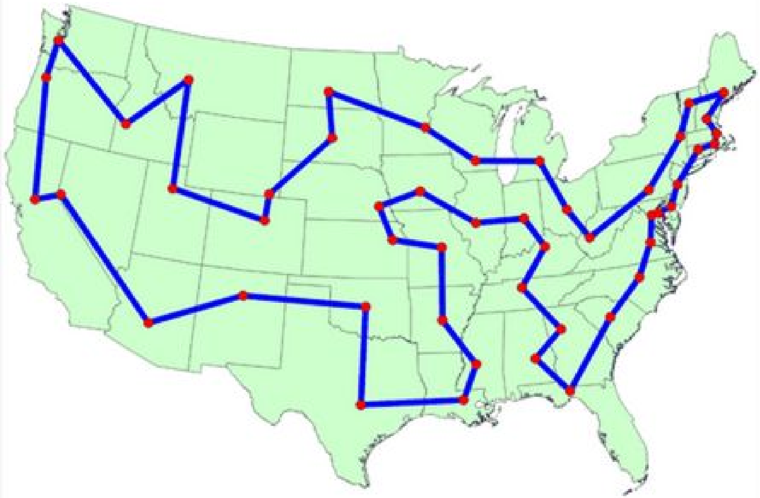
Announcements

Project 2 autograder will be up later tonight (?).

- Due Wednesday (pushed back a day for technical difficulties) at 11:59 PM.
- Tests basic cursor position activities.



CS61B

Lecture 17: Asymptotic Analysis

61B: Writing Efficient Programs

An engineer will do for a dime what any fool will do for a dollar.

Efficiency comes in two flavors:

- Programming cost (course to date).
  - How long does it take to develop your programs?
  - How easy is it to read, modify, and maintain your code?
    - More important than you might think!
    - Majority of cost is in maintenance, not development!
- Execution cost (from today to the end of the course).
  - How much time does your program take to execute?
  - How much memory does your program require?

Example of Algorithm Cost

Objective: Find a pair of duplicates in a sorted array.

- Given sorted array A, find indices i and j where A[i] = A[j].

-3	-1	2	4	5	8	10	12

Silly algorithm: Create a list of all pairs and iterate through pairs.

- (-3, -1), (-3, 2), (-3, 4), …, (10, 12)

Better algorithm?

Example of Algorithm Cost

Objective: Find a pair of duplicates in a sorted array.

- Given sorted array A, find indices i and j where A[i] = A[j].

-3	-1	2	4	5	8	10	12

Silly algorithm: Create a list of all pairs and iterate through pairs.

- (-3, -1), (-3, 2), (-3, 4), …, (10, 12)

Today's goal: Introduce standardized framework for comparing algorithmic efficiency.

Better algorithm?

- For each number A[i], just look at A[i+1], and return true the first time you see a match.

countZeros

For the next many slides, we'll be considering the runtime of the function below.

- What this function does isn't important, we just want to know about its runtime.

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

Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program.

- Tools:
  - Unix has a built in time command that measures execution time.
  - Princeton Standard library has a stopwatch class.
- Good: Easy to measure, meaning is obvious.
- Bad: Varies with machine, compiler, input data, etc.

```
public static void main(String[] args) {
    int[] array = new int[500000000];
    Stopwatch sw = new Stopwatch();
    kviate(array, 0);
    double x = sw.elapsedTime();
    System.out.println("Time: " + x);
}
```

Techniques for Measuring Computational Cost

Technique 2: Count operations for some fixed input size.

- Good: Machine independent.
- Bad: Still input dependent. Doesn't tell you actual time.

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

operation	count, N=10000
declare variable	3
assignment	3
less than	10001
equal to (==)	10000
array access	10002
increment	10001 to 20001

Techniques for Measuring Computational Cost

Technique 3: Determine symbolic execution times.

