Announcements

• Makeup Prelim 2
  ▪ Monday 11/21
  ▪ 7:30-9pm
  ▪ Upson 5130
  ▪ Please do not discuss the prelim with your classmates!

• Quiz 4 next Tuesday in class
  ▪ Topics: graphs, threads, recurrences, Franklin Olin’s middle name
Analysis of MergeSort

public static Comparable[] mergeSort(Comparable[] A, int low, int high) {
    if (low + 2 <= high) { //at least 2 elements?
        int mid = (low + high)/2;
        Comparable[] A1 = mergeSort(A, low, mid);
        Comparable[] A2 = mergeSort(A, mid, high);
        return merge(A1,A2);
    }
    ...
}

• Recurrence:
  ▪ $T(n) = c + d + e + f + 2T(n/2) + gn + h$ ← recursive case
  ▪ $T(1) = i$ ← base case

• How do we solve this recurrence?
Analysis of MergeSort

• Recurrence:
  ▪ \( T(n) = c + d + e + f + 2T(n/2) + gn + h, \ n > 1 \)
  ▪ \( T(1) = i \)

• First, simplify by dropping lower-order terms

• Simplified recurrence:
  ▪ \( T(n) = 2T(n/2) + cn + d, \ n > 1 \)
  ▪ \( T(1) = e \)

• How do we find the solution?
Solving Recurrences

• Unfortunately, solving recurrences is like solving differential equations
  ▪ No general technique works for all recurrences

• Luckily, can get by with a few common patterns

• You will learn some more techniques in CS 2800
Analysis of MergeSort

- Recurrence for MergeSort:
  - $T(n) = 2T(n/2) + cn + d, \ n > 1$
  - $T(1) = e$

- Solution is $T(n) = O(n \log n)$
Analysis of MergeSort

- Recurrence for MergeSort:
  - $T(n) = 2T(n/2) + cn + d, \ n > 1$
  - $T(1) = e$

- Solution is $T(n) = O(n \log n)$

- Tricks:
  - Take larger constants to simplify
  - Ignore small inputs
  - Use inequalities instead of equalities
Proof: Strong induction on n. Show that

If \( T(2) \leq c \) and \( T(3) \leq c \) and \( T(n) \leq 2T(n/2) + cn, \) \( n > 3 \)

then for all \( n \geq 2, \) \( T(n) \leq cn \log n \)

Basis:
\[
\begin{align*}
T(2) &\leq c \leq c 2 \log 2 \checkmark \\
T(3) &\leq c \leq c 3 \log 3 \checkmark \\
\end{align*}
\]

Induction step:
For \( n > 3, \)
\[
T(n) \leq 2T(n/2) + cn \leq 2 c(n/2) \log (n/2) + cn \quad \text{(IH)}
= cn (\log n – 1) + cn = cn \log n \checkmark
\]
Solving Recurrences

- Recurrences are important when using divide & conquer to design an algorithm.

- Solution techniques:
  - Can sometimes change variables to get a simpler recurrence.
  - Make a guess, then prove the guess correct by induction.
  - Build a recursion tree and use it to determine solution.
  - Can use the Master Method
    - A “cookbook” scheme that handles many common recurrences.

To solve $T(n) = aT(n/b) + f(n)$, compare $f(n)$ with $n^{\log_b a}$:

- Solution is $T(n) = O(f(n))$ if $f(n)$ grows more rapidly.
- Solution is $T(n) = O(n^{\log_b a})$ if $n^{\log_b a}$ grows more rapidly.
- Solution is $T(n) = O(f(n) \log n)$ if both grow at same rate.

