Divide and Conquer

- Base case
- Recursive case

TWO cases:

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

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

 Overall:

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

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Chapter 12: Printing tree

- Had a recursive algorithm
- Running time expressed as a recurrence

\[ T(n) \leq \begin{cases} \begin{align*} c \cdot n & \quad \text{if } n < u \\ p + (1 - \frac{\gamma}{u})L + \left(\frac{\beta}{u}\right)L & \quad \text{if } n \geq u \end{align*} \end{cases} \]

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

Simple Example of Divide and Conquer

- Membership testing in a sorted array: \( x \in L \)?
  - See if \( x \) is equal, less, or more than element halfway in \( L \)
  - If \( x \) is 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
  - \( T \text{ees } O(\log n) \)

Printing nodes in a tree

- Get str for left tree, for node, and for right tree
- At a leaf, return string of node: conquered
- If \( L \) is empty, return empty string
- For leaf node: for node, and for right tree
- Combine: create string for entire node

Membership testing in a sorted array: \( x \in L \)?
Example: Maximum-Subarray Problem

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

Technicalities in Recurrences

- Emphasis on recurrences for large values of \( n \)
- Ignore differences due to odd or even input size
- Ignore differences due to odd or even input size
- Emphasis on recurrences for large values of \( n \)
- Will only affect running time by a constant, which is irrelevant for \( O(\cdot) \) and \( \Theta(\cdot) \)
Brute Force

Naive Solution

Other Solutions:
- Find global min and max but global max might be before global min

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

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

- Find a nonempty continuous subarray whose values have the largest sum
- Focusing on how much price has changed since prior day
- Rather than focus on the daily price

Key Insight

- Rather than focus on the daily price
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**: ($n > 1$)

  - For simplicity, assume $n$ is a power of 2
    - Lines 1-3 take constant time
    - Line 4 takes $T(n/2)$
    - Line 5 takes $T(n/2)$
    - Line 6 takes $\Theta(n)$
    - Line 7-11 take constant time

  \[
  T(n) = 2T(n/2) + \Theta(n) + \Theta(1)
  \]

### Find Max Subarray

```plaintext
FindMaxSubarray(A, low, high)

if high == low
  return (low, high);

mid = (low + high)/2;

FindMaxSubarray(A, low, mid);
FindMaxSubarray(A, mid + 1, high);

cross_low = cross_sum[low, mid];
FindMaxSubarray(A, low, mid);
FindMaxSubarray(A, mid + 1, high);

cross_high = cross_sum[mid + 1, high];
FindMaxSubarray(A, low, mid);
FindMaxSubarray(A, mid + 1, high);

left_sum = A[low..mid];
right_sum = A[mid+1..high];

if left_sum > right_sum
  return (low, mid);

if right_sum > left_sum
  return (mid + 1, high);

return (mid, mid);
```
Master Theorem

Theorem 4.1 (Master theorem) Let $a \geq 1$, $b > 1$, and $f(n)$ 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 $b^{\lfloor \log_b n \rfloor}$ or $n/b$. Then the following asymptotic bounds hold:

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

Solving the Recurrence

- Used substitution previously
- Works for O (upper bound), but not for \Theta
- Guess the form and prove by induction
- Used substitution method previously

Technical point: More correct to use $T(n/b)$, but will affect the asymptotic behavior.

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 $O(n^{\log_b a} \epsilon)$.
Essential Point

Optimal Substructure

- The division into subproblems will give you the optimal solution.
- There is an obvious way to break the problem into subproblems.

Non-overlapping subproblems:
- Combination of optimal solutions to its subproblems.
- Solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems.

Recursion

- If problem is divided into one smaller problem, can use recursion.
- Membership testing (pick one half of the list).
- If problem is divided into several problems, use recursion.
- Use recursion when need to print both sides.
Overview

Divide and Conquer

• Algorithm for String Alignment

• Framing the Problem Mathematically

• String Alignment Problems

Chapter 15: Dynamic Programming

• Chapter 4: Divide and Conquer

Problem not solvable with Divide and Conquer

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

Finding the best route from A to B
Example: Routing

- Has the optimal substructure problem
  - Route from A to B can be divided by going through C, D, or E
  + Optimal solution from A to C followed by optimal solution from C to B
  + Optimal solution from A to C and C to B
  + Route from A to B can be divided into subproblems A to C and C to B
  - C is on the optimal path from A to B (so A to B can be optimally divided into subproblems)

Dynamic Programming

- Similar to divide-and-conquer
- Overlapping subproblems:
  - There is no single way to break the problem into subproblems
  - Combination of optimal solutions to subproblems
- Optimal Substructure: solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems

Dynamic Programming: Similar to divide-and-conquer

- Overlapping subproblems:
  - There is no single way to break the problem into subproblems
  - Combination of optimal solutions to subproblems
- Optimal Substructure: solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems
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 and A to B.
- Programming in dynamic programming actually refers to storing intermediate results in a table.
- Memoize the results (or store in a table). "Programming" in dynamic programming actually refers to storing

