Prelim 1

- Thursday, 2 October. 5:30pm or 7:30pm. Olin 255
- Review sheet is on the website.
- Everyone who had a conflict with the assigned time (76 people!) and submitted assignment P1Conflicts has been notified about what when and how to take it. Thanks, everyone, for responding so nicely.
- This week’s recitation gives practice on loop invariants and the searching/sorting algorithms for which you are responsible.
CS started in 1965. We celebrate with a symposium Wed and Thur morning. Among our alumni coming and talking:

- **Amit Singhal (PhD ’98):** Google VP for search engine
- **Robert Cook (MS ’81):** Author of Pixar’s RenderMan, Technical Oscar, past VP of Pixar for software development
- **Lars Backstrom (PhD ’09):** Director of News Feed Ranking and Infrastructure at Facebook.
- **Daniela Rus (PhD ’92):** MacArthur award ($500K). Great work in robotics. MIT professor, CSAIL Director.
- **Cynthia Dwork (PhD ’83):** Distinguished Scientist, Microsoft Research. Contributions to privacy preserving data analysis, cryptography, distributed computing, …
Gates building dedication

Wednesday morning (10:30). In front of Gates Hall, but impossible to get close.

Wednesday 4:30-5:30. Gates and Skorton talk in Bailey Hall. All sold out! But streamed on Cornellcast—see it on your laptop, smartphone.

We hope to stream the symposium all day also. We’ll let you know if it works out, how to get it.

Consequence of all this: NO CLASS ON THURSDAY! YOU ARE FREE TO USE THAT TIME TO STUDY FOR THE PRELIM!
Merge two adjacent sorted segments

/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted. */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into another array c;
    Copy values from c and b[t+1..k] in ascending order into b[h..]
}

We leave you to write this method. It is not difficult. Just have to move values from c and b[t+1..k] into b in the right order, from smallest to largest.
Runs in time O(k+1–h)
/** Sort b[h..k] */

public static mergesort(int[] b, int h, int k) {
    if (size b[h..k] < 2)
        return;
    int t = (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}
Mergesort

/** Sort b[h..k] */

public static mergesort(
    int[] b, int h, int k)
{
    if (size b[h..k] < 2)
        return;
    int t = (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}

Runtime recurrence
T(n): time to sort array of size n
T(1) = 1
T(n) = 2T(n/2) + O(n)

Can show by induction that
T(n) is O(n log n)

Alternatively, can see that T(n) is
O(n log n) by looking at tree of
recursive calls
QuickSort versus MergeSort

/** Sort b[h..k] */
public static void QS
    (int[] b, int h, int k) {
    if (k – h < 1) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}

/** Sort b[h..k] */
public static void MS
    (int[] b, int h, int k) {
    if (k – h < 1) return;
    MS(b, h, (h+k)/2);
    MS(b, (h+k)/2 + 1, k);
    merge(b, h, (h+k)/2, k);
}

One processes the array then recurses.
One recurses then processes the array.
Readings, Homework

- Textbook: Chapter 4

- Homework:
  - Recall our discussion of linked lists and A2.
  - What is the **worst** case complexity for appending an items on a linked list? For testing to see if the list contains X? What would be the **best** case complexity for these operations?
  - If we were going to talk about complexity (speed) for operating on a list, which makes more sense: worst-case, average-case, or best-case complexity? Why?
What Makes a Good Algorithm?

Suppose you have two possible algorithms or ADT implementations that do the same thing; which is better?

What do we mean by better?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

How do we measure time and space of an algorithm?
Basic Step: One “constant time” operation

**Basic step:**
- Input/output of scalar value
- Access value of scalar variable, array element, or object field
- assign to variable, array element, or object field
- do one arithmetic or logical operation
- method call (not counting arg evaluation and execution of method body)

**If-statement:** number of basic steps on branch that is executed

**Loop:** (number of basic steps in loop body) * (number of iterations) – also bookkeeping

**Method:** number of basic steps in method body (include steps needed to prepare stack-frame)
Counting basic steps in worst-case execution

Let $n = b\.length$

```java
/** return true iff v is in b */
static boolean find(int[] b, int v) {
    for (int i = 0; i < b.length; i++) {
        if (b[i] == v) return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>basic step</th>
<th># times executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0;$</td>
<td>1</td>
</tr>
<tr>
<td>$i &lt; b.length$</td>
<td>$n+1$</td>
</tr>
<tr>
<td>$i++$</td>
<td>$n$</td>
</tr>
<tr>
<td>$b[i] == v$</td>
<td>$n$</td>
</tr>
<tr>
<td>return true</td>
<td>0</td>
</tr>
<tr>
<td>return false</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>$3n + 3$</td>
</tr>
</tbody>
</table>

We sometimes simplify counting by counting only important things. Here, it’s the number of array element comparisons $b[i] == v$. That’s the number of loop iterations: $n$. 
Second solution: 
**Binary Search**

inv:
b[0..h] <= v < b[k..]

Number of iterations 
(always the same):
~log b.length

Therefore,
log b.length
array comparisons

