Outline

• Divide and Conquer design paradigm
  – matrix multiplication

• Dynamic programming design paradigm
  – Fibonacci numbers
  – weighted interval scheduling
  – knapsack
  – matrix-chain multiplication
  – longest common subsequence

Discrete Fourier Transform (DFT)

• Given $n$-th root of unity $\omega$, $DFT_n$ is a linear map from $\mathbb{C}^n$ to $\mathbb{C}^n$:

$$
\begin{pmatrix}
(\omega^0)^0 & (\omega^0)^1 & (\omega^0)^2 & \cdots & (\omega^0)^{n-1} \\
(\omega^1)^0 & (\omega^1)^1 & (\omega^1)^2 & \cdots & (\omega^1)^{n-1} \\
(\omega^2)^0 & (\omega^2)^1 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(\omega^{n-1})^0 & (\omega^{n-1})^1 & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1}
\end{pmatrix}
$$

• $(i,j)$ entry is $\omega^{ij}$

Fast Fourier Transform (FFT)

• $DFT_n$ has special structure (assume $n = 2^k$)
  – reorder columns: first even, then odd
  – consider exponents on $\omega$ along rows:

<table>
<thead>
<tr>
<th>multiples of:</th>
<th>same multiples plus:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
</tr>
</tbody>
</table>

• so we are actually computing:

$$
\begin{pmatrix}
D_{\text{even}} \\
D_{\text{odd}}
\end{pmatrix}
\begin{pmatrix}
\omega^{00} & \omega^{01} & \omega^{02} & \cdots & \omega^{0n/2} \\
\omega^{10} & \omega^{11} & \omega^{12} & \cdots & \omega^{1n/2} \\
\omega^{20} & \omega^{21} & \omega^{22} & \cdots & \omega^{2n/2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\omega^{n/2,0} & \omega^{n/2,1} & \omega^{n/2,2} & \cdots & \omega^{n/2,n/2}
\end{pmatrix}
\begin{pmatrix}
y_{\text{even}} \\
y_{\text{odd}}
\end{pmatrix}
$$

• so to compute $DFT_n \times$:

$D = \text{diagonal matrix diag}((\omega^0)^0, (\omega^1)^1, \cdots, (\omega^{n/2-1})^{n/2-1})$

Fast Fourier Transform (FFT)

• Running time?
  – $T(1) = 1$
  – $T(n) = 2T(n/2) + O(n)$
  – solution: $T(n) = O(n \log n)$
matrix multiplication

\[ \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} \]

- given \( n \times n \) matrices \( A, B \)
- compute \( C = AB \)
- standard method: \( O(n^3) \) operations

- Strassen: \( O(n^{\log_2 7}) = O(n^{2.81}) \)

Strassen’s algorithm

\[ \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} \]

- how many product operations?
- Strassen: it is possible with 7 (!!) products
  - 7 products of form: (linear combos of a entries) \( \times \) (linear combos of b entries)
  - result is linear combos of these 7 products

Key: identity holds when entries above are \( n/2 \times n/2 \) matrices rather than scalars

Strassen’s algorithm

- 7 recursive calls
- additions/subtractions are entrywise: \( O(n^2) \)

- running time recurrence?
  \[ T(n) = 7T(n/2) + O(n^2) \]

Solution: \( T(n) = O(n^{\log_2 7}) = O(n^{2.81}) \)
discovering Strassen

\[
\begin{array}{ccc}
\mathbf{a}_{11} & \mathbf{a}_{12} & \\
\mathbf{b}_{21} & \mathbf{b}_{22} & \\
\end{array}
\times
\begin{array}{ccc}
\mathbf{b}_{11} & \mathbf{b}_{12} & \\
\mathbf{b}_{21} & \mathbf{b}_{22} & \\
\end{array}
=
\begin{array}{ccc}
\mathbf{c}_{11} & \mathbf{c}_{12} & \\
\mathbf{c}_{21} & \mathbf{c}_{22} & \\
\end{array}
\]

express these as linear combinations of rank-1 matrices

e.g.:

Dynamic programming

“programming” = “planning”
“dynamic” = “over time”

