Chess Endgames

• Ken Thompson (Bell Labs) solved all chess endgame positions with $\leq 5$ pieces (1986). Some surprises:
  - $K+Q$ beats $K+2B$. (Since 1634, believed to be draw).
  - 50 moves without a pawn move or capture is not sufficient to ensure it's a draw.
  - By now, most (all?) 6-piece endgames are solved too.

• Searching all possible games with 5 pieces, going forward 50 or so moves, is infeasible (today).
  - If typical position has 4 moves, search is $2^{100}$ long
  - $2^{100} = 2^{45}$ ops/second $\times$ $2^{25}$ seconds/year $\times$ 1 billion years.

• There is a better way!
“Retrograde Analysis”

Insight: there aren't so many different positions.
   With 5 specific pieces, there are \( \leq 64^5 \) positions.
   \( 64^5 \) is about \( 10^9 \) (large, but quite feasible).

Make memory location for each position.
   “Position” includes whose move it is.
   Store count of possible moves (for Black moves).

Initialize work list to all won positions for white.

For each “win” \( P \) in list, process all positions \( Q \) that can move to \( P \) (unless \( Q \) was already known to win for white).
   If \( Q \) is a white move, mark \( Q \) as win and add it to list.
   If \( Q \) is a black move, decrease possible move count by 1
   If count == 0, mark \( Q \) as win for white and put it on list.

Why did this reduce work??

Can evaluate game tree by divide & conquer
   Win-for-white (\( P \)) {
      let \( Q_1, ..., Q_k \) be all possible moves from \( P \);
      \( W_i = \text{Win-for-white}(Q_i); \)
      if \( P \) is a white move and any \( W_i = \text{T} \) then return \( \text{T} \);
      if \( P \) is a black move and all \( W_i = \text{T} \) then return \( \text{T} \);
      return \( \text{F} \); }

Inefficient due to overlapping subproblems.
   Many subproblems share the same sub-subproblems.
**Dynamic Programming**

Motto: “It’s not dynamic and it’s not programming”

For a problem with “optimal substructure” property...
   Means that, like D&C, you can build solution to bigger problem from solutions to subproblems.

... and “overlapping subproblems”
   which makes D&C inefficient.

**Dynamic Programming**: builds a big table and fills it in from small subproblems to large.

Another approach is **memoization**:
   Start with the big subproblems, Store answers in table as you compute them. Before starting on subproblem, check if you’ve already done it.

---

**Example: Longest Common Substring**

- **Z = z₁ z₂ ... zₖ** is a substring of **X = x₁ x₂ ... xₙ** if you can get Z by dropping letters of X.  
  - Example: “Hello world” is a substring of “Help! I'm lost without our landrover”.

- **LCS problem**: given strings X and Y, find the length of the longest Z that is a substring of both (or perhaps find that Z).
LCS has “optimal substructure”

Suppose length(X) = x and length(Y) = y.

Let X[a:b] mean the a-th to b-th character of X.

Observation: If Z is a substring of X and Y, then the last character of Z is the last character of both X and Y ...

so \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) || X[x] \)

or of just X, \( \text{LCS}(X,Y) = \text{LCS}(X, Y[1:y-1]) \)

or of just Y, \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y) \)

or neither \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) \)

Dynamic Programming for LCS

Table: \( L(i,j) = \) length of LCS ( \( X[1:i], Y[1:j] \) ).

| If the last character of both X and Y is the same ... | \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) || X[x] \) |
| otherwise,                                       | \( \text{LCS}(X,Y) = \text{LCS}(X, Y[1:y-1]) \) |
| or                                               | \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y) \) |
| or                                               | \( \text{LCS}(X,Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) \) |

tells us:

\( L(i,j) = \max \{ L(i-1,j-1) + (X[i]==Y[i]), L(i,j-1), L(i-1,j) \} \)

Each table entry is computed from 3 earlier entries.
Dynamic Programming for LCS

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>W</th>
<th>O</th>
<th>R</th>
<th>L</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>V</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

= match (increment)

But – there are other substructures

• (Work out the following at board):
  - Can we have \( T(i) = \text{LCS}(X[1:i], Y[1:i]) \) ?
  - How about, “the first half of \( X \) matches up with some part of \( Y \), and the rest with the rest.”
    Suggests \( \text{LCS}(X,Y) = \text{maximum} \ (\text{for } i=1,...,y) \) of
    \( \text{LCS}(X[1:x/2], Y[1:i]) + \text{LCS}(X[x/2+1,x], Y[i+1,y]) \)
  - We can make the table 1-D instead of 2-D!
Summary

For a good dynamic programming algorithm:
- Table should be low-dimensional
  - keeps memory requirements low.
- Each table entry should be fast to calculate
  - keeps complexity low.

May decompose problems differently than a good Divide & Conquer algorithm.

How to get your name on an algorithm:
Use Dynamic Programming

- Earley’s algorithm:
  Parse a string using a CFG.
- Viterbi’s algorithm:
  Break string into codewords
- Smith-Waterman algorithm:
  Match up two strings of DNA or protein
- Floyd-Warshall:
  All-pairs shortest paths.
How to find a Dynamic Programming algorithm

- For a moderately small problem, draw a tree representing possible choices you have in finding an optimal solution.
- Look for instances of common subproblems.
- Figure out how you could store the subproblem solutions in a table.
  - make a careful statement of what the table entries mean
- Figure out where to start filling in the table.
  - typically the smallest subproblems at the tree’s bottom.
- Figure out how to fill in other table entries.
  - recursive step: how do you solve a larger subproblem given the solutions to smaller subproblems.

