A3 and Prelim

- Deadline for A3: tonight. Only two late days allowed (Wed-Thur)
- Prelim: Thursday evening. 74 conflicts!
  If you filled out P1conflict and don’t hear from us, just go to the time you said you could go.
- BRING YOUR ID CARDS TO THE PRELIM. Won’t get in without it.
- We are in THREE rooms. Olin 155/255 and Phillips 101. We will tell you on Thursday which one to go to.

Merge two adjacent sorted segments

```java
/* 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) {
    // Sort b[h..t] into another array c;
    // Copy values from c and b[t+1..k] into b[h..k]
    // in ascending order.
}
```

Merge two adjacent sorted segments

```java
// Merge sorted c and b[t+1..k] into b[h..k]
```
Mergesort

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

Merge: time proportional to n
Depth of recursion: log n
Can therefore show (later) that time taken is proportional to n log n
But space is also proportional to n

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.

What Makes a Good Algorithm?

Suppose you have two possible algorithms 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

- 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)

Piazza question had this in it:
s.substring(0, 1) == s.substring(k, k+1)
Is it constant-time? If so, is it efficient?
Piazza question had this in it:

`s.substring(0, 1) == s.substring(k, k+1)`

Is it constant-time? Yes. But VERY inefficient.

How about this instead?

`s.charAt(0) == s.charAt(k)`

Equivalent to

`s[b[0]] == s[b[k]]`

Sample Problem: Searching

**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

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*n` basic operations
   - `n` basic operations
   - `log n` basic operations
   - `5` basic operations
3. Don’t distinguish among trivially different cases, like
   - `5` or `50` operations
   - `n`, `n+2`, or `4n` operations

Definition of `O(…)`

Formal definition: `f(n)` is `O(g(n))` if there exist constants `c > 0` and `N ≥ 0` such that for all `n ≥ N`, `f(n) ≤ c · g(n)`

Graphical view

Get out far enough (for `n ≥ N`) `f(n)` is at most `c · g(n)`

Counting basic steps in worst-case execution

**Linear Search** Let `n = b.length`

<table>
<thead>
<tr>
<th>Basic Step</th>
<th># Times Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i = 0;</code></td>
<td>1</td>
</tr>
<tr>
<td><code>i &lt; b.length</code></td>
<td><code>n+1</code></td>
</tr>
<tr>
<td><code>i++</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>b[i] == v</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>return true</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>return false</code></td>
<td><code>1</code></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><code>3n + 3</code></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`.

What do we want from a definition of “runtime complexity”?

Formal definition: `f(n)` is `O(g(n))` if there exist constants `c > 0` and `N ≥ 0` such that for all `n ≥ N`, `f(n) ≤ c · 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 \(O(n^2)\)**

**Formal definition:** \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

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

**Methodology:**
- Start with \(f(n)\) and slowly transform into \(c \cdot g(n)\):
  - Use \(=\) and \(<=\) and \(<\) steps
  - At appropriate point, can choose \(N\) to help calculation
  - At appropriate point, can choose \(c\) to help calculation

\[
\begin{align*}
  f(n) &= \text{<definition of } f(n)> \\
  &= n^2 + n \\
  &\leq \text{<for } n \geq 1, n^2 \leq n^2> \\
  &= n^2 + n \\
  &= 2^n \leq 2^n \\
  &= \text{<choose } g(n) = n^2> \\
  &= 2^n g(n)
\end{align*}
\]

Choose \(N = 1\) and \(c = 2\)

**Do NOT say or write** \(f(n) = O(g(n))\)

If \(f(n) = g(n))\) is simply **WRONG**. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don’t read such things.

Here’s an example to show what happens when we use \(=\) this way.

We know that \(n+2\) is \(O(n)\) and \(n+3\) is \(O(n)\). Suppose we use

\[
\begin{align*}
  n+2 &= O(n) \\
  n+3 &= O(n)
\end{align*}
\]

But then, by transitivity of equality, we have \(n+2 = n+3\). We have proved something that is false. Not good.

---

**O(\ldots) Examples**

- **Let** \(f(n) = 3n^2 + 6n - 7\)
  - \(f(n) = O(n^2)\)
  - \(f(n) = O(n^2)\)
  - \(f(n) = O(n^2)\)
  - \(\ldots\)

- **p(n) = 4n \log n + 34n - 89\)
  - \(p(n) = O(n \log n)\)
  - \(p(n) = O(n \log n)\)
  - \(p(n) = O(n \log n)\)

- **h(n) = 20\cdot n + 40n\)
  - \(h(n) = O(2n)\)
  - \(h(n) = O(2n)\)
  - \(h(n) = O(2n)\)

Only the leading term (the term that grows most rapidly) matters

If it’s \(O(n^2)\), it’s also \(O(n^3)\) etc! However, we always use the smallest one

**Commonly Seen Time Bounds**

<table>
<thead>
<tr>
<th>(O(\ldots))</th>
<th>constant</th>
<th>excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(1))</td>
<td></td>
<td></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>
### Problem-size examples

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

<table>
<thead>
<tr>
<th>operations</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>( 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>( n^3 )</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

### 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 \( 10 \log n \) 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 \( \approx \) 17 minutes
- A’: \( 10^8 \) sec \( \approx \) 1.7 minutes
- B: \( 10^8 \) sec \( \approx \) 1.7 minutes

**Problem of size \( n=10^9 \):**

- A: \( 10^9 \) sec \( \approx \) 17 minutes
- A’: \( 10^8 \) sec \( \approx \) 1.7 minutes
- B: \( 10^8 \) sec \( \approx \) 1.7 minutes

### Algorithms for the Human Genome

- Human genome = 3.5 billion nucleotides
  ~ 1 Gb

- @1 base-pair instruction/µsec

  | \( n^2 \) | 388445 years |
  | \( n \log n \) | 30.824 hours |
  | \( n \) | 1 hour |

### 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 \( n \)

### Algorithms for the Human Genome

- 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: \( O(b) \)
Expected time \( O(b/\#b) \)

\(#b = \text{size of } b\)

**Binary Search**

// Return h that satisfies
// \[ b[0..h] \subseteq v < b[h+1..] \]
static bool bsearch(int[] b, int v) {
    int h = b.length;
    while (h > 1) {
        int e = (h+1)/2;
        if (b[e] <= v) h = e;
        else v = e;
    }
}
Always \( -\log (\#b+1) \) iterations.
Worst-case and expected times: \( O(\log \#b) \)
Analysis of Matrix Multiplication

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: $O(n^3)$

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];
    }

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        throw new Exception();
    }

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

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 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 above.
- 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

Mergesort

```java
/** 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 \( O(n \log n) \) by looking at tree of recursive calls

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