**Outline for the Evening**

- DNA
- Approximate String Matching
- Approximate String Searching
- Dynamic Programming
- Longest Common Subsequence
- DNA reconstruction
- Contiguous Ordering and PQ-trees

**DNA**

- DNA is a large molecule that can be abstractly defined as a sequence of symbols from the set, A, C, G, T, called nucleotides.
- The human genome has about 3 billion nucleotides.
  - A huge percentage of the genome is shared by all humans.
  - Some of the variation makes us different.
  - Some of the variation is inconsequential.
  - The human genome is still being discovered.

**Approximate Matching**

- Two DNA sequences approximately match if one can be transformed into the other by a short sequence of replacements and insertions of gaps.
- Example:
  - $S = AGCATG$
  - $T = AGATCGT$
- Approximate matching: $S' = AG - - CATG$ is a gap
  - $T' = AGATCGT -$

**Applications of Approximate Matching**

- DNA string alignment.
  - Given two similar DNA sequences find the best way to align them to the same length.
- DNA database searching.
  - Find DNA sequences that are similar to the query.
- Approximate text matching for searching.
  - agrep in unix
- Spell checking
  - Find the words that most closely match the misspelled word.
Scoring an Approximate Matching

- We need a way of scoring the quality of an approximate matching.
- A scoring function is a mapping $\sigma$ from $\{A, C, G, T, -\}^2$ to integers.
  - The quantity $\sigma(x, y)$ is the score of a pair of symbols, $x$ and $y$.
- Example:
  - $\sigma(x, y) = +2$ if $x = y$ and $x \in \{A, C, G, T\}$
  - $\sigma(x, y) = -1$ otherwise

Scoring Example

- Example:
  - $S' = A G - - C A T G$
  - $T' = A G A T C G T -$
  - Score $= 4 \times 2 + 4 \times (-1) = 4$
  - Is this the best match between the two strings with this scoring function?
    - $S = ASCATG$
    - $T = AGATCGT$

Approximate String Matching Problem

- Input: Two strings $S$ and $T$ in an alphabet $\Sigma$ and a scoring function $\sigma$.
- Output: Two strings $S'$ and $T'$ in the alphabet $\Sigma' = \Sigma \cup \{-\}$ with the properties:
  - $S = S'$ with the '-'s removed.
  - $T = T'$ with the '-'s removed.
  - $|S'| = |T'|$
  - The score $\sum_{i=1}^{m} \sigma(S'[i], T'[i])$ is maximized.

Algorithms for Approximate String Matching

- $O(mn)$ time and storage algorithm (using dynamic programming) invented by Needleman and Wunch, 1970.
- Fischer and Paterson, 1974, invented a very similar algorithm for computing the minimum edit distance between two strings.

Dynamic Programming for Approximate String Matching

- Assume $S$ has length $m$ and $T$ has length $n$.
- For all $i$ and $j$, $0 \leq i \leq m$ and $0 \leq j \leq n$, we find the maximum score for the sequences $S[1..i]$ and $T[1..j]$.
- The "dynamic program" fills in a $(m+1) \times (n+1)$ matrix $M$ in increasing order of $i$ and $j$ with these maximum values.
- Once the dynamic program has completed we can recover the optimal string $S'$ and $T'$ from the matrix $M$.

Max Score Recurrence

- Define $M[i, j] = \text{maximum score for a match between } S[1..i] \text{ and } T[1..j]$.
  - $M[i, 0] = \sum_{k=1}^{j} \sigma(S[k], -)$ match of $S[1..i]$ with empty string
  - $M[0, j] = \sum_{k=1}^{i} \sigma(-, T[k])$ match of $T[1..j]$ with empty string
  - $M[i, j] = \max$
    - $M[i-1, j-1] + \sigma(S[i], T[j])$
    - $M[i-1, j] + \sigma(S[i], -)$
    - $M[i, j-1] + \sigma(-, T[j])$
Dynamic Program Initialization

\[ S = AGCATG \]
\[ T = AGATCGT \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

scoring function
+2 for exact match
-1 otherwise

The Dynamic Programming Pattern

\[ d = a + 2 \text{ if } s = t \]
\[ = a - 1 \text{ otherwise} \]
\[ h = c - 1 \]
\[ v = b - 1 \]
\[ x = \max(d, h, v) \]

