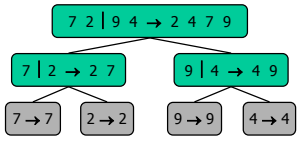
Fundamental Techniques

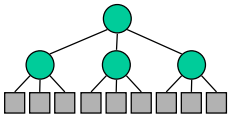
1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithm

Divide-and-Conquer



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, 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
```

$$a^n = a \times a \ldots \times a$$

- 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
```

$$a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{if } n \text{ is even} \\ a^{\frac{n}{2}} \times a^{\frac{n}{2}} \times a & \text{if } n \text{ is odd} \end{cases}$$

- Complexity: T(n) = T(n/2) + O(1) ⇒T(n) is O(log n)

Integer Multiplication

- Multiply two n-digit integers *I* and *J*.
  ex: 61438521 × 94736407

- Complexity: $O(n^2)$

```
  6 1 4 38521
×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
```

Integer Multiplication

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

$$I = I_h 10^{n/2} + I_l$$
$$J = J_h 10^{n/2} + J_l$$

- Conquer : define $I \times J$ by multiplying the parts and adding

$$I \times J = (I_h 10^{n/2} + I_l) \times (J_h 10^{n/2} + J_l)$$
$$= \underbrace{(I_h \times J_h)}_{\text{subProblem}_1} 10^n + [\underbrace{(I_h \times J_l)}_{\text{subProblem}_3} + \underbrace{(I_l \times J_h)}_{\text{subProblem}_4}] 10^{n/2} + \underbrace{(I_l \times J_l)}_{\text{subProblem}_2}$$

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

Integer Multiplication

- Improved Algorithm

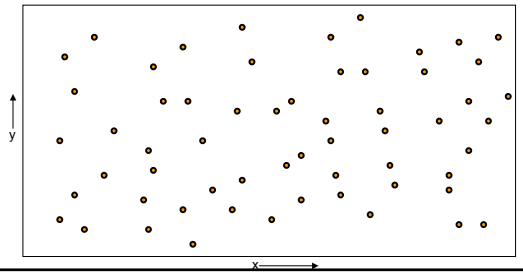
$$I \times J = (I_h \times J_h) 10^n + [(I_h \times J_l) + (I_l \times J_h)] 10^{n/2} + (I_l \times J_l)$$

$$= \underbrace{(I_h \times J_h)}_{\text{subProblem}_1} 10^n + [\underbrace{(I_h - I_l) \times (J_l - J_h)}_{\text{subProblem}_3} + \underbrace{(I_h \times J_h)}_{\text{subProblem}_1} + \underbrace{(I_l \times J_l)}_{\text{subProblem}_2}] 10^{n/2} + \underbrace{(I_l \times J_l)}_{\text{subProblem}_2}$$

- Complexity: T(n) = 3T(n/2) + cn,
  $\Rightarrow$ T(n) is $O(n^{\log_2 3})$, by the Master Theorem
- Thus, T(n) is $O(n^{1.585})$.

Closet 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)

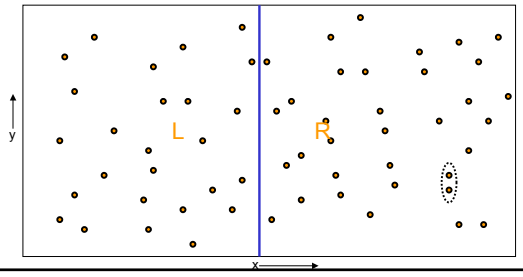
  Distance = $\sqrt{((x_i-x_j)^2+(y_i- y_j)^2)}$

Closet 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: $O(n^2)$

Closet Pair of Points

- Divide and Conquer Strategy.

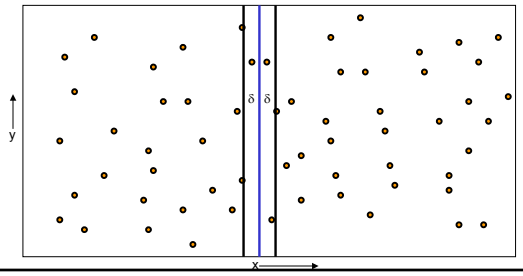


Closet Pair of Points

- 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.

Closet Pair of Points

- Let $\delta = \min(d_L , d_R)$



Closet Pair of Points

- For every point in the left region search for points in the right region in area 2δ high

Max 6 comparisons per point

Matrix Multiplication

- Another well known problem: (see Section 5.2.3)

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

- 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)

Recursive Algorithms

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

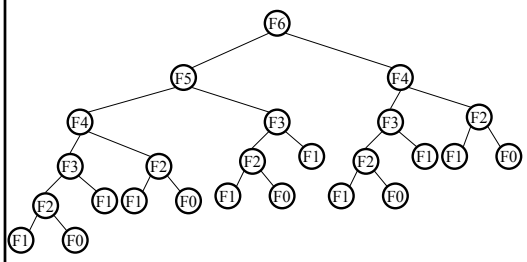
F(40) ~ 75 seconds
F(70) ~ 4.4 years

Complexity: $T(n) \leq 2T(n-1) + O(1)$

$\Rightarrow T(n)$ is $O(2^n)$

$\Rightarrow T(n) = 0.7236 * 1.618^n - 1$

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:

```
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:

```
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 $n>k>0$

Computing Binomial Coefficient

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

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

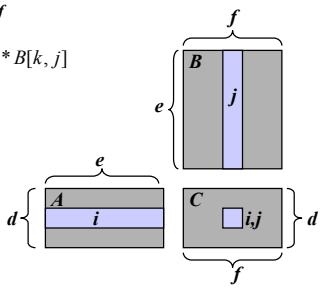
- Best used for solving optimization problems.
- Optimization Problem defn:
  - Many solutions possible
  - Choose a solution that minimizes the cost

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

  - $O(d.f.e)$ time



Matrix Chain-Products

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

An Enumeration Approach

- Matrix Chain-Product Alg.:
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n'-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of paranthesizations 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

•Find a recursive solution
•Calculate *bottom up* and avoid recalculation

- Let A_i is 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

  $A_0*\ldots*A_i*A_{i+1}*\ldots*A_{n-1} = (A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
- Let $N_{0,n-1}$ is number of operation for $A_0*\ldots*A_i*A_{i+1}*\ldots*A_{n-1}$
  $N_{0,i}$ for $A_0*\ldots*A_i$ and $N_{i+1,n-1}$ for $A_{i+1}*\ldots*A_{n-1}$

Then total operations: $N_{0,n-1} = N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n$

$$N_{0,n-1} = \min_{0 \le k < n-1} \{N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n\}$$

A "Recursive" Approach

•Find a recursive solution
•Calculate *bottom up* and avoid recalculation

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

A Dynamic Programming Algorithm

•Find a recursive solution
•Calculate *bottom up* and avoid recalculation

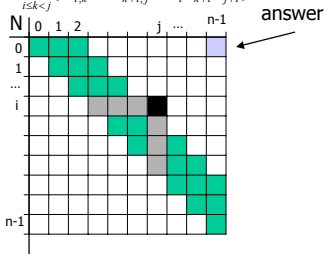
•Running time: $O(n^3)$

```
Algorithm matrixChain(S):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal paranthization of S
    for i ← 0 to n-1 do
        N_{i,i} ← 0                          Length 1 are easy, so start with them
    for b ← 1 to n-1 do                      Then do length 2,3,… sub-problems.
        for i ← 0 to n-1-b do
            j ← i+b
            N_{i,j} ← ∞
            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+1}}
```

A Dynamic Programming Algorithm Visualization

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious 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.