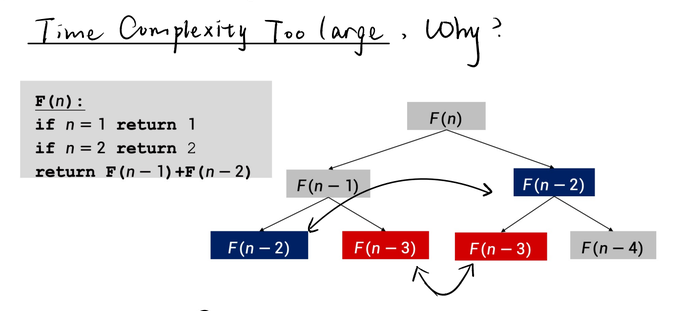
Dynamic Programming

Fibonacci numbers: 1, 1, 2, 3, ...

$F(n) = F(n-1) + F(n-2)$

If we solve $F(n)$ using recurrence: $T(n) = O(\varphi^n)$ ...

Time Complexity Too large. Why?



Too many SAME subproblems

Alternative Way: Dynamic Programming

```
F(n):
  Allocate an array F of size n
  F[1] = 1
  F[2] = 1
  for i = 3 to n:
      F[i] = F[i-1] + F[i-2]
  return F[n]
```

Running Time: $O(n)$

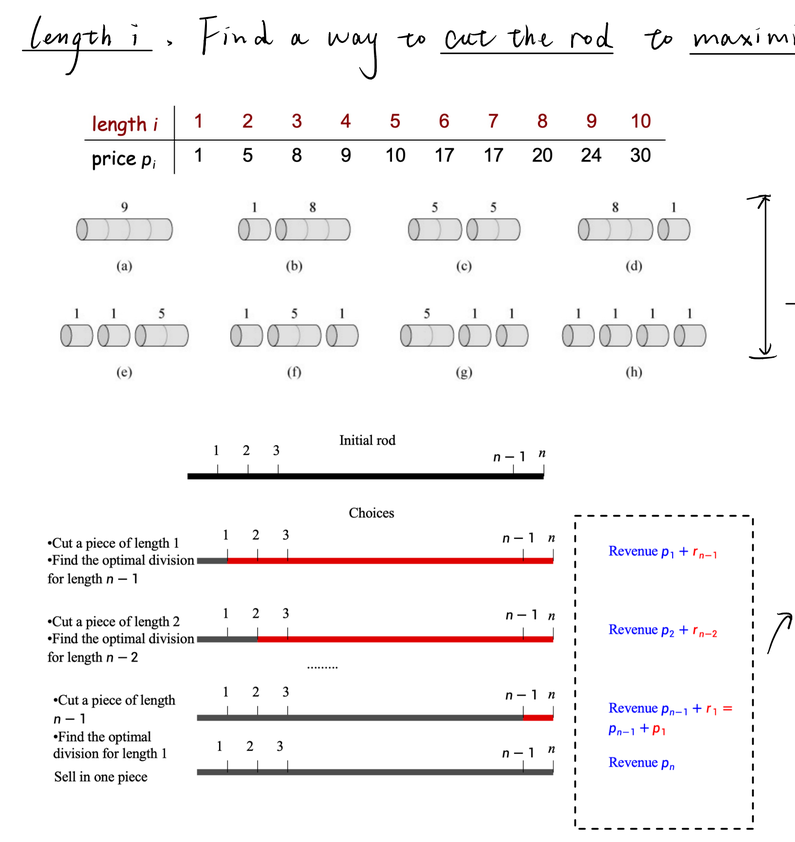
Space: $O(n)$

⇒ DP: Dynamic Programming ("Planning", not coding!)

Solve recurrence by storing solutions

The rod cutting Problem:

A rod of length n, price p_i for $i = 1, ..., n$, representing price of rod of length i. Find a way to cut the rod to maximize the total revenue.