```java
/** b is sorted. Return h satisfying b[0..h] <= v < b[h+1..] */
static int bsearch(int[] b, int v) {
    int h = -1;
    int k = b.length;
    while (h+1 != k) {
        int e = (h+k)/2;
        if (b[e] <= v) h = e;
        else k = e;
    }
    return h;
}
```
What do we want from a definition of “runtime complexity”?

1. Distinguish among cases for large $n$, not small $n$

2. Distinguish among important cases, like
   - $n^2$ basic operations
   - $n$ basic operations
   - $\log n$ basic operations
   - 5 basic operations

3. Don’t distinguish among trivially different cases.
   - 5 or 50 operations
   - $n$, $n+2$, or $4n$ operations
Definition of \( O(\ldots) \)

Formal definition: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c \) and \( N \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

Graphical view

Get out far enough — for \( n \geq N \) — \( c \cdot g(n) \) is bigger than \( f(n) \)
What do we want from a definition of “runtime complexity”?

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c$ and $N$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Roughly, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower, to within a constant factor.
Prove that \((n^2 + n)\) is \(O(n^2)\)

Example: Prove that \((n^2 + n)\) is \(O(n^2)\)

\[
\begin{align*}
  f(n) &= \text{<definition of } f(n)> \\
  n^2 + n &= \text{<for } n \geq 1> \\
  n^2 + n^2 &= \text{<arith>} \\
  2*n^2 &= \text{<choose } g(n) = n^2> \\
  2*g(n) &= \text{Choose } N = 1 \text{ and } c = 2
\end{align*}
\]
Prove that $100n + \log n$ is $O(n)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c$ and $N$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

\[
\begin{align*}
\text{f(n)} &= \text{<put in what f(n) is>} \\
&= 100n + \log n \\
&= \text{<We know } \log n \leq n \text{ for } n \geq 1> \\
&= 100n + \log n \\
&= \text{<arith>} \\
&= 101n \\
&= \text{<g(n) = n>} \\
&= 101g(n)
\end{align*}
\]

Choose

$N = 1$ and $c = 101$
Let $f(n) = 3n^2 + 6n - 7$
- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$
- ...

$p(n) = 4n \log n + 34n - 89$
- $p(n)$ is $O(n \log n)$
- $p(n)$ is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$
- $h(n)$ is $O(2^n)$

$a(n) = 34$
- $a(n)$ is $O(1)$
Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

<table>
<thead>
<tr>
<th>alg</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>$3n^2$</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>
## Commonly Seen Time Bounds

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>excellent</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
<td>excellent</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>good</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$n \log n$</td>
<td>pretty good</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td>OK</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>cubic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>exponential</td>
<td>too slow</td>
</tr>
</tbody>
</table>
Worst-Case/Expected-Case Bounds

May be difficult to determine time bounds for all imaginable inputs of size $n$

Simplifying assumption #4: Determine number of steps for either
- worst-case or
- expected-case or average case

- Worst-case
  - Determine how much time is needed for the worst possible input of size $n$

- Expected-case
  - Determine how much time is needed on average for all inputs of size $n$
Simplifying Assumptions

Use the size of the input rather than the input itself – $n$

Count the number of “basic steps” rather than computing exact time

Ignore multiplicative constants and small inputs (order-of, big-O)

Determine number of steps for either
- worst-case
- expected-case

These assumptions allow us to analyze algorithms effectively
Worst-Case Analysis of Searching

Linear Search
// return true iff v is in b
static bool find (int[] b, int v) {
    for (int x : b) {
        if (x == v) return true;
    }
    return false;
}

worst-case time: O(n)

Binary Search
// Return h that satisfies
//      b[0..h] <= v < b[h+1..]
static bool bsearch(int[] b, int v {
    int h= -1; int t= b.length;
    while ( h != t-1 ) {
        int e= (h+t)/2;
        if (b[e] <= v) h= e;
        else t= e;
    }

Always takes ~log(n+1) iterations.
Worst-case and expected times:
O(log n)
Comparison of linear and binary search

![Linear vs. Binary Search Graph]

- **X-axis**: Number of Items in Array
- **Y-axis**: Max Number of Comparisons
- **Legend**:
  - Linear Search (紫色方块)
  - Binary Search (黄色三角形)

The graph compares the efficiency of linear and binary search algorithms as the number of items in the array increases. Binary search, indicated by the yellow triangles, shows significantly fewer comparisons as the number of items grows, demonstrating its efficiency over linear search, marked by purple squares.
Comparison of linear and binary search
Multiply n-by-n matrices A and B:

Convention, matrix problems measured in terms of n, the number of rows, columns

- Input size is really $2n^2$, not $n$
- Worst-case time: $O(n^3)$
- Expected-case time: $O(n^3)$

