Lecture 23: Recurrences
Analysis of Merge-Sort

```
public static Comparable[] mergeSort(Comparable[] A, int low, int high) {
    if (low < high) { //at least 2 elements?
        int mid = (low + high)/2;
        Comparable[] A1 = mergeSort(A, low, mid);
        Comparable[] A2 = mergeSort(A, mid+1, high);
        return merge(A1,A2);
    }
    ....
}
```

• Recurrence describing computation time:
  – $T(n) = c + d + e + f + 2T(n/2) + g\ n + h \quad \leftarrow \text{recurrence}$
  – $T(1) = i \quad \leftarrow \text{base case}$

• How do we solve this recurrence?
Analysis of Merge-Sort

• Recurrence:
  – $T(n) = c + d + e + f + 2T(n/2) + gn + h$
  – $T(1) = i$

• First, simplify by dropping lower-order terms and replacing constants by their max
  – $T(n) = 2T(n/2) + an$
  – $T(1) = b$

• Simplify even more. Consider only the number of comparisons.
  – $T(n) = 2T(n/2) + n$
  – $T(1) = 0$

• 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 learn some more techniques in CS2800
Analysis of Merge-Sort

• Recurrence for number of comparisons of MergeSort
  – \( T(n) = 2T(n/2) + n \)
  – \( T(1) = 0 \)
  – \( T(2) = 2 \)

• To show: \( T(n) \) is \( O(n \log(n)) \) for \( n \in \{2,4,8,16,32,...\} \)
  – Restrict to powers of two to keep algebra simpler

• Proof: use induction on \( n \in \{2,4,8,16,32,...\} \)
  – Show \( P(n) = \{T(n) \leq c \ n \log(n)\} \) for some fixed constant \( c \).
  – Base: \( P(2) \)
    • \( T(2) = 2 \leq c \ 2 \log(2) \) using \( c=1 \)
  – Strong inductive hypothesis: \( P(m) = \{T(m) \leq c \ m \log(m)\} \) is true for all \( m \in \{2,4,8,16,32,...,k\} \).
  – Induction step: \( P(2) \wedge P(4) \wedge ... \wedge P(k) \rightarrow P(2k) \)
    • \( T(2k) \leq 2T(2k/2) + (2k) \leq 2(c \ k \log(k)) + (2k) \leq c \ (2k) \log(k) + c \ (2k) \)
      = \( c \ (2k) (\log(k) + 1) = c \ (2k) \log(2k) \) for \( c \geq 1 \)
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

Master Method:
To solve $T(n) = a \cdot T(n/b) + f(n)$

• 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

Some common cases:

• \( T(n) = T(n - 1) + 1 \)  \( T(n) \) is \( O(n) \)  Linear Search
• \( T(n) = T(n - 1) + n \)  \( T(n) \) is \( O(n^2) \)  QuickSort worst-case
• \( T(n) = T(n/2) + 1 \)  \( T(n) \) is \( O(\log n) \)  Binary Search
• \( T(n) = T(n/2) + n \)  \( T(n) \) is \( O(n) \)
• \( T(n) = 2 \, T(n/2) + n \)  \( T(n) \) is \( O(n \log n) \)  MergeSort
• \( T(n) = 2 \, T(n - 1) \)  \( T(n) \) is \( O(2^n) \)
<table>
<thead>
<tr>
<th>5n</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>300</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>250</td>
<td>500</td>
<td>1500</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>282</td>
<td>665</td>
<td>2469</td>
<td>9966</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2500</td>
<td>10,000</td>
<td>90,000</td>
<td>1,000,000</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>125,000</td>
<td>1,000,000</td>
<td>27 million</td>
<td>1 billion</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></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></th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^2 )</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 )</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 \( \times 10^{17} \) secs)

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

• Mathematical definition:
  – fib(0) = 0
  – fib(1) = 1
  – fib(n) = fib(n – 1) + fib(n – 2), n ≥ 2

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

• Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, ...

