Analysis of Merge-Sort

- Recurrence describing computation time:
  \[ T(n) = c + d + e + f + 2T(n/2) + gn + h \]
  \( \leftarrow \) recurrence
  \( T(1) = i \) \( \leftarrow \) base case

- How do we solve this recurrence?

```
public static Comparable[] mergesort(Comparable[] A, int low, int high) {
    if (low < high) {
        // at least 2 elements?
        int mid = (low + high)/2;
        cost = d
        Comparable[] A1 = mergesort(A, low, mid);
        cost = T(n/2) + e
        Comparable[] A2 = mergesort(A, mid+1, high);
        cost = T(n/2) + f
        return merge(A1,A2);
    }
    ....
    cost = i
}
```

Analysis of Merge-Sort

- Recurrence:
  \[ T(n) = c + d + e + f + 2T(n/2) + gn + h \]
  \( \leftarrow \) recurrence
  \( T(1) = i \) \( \leftarrow \) base case

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

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

• How do we find the solution?

Analysis of Merge-Sort

- Recurrence:
  \[ T(n) = 2T(n/2) + n \]
  \( T(1) = 0 \)

• 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

  \[ T(n) = 2T(n/2) + n \]
  \( \leq \) \( c \) \( n \log(n) \) for some fixed constant \( c \).

  \( T(2) \) \( = \) \( 2 \log(2) \) using \( c = 1 \)

- Induction step: \( T(2^{k}) \leq c(2k) \) \( \log(k) \) \( + \) \( c \)
  \( \leq \) \( c \) \( 2k \log(k+1) \) \( + \) \( c \)
  \( \leq \) \( c \) \( 2k \log(k) \) \( + \) \( c \)

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

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 \( T(n) = aT(n/b) + f(n) \)
- 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 \( T(n) = O(n \log(n)) \)
  - if both grow at same rate

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

Some common cases:

- $T(n) = T(n-1) + 1$  \quad $T(n)$ is $O(n)$  \quad Linear Search
- $T(n) = T(n-1) + n$  \quad $T(n)$ is $O(n^2)$  \quad QuickSort worst-case
- $T(n) = T(n/2) + 1$  \quad $T(n)$ is $O(\log n)$  \quad Binary Search
- $T(n) = T(n/2) + n$  \quad $T(n)$ is $O(n)$
- $T(n) = 2T(n/2) + n$  \quad $T(n)$ is $O(n \log n)$  \quad MergeSort
- $T(n) = 2T(n-1)$  \quad $T(n)$ is $O(2^n)$

How long would it take @ 1 instruction / \(\mu\)sec?

<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>(\mu)sec</td>
<td>1/10,000 sec</td>
<td>1/2,000 sec</td>
<td>1/500 sec</td>
<td>1/100 sec</td>
<td>1/10 sec</td>
</tr>
<tr>
<td>(\mu)sec</td>
<td>0.1 sec</td>
<td>0.2 sec</td>
<td>0.5 sec</td>
<td>1 sec</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>(\mu)sec</td>
<td>1/1,000 sec</td>
<td>1 sec</td>
<td>3.3 trillion years</td>
<td>400 trillion centuries</td>
<td>10 billion</td>
</tr>
<tr>
<td>(\mu)sec</td>
<td>0.2 sec</td>
<td>0.3 billion years</td>
<td>(\approx) 10^85 number of centuries</td>
<td>(\approx) 10^201 number of centuries</td>
<td>(\approx) 10^744 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:
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$, $n \geq 2$

  ```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, ...

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?
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: $P(0) \Rightarrow \ldots \Rightarrow P(k) \Rightarrow P(k+1)$
      - $T(k+1) \leq T(k) + T(k-1) + a \leq c \cdot 2^{k+1} + a = c \cdot 2^{k+1} + a \leq c \cdot 2^{k+2}$

- Simplification: Ignore constant effort in recursive case.
  - $T(0) = a$
  - $T(1) = a$
  - $T(n) = T(n-1) + T(n-2)$
- Want to show $T(n) \leq c \cdot \phi^n$ for all $n \geq 0$.
  - have $\phi^2 = 1 + \sqrt{5}$
  - multiplying by $\phi^n \rightarrow \phi^{n+1} = \phi \phi^n + \phi \phi^{n-1}$

- Base:
  - $T(0) = a \leq c \cdot \phi^0$ for $c = a$
  - $T(1) = a \leq c \cdot \phi^0$ for $c = a$
- Induction step:
  - $T(n+2) = T(n+1) + T(n) \leq c \cdot \phi^{n+1} + c \cdot \phi^n = c \cdot \phi^{n+2}$

Fibonacci Recurrence is $O(\phi^n)$

- Can we do 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 $\binom{0}{1} \binom{1}{0} + \binom{1}{1} \binom{1}{1} + \binom{1}{2} \binom{1}{1}$
  - 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 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 $\binom{0}{1} \binom{1}{0} + \binom{1}{1} \binom{1}{1} + \binom{1}{2} \binom{1}{1}$
- 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.

The Golden Ratio

- Actually, can prove a tighter bound than $O(2^n)$.
  - $\phi = \frac{1 + \sqrt{5}}{2}$
  - $\phi^2 = \phi + 1$
  - $\phi^2 = \frac{1 + \sqrt{5}}{2}$
- The ratio of sum of sides $(a+b)$ to longer side $(b)$
  - $\phi = 1.618\ldots$
  - $\phi = 1 + \sqrt{5}$
  - $\phi = \frac{1 + \sqrt{5}}{2}$

Can We Do Better?

- Number of times loop is executed? $n-1$
- Number of basic steps per loop? Constant
- Complexity of iterative algorithm = $O(n)$

Much, much, much, much, better than $O(\phi^n)!$

But We Are Not Done Yet...

- Would you believe constant time?
  - $f_n = \frac{\phi^n - \phi'^n}{\sqrt{5}}$
  - $\phi' = \frac{1 - \sqrt{5}}{2}$
  - $\phi = \frac{1 + \sqrt{5}}{2}$
  - $\phi^2 = \phi + 1$
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:

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

- 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

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

- 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!

- Recurrence for the runtime of Strassen’s Alg
  - $M(n) = 7 \cdot M(n/2) + cn^2$
  - Solution is $M(n) = O(\log_7^5 n) = O(n^{2.81})$
- 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!