Dynamic Program Example (1)

\[ S = AGCATG \]
\[ T = AGATCGT \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Program Example (2)

\[ S = AGCATG \]
\[ T = AGATCGT \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Program Example (3)

\[ S = AGCATG \]
\[ T = AGATCGT \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Program Example (4)

\[ S = AGCATG \]
\[ T = AGATCGT \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Program Example (5)

**S** = AGCATG

**T** = AGATCGT

scoring function

+2 for exact match
-1 otherwise

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Max score for any matching

By row

for i = 1 to m do
  for j = 1 to n do
    M[i,j] := ...

By column

for j = 1 to n do
  for i = 1 to m do
    M[i,j] := ...

By diagonal

Which order is best?

How to Find the Matching

- To find S' and T' we build a matching graph.

```
x = a + 2 if s = t
  = a - 1 otherwise?

x = c - 1?

x = b - 1?
```

If the answer is yes, include the corresponding edge.

Computing the Matching Graph (1)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>-4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>-6</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Computing the Matching Graph (2)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>-4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>-6</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Computing the Matching Graph (3)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>-4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>-6</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Which order is best?
Computing the Matching Graph

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Computing the Matching Path

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm to find Matching

• Follow any path in the matching graph starting at \((m,n)\).
• The path will end up at \((0,0)\).
• Output each pair \((i,j)\) visited to make a list of pairs forming a matching path.

Computing the Matching

\[
p = \text{length of the matching path } P
\]

1. \(i := 1\);
2. For \(k \) from 1 to \(p\):
   1. If \(P[k].\text{first} = P[k-1].\text{first}\) then
      1. \(S'[k] := -\)
      2. \(i := i + 1\);
   2. Else
      1. \(S'[k] := S[i];
      2. \(i := i + 1\);
3. If \(P[k].\text{second} = P[k-1].\text{second}\) then
   1. \(T'[k] := -\)
   2. \(j := j + 1\);
   Else
   1. \(T'[k] := T[j];
   2. \(j := j + 1\);

Creating the Matching

\[
p = 1 2 3 4 5 6
\]

\[
S = A G C A T G
\]

\[
T = A G A T C G T
\]

\[
S' = A G C A T - G -
\]

\[
T' = A G - A T C G T
\]

Score = 5 \times 2 + 3 \times (-1) = 7

Example of Multiple Paths

\[
p = 1 2 3 4 5
\]

\[
S = C A T G T
\]

\[
T = C A T G T
\]

\[
S' = C A T G T
\]

\[
T' = - C A T G T
\]

Score = 3 \times 2 + 4 \times (-1) = 2
Exercise

- Find an optimal approximate matching for:
  - A G T T C
  - A C T A T C

Approximate String Searching

- Input: Query string Q and target string T in an alphabet Σ and a scoring function σ, and a minimum score r.
- Output: The set of k such that for some i ≤ k \( \sigma(Q[T[i..k]]) > r \). That is, an approximate match of some substring of T that ends at index k has a score of at least r.
  - \( \sigma(X,Y) \) is the maximum score for all matchings between X and Y.

Search Algorithm

- We change the previous dynamic program slightly.
  - \( M[i,j] = \sum_{k=0}^{\min(i,j)} \sigma(Q[k..i],T[j-k]) \)
  - \( M[i,j] = 0 \) if we don’t care where the match begins in T
  - \( M[i,j] = \max \{ M[i-1,j-1] + \sigma(Q[i..j],T[j..i]), M[i-1,j] + \sigma(Q[i..j],T[j..i]), M[i,j-1] + \sigma(Q[i..j],T[j..i]) \} \)

Choose all k such that \( M[m,k] \geq r \) where m is the length of Q.

