CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Ate
18 July 2012

Data structures: a quantitative perspective.

Data structures so far

- Up to now, we've focused on data structures from a software construction perspective:
  - Data structures as ADTs.
  - Separation of implementation from interface.
  - Encoding of the user's data in a sequence of bits.

Data structures: a quantitative analysis

- Just as important is the quantitative performance of those structures, e.g.:
  - **Time cost**: If I have a linked list of 100 elements, how long will it take to find a particular element? What if the list is 1000 elements long? 10,000?
  - **Space cost**: How much overhead (e.g., in `Nodes`) is there in a `DoublyLinkedList12` versus an `ArrayList`?

Data structures: a quantitative analysis

- In this lecture we will discuss *algorithmic analysis*, in particular, methods of estimating the time cost of algorithms.
- Data structures and algorithms are invariably coupled:
  - Without an algorithm, the data are useless.
  - Without a data structure, the algorithm can't accomplish anything -- they need "space" to execute.

Measuring time cost

- Instead of measuring time cost in terms of seconds, milliseconds, etc., we will count the "number of abstract operations".
- Examples of "abstract operations" include:
  - `i = i + 1;  // Assignment and/or arithmetic`
  - `if (i > 5) {  // Comparison`
- On the other hand, calling another method -- i.e., *another algorithm* -- would *not* be considered a single, abstract operation:
  - `otherMethod();  // Have to look inside otherMethod!`

Measuring time cost

- The number of "abstract operations" is largely independent of:
  - The particular computer on which an algorithm is running
  - The particular programming language in which an algorithm was implemented

Measuring time cost

- We are interested in *how the time cost grows as the size of the input* to the algorithm grows:
  - For instance, if we want to sort a list of numbers, and the size of the list is n, then we want to describe, as a function of n, how many operation the sort procedure will take.
  - Possible answers might include:
    - $2n + 3$
    - $n^2 + 3n - 1$
    - ...

Measuring time cost

- We are interested in *how the time cost grows as the size of the input* to the algorithm grows:
  - When analyzing data structures and their associated **`add`**/**`get`**/**`remove`** algorithms, the input size *n* will often be the *number of data already stored* in the ADT.

Three cases

- When estimating the time cost of an algorithm on an input of size *n*, we will consider three cases:

  1. **Worst case**: how many operations will the algorithm take on the "hardest" possible input (of size *n*)?

Three cases

- When estimating the time cost of an algorithm on an input of size *n*, we will consider three cases:

    2. **Best case**: how many operations will the algorithm take on the "easiest" possible input (of size *n*)?

Three cases

- When estimating the time cost of an algorithm on an input of size *n*, we will consider three cases:

  3. **Average case**: compute how long the algorithm would take on *every possible* input of size *n*; then, compute the *sum* of these time costs weighted by how *probable* each input would arise.

Three cases

- When estimating the time cost of an algorithm on an input of size *n*, we will consider three cases:

  3. **Average case**: compute how long the algorithm would take on *every possible* input of size *n*; then, compute the *sum* of these time costs weighted by how *probable* each input would arise.

Typically very difficult to compute exactly.

Example 1

- Let's count the number of abstract operations needed to compute the average of students' grades...

Example 1

operations

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

  return sum / grades.length;
}
```

Example 1

\# operations

```
// Assume grades.length > 0
float computeAverageGrade (float[] grades) {
  float sum = 0;                                      1
  for (int i = 0; i < grades.length; i++) {          1+2n+1
    sum += grades[i];                                 2n
  }