```plaintext
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    c[i][j] = 0;
    for (k = 0; k < n; k++)
      c[i][j] += a[i][k]*b[k][j];
  }
```
Remarks

Once you get the hang of this, you can quickly zero in on what is relevant for determining asymptotic complexity

- Example: you can usually ignore everything that is not in the innermost loop. Why?

One difficulty:

- Determining runtime for recursive programs
  Depends on the depth of recursion
Why bother with runtime analysis?

Computers so fast that we can do whatever we want using simple algorithms and data structures, right?

Not really – data-structure/algorithm improvements can be a very big win

Scenario:
- A runs in $n^2$ msec
- A' runs in $n^2/10$ msec
- B runs in $10 \cdot n \log n$ msec

Problem of size $n=10^3$
- A: $10^3$ sec $\approx$ 17 minutes
- A': $10^2$ sec $\approx$ 1.7 minutes
- B: $10^2$ sec $\approx$ 1.7 minutes

Problem of size $n=10^6$
- A: $10^9$ sec $\approx$ 30 years
- A': $10^8$ sec $\approx$ 3 years
- B: $2 \cdot 10^5$ sec $\approx$ 2 days

1 day $= 86,400$ sec $\approx 10^5$ sec
1,000 days $\approx$ 3 years
Human genome
= 3.5 billion nucleotides
~ 1 Gb

@1 base-pair
instruction/µsec

- $n^2 \rightarrow 388445$ years
- $n \log n \rightarrow 30.824$ hours
- $n \rightarrow 1$ hour
Limitations of Runtime Analysis

Big-O can hide a very large constant
- Example: selection
- Example: small problems

The specific problem you want to solve may not be the worst case
- Example: Simplex method for linear programming

Your program may not be run often enough to make analysis worthwhile
- Example: one-shot vs. every day
- You may be analyzing and improving the wrong part of the program
- Very common situation
- Should use profiling tools
What you need to know / be able to do

- Know the definition of $f(n)$ is $O(g(n))$

- Be able to prove that some function $f(n)$ is $O(g(n))$. The simplest way is as done on two slides.

- Know worst-case and average (expected) case $O(\ldots)$ of basic searching/sorting algorithms: linear/binary search, partition alg of Quicksort, insertion sort, selection sort, quicksort, merge sort.

- Be able to look at an algorithm and figure out its worst case $O(\ldots)$ based on counting basic steps or things like array-element swaps/
Lower Bound for Comparison Sorting

**Goal:** Determine minimum time *required* to sort *n* items

**Note:** we want worst-case, not best-case time

- Best-case doesn’t tell us much. E.g. Insertion Sort takes O(n) time on already-sorted input
- Want to know *worst-case time* for *best possible* algorithm

- How can we prove anything about the *best possible* algorithm?

- Want to find characteristics that are common to *all* sorting algorithms

- Limit attention to *comparison-based algorithms* and try to count number of comparisons
Comparison Trees

- Comparison-based algorithms make decisions based on comparison of data elements
- Gives a *comparison tree*
- If algorithm fails to terminate for some input, comparison tree is infinite
- Height of comparison tree represents **worst-case number of comparisons** for that algorithm
- Can show: *Any correct comparison-based algorithm must make at least* $n \log n$ *comparisons in the worst case*
Lower Bound for Comparison Sorting

- Say we have a correct comparison-based algorithm
- Suppose we want to sort the elements in an array $b[]$
- Assume the elements of $b[]$ are distinct
- Any permutation of the elements is initially possible
- When done, $b[]$ is sorted
- But the algorithm could not have taken the same path in the comparison tree on different input permutations
How many input permutations are possible? \( n! \sim 2^{n \log n} \)

For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree.

To have at least \( n! \sim 2^{n \log n} \) leaves, it must have height at least \( n \log n \) (since it is only binary branching, the number of nodes at most doubles at every depth).

Therefore its longest path must be of length at least \( n \log n \), and that it its worst-case running time.