Example of Approximate Matching

Q = AGTA
T = AGATCGTAG
r = 5
scoring function:
+2 for exact match
-1 otherwise

Output is 3, 8, 9, 10

Recovering the Matchings

Notes on Approximate Matching

- Time complexity O(mn)
- Storage complexity O(mn)
  - Storage in the dynamic program can be reduced to O(m+n) by just keeping the frontier.
  - Recovering the matching can be done in time O(m+n) cleverly.
FASTA and BLAST

- Two of best known approximate search algorithms for DNA database searching
- Both use the idea of exclusion search
  - Parameter $k$ for number of possible errors
  - Exact search on $k+1$ substrings. At least one must succeed
  $k = 4$

1. Find all the exact matches for at least one of the strings
2. For each such match do an approximate matching

Example

- $k = 2$

<table>
<thead>
<tr>
<th>AGTTATGCC</th>
<th>AGT TAT GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTAGCAAGATGCC</td>
<td>AGTTAGCTAGAGTTAGTG</td>
</tr>
</tbody>
</table>

Dynamic Programming

- A strategy for designing algorithms.
- A technique, not an algorithm.
- The word “programming” is historical and predates computer programming.
- Ideal when the problem breaks down into recurring small sub-problems.

Longest Common Subsequence

- Longest common subsequence (LCS) problem:
  - Given two sequences $x[1..m]$ and $y[1..n]$, find the longest subsequence which occurs in both (not necessarily contiguous).
  - Example: $x = A B C B D A B$, $y = B D C A B A$
  - $B C$ and $A A$ are both subsequences of both
  - What is the LCS? $BCAB$, $BCBA$
  - Brute-force algorithm: For every subsequence of $x$, check if it’s a subsequence of $y$
    - How many subsequences of $x$ are there?
    - What will be the running time of the brute-force alg?

LCS Algorithm

- Brute-force algorithm: $2^m$ subsequences of $x$ each takes $O(n)$ to search in $y$: $O(n \cdot 2^m)$
- We can do better: for now, let’s only worry about the problem of finding the length of the LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS.
- Notice LCS problem has optimal substructure
  - Subproblems: LCS of pairs of prefixes of $x$ and $y$

Finding LCS Length

- Define $c[i,j]$ to be the length of the LCS of $X_i = x[1..i]$ and $Y_j = y[1..j]$
  - What is the length of LCS of $x$ and $y$?
  - $c[m,n]$
- Theorem:
  - $c[i,j] = \begin{cases} 
  c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\
  \max(c[i,j-1],c[i-1,j]) & \text{otherwise}
  \end{cases}$
LCS Recurrence

\[ c[i,j] = \begin{cases} 
  c[i-1,j-1] + 1 & \text{if } x[i] = y[j], \\
  \max(c[i-1,j], c[i,j-1]) & \text{otherwise}
\end{cases} \]

Proof: When calculating \( c[i,j] \), there are two cases to consider:

- **First case**: \( x[i] = y[j] \): one more symbol in strings \( X \) and \( Y \) matches, so the length of LCS \( X \) \( i \) and \( Y \) \( j \) equals to the length of LCS of smaller strings \( X_{i-1} \) and \( Y_{j-1} \), plus 1.

- **Second case**: \( x[i] \neq y[j] \)
  - As symbols don’t match, our solution is not improved, and the length of LCS \( (X_i, Y_j) \) is the maximum of LCS \( (X_{i}, Y_{j-1}) \) and LCS \( (X_{i-1}, Y_{j}) \).

Why not just take the length of LCS \( (X_{i-1}, Y_{j-1}) \) ?

