Fundamental Techniques

1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithm

Divide-and-Conquer

- Divide: divide the input data $S$ into two or more disjoint subsets $S_1, S_2, \ldots$
- Recur: solve the subproblems recursively
- Conquer: combine the solutions for $S_1, S_2, \ldots$ into a solution for $S$

- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations
Computing integer power $a^n$

• Brute force algorithm

  Algorithm power(a,n)
  value ← 1
  for i ← 1 to n do
    value ← value × a
  return value

• Complexity: O(n)

Computing integer power $a^n$

• Divide and Conquer algorithm

  Algorithm power(a,n)
  if (n = 1)
    return a
  partial ← power(a,floor(n/2))
  if n mod 2 = 0
    return partial × partial
  else
    return partial × partial × a

• Complexity: $T(n) = T(n/2) + O(1)$
  $\Rightarrow T(n) = O(\log n)$

Integer Multiplication

• Multiply two n-digit integers $I$ and $J$.

  ex: 61438521 × 94736407

  6 1 4 3 8 5 2 1
  × 9 4 7 3 6 4 0 7
  4 3 0 0 6 9 6 4 7
  0 0 0 0 0 0 0 0
  2 4 5 7 5 4 0 8 4
  ... 5 8 2 0 4 6 4 7 3 0 9 3 4 0 4 7

• Complexity: $O(n^2)$
**Integer Multiplication**

- **Divide:** Split \( I \) and \( J \) into high-order and low-order digits.
  - e.g. \( I = 61438521 \) is divided into \( I_h = 6143 \) and \( I_l = 8521 \)
  - i.e. \( I = I_h \times 10^4 + I_l \)

- **Conquer:** define \( I \times J \) by multiplying the parts and adding
  \[
  I \times J = (I_h \times J_h)10^8 + (I_h \times J_l + I_l \times J_h)10^4 + (I_l \times J_l)
  \]

Complexity: \( T(n) = 4T(n/2) + n \Rightarrow T(n) \) is \( O(n^2) \).

**Improved Algorithm**

- \( I \times J = (I_h \times J_h)10^8 + (I_h \times J_l + I_l \times J_h)10^4 + (I_l \times J_l) \)

Complexity: \( T(n) = 3T(n/2) + cn \),

\( \Rightarrow T(n) \) is \( O(n \log_2 3) \), by the Master Theorem

Thus, \( T(n) \) is \( O(n^{1.585}) \).

**Closest Pair of Points**

- Given \( n \) points find the 2 that are closest.
Distance Between Two Points

- If the points are: \((x_i, y_i)\) and \((x_j, y_j)\)

\[
\text{Distance} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}
\]

Closest Pair of Points

- Brute-Force Strategy:
  i) Compute distance between in each pair
  ii) Examine \(n(n-1)/2\) pair of points
  iii) Determine the pair for which the distance is minimum

- Complexity: \(O(n^2)\)

Closest Pair of Points

- Divide and Conquer Strategy.
Closet Pair of Points

- Divide and Conquer Strategy,
  i) Divide the point set to roughly two sub-sets L and R
  ii) Recursively Determine the Closest Pair of Points in L and in R. Let the distances be $d_L$, $d_R$
  iii) Determine the closest pair such that one is L and the other in R. Let the distances be $d_c$
  iv) From the three closest pairs select the one with least distance.

• Let $\delta = \min(d_L, d_R)$

• For every point in the left region search for points in the right region in area $2\delta$ high

Max 6 comparisons per point
Matrix Multiplication

• Another well known problem: (see Section 5.2.3)

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\begin{bmatrix}
E & F \\
G & H
\end{bmatrix}
= \begin{bmatrix}
I & J \\
K & L
\end{bmatrix}
\]

• Brute force Algorithm: O(n^3)
• Divide and conquer algorithm; O(n^{2.81})

Dynamic Programming

Dynamic Algorithm

• Most difficult of the fundamental techniques. [Ref: Sahni, “Data Structure and Algorithms and Applications”]
• Takes a problem that seems to require exponential time and produces polynomial-time algorithm to solve it.
• The resulting algorithm is simple. Often requires a few lines of code.
Recursive Algorithms

• Best when sub-problems are disjoint.
• Example of Inefficient Recursive Algorithm:
  Fibonacci Number: \( F(n) = F(n-1) + F(n-2) \)

```plaintext
Recursive Algorithms

function F(n)
    if n = 0
        return 0
    else if n = 1
        return 1
    return F(n-1) + F(n-2)

Complexity: \( T(n) \leq 2T(n-1) + O(1) \)
⇒ \( T(n) = O(2^n) \)
⇒ \( T(n) = 0.7236 \times 1.618^{n-1} \)
```

F(40) ~ 75 seconds
F(70) ~ 4.4 years

Trace of Recursive Version

If we could store the list of all pre-computed values !!
Dynamic Programming

• Find a recursive solution that involves solving the same problem many times.
• Calculate *bottom up* and avoid recalculation

