Today’s Outline

- Greedy
- Divide & Conquer
- Dynamic Programming

Greedy Algorithms

Repeat until problem is solved:
  - Consider possible next steps
  - Choose best-looking alternative and commit to it

Greedy algorithms are normally fast and simple.

Sometimes appropriate as a heuristic solution or to approximate the optimal solution.

Hill-Climbing

Scheduling Problem

- Given:
  - a group of tasks \{T_1, \ldots, T_n\}
  - each with a duration \{d_1, \ldots, d_n\}
  - a single processor without interrupts
- Select an order for the tasks that minimizes average completion time

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>T_5</th>
<th>T_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>13</td>
<td>5</td>
<td>3</td>
<td>14</td>
<td>10</td>
</tr>
</tbody>
</table>

Average time to completion:
Greedy Solution

Divide & Conquer
- Divide problem into multiple smaller parts
- Solve smaller parts
  - Solve base cases directly
  - Otherwise, solve subproblems recursively
- Merge solutions together (Conquer!)

Often leads to elegant and simple recursive implementations.

Divide & Conquer in Action

Memoizing/
Dynamic Programming
- Define problem in terms of smaller subproblems
- Solve and record solution for base cases
- Build solutions for subproblems up from solutions to smaller subproblems

Can improve runtime of divide & conquer algorithms that have shared subproblems with optimal substructure.

Usually involves a table of subproblem solutions.

Dynamic Programming in Action
- Sequence Alignment
- Fibonacci numbers
- All pairs shortest path
- Optimal Binary Search Tree
- Matrix multiplication

Fibonacci Numbers

\[
F(n) = F(n - 1) + F(n - 2) \\
F(0) = 1 \quad F(1) = 1
\]
Fibonacci Numbers

\[ F(n) = F(n - 1) + F(n - 2) \]
\[ F(0) = 1 \quad F(1) = 1 \]

Divide & Conquer

```c
int fib(int n) {
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Memoized

```c
int fib(int n) {
    // Create an array however your favorite language allows
    // int fibs[n];
    if (n <= 1)
        return 1;
    if (fibs[n] == 0)
        fibs[n] = fib(n - 1) + fib(n - 2);
    return fibs[n];
}
```

Sequence Alignment

Biology 101:

- DNA Sequence
  - String using letters (nucleotides): A,C,G,T
  - For example: ACGGGCATTTACGTGA
  - DNA can mutate!
    - Change a letter: ACGGCAT → ACGTCAT
    - Insert a letter: ACGGGCAT → ACGGGAAT
    - Delete a letter: ACGGGCAT → ACGGGA

- A few mutations makes sequences “different”, but “similar”
- Similar sequences often have similar functions

What is Sequence Alignment?

- Use underscores (\_) or wildcards to match up 2 sequences
- The “best alignment” of 2 sequences is an alignment which minimizes the number of “underscores”
- For example: ACGGTTT and TCCCTTT
  - Best alignment: A\_CCC\_TTT
  - (3 underscores) \_TCCC\_TTT

Solutions

- Naive solution
  - Try all possible alignments
  - Running time: exponential
- Dynamic Programming Solution
  - Create a table
  - Table(x,y): best alignment for first x letters of string 1, and first y letters of string 2
  - Running time: polynomial
Example Alignment

Match ACCGTTAG with ACTGTTAA
(1) match ‘G’ with ‘_’: 1 + align(ACCGTTA,ACTGTTAA)
(2) match ‘A’ with ‘_’: 1 + align(ACCGTGA,ACTGTTA)

• Suppose we have already determined the best alignment for:
  – First x letters of string1 with first y-1 letters of string2
  – First x-1 letters of string1 with first y-1 letters of string2
  – First x-1 letters of string1 with first y letters of string2

If (string1[x] == string2[y]) then TABLE[x,y] = TABLE[x-1,y-1]
Else TABLE[x,y] = min(1+TABLE[x,y-1], 1+TABLE[x-1,y])

-----

**Example GGCAT and TGCAA**

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**Example Pseudocode (bottom-up)**

```c
int align(String X, String Y, TABLE[1..x,1..y]) {
  int i,j;
  // Initialize top row and leftmost column
  for (i=1; i<=x, ++i) TABLE[i,1] = i;
  for (j=1; j<=y; ++j) TABLE[1,j] = j;
  for (i=2; i<=x; ++i) {
    for (j=2; j<=y; ++j) {
      if (X[i] == Y[j])
        TABLE[i,j] = TABLE[i-1,j-1];
      else
        TABLE[i,j] = min(TABLE[i-1,j], TABLE[i,j-1]) + 1;
    }
  }
  return TABLE[x,y];
}
```

**Example Pseudocode (top-down)**

```c
int align(String X, String Y, TABLE[1..x,1..y]) {
  Compute TABLE[x-1,y-1] if necessary
  Compute TABLE[x-1,y] if necessary
  Compute TABLE[x,y-1] if necessary
  if (X[x] == Y[y])
    TABLE[x,y] = TABLE[x-1,y-1];
  else
    TABLE[x,y] = min(TABLE[x-1,y], TABLE[x,y-1]) + 1;
  return TABLE[x,y];
}
```

**Dynamic Programming Wrap-Up**

• Re-use expensive computations
• Store optimal solutions to sub-problems in table
• Use optimal solutions to sub-problems to determine optimal solution to slightly larger problem