Exercise: Write the Program

1. \( m = \text{length}(X) \) // # of symbols in \( X \)
2. \( n = \text{length}(Y) \) // # of symbols in \( Y \)
3. for \( i = 1 \) to \( m \) \( c[i,0] = 0 \) // special case: \( X_0 \)
4. for \( j = 1 \) to \( n \) \( c[0,j] = 0 \) // special case: \( Y_0 \)

Finish it

Exercise: Create a Dynamic Program

- Design a dynamic program for knapsack problem.
- Input: \((s_1,c_1), (s_2,c_2), \ldots, (s_n,c_n)\), \( S \)
- Output: find a subset \( X \) of \( \{1,2,\ldots,n\} \) such that \( \sum_{i \in X} s_i \leq S \) and \( \sum_{i \in X} c_i \) is maximized
- Hint: For \( i \leq n \) and \( k \leq S \) recursively define
  \[ c(i,k) = \max_{i \in X} \{ c(i,k) : X \subseteq \{1,2,\ldots,i\} \text{ and } \sum_{j \in X} s_j = k \} \]

DNA Sequence Reconstruction

- DNA can only be sequenced in relatively small pieces, up to about 1,000 nucleotides.
- By chemistry a much longer DNA sequence can be broken up into overlapping sequences called clones. Clones are 10’s of thousands of nucleotides long.
Tagging the Clones

- By chemistry the clones can be tagged by identifying a region of the DNA uniquely.
- Each clone is then tagged correspondingly.

Problem to Solve

- Given a set of tagged clones, find a consistent ordering of the tags that determines a possible ordering of the DNA molecule.

<table>
<thead>
<tr>
<th>Clone</th>
<th>Tag</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[E, G]</td>
<td>E G F H A I B D C</td>
</tr>
<tr>
<td>2</td>
<td>[F, G, H]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[A, I]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[C, D]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[E, G]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[A, H, I]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[B, D]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[F, H]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[A, B, D, I]</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[C, D]</td>
<td></td>
</tr>
</tbody>
</table>

Contiguous Ordering Solutions

Contiguous ordering problem

U = \{A, B, C, D, E, F, G, H, I\}

S = \{(E, G),
(F, G, H),
(A, I),
(C, D),
(E, G),
(A, H, I),
(B, D),
(F, H),
(A, B, D, I),
(C, D)\}

Solution

E G F H A I B D C

Alternate Solutions

E G F H I A B D C

C D B I A H F G E

C D B A I H F G E

Linear Time Algorithm

- Booth and Lueker, 1976, designed an algorithm that runs in time \(O(n+m+s)\).
  - \(n\) is the size of the universe, \(m\) is the number of sets, and \(s\) is the sum of the sizes of the sets.
- It requires a novel data structure called the PQ tree that represents a set of orderings.
- PQ trees can also be used to test whether an undirected graph is planar.

PQ Trees

- PQ trees are built from three types of nodes

P node

Children can be reordered.

Q node

Children can be reversed.

Leaf

Each leaf has a unique label.

Example PQ-Tree

The frontier of T defines the ordering \(F(T) = FCABDE\), just read the leaves left to right.

\(T'\) is equivalent to \(T\) if \(T\) can be transformed into \(T'\) by reordering the children of P nodes and reversing the children of Q nodes.
Equivalent PQ Trees

Orderings Defined by a PQ Tree

• Given a PQ tree T the orderings defined by T is
  - \( PQ(T) = \{ F(T') : T' \text{ is equivalent to } T \} \)

There are 6 \( \times 2 \times 2 = 24 \) distinct orderings in \( PQ(T) \).

Generally, if a PQ tree T has q Q node and p P nodes with number of children \( c_1, c_2, \ldots, c_p \) then the number of orderings in \( PQ(T) \) is

\[ 2^{q-1} c_1! c_2! \cdots c_p! \cdot n! = 1 \times 2 \times \ldots \times n \]

Example Solution

PQ Tree Solution for the Contiguous Ordering Problem

• Input: A universe U and a set \( S = \{ S_1, S_2, \ldots, S_m \} \) of subsets of U.
• Output: A PQ tree T with leaves U with the property that \( PQ(T) \) is the set of all orderings of U where each set in S is contiguous in the ordering.

