CS 341: ALGORITHMS

Lecture 6: divide and conquer

Slides by Trevor Brown (some material from Doug Stinson and Semih Salihoglu)

trevor.brown@uwaterloo.ca

(http://tbrown.pro)
FINISHING UP LAST TIME
**REVISITING THE RECURSION TREE METHOD**

- Some recurrences with complex \( f(n) \) functions (such as \( f(n) = \log n \)) can still be solved “by hand”

- Example: Let \( n = 2^i \); \( T(1) = 1 \); \( T(n) = 2T\left(\frac{n}{2}\right) + n \log n \)

<table>
<thead>
<tr>
<th>Level</th>
<th># of nodes</th>
<th>runtime per node</th>
<th>total runtime for level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( \frac{n}{2} \log \frac{n}{2} )</td>
<td>( n \log \frac{n}{2} )</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>( \frac{n}{4} \log \frac{n}{4} )</td>
<td>( n \log \frac{n}{4} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( k )</td>
<td>( 2^k )</td>
<td>( \frac{n}{2^k} \log \frac{n}{2^k} )</td>
<td>( n \log \frac{n}{2^k} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \log n )</td>
<td>( n )</td>
<td>1</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Must **sum** the values over all levels!

**Top level recursive call**

**Base case:** leaves of recursion tree
Recall \( n = 2^j \); \( T(1) = 1 \); \( T(n) = 2T \left( \frac{n}{2} \right) + n \log n \)

\[
T(n) = \sum_{k=0}^{\log n-1} n \log \frac{n}{2^k} + n = n \sum_{k=0}^{\log n-1} \log \frac{n}{2^k} + n
\]

\[
= n \left( \sum_{k=0}^{\log n-1} \log n - \sum_{k=0}^{\log n-1} \log 2^k \right) + n
\]

\[
= n \left( \log n \log n - \sum_{k=0}^{\log n-1} k \right) + n
\]

\[
= n \left( \log^2 n - \frac{(\log n - 1) \log n}{2} \right) + n
\]

\[
\in \Theta(n \log^2 n)
\]
FAST MATRIX MULTIPLICATION

Matrix Multiplication

Neural Network
Matrix Multiplication

- Input: $A$ and $B$
- Output: their product $C = AB$
- Word-RAM model (64-bit ints)
- Naïve algorithm for $n \times n$ matrices:
  - For each output cell $C_{ij}$
    
    \[
    C_{ij} = \text{DotProd}(\text{row}_i(A), \text{col}_j(B)^T) 
    \]
    
    \[
    = \sum_{k=1}^{n} A_{ik}B_{kj}
    \]
  - Running time?
ATTEMPTING A BETTER SOLUTION

• What if we first **partition** the matrix into **sub-matrices**
• Then **divide and conquer** on the **sub-matrices**
• Example of partitioning: 4x4 matrix into four 2x2 matrices

\[
\begin{bmatrix}
M_1 & M_2 \\
M_3 & M_4 \\
\end{bmatrix} =
\begin{bmatrix}
a_1 & b_1 & a_2 & b_2 \\
c_1 & d_1 & c_2 & d_2 \\
a_3 & b_3 & a_4 & b_4 \\
c_3 & d_3 & c_4 & d_4 \\
\end{bmatrix}
\]
D&C Matrix Multiplication: Problem Decomposition

Let

\[ A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \]

If \( A, B \) are \( n \) by \( n \) matrices, then \( a, b, \ldots, h, r, s, t, u \) are \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices, where

\[
\begin{align*}
r &= ae + bg \\
t &= ce + dg
\end{align*}
\]

\[
\begin{align*}
s &= af + bh \\
u &= cf + dh
\end{align*}
\]

We require 8 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).

What is the complexity of the resulting divide-and-conquer algorithm?
DERIVING A RECURRENCE

• MatrixMultiply performs $T(n)$ work, consisting of:
  
  • **Eight recursive calls** to perform multiplications on $\frac{n}{2} \times \frac{n}{2}$ matrices
  
  • **Four additions** of $\frac{n}{2} \times \frac{n}{2}$ matrices
    • Work: $4 \left( \frac{n}{2} \cdot \frac{n}{2} \right) \in \Theta(n^2)$

  $\begin{align*}
  r &= ae + bg \\
  t &= ce + dg \\
  s &= af + bh \\
  u &= cf + dh
  \end{align*}$

  $T(n) = 8T \left( \frac{n}{2} \right) + \Theta(n^2)$
**SOLVE USING THE (SIMPL.) MASTER THEOREM**

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y), \text{ where } n \text{ is a power of } b.$$  

Denote $x = \log_b a$. Then

$$T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } y < x \\
\Theta(n^x \log n) & \text{if } y = x \\
\Theta(n^y) & \text{if } y > x.
\end{cases}$$

Recurrence: $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$

$a = 8 \quad b = 2 \quad y = 2 \quad x = \log_2 8 = 3$

$x > y \text{ so } T(n) \in \Theta(n^x)$

$a = ? \quad b = ? \quad y = ? \quad x = ?$

$T(n) \in \Theta(n^3)$
Define

\[ A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \]

\[ P_1 = a(f - h) \]
\[ P_3 = (c + d)e \]
\[ P_5 = (a + d)(e + h) \]
\[ P_7 = (a - c)(e + f) \]
\[ P_2 = (a + b)h \]
\[ P_4 = d(g - e) \]
\[ P_6 = (b - d)(g + h) \]

Then, compute

\[ r = P_5 + P_4 - P_2 + P_6 \]
\[ s = P_1 + P_2 \]
\[ t = P_3 + P_4 \]
\[ u = P_5 + P_1 - P_3 - P_7 \]

This simplifies to

\[ t = ce + dg \]

We now require only 7 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C' = AB \).
The recurrence is $T(k) = 7T(k/2) + \Theta(k^2)$, so $T(k) \in \Theta(k^{\log_2 7}) = \Theta(n^{2.81})$ by the Master Theorem.