$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}$ where $r_0 = p_0$

⇒ Recurrence + We're going to use Dynamic Programming

Algorithm:

Let $r[0 \ldots n]$ be a new array

$r[0] = 0$

for $i = 1$ to n do

  $q \leftarrow -\infty$

  for $j = 1$ to i do

    $q \leftarrow \max(q, p[j] + r[i-j])$

  $r[i] = q$

return $r[n]$

Running Time: $O(n^2)$

This Algorithm ONLY finds max revenue!

i.e. We don't know how to cut to realize the maximum revenue

Want to find "Solution" that yields max revenue (How to cut)

Let $r[0 \ldots n]$, $s[1 \ldots n]$ be two empty arrays

$r[0] \leftarrow 0$

for $i = 1$ to n do:

  $q \leftarrow -\infty$

  for $j = 1$ to i do

    if $p[j] + r[i-j] > q$ then

      $q \leftarrow p[j] + r[i-j]$

      $s[i] \leftarrow j$ // Record the way of optimal cutting on length i

  $r[i] \leftarrow q$

$k \leftarrow n$

while $k > 0$:

  print $s[k]$

  $k \leftarrow k - s[k]$ // cut the rod

(print out the optimal solution)

return $r[n]$

Maximum Sum Problem

Find the subset with maximum sum of non-adjacent elements.

If using Greedy Algorithm ⇒ Not Global Optimal!

ex. $\{2, 9, 8, 6, 1\}$ Greedy: $\{2, 8, 1\}$, Sum = 11

Actual Optimal Solution: $\{2, 9, 6\}$, Sum = 17

Let A_i be the subset containing first i elements in A

$A_i = \{a_1, a_2, a_3, \ldots, a_i\}$

Let S_i be the solution of A_i

Let W_i be the Sum of S_i

Key Observation:

for element a_i:

① if $a_i \notin S_i$, then $a_{i-1} \notin S$ + $W_i = a_i + W_{i-2}$

② if $a_i \notin S_i$, then $W_i = W_{i-1}$, since a_i does not be counted in W_i

⇒ $W_i = \max\{W_{i-1}, W_{i-2} + a_i\}$

Then We can use DP to solve this recurrence!

Algorithm:

Create two empty array $W[1 \ldots n]$, $b[1 \ldots n]$ (Storing W_i; Recording if $a_i \in S_i$)

$W[0] \leftarrow 0$, $b[0] \leftarrow$ true

if $a_1 > a_2$ then $W[1] \leftarrow a_1$, $b[1] \leftarrow$ true

if $a_1 < a_2$ then $W[1] \leftarrow a_2$, $b[1] \leftarrow$ false

for $i = 3$ to n do

  if $W[i-2] + a_i > W[i-1]$ then:

    $W[i] \leftarrow W[i-2] + a_i$

    $b[i] \leftarrow$ true

  else

    $W[i] \leftarrow W[i-1]$

    $b[i] \leftarrow$ false

Print out the solution

Minimum number of Coins

Amount n and k denomination $d_1 \ldots d_k$

Minimum number of coins for amount n

If Greedy ⇒ Not global optimal

Recurrence: $C[n] = 1 + \min\{C[n - d_i] : i \in 1 \ldots k\}$

Where $C[n]$ denotes Minimum number of coins for amount n

Algorithm:

DP: coin(n)

Create empty array $C[1 \ldots n]$, $S[1 \ldots n]$

$C[0] = 0$

for $i = 1$ to n:

  ...

Running Time: $O(kn)$

While $n > 0$:

  print $S[n]$

  $n \leftarrow n - S[n]$

Weighted Interval Scheduling

Label jobs by finishing time. $f_1 \le f_2 \le \cdots \le f_n$.

Def: $P(j)$: largest index $i < j$ that job i is compatible with job j

⇒ for all j' that $P(j) < j' < j$, job j' is incompatible with job j

Def. $V[j]$ = value of optimal solution to the problem on jobs 1, 2, ..., j.

Step 1 Recurrence: Solve the problem for jobs {1}, {1,2}, ..., {1,2, ..., n}.

Case 1: Select job j.

- can't use incompatible jobs $\{p(j) + 1, p(j) + 2, \ldots, j - 1\}$
- must include optimal solution to problem on jobs 1, 2, ..., $p(j)$

Case 2: Do not select job j.