Example: Viterbi’s Algorithm

- Given a Hidden Markov Model (HMM) and a string \( s = s_1 \ s_2 \ldots \ s_n \), find most probable path taken by the Markov model that produces \( s \).
  - HMM is a set of states, a set of transition probabilities between the states, and, for each state and each output character, a probability of producing that character at that state.
  - The probability of a path is the product of the transition probabilities along the path and the probabilities of producing the observed character are the states along the path.
Discovering Viterbi’s algorithm

Example HMM:

State 1
Pr{A} = .2
Pr{B} = .8

State 2
Pr{A} = .5
Pr{B} = .8
Pr{C} = .5

State 3
Pr{A} = .3
Pr{B} = .4
Pr{C} = .3

The two red subtrees are identical!
They both represent the choices for producing "CA" starting in state 2.

How to find a Dynamic Programming algorithm

• Figure out how you could store the subproblem solutions in a table.
  - Index the table by a state and a level in the tree

• Make a careful statement of what the table entries mean.
  - \( T(i,k) = \text{Probability of the most-probable path that starts in state } i \text{ and produces the string } s_k, s_{k+1}, ..., s_n \)

• Figure out where to start filling in the table.
  - You can fill in \( T(i,n) = \text{Pr\{state } i \text{ produces } s_n\} \) for each \( i \).

• Figure out how to fill in other table entries.
  - \( T(i,k) = \text{Pr\{i produces } s_i\} \times \max_{\text{state } j} \{\text{Pr\{i moves to } j\} \times T(j,k+1)\} \)
Yet another example

Disclaimer: dynamic programming is NOT the best way to solve this problem!

• Given matrices $M_1$, $M_2$, ..., $M_n$, where $M_i$ has $r_i$ rows and $c_i$ columns, find the best order of performing the computation.
  - It will be some parenthesization, e.g. $(M_1 \times M_2) \times (M_3 \times (\cdots \times M_n)\cdots)$
  - Note that for $i = 1,...,n-1$, $c_i = r_{i+1}$.
  - Assume time to multiply an $r$ by $s$ matrix with an $s$ by $t$ matrix is $rst$.

• We worked out an example, found subproblems like “What is time to compute $M_i \times M_{i+1} \times \cdots \times M_k$?”

• This led us to an $\Theta(n^3)$ algorithm.

---

Protein (or DNA) String Matching

Biological sequences:

A protein is a string of amino acids (20 “characters”).
DNA is a string of base pairs (4 “characters”).

Databases of known sequences:

SDSC handles the Protein Data Base (PDB).
Human Genome ($3 \times 10^9$ base pairs) computed in '01.

String matching problem:

Given a query string, find similar strings in database.
Smith-Waterman algorithm is most accurate method.
FastA and BLAST are faster approximations.
Smith-Waterman Algorithm

• Like LCS, but more sophisticated:

• Each pair \((x,y)\) of characters has “affinity” \(A(x,y)\).
  - easy to handle – add affinity instead of “1”.

• Skipping characters incurs a penalty
  - \(c_0 + k c_1\) penalty for skipping over \(k\) characters.

\[
\begin{array}{cccccc}
-4 & -4 & -4 & -6 \quad \text{using} \quad c_0=2, c_1=1 \\
M Y C & R O & A R & E L I C \\
5 & 3 & 5 & 5 & 5 & 5 & 5 & 2 \\
M I C & R O & A R & E L I S \\
\end{array}
\]

\[
\begin{array}{cccc}
P & A L L & M \\
-3 & -5 & -3 \\
\end{array}
\]

Score = 50 - 29 = 21

Smith-Waterman Algorithm

We could handle skips by:

\[
SW(i,j) = \max \{ 0, SW(i-1, j-1) + A(X_i, Y_j), \\
\max_{k=1}^{j} \{ SW(i-k, j) - c_0 - c_1 k \}, \\
\max_{k=1}^{i} \{ SW(i, j-k) - c_0 - c_1 k \}, \\
\max_{k=1}^{i} \max_{m=1}^{j} \{ SW(i-k, j-m) - 2c_0 - c_1 (k+m) \} \}
\]

This would take \(O(n^4)\) time.

Fortunately, the last term (the double-max) is redundant.

Go from \(SW(i-k, j-m)\) to \(SW(i, j-k)\) and then to \(SW(i, j)\).

Eliminating it reduces the complexity to \(O(n^3)\).
Smith-Waterman in $O(n^2)$ time

At cell $i,j$ of table keep three numbers:

- $s(i,j) =$ score of best matching of $X[1:i]$ and $Y[1:j]$.
- $t(i,j) =$ best score of $X[1:i]$ and $Y[1:j]$ that skips $X_i$.
- $u(i,j) =$ best score of $X[1:i]$ and $Y[1:j]$ that skips $Y_j$.

At each cell, compute:

- $t(i,j) = \max \{ s(i-1,j)-c_0-c_1, t(i-1,j)-c_1 \}$
- $u(i,j) = \max \{ s(i,j-1)-c_0-c_1, u(i,j-1)-c_1 \}$
- $s(i,j) = \max \{ s(i-1,j-1)+A(x_i,y_j), t(i,j), u(i,j), 0 \}$