  return sum / grades.length;                         1
}
```

By definition of Java array, each access takes 1 operation.

Total:
4n+4

Example 1

operations

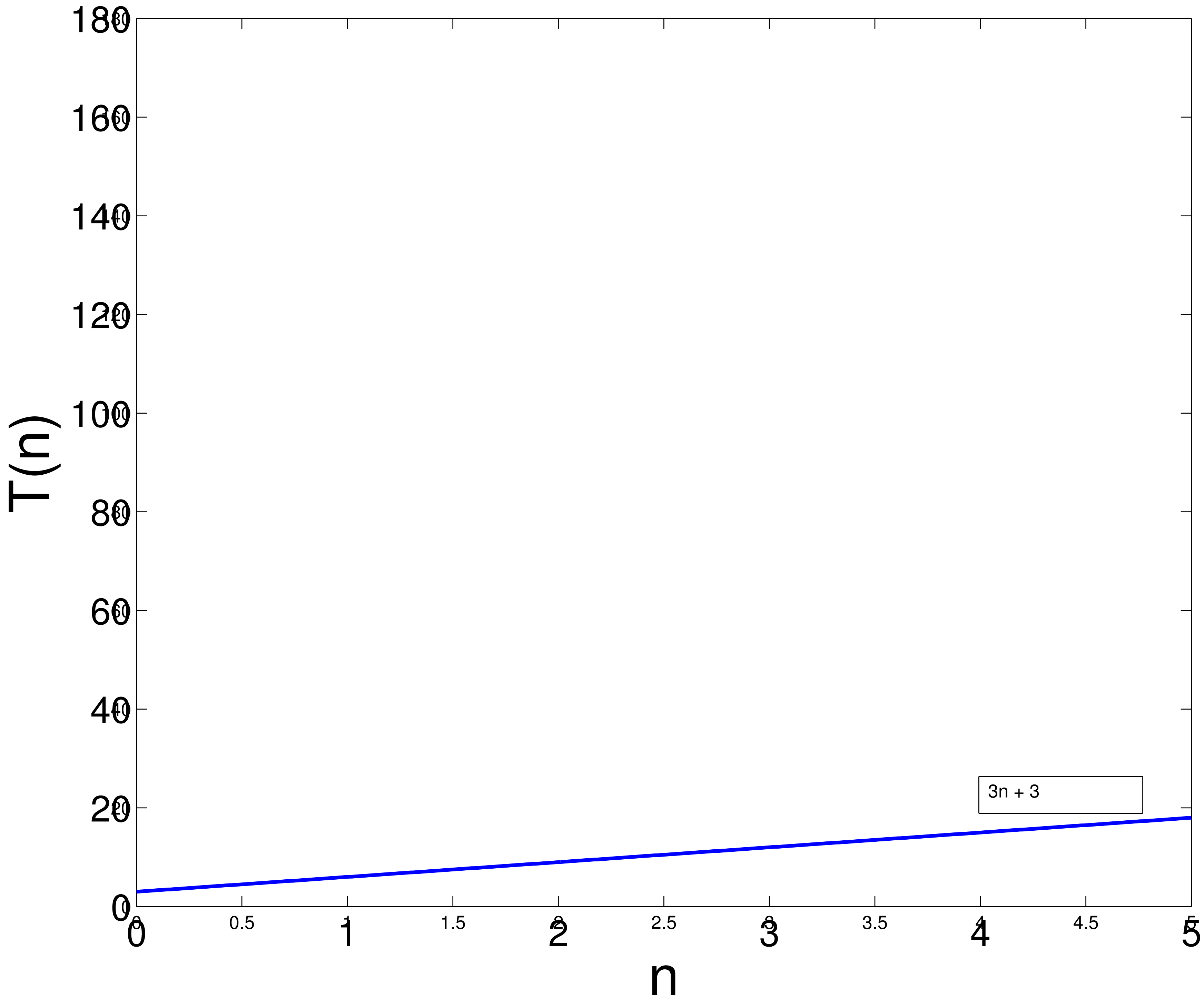
```
// Assume grades.length > 0
float computeAverageGrade (float[] grades) {
  float sum = 0;                                        1
  for (int i = 0; i < grades.length; i++) {             1+2n+1
    sum += grades[i];                                   2n
  }

  return sum / grades.length;                           1
}
```

By definition of Java array, each access takes 1 operation.

Total: 4n+4

- In this algorithm, best case = worst case = average case.
- Only the *size* (n) of the input affects the time cost, not the *particular* input.

Example 1



ne cost.

Example 2

operations