- must include optimal solution to problem on jobs 1, 2, ..., $j - 1$

Key Observation

$V[j] = \max\{v_j + V[p(j)], V[j-1]\}, V[0] = 0$

Step 2: Solve the problem by incrementally including jobs 1, 2, ..., n

```
sort all jobs by finish time
V[0] ← 0
for j ← 1 to n
    V[j] ← max{v_j + V[p(j)], V[j − 1]}
return V[n]
```

Similarly With Maximum Sum

Algorithm:

Sort all jobs by finish time

$V[0] = 0$

for $j \leftarrow 1$ to n do:

  if $f_j + V[p(j)] > V[j-1]$ then:

    $V[j] \leftarrow f_j + V[p(j)]$

    $keep[j] \leftarrow 1$ // keep j

  else:

    $V[j] \leftarrow V[j-1]$

    $keep[j] \leftarrow 0$ // don't keep j

$j \leftarrow n$

while $j > 0$ do:

  if $keep[j] = 1$ then:

    print (j)

    $j \leftarrow P(j)$

  else:

    $j \leftarrow j - 1$

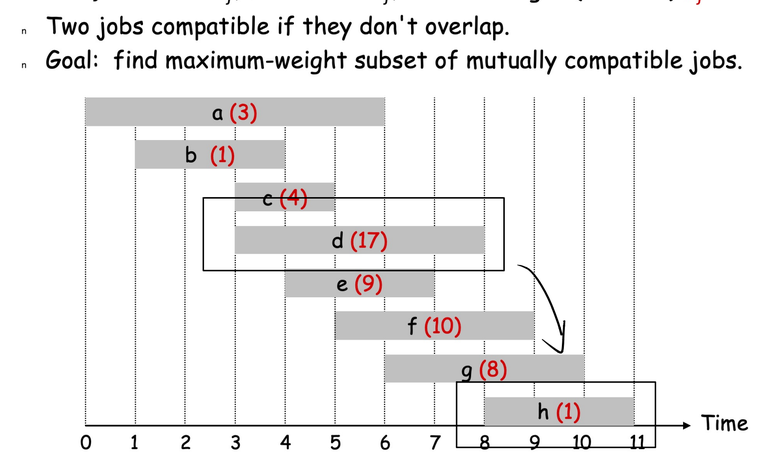
Running time is dominated by Sorting

$T(n) = O(n \log n)$

Weighted Interval Scheduling

Weighted interval scheduling problem.

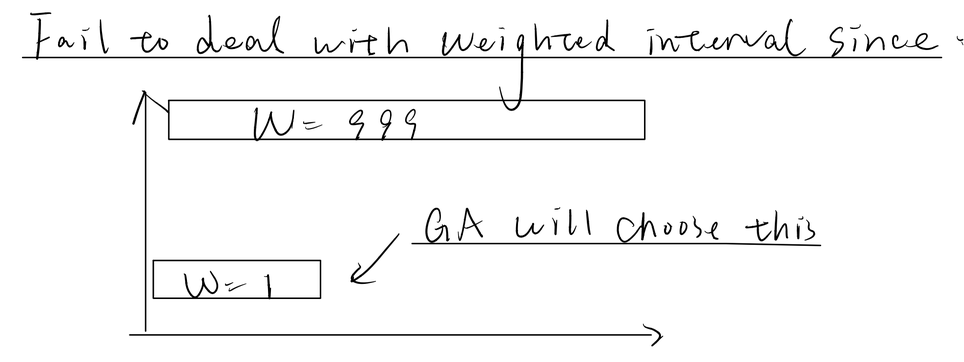
- Job j starts at s_j, finishes at f_j, and has weight (or value) v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.



Greedy Algorithm:

Sorted by Finish time, for i = n, Add job if it's compatible with previous chosen jobs

Fail to deal with weighted interval since



GA will choose this

Main Idea of DP:

(1) Find Recurrence (Main Step)

Some Recurrence Example

① Fibonacci numbers: 1, 1, 2, 3, 5, 7, ...

