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

```java
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);
    }
    // at most 1 element
    cost = 1;
    return A;
}
```

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

• How do we solve this recurrence?

• Recurrence for MergeSort:
  \[ T(n) = 2T(n/2) + cn + d, \quad n > 1 \]
  \[ T(1) = a \]

• Solution is \( T(n) = O(n \log n) \)
Recurrence for MergeSort:

\[ T(n) = 2T(n/2) + cn + d, \quad n > 1 \]

\[ T(1) = e \]

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

Tricks:
- Take larger constants to simplify
- Ignore small inputs
- Use inequalities instead of equalities

Analysis of MergeSort

Proof: Strong induction on \( n \). Show that

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

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

Basis:
\( T(2) \leq c \leq c2 \log 2 \)
\( T(3) \leq c \leq c3 \log 3 \)

Induction step:
\( T(n) \leq 2T(n/2) + cn \leq 2c(n/2) \log (n/2) + cn \) (IH)
\( = cn (\log n – 1) + cn = cn \log n \)

Analysis of MergeSort

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 \( n \log b \)

\[ T(n) = O(f(n)) \]

\[ T(n) = O(\sqrt[n]{a} n) \]

\[ T(n) = T(n – 1) + 1 \rightarrow T(n) = O(n) \]

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

\[ T(n) = T(n/2) + n \rightarrow T(n) = O(n) \]

\[ T(n) = 2T(n/2) + n \rightarrow T(n) = O(n \log n) \]

MergeSort

\[ T(n) = 2T(n/2) \rightarrow T(n) = O(2^n) \]

Recurrence Examples

- Protons in the known universe ~ 126 digits
- Usec since the big bang ~ 24 digits
- The big bang was 15 billion years ago (5 \( \cdot \) \( 10^{17} \) secs)

How long would it take @ 1 instruction / usec?

How long would it take @ 1 instruction / usec?

- Source: D. Harel, Algorithmics
The Fibonacci Function

Mathematical definition:
\[
\text{fib}(0) = 0 \\
\text{fib}(1) = 1 \\
\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \ n \geq 2
\]

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

Execution of fib(4):

```
fib(4) → fib(3) → fib(2) → 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

\[ \phi = \frac{a+b}{b} = \frac{b}{a} \]
\[ \phi^2 = \phi + 1 \]
\[ \phi = \frac{1 + \sqrt{5}}{2} = 1.618... \]

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

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix} \begin{bmatrix}
1 \\
0
\end{bmatrix} = \begin{bmatrix}
1 \\
0
\end{bmatrix} \begin{bmatrix}
1 & 1 \\
0 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 1 \\
0 & 1
\end{bmatrix}
\]

Repeated squaring of the matrix gives \(O(\log n)\)

But We Are Not Done Yet...

- Would you believe constant time?

\[
\psi = \sqrt[3]{5} - 1
\]

\[
\psi' = \sqrt[3]{5} + 1
\]

Now Apply This Recursively – Divide and Conquer!

Matrix Multiplication in Less Than \(O(n^3)\)
(\text{Strassen's Algorithm})

- Idea: Naive \(2 \times 2\) matrix multiplication takes 8 scalar multiplications, but we can do it in 7:

\[
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix} \begin{bmatrix}
e & f \\
g & h
\end{bmatrix} = \begin{bmatrix}
ae + bg + ch + dh & af + bh + cg + dh \\
cg - eh + ds - fh
\end{bmatrix}
\]

where

\[
s_1 = (b - d)(g + h)
\]
\[
s_2 = (a - d)(e + h)
\]
\[
s_3 = (c - a)(e + f)
\]
\[
s_4 = h(a + b)
\]
\[
s_5 = (a + d)(e + h)
\]
\[
s_6 = d(g - e)
\]
\[
s_7 = e(c + d)
\]

\[
s_1 + s_2 - s_4 + s_6
\]
\[
s_4 + s_5
\]
\[
s_2 - s_3 + s_5 - s_7
\]

Now Apply This Recursively – Divide and Conquer!

\[
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix} = \begin{bmatrix}
s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\
s_2 - s_3 + s_5 - s_7
\end{bmatrix}
\]

Is That the Best You Can Do?

- How about \(3 \times 3\) for a base case?
  - best known is 23 multiplications
  - not good enough to beat Strassen

- In 1978, Victor Pan discovered how to multiply \(70 \times 70\) 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 known lower bound is still \(\Omega(n^2)\)
Moral: Complexity Matters!

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