Efficient Version

• Linear Time Iterative:

```plaintext
Algorithm Fibonacci(n)
    Fn[0] ← 0
    Fn[1] ← 1
    for i ← 2 to n do
        Fn[i] ← Fn[i-1] + Fn[i-2]
    return Fn[n]
```

Fibonacci(40) < microseconds
Fibonacci(70) < microseconds

Efficient Version

• Linear Time Iterative:

```plaintext
Algorithm Fibonacci(n)
    Fn_1 ← 1
   Fn_2 ← 0
   Fn ← 1
    for i ← 2 to n do
        Fn ← Fn_1 + Fn_2
        Fn_2 ← Fn_1
        Fn_1 ← Fn
    return Fn
```
Computing Binomial Coefficient

\[(a+b)^n = C(n,0)a^n + \ldots + C(n,i)a^{n-i}b^i + \ldots + C(n,n)b^n\]

Recursive Solution:
- \(C(n,0) = 1\) for all \(n\)
- \(C(n,n) = 1\) for all \(n\)
- \(C(n,k) = C(n-1,k-1) + C(n-1,k)\) for \(n > k > 0\)

Dynamic Programming Solution:

```
for i ← 0 to n do
    C[n,0] ← 1
    C[n,n] ← 1
for i ← 2 to n
    for j ← 1 to i-1
        C[i,j] ← C[i-1,j-1] + C[i-1,j]
```

Dynamic Programming

- Best used for solving optimization problems.
- Optimization Problem defn:
  - Many solutions possible
  - Choose a solution that minimizes the cost
Matrix Chain-Products
• Review: Matrix Multiplication.
  – \( C = A \cdot B \)
  – \( A \) is \( d \times e \) and \( B \) is \( e \times f \)
  \[
  C[i, j] = \sum_{k=1}^{e} a[i,k] \cdot b[k,j]
  \]
  – \( O(d,e,f) \) time

\[
\begin{array}{c}
\text{\( a \)} \\
\text{\( b \)} \\
\text{\( c \)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{\( d \)} \\
\text{\( e \)} \\
\text{\( f \)} \\
\end{array}
\]

Matrix Chain-Products
• Matrix Chain-Product:
  – Compute \( A_0 \cdot A_1 \cdot \ldots \cdot A_{n-1} \)
  – \( A_i \) is \( d_i \times d_{i+1} \)
  – Problem: How to parenthesize?
• Example
  – \( B \) is \( 3 \times 100 \)
  – \( C \) is \( 100 \times 5 \)
  – \( D \) is \( 5 \times 5 \)
  – \( B \cdot (C \cdot D) \) takes \( 1500 + 2500 = 4000 \) ops
  – \( (B \cdot C) \cdot D \) takes \( 1500 + 75 = 1575 \) ops

An Enumeration Approach
• Matrix Chain-Product Alg.:
  – Try all possible ways to parenthesize
    \( A_0 \cdot A_1 \cdot \ldots \cdot A_{n-1} \)
  – Calculate number of ops for each one
  – Pick the one that is best
• Running time:
  – The number of parenthesizations is equal to the number of binary trees with \( n \) nodes
  – This is exponential!
  – It is called the Catalan number, and it is almost \( 4^n \).
  – This is a terrible algorithm!
A “Recursive” Approach

- Let $A_i$ be a $d_i \times d_{i+1}$ matrix.
- There has to be a final multiplication to arrive at the solution.
- Say, the final multiply is at index $i$.

$$A_i \cdots A_1 * A_0$$

- Let $N_{i+1,n}$ be the number of operation for $A_i \cdots A_1 * A_0$.
- Let $N_{i,n-1}$ be the number of operation for $A_i \cdots A_1$.

Then total operations:

$$N_{i,n-1} = N_{i,k} + N_{k+1,n-1} + d_i d_{k+1} d_n$$

A Dynamic Programming Algorithm

```
Algorithm matrixChain(S):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal paranthesisization of S
    for i ← 0 to n-1 do
        $N_{i,i} ← 0$
    for b ← 1 to n-1 do
        for i ← 0 to n-b do
            j ← i + b
            $N_{i,j} ← \infty$
            for k ← i to j-1 do
                $N_{i,j} ← \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_j\}$
```

• Find a recursive solution
• Calculate bottom up and avoid recalculation
• Running time: $O(n^3)$
A Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the N array by diagonals
- N_{i,j} gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: O(n^3)
- Getting actual parenthesization can be done by remembering "k" for each N entry

Dynamic Programming

- Is based on Principle of Optimality
- Generally reduces the complexity of exponential problem to polynomial problem
- Often computes data for all feasible solutions, but stores the data and reuses

When to Use Dynamic Programming?

- Brute Force solution is prohibitively expensive
- Problem must be divisible into multiple stages
- Choices made at each stage include the choices made at previous stages.