- Give statement counts or runtime in terms of input size.
- Good: Relates runtime to input sizes. **Tells you how the algorithm scales.**
- Bad: Doesn’t tell you about actual times.

```java
int count = 0, N = a.length;
for (int i = 0; i < N; i++) {
    if (a[i] == k) {
        count += 1;
    }
}
a[k] += count;
```

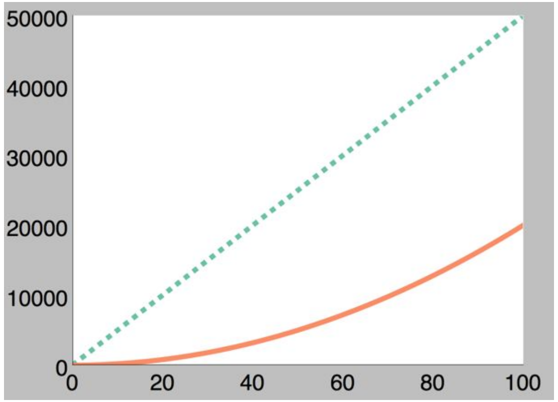
operation	count	count, N=10000
declare variable	3	3
assignment	3	3
less than	N+1	10001
equal to (==)	N	10000
array access	N+2	10002
increment	N+1 to 2N+1	10001 to 20001

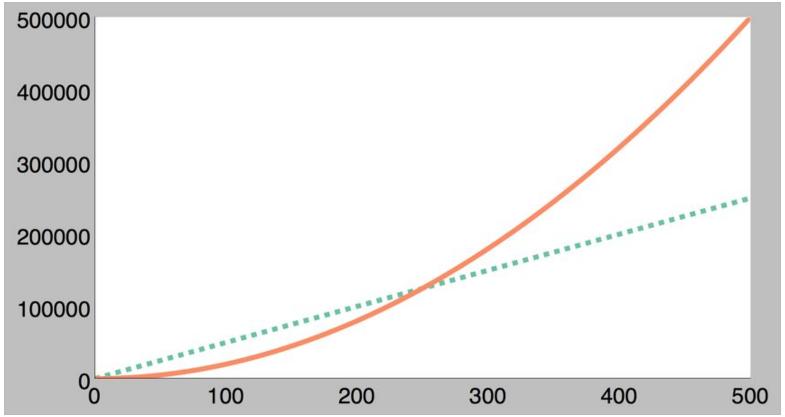
Why Scaling?

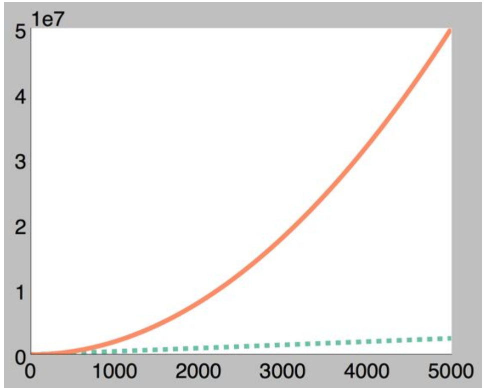
Suppose we have two algorithms that zerpify an array of size N.

- One algorithm takes 500N operations.
- The other takes $2N^2$ operations.

For small N, the $2N^2$ will be faster, but as dataset grows, the N^2 is going to fall further and further behind.







Scaling Across Many Domains

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

We'll informally refer to the "shape" of a runtime function as its ***order of growth*** (will formalize soon).

- Effect is dramatic! Determines whether a problem can be solved at all.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Simplifications (Intuitive)

Since we're primarily interested in scaling, we can simplify:

- Pick some representative operation, e.g. increment. ← arbitrary
  - Example of a bad choice: variable declaration.
  - We call this our ***cost model***.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;

hist[count] += 1;
```

operation	count	count, N=10000
declare variable	3	3
assignment	3	3
less than	N+1	10001
equal to	N	10000
array access	N+1	10001
increment	N+1 to 2N+1	10001 to 20001

Simplifications (Intuitive)

Since we're primarily interested in scaling, we can simplify:

- Pick some representative operation (cost model), e.g. increment. ← arbitrary
- Don't worry about small inputs.
    - Assume N is a lot bigger than 1.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;

hist[count] += 1;
```

operation	count	count, N=10000
increment	N+1 to 2N+1	10001 to 20001

Simplifications (Intuitive)