```
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == number) {
      return i;
    }
  }
  return -1;  // not found
}
```

- In this algorithm, the time cost depends on the *particular* inputs `numbers` and `number`.
- Let's first consider the *worst case*.

Example 2

operations

```
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == number) {
      return i;
    }
  }
  return -1;  // not found
}
```

- In this algorithm, the time cost depends on the *particular* inputs **`numbers`** and **`number`**.
- Let's first consider the *worst case*.
  - Here, the worst case is when **`number`** is not found.

Example 2

```
                                                          # operations
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {            1+2n+1
    if (numbers[i] == number) {                          n
      return i;                                          0
    }
  }
  return -1;  // not found                               1
}
                                                         Total:
                                                         3n+3
```

- In this algorithm, the time cost depends on the *particular* inputs `numbers` and `number`.
- Let's first consider the *worst case*.
  - Here, the worst case is when `number` is not found.

Example 2

operations

```
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == number) {
      return i;
    }
  }
  return -1;  // not found
}
```

- In this algorithm, the time cost depends on the *particular* inputs **`numbers`** and **`number`**.
- Let's first consider the *best case*.
  - Best case is when **`number`** is at index 0 of **`numbers`**.

Example 2

operations

```
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {       1+1
    if (numbers[i] == number) {                    1
      return i;                                    1
    }
  }
  return -1;  // not found
}
```

```
Total:
4
```

- In this algorithm, the time cost depends on the *particular* inputs `numbers` and `number`.
- Let's first consider the *best case*.
  - Best case is when `number` is at index 0 of `numbers`.

Example 2

operations

```
// Returns -1 if number not found in numbers
int find (int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == number) {
      return i;
    }
  }
  return -1;  // not found
}
```

- In this algorithm, the time cost depends on the *particular* inputs `numbers` and `number`.
- Finding the *average case* time cost is more difficult.
  - We'll handle that later...

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;                                          1
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      sum += numbers[i] * numbers[j];
    }
  }
  return sum;                                           1
}
```

Example 3

\# operations

```
int someMethod (int[] numbers) {
  int sum = 0;                                          1
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      sum += numbers[i] * numbers[j];                   n*n*4
    }
  }
  return sum;                                           1
}
```

Example 3

	# operations
`int someMethod (int[] numbers) {`	
`  int sum = 0;`	1
`  for (int i = 0; i < numbers.length; i++) {`	
`    for (int j = 0; j < numbers.length; j++) {`	n*(1+2n+1)
`      sum += numbers[i] * numbers[j];`	n*n*4
`    }`	
`  }`	
`  return sum;`	1
`}`	

Example 3

```
                                                   # operations
int someMethod (int[] numbers) {
  int sum = 0;                                     1
  for (int i = 0; i < numbers.length; i++) {       1+2n+1
    for (int j = 0; j < numbers.length; j++) {     n*(1+2n+1)
      sum += numbers[i] * numbers[j];              n*n*4
    }
  }
  return sum;                                      1
}
```

Example 3

```
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;
}
```

\# operations

1

1+2n+1

n*(1+2n+1)

n*n*4

1

Total:

$4n^2+2n^2+n$

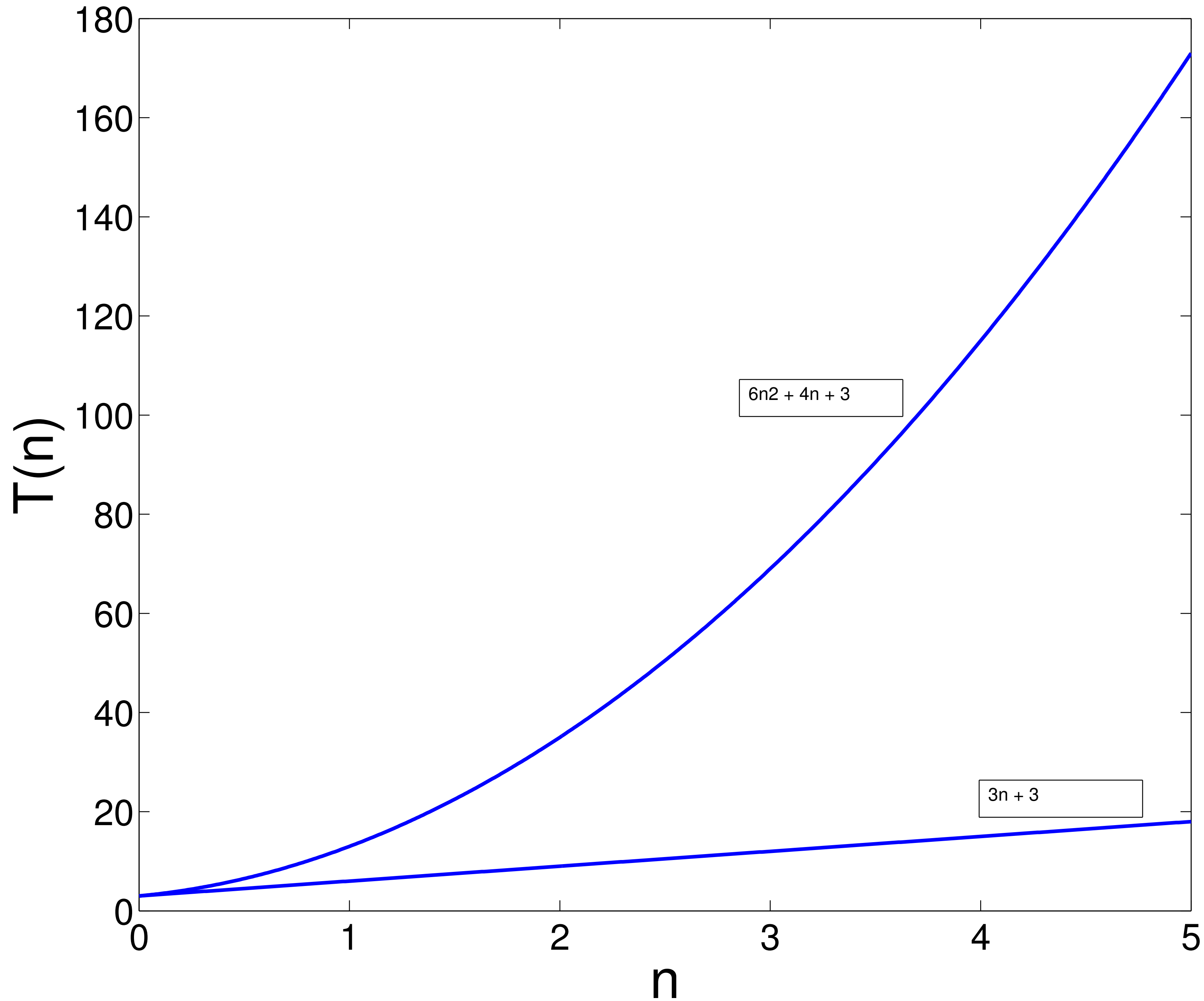
$+n+1+2n$

$+1+1 =$

$6n^2+4n+3$

This is an example of **quadratic** time cost.

Quadratic versus linear time



Asymptotic performance analysis

- This level of detail is usually more than we need when comparing algorithms:
  - We *don't* care if the time cost is n, or $3n$, or $0.1n$ -- the main thing is that it's "some constant times n".
  - We *do* care whether it's n or n^2 or 2^n.
- We are interested in *asymptotic analysis* ($n \to \infty$):
  - We mostly care about the algorithm's time cost when n is very large.
  - If n is small, then the algorithm will be fast anyway.

