Overview

⇒ Chapter 4: Divide and Conquer
• Chapter 15: Dynamic Programming
• String Alignment Problems
• Framing the Problem Mathematically
• Algorithm for String Alignment

Divide and Conquer

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, just solve the subproblem in a straightforward manner

**Combine** the solutions to the subproblems into the solution for the original problem

• Two cases:
  - Recursive case
  - Base case
Recursion versus Iteration

• Which of these two methods can be done using Iteration?

```python
def InOrderWalk(self):
    if self.left is not None:
        self.left.InOrderWalk()
    print(self.key)
    if self.right is not None:
        self.right.InOrderWalk()

def Search(self, k):
    if k == self.key:
        return self
    if k < self.key and self.left is not None:
        return self.left.Search(k)
    if k > self.key and self.right is not None:
        return self.right.Search(k)
    return None
```

Simple Example of Divide and Conquer

• Membership testing in a sorted array: $x \in L$?
  + See if $x$ is equal, less, or more than the element halfway in L
  + If less, test $x$ with first half of L
  + We either find the element, or get to an empty array: conquer
  + Combine by passing answer back up the recursion
  + Takes $O(\log(n))$

• Printing nodes in a tree
  - get str for left tree, for node, and for right tree
  - at a leaf, return string of node: conquered
  - combine: create string for entire node
Recursion

• If problem is divided into one smaller problem, can use loop
  - Membership testing (pick one half of the list)
• If problem is divided into several problems, use recursion
  - Printing tree, need to print both sides

Recurrences

• recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
  - natural way to characterize running time of divide and conquer algorithm
• Chapter 12: Printing tree
  - Had a recursive algorithm
  - Running time expressed as a recurrence

\[
T(n) \leq \begin{cases} 
    c & n = 1 \\
    T(k) + T(n - k - 1) + d & n > 1 
\end{cases}
\]

- We used ‘substitution method’ to solve this
  + Guess the solution
  + Use induction to prove solution is correct
- Textbook gives two other methods for solving them
Technicalities in Recurrences

• Emphasis on recurrences for large values of n
  - Ignore differences due to odd or even input size
  - Ignore differences for boundary conditions (on small n)
    + Will only effect running time by a constant, which is irrelevant for $O$ and $\Theta$

Example: Maximum-Subarray Problem

• Find the biggest upshift in prices
  - Perhaps to see how well you did in a trading a stock versus the optimum
  - Can just buy/sell at end of day, over fixed period of time (say 100 days)
  - Can hold onto stock for any number of days
  - Must buy stock before selling it (no short sales)
Brute force

- Look at every pair of dates to find best one
  - array price has the end-of-day prices
  - Running time $\Theta(n^2)$

```python
best = -1
for j in range(99):
    for i in range(j+1,100):
        gain = price[i] - price[j]
        if gain > best:
            best = gain
```

Naive Solution

- Find global min and max, but global max might be before global min
- Other solutions?
Key Insight

- Rather than focus on the daily price
  - Focus on how much price has changed since prior day
  - Let \( \delta(i) = \text{price}(i) - \text{price}(i-1) \)
  - Find a nonempty continuous subarray whose values have the largest sum
    + Referred to as ‘Maximum subarray’

- For dividing the problem
  - Split delta array in half
  - Max subarray is either entirely on one side or spans halfway point

<table>
<thead>
<tr>
<th>Day</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>100</td>
<td>113</td>
<td>110</td>
<td>85</td>
<td>105</td>
<td>102</td>
<td>86</td>
<td>81</td>
<td>63</td>
<td>81</td>
<td>101</td>
<td>94</td>
<td>106</td>
<td>101</td>
<td>79</td>
<td>94</td>
<td>90</td>
</tr>
<tr>
<td>Change</td>
<td>13</td>
<td>-3</td>
<td>-25</td>
<td>20</td>
<td>-3</td>
<td>-16</td>
<td>-23</td>
<td>18</td>
<td>20</td>
<td>-7</td>
<td>12</td>
<td>-5</td>
<td>-22</td>
<td>15</td>
<td>-4</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Spanning Midpoint