- basic idea:
  - identify subproblems
  - express solution to subproblem in terms of other “smaller” subproblems
  - build solution bottom-up by filling in a table
- defining subproblem is the hardest part
Dynamic programming

- Simple example: computing Fibonacci #s
  
  \( f(1) = f(2) = 1 \)
  
  \( f(i) = f(i-1) + f(i-2) \)

- recursive algorithm:
  
  Fib(n)
  
  1. if \( n = 1 \) or \( n = 2 \) return(1)
  2. else return(Fib(n-1) + Fib(n-2))

  \( \text{running time?} \)

---

Weighted interval scheduling

- job \( j \) starts at \( s_j \), finishes at \( f_j \), weight \( v_j \)
- jobs compatible if they don't overlap

Goal: find maximum weight subset of mutually compatible jobs.

---

Weighted interval scheduling

- label jobs by finishing time \( f_j \)

Definition: \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\( p(8) = 5 \)

---

Weighted interval scheduling

- subproblem \( j \): jobs 1...\( j \)

\( \text{OPT}(j) = \text{value achieved by optimum schedule} \)

- relate to smaller subproblems

  - case 1: use job \( j \)
    - can't use jobs \( p(j)+1, ..., j-1 \)
    - must use optimal schedule for 1...\( p(j) = \text{OPT}(p(j)) \)

  - case 2: don't use job \( j \)
    - must use optimal schedule for 1...\( j-1 = \text{OPT}(j-1) \)
Weighted interval scheduling

- job \( j \) starts at \( s_j \), finishes at \( f_j \), weight \( v_j \)
- \( \text{OPT}(j) = \max \{v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\} \)

recursive solution?

running time?

Knapsack

- item \( i \) has weight \( w_i \) and value \( v_i \)
- goal: pack knapsack of capacity \( W \) with maximum value set of items
  - greedy by weight, value, or ratio of weight/value all fail

- subproblems:
  - optimum among items 1...i-1?

subproblems:

- optimum among items 1...i-1, with total weight \( w \)
  - case 1: don't use item \( i \)
    - \( \text{OPT}(i) = \text{OPT}(i-1) \)
  - case 2: do use item \( i \)
    - \( \text{OPT}(i) = ? \)

subproblems, second attempt:

- optimum among items 1...i-1, with total weight \( w \)
  - \( \text{OPT}(i, w) = \max \{v_i + \text{OPT}(i-1, w - w_i), \text{OPT}(i-1, w)\} \)
  - order to fill in the table?
Knapsack

Knapsack($v_1, w_1, ..., v_n, w_n, W$)
1. OPT(i, 0) = 0 for all i
2. for i = 1 to n
3. for w = 1 to W
4. if $w_i > w$ then OPT(i, w) = OPT(i-1, w)
5. else OPT(i, w) = ($v_i +$ OPT(i-1, $w-w_i), OPT(i-1, w))
6. return(OPT(n, W))

- Running time?
  - $O(nW)$
- space: $O(nW)$ – can improve to $O(W)$ (how?)
- how do we actually find items?

Matrix-chain multiplication

- Sequence of matrices to multiply
e.g.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>11</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

- goal: find best parenthesization
  - e.g.: ((A-B)C)D = 10·3·11 + 10·11·9 + 10·9·1 = 1410
  - e.g. (A-(B(CD)) = 11·9·1 + 3·11·1 + 10·3·1 = 162

April 29, 2014 CS38 Lecture 9 34

Matrix-chain multiplication

- Sequence of $n$ matrices to multiply, given by $a_1, a_2, ..., a_{n+1}$
- Goal: output fully parenthesized expression with minimum cost
  - fully parenthesized = single matrix: (A) or
  - product of two fully parenthesized: (…)(…)
- try subproblems for ranges:
  $OPT(1,n) = \min_k OPT(1,k) + OPT(k+1,n) + a_1a_{k+1}a_{n+1}$

April 29, 2014 CS38 Lecture 9 33

Matrix-chain multiplication

- running time?
  - $O(n^3)$
- print out the optimal parenthesization?
  - store chosen $k$ in each cell

April 29, 2014 CS38 Lecture 9 35