Details: $a = 7$, $b = 2$, so $x = \log_2 7 = 2.81$, $y = 2$, $x > y$ so we are in case 1 and $T(n) = \Theta(n^x) = \Theta(n^{2.81})$.

**Strassen's algorithm** was improved in 1990 by Coppersmith-Winograd. Their algorithm has complexity $O(n^{2.376})$. Some slight improvements have been found more recently.

<table>
<thead>
<tr>
<th>How much better is $\Theta(n^{2.81})$ than $\Theta(n^3)$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ( n = 10,000 )</td>
</tr>
<tr>
<td>$n^{2.81} \approx 174 \text{ billion}$</td>
</tr>
<tr>
<td>$n^3 = 1 \text{ trillion} (~6x \text{ more})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How much better is $\Theta(n^{2.376})$ than $\Theta(n^3)$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ( n = 10,000 )</td>
</tr>
<tr>
<td>$n^{2.376} \approx 3.2 \text{ billion}$</td>
</tr>
<tr>
<td>$n^3 = 1 \text{ trillion} (~312x)$</td>
</tr>
</tbody>
</table>
THE SELECTION PROBLEM

NATURAL SELECTION
in progress...
**THE SELECTION PROBLEM**

- Input: An array $A$ containing $n$ distinct integer values, and an integer $k$ between 1 and $n$
- Output: The $k$-th smallest integer in $A$
- **Minimum** is a special case where $k = 1$
- **Median** is a special case where $k = \frac{n}{2}$
- **Maximum** is a special case where $k = n$
- Simple algorithm for solving selection?
QuickSelect

Suppose we choose a **pivot** element $y$ in the array $A$, and we **restructure** $A$ so that all elements less than $y$ precede $y$ in $A$, and all elements greater than $y$ occur after $y$ in $A$. (This is exactly what is done in **Quicksort**, and it takes **linear time**.)


Then the $k$th smallest element of $A$ is

$$
\begin{cases}
  y & \text{if } k = posn \\
  \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\
  \text{the } (k - posn)\text{th smallest element of } A_R & \text{if } k > posn.
\end{cases}
$$

We make (at most) one recursive call at each level of the recursion.
QuickSelect(k, A[1..n])
    if n = 1 then return A[1] // base case

    y = A[1] // pick an arbitrary pivot
    (AL, AR, posn) = Restructure(A, y)

    if k == posn return y
    else if k < posn then return QuickSelect(k, AL)
    else return QuickSelect(k - posn, AR)

Restructure(A[1..n], y)
    AL = new array[1..n] // allocate more than enough
    AR = new array[1..n] // to avoid need for expansion
    nL = 0, nR = 0

    for i = 1 .. n
        if A[i] < y then AL[nL++] = A[i]
        else A[i] > y then AR[nR++] = A[i]

    return (AL, AR, nL+1) // nL+1 is the new index of y
Average-case Analysis of QuickSelect

We say that a pivot is **good** if \( posn \) is in the middle half of \( A \), i.e., \( n/4 \leq posn \leq 3n/4 \).

The probability that a pivot is good is 1/2.

On average, after **two iterations**, we will encounter a good pivot.

If a pivot is good, then \( |A_L| \leq 3n/4 \) and \( |A_R| \leq 3n/4 \).

With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.

Let’s consider the **average-case** recurrence relation:

\[
T(n) = T(3n/4) + \Theta(n).
\]

Apply the **Master Theorem** with \( a = 1 \), \( b = 4/3 \) and \( y = 1 \). Here \( x = \log_{4/3} 1 = 0 < 1 = y \) so we are in case 3.

This yields \( T(n) \in \Theta(n) \) on average.
Here is a more rigorous proof of the average-case complexity: We say the algorithm is in phase $j$ if the current subarray has size $s$, where

$$n \left( \frac{3}{4} \right)^{j+1} < s \leq n \left( \frac{3}{4} \right)^j.$$  

Let $X_j$ be a random variable that denotes the amount of computation time occurring in phase $j$. If the pivot is in the middle half of the current subarray, then we transition from phase $j$ to phase $j+1$. This occurs with probability $1/2$, so the expected number of recursive calls in phase $j$ is 2. The computing time for each recursive call is linear in the size of the current subarray, so $E[X_j] \leq 2cn(3/4)^j$ (where $E[\cdot]$ denotes the expectation of a random variable). The total time of the algorithm is given by $X = \sum_{j \geq 0} X_j$. Therefore

$$E[X] = \sum_{j \geq 0} E[X_j] \leq 2cn \sum_{j \geq 0} (3/4)^j = 8cn \in O(n).$$

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}, \text{ for } |r| < 1.$$