Since we're primarily interested in scaling, we can simplify:

- Pick some representative operation (cost model), e.g. increment. arbitrary
- Don't worry about small inputs.
  - Assume N is a lot bigger than 1.
- Ignore constant scaling factors. 2N has **same shape** as N.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;

hist[count] += 1;
```

operation	approx count	count, N=10000
increment	N to 2N	10001 to 20001

Simplifications (Intuitive)

Since we're primarily interested in scaling, we can simplify:

- Pick some representative operation (cost model), e.g. increment. ← arbitrary
- Don't worry about small inputs.
  - Assume N is a lot bigger than 1.
- Ignore constant scaling factors. 2N has **same shape** as N.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;

hist[count] += 1;
```

operation	approx count	count, N=10000
increment	N	10001 to 20001

Formalizing our Intuitive Simplifications

Converted “N+1 to 2N+1” to just “N” using some intuitive rules.

- Basic idea: For really huge N, “N+1 to 2N+1” looks-just-like N.
- How do we describe this process with mathematical precision?
    - Let’s try another example before tackling this question.

Intuitive Shapeness

Suppose we have a function $f(N) = 3813 + 387N + 2N^2$.

For very large N, what function does f(N) "look like"?

a. $g(N) = c$. A constant.
b. $g(N) = N$. A line.
c. $g(N) = N^2$. A parabola.
d. None of these.

Intuitive Shapeness

Suppose we have a function $f(N) = 3813 + 387N + 2N^2$.

For very large N, what function does f(N) look-just-like?

a. $g(N) = c$. A constant.
b. $g(N) = N$. A line.
c. $g(N) = N^2$. A parabola.
d. None of these.

See: [Wolfram Alpha]

Or if you know Calculus, can use limits.

Formalizing our Intuitive Simplifications

Converted “N+1 to 2N+1” to just “N” using some intuitive rules.

- Basic idea: For really huge N, “N+1 to 2N+1” looks-just-like N.
- $3813 + 387N + 2N^2$ looks-just-like N^2.
- How do we describe this process with mathematical precision?

Asymptotic notation (sometimes referred to as Bachmann-Landau notation) to describe ***order of growth***.

- Big O: Used for bounding above (less than).
- Big Omega: Used for bounding below (greater than).
- **Big Theta:** Used for bounding both above and below (equals).
    - Big Theta captures what we just did with our intuitive simplifications.

Big-Theta

Suppose we have some performance measurement *R(N)*, where N is the size of our problem.

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

means there exist positive constants k1 and k2 such that:

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

for all values of N that are 'very large'.

to be more precise, for all N greater than some N_0

Big-Theta Challenge

Suppose we have some performance measurement $R(N)$, where N is the size of our problem.

- Suppose R(N) = 2N + 1. Find a simple f(N) and corresponding k_1 and k_2.

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

means there exist positive constants k1 and k2 such that:

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

for all values of N that are 'very large'.

to be more precise, for all N greater than some N_0

Big-Theta Challenge Solution

Suppose we have some performance measurement $R(N)$, where N is the size of our problem.

- Suppose R(N) = 2N + 1. Find a simple f(N) and corresponding k_1 and k_2.
  - One solution: f(N) = , k_1 = , k_2=

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

means there exist positive constants k1 and k2 such that:

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

for all values of N that are 'very large'.

to be more precise, for all N greater than some N_0

Big-Theta Challenge #2

- Suppose we have $R(N) = 3813 + 387N + 2N^2$.
  - Find a simple $f(N)$ and corresponding k_1 and k_2.

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

means there exist positive constants k1 and k2 such that:

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

for all values of N that are ‘very large’.

to be more precise, for all N greater than some N_0

Big-Theta Challenge #2 Solution

- Suppose we have $R(N) = 3813 + 387N + 2N^2$.
  - Find a **simple** $f(N)$ and corresponding k_1 and k_2.