Asymptotic performance analysis

- Instead of saying $T(n) = 3n+3$
  we will say $T(n) = O(n)$ ("*T* is big-'*O*' of *n*"),
  i.e., $T(n)$ is basically *linear*.

- Instead of saying $T(n) = 2n-1$
  we will say $T(n) = O(n)$ ("*T* is big-'*O*' of *n*"),
  i.e., $T(n)$ is basically *linear*.

- Instead of saying $T(n) = 1/2\ n-0.2353$
  we will say $T(n) = O(n)$ ("*T* is big-'*O*' of *n*"),
  i.e., $T(n)$ is basically *linear*.

Asymptotic performance analysis

- Instead of saying $T(n) = 6n^2$
  we will say $T(n) = O(n^2)$ ("T is big-'O' of n^2"),
  i.e., $T(n)$ is basically *quadratic*.

- Instead of saying $T(n) = 2n^2+3n+13535$
  we will say $T(n) = O(n^2)$ ("T is big-'O' of n^2"),
  i.e., $T(n)$ is basically *quadratic*.

Here, the quadratic term *dominates* the linear term -- as n grows large, n^2 will become much larger than n.

Asymptotic performance analysis

- Instead of saying $T(n) = 6 \log n + 3$
  we will say $T(n) = O(\log n)$ ("T is big-'O' of log n"),
  i.e., $T(n)$ is basically *logarithmic*.

- Instead of saying $T(n) = n \log n + n - 23$
  we will say $T(n) = O(n \log n)$ ("T is big-'O' of n log n"),
  i.e., $T(n)$ is basically *loglinear*.

- Instead of saying $T(n) = n + n^2 - 3$
  we will say $T(n) =$

Asymptotic performance analysis

- Instead of saying $T(n) = 6 \log n + 3$
  we will say $T(n) = O(\log n)$ ("*T* is big-'*O*' of log *n*"),
  i.e., $T(n)$ is basically *logarithmic*.

- Instead of saying $T(n) = n \log n + n - 23$
  we will say $T(n) = O(n \log n)$ ("*T* is big-'*O*' of *n* log *n*"),
  i.e., $T(n)$ is basically *loglinear*.

- Instead of saying $T(n) = n + n^2 - 3$
  we will say $T(n) = O(n^2)$ ("*T* is big-'*O*' of n^2"),
  i.e., $T(n)$ is basically quadratic.

The *ordering* (first v second) of the terms is unimportant.
What matters is what the *dominant* term is.

Different asymptotic costs

- Asymptotic analysis assigns algorithms to different "complexity classes":
  - $O(1)$ - constant - performance of algorithm does not depend on input size.
  - $O(n)$ - linear - doubling n will double the time cost.
  - $O(\log n)$ - logarithmic
  - $O(n \log n)$ -- loglinear
  - $O(n^2)$ - quadratic
  - $O(2^n)$ - exponential
- Algorithms that differ in complexity class can have *vastly* different run-time performance (for large n).