- Maximum subarray that spans both sides (includes \( \delta(mid) \) and \( \delta(mid+1) \))
  - First, go backward from \( mid \) and find max sum
  - Then, go forward from \( mid+1 \) and find max sum
  - Can be done in \( \Theta(n) \)
  - Not smaller instance of original problem, as it has an added restriction

```plaintext
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
1  left-sum = -\infty
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum = -\infty
9  sum = 0
10     for j = mid + 1 to high
11         sum = sum + A[j]
12         if sum > right-sum
13             right-sum = sum
14             max-right = j
15     return (max-left, max-right, left-sum + right-sum)
```
Find Max Subarray

FIND-MAXIMUM-SUBARRAY(A, low, high)
1 if high == low
2 return (low, high, A[low]) // base case: only one element
3 else mid = ⌊(low + high)/2⌋
4 (left-low, left-high, left-sum) =
5 FIND-MAXIMUM-SUBARRAY(A, low, mid)
6 (right-low, right-high, right-sum) =
7 FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
8 (cross-low, cross-high, cross-sum) =
9 FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
10 if left-sum ≥ right-sum and left-sum ≥ cross-sum
11 return (left-low, left-high, left-sum)
12 elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
13 return (right-low, right-high, right-sum)
14 else return (cross-low, cross-high, cross-sum)

• Divide problem, conquer each part, combine

Running Time

• Let T(n) be running time of algorithm on input size of n
  - For simplicity, assume n is a power of 2

FIND-MAXIMUM-SUBARRAY(A, low, high)
1 if high == low
2 return (low, high, A[low]) // base case: only one element
3 else mid = ⌊(low + high)/2⌋
4 (left-low, left-high, left-sum) =
5 FIND-MAXIMUM-SUBARRAY(A, low, mid)
6 (right-low, right-high, right-sum) =
7 FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
8 (cross-low, cross-high, cross-sum) =
9 FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
10 if left-sum ≥ right-sum and left-sum ≥ cross-sum
11 return (left-low, left-high, left-sum)
12 elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
13 return (right-low, right-high, right-sum)
14 else return (cross-low, cross-high, cross-sum)
Running Time

Let $T(n)$ be the running time of the algorithm on input size of $n$
- Base case (just one element)
  $T(1) = \Theta(1)$ (Lines 1 to 3 take constant time)
- Recursive step: ($n > 1$)
  + Lines 1-3 take $\Theta(1)$
  + Line 4 takes $T(n/2)$
  + Line 5 takes $T(n/2)$
  + Line 6 takes $\Theta(n)$
  + Line 7-11 take constant time
  $T(n) = 2T(n/2) + \Theta(n) + \Theta(1)$
  $= 2T(n/2) + \Theta(n)$

Solving the Recurrence

- Used substitution method previously
  - Guess the form, and prove by induction
    - Works for $O$ (upper bound), but not for $\Theta$
- Master theorem: Another way of solving recurrences
  - Cookbook method for recurrences of form $T(n) = aT(n/b) + f(n)$
    where $a \geq 1$, $b > 1$ and $f(n)$ is asymptotically positive function
  - Captures any algorithm that divides problem into $a$ smaller ones of size $n/b$, and solves them recursively
  - Technical point: More correct to use $T(\lfloor n/b \rfloor)$, but will not affect the asymptotic behavior
Master Theorem

Theorem 4.1 (Master Theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. □

So for $T(n) = 2T(n/2) + \Theta(n)$

$+ a = 2, b = 2, f(n) = \Theta(n) = \Theta(n^{\log_2 2})$

So use second case: $T(n) = \Theta(n \log n)$

Essential Point of Divide and Conquer

- **Optimal Substructure:**
  - solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems

- **Non-overlapping subproblems:**
  - There is an obvious way to break the problem into subproblems (do not have to search over different subproblems)
  - The division into subproblems will give you the optimal solution
Problem not solvable with Divide and Conquer

- Finding the best route from A to B
- Divide problem,
  - Unclear what C should be used to split problem into A to C, and C to B
  - Many different ways to divide into subproblems
  - Choice will affect whether you find the optimal solution

Overview

- Chapter 4: Divide and Conquer
  ⇒ Chapter 15: Dynamic Programming
- String Alignment Problems
- Framing the Problem Mathematically
- Algorithm for String Alignment
Dynamic Programming

• Similar to divide-and-conquer

• Optimal Substructure:
  - solution to a given optimization problem can be obtained by the
  combination of optimal solutions to its subproblems

• Overlapping subproblems:
  - There is not a single obvious way to break the problem into subproblems

Example: Routing

• Has the optimal substructure problem
  - If C is on the optimal path from A to B (so A to B can be optimally
    divided into subproblems A to C and C to B)
    + Optimal solution to A to B is:
      optimal solution from A to C followed by optimal solution from C to B

• Overlapping subproblems:
  - Route from A to B can be divided by going through C, or D, or E
    + Don’t know which division is best
Key Insight into Efficient Solution

- You have overlapping subproblems
e.g., A to C and C to B; versus A to D and D to B
  - there might be subproblems in common between these subproblems
e.g., A to X might be used in A to C and A to D
e.g., X to Y might be used in A to C and A to D and C to B and D to B
- Save solutions to these subproblems and do not recompute!
  - Memoize the results (or store in a table)
  - 'Programming' in dynamic programming actually refers to storing intermediate results in a table

Brute Force

- Try each possible partition at each level
- Could lead to exponential time algorithm
Top Down or Bottom Up

- In a top down implementation
  - Before doing a subproblem, check table to see if already done it
- In bottom up
  - Start with small problems, and build up to larger ones
  - More straightforward

Overview

- Chapter 4: Divide and Conquer
- Chapter 15: Dynamic Programming
  ⇒ String Alignment Problems
- Framing the Problem Mathematically
- Algorithm for String Alignment
Matching DNA Sequences

- DNA of one organism might be:
  ACCGGTCGAGTCGCGGAAGCCGGCCGAA
- Of another:
  GTCGTTCCGAATGCCTGTGCTCTGTAA
- How similar are the strands?
  - What sequence of bases are common in both strands?
  - Find an alignment with the maximum number of matches

```
ACCGGTCGAGTCGCGGAAGCCGGCCGAA
     GTCGTTCCGAATGCCTGTGCTCTGTAA
```

```
GTCGTTCCGAATGCCTGTGCTCTGTAA
     GTCGTTCCGAATGCCTGTGCTCTGTAA
```

Automatic Speech Recognition

- Also uses string alignment in scoring output with respect to a reference transcription
Sentence Recall Test

- Examiner says a sentence to the participant:
  *the little boy went to the store*

- Participant tries to repeat it verbatim:
  *the boy went to the store*
  *the boy went to a store*
  *a little lad goed to a store*

- Align the two sentences to find what the mistakes are
  - Interested in *insertions, deletions, and substitutions*
  - *the* → *a*
  - *boy* → *lad*
  - *little* → *ε*
  - *went* → *goed*
How do we frame the problem?

- For any alignment between two sequences
  - Score it based on the number of insertions, deletions, and substitutions
    - Penalty of 1 for insert/delete/sub and no penalty for match
- Find the alignment with the smallest penalty
- Let’s be more precise about what an alignment is
  - Let $a, b \in \Sigma^*$ and $a = a_1...a_n$ and $b = b_1...b_m$
  - Which indexes of $a$ and $b$ are part of a match or substitution
    - $A \subset \{1, 2, 3, ..., n\}$ and $B \subset \{1, 2, 3, ..., m\}$
    - $2^n$ possible values for $A$ and $2^m$ for $B$
    - $2^{n+m}$ possible alignments