- Not an exact statement of the theorem – $f(n)$ must be “well-behaved.”
Recurrence Examples

• \( T(n) = T(n - 1) + 1 \) \( \rightarrow \) \( T(n) = O(n) \) Linear Search

• \( T(n) = T(n - 1) + n \) \( \rightarrow \) \( T(n) = O(n^2) \) QuickSort worst-case

• \( T(n) = T(n/2) + 1 \) \( \rightarrow \) \( T(n) = O(\log n) \) Binary Search

• \( T(n) = T(n/2) + n \) \( \rightarrow \) \( T(n) = O(n) \)

• \( T(n) = 2 \, T(n/2) + n \) \( \rightarrow \) \( T(n) = O(n \log n) \) MergeSort

• \( T(n) = 2 \, T(n - 1) \) \( \rightarrow \) \( T(n) = O(2^n) \)
<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>300</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>50</td>
<td>250</td>
<td>500</td>
<td>1500</td>
<td>5000</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
<td>282</td>
<td>665</td>
<td>2469</td>
<td>9966</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>2500</td>
<td>10,000</td>
<td>90,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>125,000</td>
<td>1,000,000</td>
<td>27 million</td>
<td>1 billion</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>a 16-digit number</td>
<td>a 31-digit number</td>
<td>a 91-digit number</td>
<td>a 302-digit number</td>
</tr>
<tr>
<td>3.6 million</td>
<td>3.6 million</td>
<td>a 65-digit number</td>
<td>a 161-digit number</td>
<td>a 623-digit number</td>
<td>unimaginably large</td>
</tr>
<tr>
<td>10 billion</td>
<td>10 billion</td>
<td>an 85-digit number</td>
<td>a 201-digit number</td>
<td>a 744-digit number</td>
<td>unimaginably large</td>
</tr>
</tbody>
</table>

- protons in the known universe ~ 126 digits
- µsec since the big bang ~ 24 digits

- Source: D. Harel, *Algorithmics*
How long would it take @ 1 instruction / µsec?

<table>
<thead>
<tr>
<th>n</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1/10,000 sec</td>
<td>1/2500 sec</td>
<td>1/400 sec</td>
<td>1/100 sec</td>
<td>9/100 sec</td>
</tr>
<tr>
<td>n^5</td>
<td>1/10 sec</td>
<td>3.2 sec</td>
<td>5.2 min</td>
<td>2.8 hr</td>
<td>28.1 days</td>
</tr>
<tr>
<td>2^n</td>
<td>1/1000 sec</td>
<td>1 sec</td>
<td>35.7 yr</td>
<td>400 trillion centuries</td>
<td>a 75-digit number of centuries</td>
</tr>
<tr>
<td>n^n</td>
<td>2.8 hr</td>
<td>3.3 trillion years</td>
<td>a 70-digit number of centuries</td>
<td>a 185-digit number of centuries</td>
<td>a 728-digit number of centuries</td>
</tr>
</tbody>
</table>

- The big bang was 15 billion years ago \( (5 \cdot 10^{17} \text{ secs}) \)

- Source: D. Harel, *Algorithmics*
The Fibonacci Function

Mathematical definition:
\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n - 1) + \text{fib}(n - 2), \quad n \geq 2
\end{align*}
\]

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, …

```c
int fib(int n) {
    if (n == 0 || n == 1) return n;
    else return fib(n-1) + fib(n-2);
}
```
Recursive Execution

```cpp
int fib(int n) {
    if (n == 0 || n == 1) return n;
    else return fib(n-1) + fib(n-2);
}
```

Execution of fib(4):

```
  fib(4)
  /   
fib(3)  fib(2)
/       /   
fib(2)  fib(1)  fib(1)  fib(0)
/     /     /     /
fib(1) fib(0) fib(1) fib(0)
```
The Fibonacci Recurrence

```c
int fib(int n) {
    if (n == 0 || n == 1) return n;
    else return fib(n-1) + fib(n-2);
}
```

- $T(0) = c$
- $T(1) = c$
- $T(n) = T(n – 1) + T(n – 2) + c$

- Solution is exponential in $n$
- But not quite $O(2^n)$...
The Golden Ratio

\[ \varphi = \frac{a+b}{b} = \frac{b}{a} \]

\[ \varphi^2 = \varphi + 1 \]

\[ \varphi = \frac{1 + \sqrt{5}}{2} \]

\[ = 1.618... \]

ratio of sum of sides \((a+b)\) to longer side \((b)\)