Different asymptotic costs

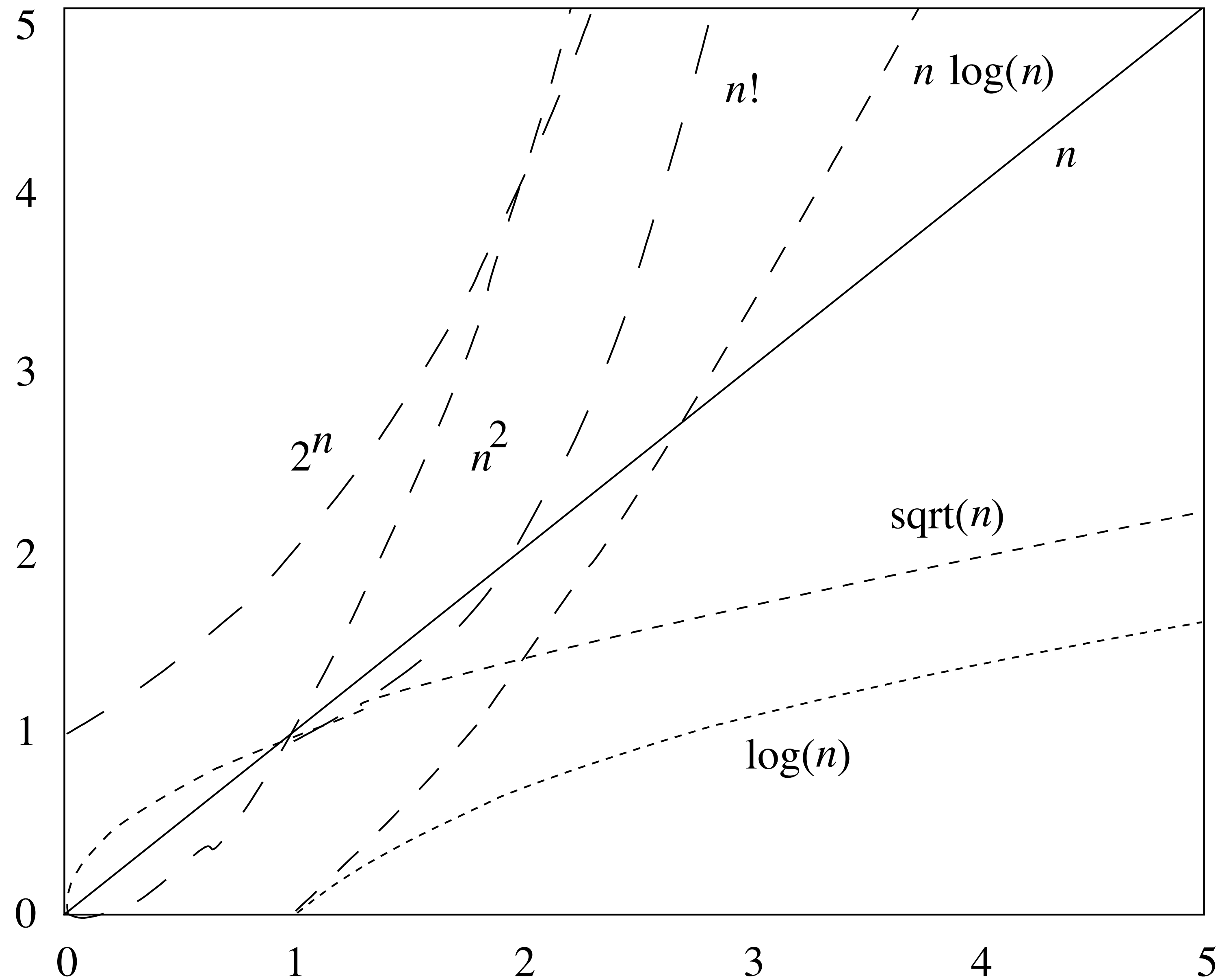


Figure 5.2 Near-origin details of common curves. Compare with Figure 5.3.

from Bailey (2007)