High Level PQ tree Algorithm

• Input is \( U = \{ A_1, A_2, \ldots, A_n \} \), and subsets \( S_1, S_2, \ldots, S_m \) of U.
• Initialization:
  - \( T \) = P node with children \( A_1, A_2, \ldots, A_n \)
• Calculate m restrictions:
  - for \( j = 1 \) to \( m \) do
  - \( T := \text{Restrict}(T,S) \)
• At the end of iteration \( k \):
  - \( PQ(T) \) = the set of ordering of U where each set \( S_1, S_2, \ldots, S_k \) are contiguous.
Marking Nodes

- Given a set $S$ and PQ tree $T$ we can mark nodes either full or partial.
  - A leaf is full if it is a member of $S$.
  - A node is full if all its children are full.
  - A node is partial if either it has both full and non-full children or it has a partial child.
  - A node is doubly partial if it has two partial children.

Marks of Nodes

Mark the leaves in $S$ full. Bottom up mark the nodes full or partial. The members of $S$ will become contiguous.

Structure of the Marked PQ Tree

Restrict($T, S$)

- Mark the full and partial nodes from the bottom up.
  - In the process the marked leaves become contiguous.
- Locate the key node.
  - Deepest node with the property that all the full leaves are descendants of the node.
- Restrict the key node.
  - In the process of restricting the key node we will have to recursively direct partial nodes.
  - Directing a node returns a sequence of nodes.

Restricting a P Node with Partial Children

Restricting a P node with no Partial Children
Restricting a Q node

Directing a P Node

Directing a Q Node

Example (1)

Example (2)

Example (3)
Example (4)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_2 = \{C,D,F,G,I,J\} \]

Example (5)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_2 = \{C,D,F,G,I,J\} \]

Example (6)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_2 = \{C,D,F,G,I,J\} \]

Example (7)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_3 = \{A,B,E,G\} \]

Example (8)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_3 = \{A,B,E,G\} \]

Example (9)

\[ U = \{A,B,C,D,E,F,G,H,I,J\} \]
\[ S_3 = \{A,B,E,G\} \]
Example (10)

\[ U = \{A, B, C, D, E, F, G, H, I, J\} \]
\[ S_1 = \{A, B, E, G\} \]

Example (11)

\[ U = \{A, B, C, D, E, F, G, H, I, J\} \]
\[ S_3 = \{A, B, E, G\} \]

Example (12)

\[ U = \{A, B, C, D, E, F, G, H, I, J\} \]
\[ S_1 = \{A, C, E, G, I\} \]
\[ S_2 = \{C, D, F, G, I, J\} \]
\[ S_3 = \{A, B, E, G\} \]

Exercise

- Restrict with to make \(\{A, B, D, E, G\}\) contiguous

Linear Number of Nodes Processed

- Let \(n\) be the size of the universe, \(m\) the number of sets, and \(s\) the sum of the sizes of the sets.
  - Number of full nodes processed \(\leq 2s\).
  - Number of key nodes processed = \(m\).
  - Number of partial nodes with partial children processed below the key node \(\leq m + n\).
  - Number of partial nodes with no partial children \(\leq 2m\).
  - Number of partial nodes processed above the key node \(\leq m + n\).

Number of Processed Nodes Amortized

\[ n \text{ size of universe} \]
\[ m \text{ number of sets} \]
\[ s \text{ sum of size of sets} \]
Partials with Partial Children Below the Key Node

- Amortized complexity argument.
- Consider the quantities:
  - \( q \) = number of Q nodes,
  - \( c_p \) = number of children of P nodes.
- We examine the quantity \( x = q + c_p \)
- \( x \) is initially \( n \) and never negative.
- Each restrict of a key node increases \( x \) by at most 1.
- Each direct of a partial node with a partial child decreases \( x \) by at least 1.
- Since there are \( m \) restricts of a key node then there are most \( n + m \) directs of partials with partial children.

Restricting a P Node with Partial Children

Restricting a P node with no Partial Children

Restricting a Q node

Directing a P Node

Directing a Q Node
PQ Tree Notes

- In algorithmic design only a linear number of nodes are ever processed.
- Designing the data structures to make the linear time processing a reality is very tricky.
- PQ trees solve the idealized DNA ordering problem.
- In reality, because of errors, the DNA ordering problem is NP-hard and other techniques are used.

Example of Data Structure Trick

- Linking the children of a Q node

- Linking of siblings can be in any order.
- Middle children don’t know parents.
- End children know parents.