Example

- Alignment was implicit in the formatting
  - Blank spots have an epsilon in there alignment
- One alignment (8 deletes, 6 inserts, 1 subs)
  \[
  \begin{align*}
  ACCGGTCTCGAGTCGCGGAA & \quad GCCG \quad GC \quad GC \quad AA \\
  GTCG & \quad TT CG \quad GAATGCCGTTGCTCTGTAAAA
  \end{align*}
  \]
- A worse one (3 deletes, 2 inserts, 13 subs)
  \[
  \begin{align*}
  ACCGGT & \quad CGAGTGC CGCGGAA AGCCG \quad GCCGAA \\
  GTCG & \quad TT CG \quad GAATGCCGTTGCTCTGT AAA
  \end{align*}
  \]
- Pick the best alignment (one with lowest score)
A Better Way to View an Alignment

• a (prompt) has \( n \) words/characters, \( b \) (response) has \( m \) words

• View prompt as columns and response as rows (\( n \times m \) array)

• An alignment is a path through the cells where you can go:
  - Left (consume a word of the prompt but no word of response)
    + deletion
  - Down (consume a word of the response but no word of prompt)
    + insertion
  - Diagonal left-down (consume a word of response and prompt)
    + If words are the same: match, otherwise substitution

Example

<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>little</th>
<th>boy</th>
<th>went</th>
<th>to</th>
<th>the</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>little</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lad</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>goed</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>yeah</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Overview

- Chapter 4: Divide and Conquer
- Chapter 15: Dynamic Programming
- String Alignment Problems
- Framing the Problem Mathematically

⇒ Algorithm for String Alignment

Brute Force

- How many paths through array are there?
- You can just go right, down or diagonally right down
  - Path is at most \( n + m \) in length (no matches or substitutions)
  - At each point in path, at most 3 options (left, down, diagonal)
  - At most \( 3^{n+m} \) paths
  - For each path, compute score
    + For each left or right move, add 1
    + For each diagonal move: determine if it is a match (0) or substitution (1)
- Can do some optimization
  - Keep track of best path so far, and prune paths if they are worse
  - Can do this as a depth-first search
    + Gets rid of some redundancy (of the first parts of the path)
    + But bottom right hand corner will be redone many times
### Dynamic Programming

**Initialization**

- Each cell will have optimal score to get from (0,0).
- Fill in 0th row. Each move left implies we are deleting.
- Fill in 0th column. Each move down implies we are inserting.

<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>little</th>
<th>boy</th>
<th>went</th>
<th>to</th>
<th>the</th>
<th>store</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>little</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>lad</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yeah</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Filling in cells

- Fill in cell \((i, j)\) if cells above \((i, j-1)\), left \((i-1, j)\), diag \((i-1, i-j)\) filled in
- 3 possible ways to get to cell
  - From above: take score \((i, j-1)\) and subtract 1 (insert)
  - From left: take score of \((i-1, j)\) and subtract 1 (delete)
  - From left-above: take score of \((i-1, j-1)\) if match, else subtract 1 (substitution)
  - Take lowest

|      | the | little | boy | ...
|------|-----|--------|-----|------
| little | 1   | 1      | 3   | ...
| lad   | 2   |        |     |      
| goed  | 3   |        |     |      
| ...   | ... |        |     |      |

Efficiency?

- Initialize row 0 \(\Theta(n)\)
- Initialize col 0 \(\Theta(m)\)
- For \(m \times n\) cells
  - Determine value of each cell \(\Theta(nm)\)
- Assume \(n\) and \(m\) are similar in size
  - \(\Theta(n^2)\) rather than \(\Theta(3^n)\)
How is it a Dynamic Programming Solution

Any optimal path going from \( A_{0,0} \) to \( A_{i,j} \) can be divided into the first part and the last step. The last step must be either a down, left, or diagonal, so from \( A_{i,j-1}, A_{i-1,j} \) or \( A_{i-1,j-1} \). This is what the algorithm computes.