Different asymptotic costs

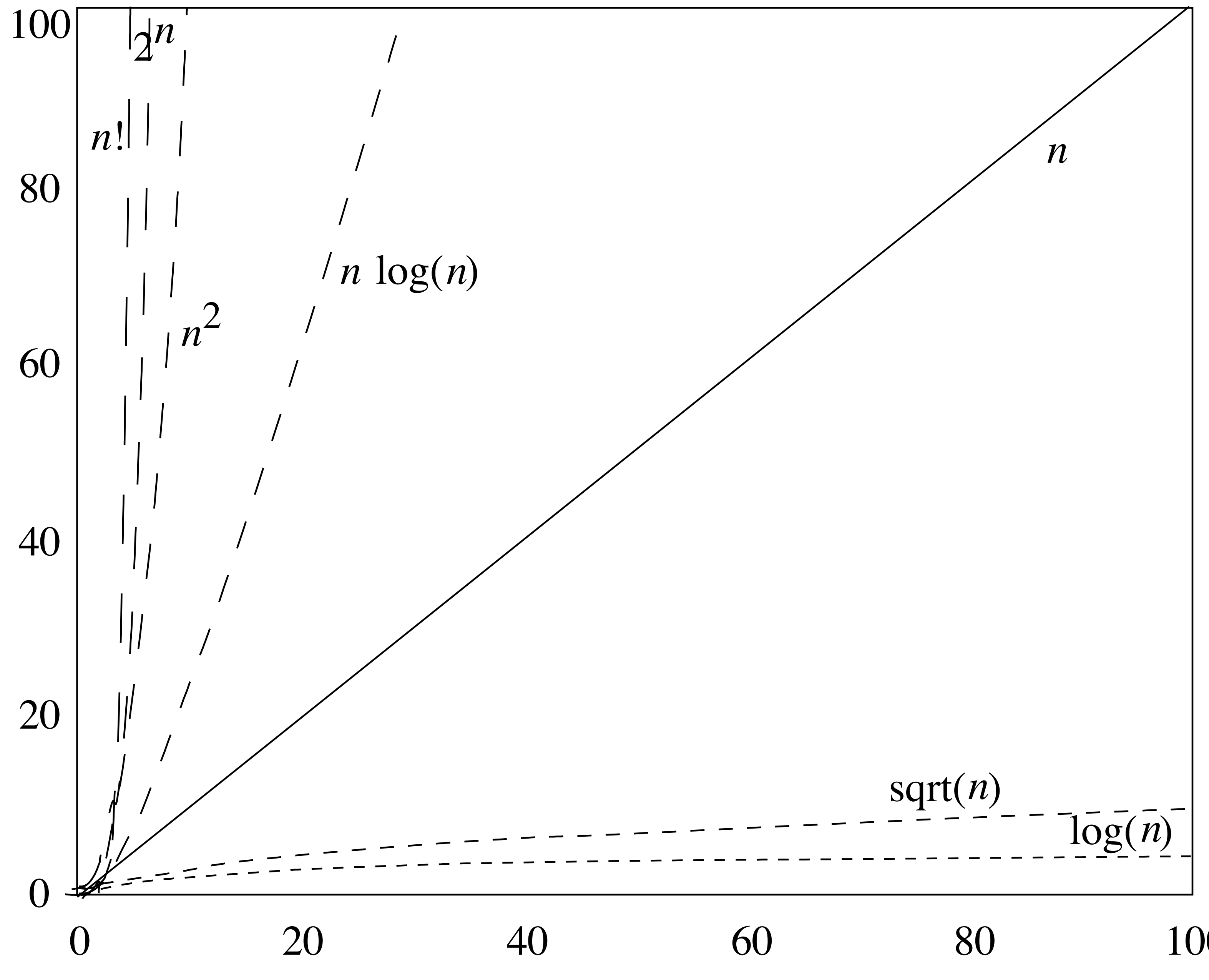


Figure 5.3 Long-range trends of common curves. Compare with Figure 5.2.

from Bailey (2007)

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 $T(n) = n^2 + 2n$, 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 $n > n_0$.

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

O(1) -- no matter how many elements the list already contains, the cost is just 2 “abstract operations”.

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*):

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

- 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.*

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

Questions to ponder

- What is the time cost of adding to the back of a *singly*-linked list, as a function of the number of elements already in the list?
  - With just a **`_head`** pointer?
  - With both **`_head`** and **`_tail`**?

