1)$	$O(1)$	Adding is fast.
`find(o)`	$O(n)$	$O(n)$	Finding is slow.
`remove(o)`	$O(n)$	$O(n)$	Removing is slow.

Array-list and linked-list scorecard

- There are many occasions where the user will *add* new data relatively *rarely*, but want to *find* data already in the data structure relatively *frequently*.
- In order to improve the asymptotic time cost of the **`find(o)`** and **`remove(o)`** operations, we will make use of order relationships between data elements.
  - Once we've *found* an element within a data structure, it is typically easy for the data structure to *remove* it.

Why `find` something?

- It may strike some as odd that an ADT would support the method `T find (T o)`.
- After all, if the user knows the object `o` he/she is looking for, then why call `find` at all?
- *Answer*: sometimes the user knows *part* of the information about an object `o`, but does not have the whole record.
  - This illustrates the difference between a record's *key* and its *value*.

Keys and values

- The part of the **`Student`** object that the user always knows is called the *key* (e.g., student ID number at Student Health).
- The rest of the **`Student`** record is called the *value*.

```
class Student {
  String _studentID;
  String _firstName, _lastName;
  String _address;

  Student (String studentID) {
    _studentID = studentID;
  }

  Student (String studentID, String firstName, String lastName,
           String address) {
    _studentID = studentID;
    _firstName = firstName;
    _lastName = lastName;
    _address = address;
  }
}
```

Key

Value

Keys and values

- The user may store many **`Student`** objects inside a **`List12`** container, e.g.:

```
list.add(new Student("A123", "Bill", "Carter", "123 Main St"));
list.add(new Student("A213", "Jimmy", "Clinton", "124 Main St"));
...
list.add(new Student("B092", "Hillary", "Nixon", "125 Main St"));
```

- Later, the user may wish to find a particular **`Student`** object using just the key, e.g., the student ID:

```
final Student cse12Student = list.find(new Student("A123"));
```

Student containing both the key and value.

Student initialized with just the key.

Keys and values

- For some data structures, the key and value are completely separate:
- Example:
  - A "hash map/table" (covered later in this course) allows $O(1)$-time retrieval of any *value* given its *key*.
  - To add a new entry to the table, the user calls **`put(key, value)`**, e.g.:

Key

Value

```
hashMap.put("A123",
            new Student("A123", "Bill", "Carter",
                        "123 Main St")
           );
```

Finding a particular key

- Given a request to find a particular key, and given that keys often have an *order relation* defined between them, it seems silly to search through the container *as if the keys were all unrelated*.
- *Example*: Suppose we are searching for the student ID "`C237`". Do we really need to start at the very beginning?

	`A101`	`B972`	`D192`
	`A102`	`C092`	`...`
Search ↓	`A125`	`C100`	
	`A192`	`C200`	
	`A204`	`C203`	
	`B135`	`C237`	
	`B193`	`C292`	

Finding a particular key

- Given a request to find a particular key, and given that keys often have an *order relation* defined between them, it seems silly to search through the container *as if the keys were all unrelated*.
- *Example*: Suppose we are searching for the student ID "C237". Do we really need to start at the very beginning?

Search

A101	B972	D192
A102	C092	...
A125	C100	
A192	C200	
A204	C203	
B135	C237	
B193	C292	

No -- the *natural order* among *keys* imposes structure on the "search problem" that lets us find a particular key much more quickly.

Binary order relations

- An example of a binary order relationship is the Java < operator, e.g.:

```
int  a = 3, b = 4;
if (a < b) {
   ...
}
```

- However, the < operator is only valid on primitive numeric variables (**`int`**, **`float`**, **`double`**, etc.).

Binary order relations

- More generally, two Java `Objects` can be compared if they are `Comparable`, using the `compareTo` method: `int compareTo (T o);`
- `o1.compareTo(o2)` is:
  - < 0 if `o1` is "less than" `o2`
  - == 0 if `o1` is "equal to" `o2`
  - > 0 if `o1` is "greater than" `o2`
- Classes that implement the `compareTo(o)` method can implement the `Comparable<T>` interface.

Comparable<T>

Each `Student` might be “comparable to” objects of a different class, e.g., `UCSDMember` (since faculty and staff also have ID numbers).

- Example:

```
class Student implements Comparable<Student> {
  ...
  int compareTo (T other) {
    // Compare this._studentID to
    // other._studentID -- return -1, 0, or 1
    // if this._studentID is “less than”,
    // “equal to”, or “greater than”
    // other._studentID, respectively.
    ...
  }
}
```

Comparable<T>

- Example:

```
class Student implements Comparable<Student> {
  ...
  int compareTo (T other) {
    return _studentID.compareTo(
      other._studentID
    );
  }
}
```

In this particular case, we can just delegate to the `String.compareTo(o)` method, since `String` implements `Comparable<String>`.

Comparable<T>

- Now, we can compare two **Student** objects:

```
if (student1.compareTo(student2) < 0) {
  ...
}
```