  $f(N) = N^2$, $k_1 = 1$, $k_2 = 3$

- See N2Example.py or N2Example.java for a visualization.

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

means there exist positive constants k1 and k2 such that:

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

for all values **of N that are 'very large'**.

to be more precise, for all N greater than some N_0

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 checkDuplicate(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;
}
```

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.

```java
public boolean checkDuplicate(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;
}
```

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.

```java
public boolean dupFinderBetter(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;
}
```

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).**

```java
public boolean dupFinderBetter(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;
}
```

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)?
  - It depends! In the **worst case**, $R(N) \in \Theta(N^2)$.
  - However, if input is sorted and contains duplicates, $R(N) \in \Theta(1)$.

```
public boolean dupFinderBetter(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;
}
```

Big O

Big O: Similar to Big Theta, but only bounds from above. Examples:

- $N^2 \in O(N^2)$
- $N^2 \in O(N^{500})$
- $\lg N \in O(N^2)$

Informally: The order of growth of R(N) is less than or equal to f(N).

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

means there exists a positive constant such that:

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

for all values of N that are ‘very large’.

Question

Which statement is more precise?

A. Every house in the neighborhood is worth less than $1,000,000.
B. The most expensive house in the neighborhood is worth $1,000,000.

Question

Which statement is more precise?

A. Every house in the neighborhood is worth less than $1,000,000.

B. The most expensive house in the neighborhood is worth $1,000,000.

Question

Which statement is more precise?

A. $R(N) \in O(N^2)$.

B. In the worst case, $R(N) \in \Theta(N^2)$.

Question

Which statement is more precise?

A. $R(N) \in O(N^2)$.

B. In the worst case, $R(N) \in \Theta(N^2)$.

Note: Even though B is a stronger statement than A, for convenience, people usually just say A.

Example:

- Runtime of checkDuplicates is $O(N^2)$.
- This statement is true, but runtime is also $O(N^5)$, and $O(2^N)$.
- Stronger (but wordier) statement: In the worst case, runtime of checkDuplicates is $\Theta(N^2)$.

Summary and What's Next

	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
Tilde $\sim f(N)$	Ratio converges to 1 for very large N.	$\sim 2N^2$	$2N^2$ $2N^2 + 5$

More on Wednesday

Summary and What's Next

Asymptotics provide an approximation for the scaling behavior of an algorithm.

- Will serve as our primary measure for comparing efficiency of algorithms and data structures.

Coming up:

- Wednesday: Big-omega and tilde notation. Theoretical and empirical code analysis.
- Friday: Amortized algorithm analysis.

Citations

TSP problem solution, title slide: http://support.sas.com/documentation/cdl/en/ornoaug/65289/HTML/default/viewer.htm#ornoaug_optnet_examples07.htm#ornoaug.optnet.map002g

Table of runtimes for various orders of growth: Kleinberg & Tardos, Algorithm Design.

Big Oh/Big Omega graphs: Paul Hilfinger

Selecting a cost model.

When picking a representative operation (or operations) to count.

- Using Big Theta: Doesn't matter if we pick one or many operations.
- Make sure the operation we're counting R(N) is in the same family as the function for the runtime T(N).
    - $R(N) \in \Theta(T(N))$

operation	count
declare variable	N+2
assignment	N+2
less than	(N+1)(N+2)/2
equal to	N(N-1)/2
array access	N(N-1)
increment	N(N-1)/2 to N(N-1)

Computational Cost

```java
public static String concatenateNoSpace(String s1, String s2) {
    for (int i = 0; i < s2.length(); i++)
        if (s2.charAt(i) != ‘ ’)
            s1 = s1 + s2.charAt(i);
        return s1;
    }
```

```
$ java-226 concatenateNoSpace
s2.length      Time (s)
    10000          0.14
    20000          0.41
    40000          1.62
    80000          6.59
```

Common problem

- Novice programmer does not understand performance characteristics of data structure
    - In this case, results in poor performance that rapidly gets worse as input grows.