= 

ratio of longer side \((b)\) to shorter side \((a)\)
Fibonacci Recurrence is $O(\varphi^n)$

Want to show that for all $n$, $T(n) \leq c\varphi^n$

Have $\varphi^2 = \varphi + 1$

Multiplying by $c\varphi^{n-2}$, $c\varphi^n = c\varphi^{n-1} + c\varphi^{n-2}$

Basis:
- $T(0) = c = c\varphi^0$
- $T(1) = c \leq c\varphi^1$

Induction step:
- $T(n) = T(n-1) + T(n-2) \leq c\varphi^{n-1} + c\varphi^{n-2} = c\varphi^n$
**Can We Do Better?**

```java
if (n <= 1) return n;
int parent = 0;
int current = 1;
for (int i = 2; i <= n; i++) {
    int next = current + parent;
    parent = current;
    current = next;
}
return (current);
```

- Number of times loop is executed? \( n - 1 \)
- Number of basic steps per loop? constant
- Complexity of iterative algorithm = \( O(n) \)
- Much, much, much better than \( O(\varphi^n) \)!
...But We Can Do Even Better!

\[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}
\begin{pmatrix}
f_n \\
f_{n+1}
\end{pmatrix} =
\begin{pmatrix}
f_{n+1} \\
f_{n+2}
\end{pmatrix} =
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^n
\begin{pmatrix}
f_0 \\
f_1
\end{pmatrix} =
\begin{pmatrix}
f_n \\
f_{n+1}
\end{pmatrix}
\]

Repeated squaring of the matrix gives $O(\log n)$
But We Are Not Done Yet...

- Would you believe constant time?

\[ f_n = \frac{\varphi^n - \varphi'^n}{\sqrt{5}} \]

- where \( \varphi = \frac{1 + \sqrt{5}}{2} \) \quad \varphi' = \frac{1 - \sqrt{5}}{2} \]
Matrix Multiplication in Less Than \(O(n^3)\) (Strassen's Algorithm)

- Idea: Naive 2 x 2 matrix multiplication takes 8 scalar multiplications, but we can do it in 7:

\[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\begin{pmatrix}
e & f \\
g & h
\end{pmatrix}
= \begin{pmatrix}
s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\
s_6 + s_7 & s_2 - s_3 + s_5 - s_7
\end{pmatrix}
\]

where

\[
\begin{align*}
s_1 &= (b - d)(g + h) \\
s_2 &= (a + d)(e + h) \\
s_3 &= (a - c)(e + f) \\
s_4 &= h(a + b) \\
s_5 &= a(f - h) \\
s_6 &= d(g - e) \\
s_7 &= e(c + d)
\end{align*}
\]
Now Apply This Recursively – Divide and Conquer!

\[
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix}
\begin{pmatrix}
E & F \\
G & H
\end{pmatrix}
=
\begin{pmatrix}
S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\
S_6 + S_7 & S_2 - S_3 + S_5 - S_7
\end{pmatrix}
\]
Now Apply This Recursively – Divide and Conquer!

- Gives recurrence $M(n) = 7 M(n/2) + cn^2$ for the number of multiplications
- Solution is $M(n) = O(n^{\log 7}) = O(n^{2.81...})$
- Number of additions is $O(n^2)$, bound separately
Is That the Best You Can Do?

• How about 3 x 3 for a base case?
  ▪ best known is 23 multiplications
  ▪ not good enough to beat Strassen

• In 1978, Victor Pan discovered how to multiply 70 x 70 matrices with 143640 multiplications, giving $O(n^{2.795...})$

• Best bound to date (obtained by entirely different methods) is $O(n^{2.376...})$ (Coppersmith & Winograd 1987)

• Best know lower bound is still $\Omega(n^2)$
Moral: Complexity Matters!

• But you are acquiring the best tools to deal with it!