Brute Force

- Could lead to exponential time algorithm.
- Try each possible partition at each level.
Algorithm for String Alignment

Framing the Problem Mathematically

String Alignment Problems

Chapter 15: Dynamic Programming

Chapter 4: Divide and Conquer

Overview

Top Down or Bottom Up

- In a top down implementation, before doing a subproblem, check if the table is already done.
- In a bottom up implementation, start with small problems, and build up to larger ones.
Automatic Speech Recognition

Reference Transcription

- Also uses string alignment in scoring output with respect to a

<table>
<thead>
<tr>
<th>Automatic Speech Recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Matching DNA Sequences

- DNA of one organism might be:

<table>
<thead>
<tr>
<th>ACCGGTCGAGTGCGCGGAAGCCGGCCGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTCGTTCGGAA TGCCGTTGCTCTGTAA</td>
</tr>
</tbody>
</table>

- How similar are the strands?

- What sequence of bases are common in both strands?

- Find an alignment with the maximum number of matches:

<table>
<thead>
<tr>
<th>GTTGCAATCGGCTTCCGAAGCGCTGCCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTCGTTCGGAA TGCCGTTGCTCTGTAA</td>
</tr>
</tbody>
</table>

- Of another:

<table>
<thead>
<tr>
<th>ACCGGTCGAGTGCGCGGAAGCCGGCCGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTCGTTCGGAA TGCCGTTGCTCTGTAA</td>
</tr>
</tbody>
</table>

Matching DNA Sequences
Overview

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

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
- When the two sentences do not line up:
  - A little lad goed to a store
  - The boy went to the store
  - The little boy went to the store

Examiner says a sentence to the participant:

Examiner says the problem:

```plaintext
good ← went
boy ← had
little ← e
lad ← boy
```
Example

- Alignment was implicit in the formatting
  - Blank spots have an epsilon in their alignment
- One alignment (8 deletes, 6 inserts, 1 subs)
  - ACCTG TTG GAATGCCGTTGCTCTGTAAA
  - GCCG GCCG GCA A
- A worse one (3 deletes, 2 inserts, 13 subs)
  - ACCTG GT CGAG T GCG CGGAA GCCG GCCGAA
  - GT CG TTCG GAAT GCCG T GCT C TGT AAA

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
- Which indices of \( a \) and \( b \) are part of a match or substitution
  - Let \( a_q \) and \( b_q \) be the first non-blank or non-epsilon symbols
  - For any alignment between two sequences
- Let's be more precise about what an alignment is
- Find the alignment with the smallest penalty
- Penalty of 1 for insert/delete/sub and no penalty for match
- Score is based on the number of insertions, deletions, and substitutions
### Example

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>little boy went to the store</td>
<td>yeah to store to good led little boy went to the store</td>
</tr>
</tbody>
</table>

**A Better Way to View an Alignment**

- **Prompt** has $u$ words/characters, $q$ (response) has $m$ words
- **Response** has $v$ words/characters

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

### A Better Way to View an Alignment

- View prompt as columns and response as rows ($n \times m$ array)
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 $3^n \cdot m$ paths.
- For each path, compute score.
- For each diagonal move, determine if it is a match or substitution.

Can do some optimization:
- Keep track of best path so far, and prune out partial paths if they are worse.
- Can do this as a depth-first search.
- Prune out partial paths if they are worse.
- Can do this as a depth-first search.
- Gets rid of some redundancy.
- But bottom right hand corner will be redone many times.

Overview

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

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

**Dynamic Programming**

- Optimal substructure:
  - Optimal path from (i,j) to (k,l) is in the optimal solution. Optimal path from (i,j) to (k,l) is a subproblem of a lot of other problems.
  - Optimal subproblems:
  - Overlapping subproblems:
  - If (i,j) and (k,l) are in the optimal solution, optimal path from (i,j) to (k,l) is a subproblem of a lot of other problems.
Efficiency

- Rather than \( \Theta(n^2) \)
- Assume \( n \) and \( m \) are similar in size
- \( \Theta(nm) \)
- For \( m \times n \) cells
- \( \Theta(mn) \)
- Initialize col 0
- \( \Theta(m) \)
- Initialize row 0
- \( \Theta(n) \)

Filling in cells

From left-above: take score of \((i,j-1)\) if match, else substitute 1 (substitution)
From left: take score of \((i-1,j)\) and substitute 1 (delete)
From above: take score of \((i-1,j-1)\) and substitute 1 (insert)
From left-above: take score of \((i-1,j-1)\) if match, else subtract 1 (substitution)

Taking lowest 3 possible ways to get to cell

Fill in cell \((i,j)\) if cells above \((i,j-1), (i-1,j)\), and \((i-1,j-1)\) have in

little 0
lad 2
...
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 must be either a down, left, or diagonal, so from \(A_{i-1,j-1}\),

or \(A_{i-1,j}\) or \(A_{i,j-1}\).

This is what the algorithm computes.