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

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

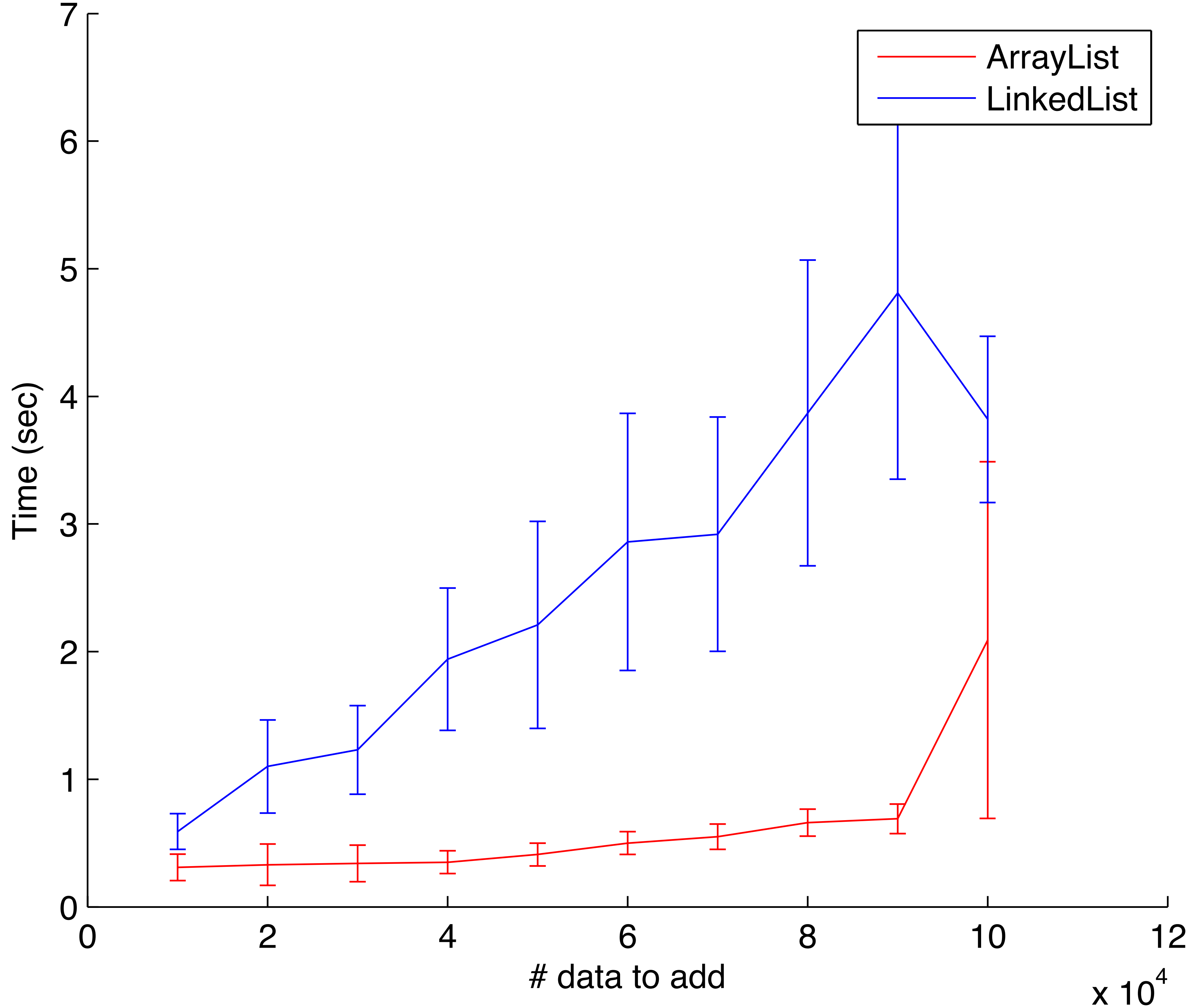
where "T-bar" is the average (computed on earlier slide).

- 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