Fibonacci (Leonardo Pisano) 1170–1240?
Statue in Pisa, Italy
Giovanni Paganucci 1863
Recursive Execution

```c
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)
The Fibonacci Recurrence

• Recurrence for computation time:
  – $T(0) = a$
  – $T(1) = a$
  – $T(n) = T(n-1) + T(n-2) + a$

• What is computation time?

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

• Recurrence for computation time of fib
  – $T(0) = a$
  – $T(1) = a$
  – $T(n) = T(n - 1) + T(n - 2) + a$

• To show: $T(n)$ is $O(2^n)$

• Proof: use induction on $n$
  – Show $P(n) = \{T(n) \leq c \cdot 2^n\}$ for some fixed constant $c$.
  – Basis: $P(0)$
    • $T(0) = a \leq c \cdot 2^0$ using $c=a$
  – Basis: $P(1)$
    • $T(1) = a \leq c \cdot 2^1$ using $c=a$
  – Strong inductive hypothesis: $P(m) = \{T(m) \leq c \cdot 2^m\}$ is true for all $m \leq k$.
  – Induction step: $P(0) \wedge \ldots \wedge P(k) \Rightarrow P(k+1)$
    • $T(k+1) \leq T(k) + T(k-1) + a \leq c \cdot 2^k + c \cdot 2^{k-1} + a = c \cdot 2^k + c \cdot 2^{k-1} + a \leq c \cdot 2^{k+1}$
      for any $c \geq \frac{1}{4} a$ and any $n \geq 2$. 
The Golden Ratio

Actually, can prove a tighter bound than $O(2^n)$.

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

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

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

= 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 $O(\phi^n)$

- Simplification: Ignore constant effort in recursive case.
  - $T(0) = a$
  - $T(1) = a$
  - $T(n) = T(n - 1) + T(n - 2)$

- Want to show $T(n) \leq c\phi^n$ for all $n \geq 0$.
  - have $\phi^2 = \phi + 1$
  - multiplying by $c\phi^n$ $\rightarrow c\phi^{n+2} = c\phi^{n+1} + c\phi^n$

- Base:
  - $T(0) = c = c\phi^0$ for $c = a$
  - $T(1) = c \leq c\phi^1$ for $c = a$

- Induction step:
  - $T(n+2) = T(n+1) + T(n) \leq c\phi^{n+1} + c\phi^n = c\phi^{n+2}$
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);
```

Time Complexity:

- Number of times loop is executed? \( n - 1 \)
- Number of basic steps per loop? Constant

\( \rightarrow \) Complexity of iterative algorithm = \( O(n) \)

Much, much, much, much, better than \( O(\phi^n) \)!
...But We Can Do Even Better!

- Denote with \( f_n \) the \( n \)-th Fibonacci number
  - \( f_0 = 0 \)
  - \( f_1 = 1 \)
  - \( f_{n+2} = f_{n+1} + f_n \)

- Note that
  \[
  \begin{pmatrix}
    0 & 1 \\
    1 & 1
  \end{pmatrix}
  \begin{pmatrix}
    f_n \\
    f_{n+1}
  \end{pmatrix}
  =
  \begin{pmatrix}
    f_{n+1} \\
    f_n + f_{n+1}
  \end{pmatrix},
  \text{thus}
  \begin{pmatrix}
    0 & 1 \\
    1 & 1
  \end{pmatrix}^n
  \begin{pmatrix}
    f_0 \\
    f_1
  \end{pmatrix}
  =
  \begin{pmatrix}
    f_n \\
    f_{n+1}
  \end{pmatrix}
  \]

- Can compute \( n \)-th power of matrix by repeated squaring in \( O(\log n) \) time.
  - Gives complexity \( O(\log n) \)
  - A little cleverness got us from exponential to logarithmic.
But We Are Not Done Yet...

• Would you believe constant time?

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

where \( \varphi = \frac{1 + \sqrt{5}}{2} \) and \( \varphi' = \frac{1 - \sqrt{5}}{2} \)
Matrix Mult in Less Than $O(n^3)$

• Idea (Strassen's Algorithm): 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_5 &= a(f - h) \\
  s_2 &= (a + d)(e + h) & s_6 &= d(g - e) \\
  s_3 &= (a - c)(e + f) & s_7 &= e(c + d) \\
  s_4 &= h(a + b)
  \end{align*}
Now Apply This Recursively – Divide and Conquer!

- Break $2^{n+1} \times 2^{n+1}$ matrices up into $4 \ 2^n \times 2^n$ submatrices
- Multiply them the same way
  $\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$
  
- where
  $S_1 = (B - D)(G + H)$  \hspace{1cm}  $S_5 = A(F - H)$
  $S_2 = (A + D)(E + H)$  \hspace{1cm}  $S_6 = D(G - E)$
  $S_3 = (A - C)(E + F)$  \hspace{1cm}  $S_7 = E(C + D)$
  $S_4 = H(A + B)$
Now Apply This Recursively – Divide and Conquer!

• Recurrence for the runtime of Strassen’s Alg
  – $M(n) = 7 \cdot M(n/2) + cn^2$
  – Solution is $M(n) = \Theta(n^{\log 7}) = \Theta(n^{2.81})$

• Number of additions
  – Separate proof
  – Number of additions is $O(n^2)$
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!