$F(n) = F(n-1) + F(n-2)$

② The rod cutting Problem

$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}$ where $r_0 = p_0$

③ Maximum Sum Problem

$W_i = \max\{W_{i-1}, W_{i-2} + a_i\}$

④ Minimum number of Coins

$C[n] = 1 + \min\{C[n - d_i] : i \in 1 \ldots k\}$

⑤ Weighted Interval Scheduling

$V[j] = \max\{v_j + V[p(j)], V[j-1]\}$, $V[0] = 0$

(2) Store from bottom, Storing solutions for each subproblems

Dynamic Programming (2D)

0/1 Knapsack Problem

A set of n items, each has weight w_i, value v_i

Find $x_1, \ldots, x_n \in \{0, 1\}$, $\sum x_i w_i \le W$

Then maximize $\sum x_i v_i$

Greedy Can not solve this problem → Only can solve fragment knapsack ($x_i \in [0, 1]$)

If 1D dynamic programming

$V[n] = \max\{v_i + V[n - w_i], \ldots, V[n - w_n] + v_n\}$

⇒ may choose Same item many times!

We need 2D dynamic programming

Recurrence: $V[i, w] = \max\{V[i-1, w - w_i] + v_i, V[i-1, w]\}$

Base Case: $V[i, w] = 0$ if $i = 0$ or $w = 0$

Algorithm:

Let $V[0 \ldots n, 0 \ldots W]$ be a new 2D array of all 0

for $i = 1$ to n do

  for $j = 1$ to W do

    if $w[i] \le w$ and $V[i-1, j - w[i]] + v[i] > V[i-1, j]$ then:

      $V[i, j] \leftarrow V[i-1, j - w[i]] + v[i]$

    else:

      $V[i, j] \leftarrow V[i-1, j]$

return $V[n, W]$

Running time $O(nW)$ ⇒ Can be improved to $O(n+W)$

We want to know Specific Solution

Let $V[0 \ldots n, 0 \ldots W]$ be a new 2D array of all 0

for $i = 1$ to n do

  for $j = 1$ to W do

    if $w[i] \le w$ and $V[i-1, j - w[i]] + v[i] > V[i-1, j]$ then

      $V[i, j] \leftarrow V[i-1, j - w[i]] + v[i]$

      $keep[i, w] \leftarrow 1$

    else:

      $V[i, j] \leftarrow V[i-1, j]$

      $keep[i, w] \leftarrow 0$

$k \leftarrow W$

for $i \leftarrow n$ down to 1 do:

  if $keep[i, k] = 1$ then

    print i

    $k \leftarrow k - w[i]$

Problem: Suppose that you are given two knapsacks with the same max weight W. Give an $O(nW^2)$ dynamic programming algorithm for finding the maximum value of items that can be carried by the two knapsacks. $V[i, w^1, w^2]$ is the largest obtained value for knapsack 1 with capacity w^1, knapsack 2 with capacity w^2 choosing ONLY from items 1 ... i.

$V[i, w^1, w^2] = \max\{V[i-1, w^1, w^2], V[i-1, w^1 - w_i, w^2], V[i-1, w^1, w^2 - w_i]\}$

⇒ Find order for filling the table

For $i \leftarrow 1$ to n:

  For $w^1 \leftarrow 1$ to W:

    For $w^2 \leftarrow 1$ to W:

⇒ Return $V[i, W, W]$

Longest Common Sequences

Given two sequence $X = (x_1, x_2, \ldots, x_m)$ $Y = (y_1, y_2, \ldots, y_n)$

$Z = (z_1, z_2, \ldots, z_k)$ is a Common Sequence of X and Y if:

$x_{i_p} = y_{j_p} = z_p$, for all $p = 1, 2, \ldots, k$

$i_1 < i_2 < \ldots < i_k$, $j_1 < j_2 < \ldots < j_k$

ex. A: A B B A C B D A B

B: B D C A B A

⇒ Z: B C B A

Let $C[i, j]$ be the length of Longest common sequence of $X[1 \ldots i]$, $Y[1 \ldots j]$

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + C[i-1, j-1] & \text{if } X[i] = Y[j] \\ \max[C[i-1, j], C[i, j-1]] & \text{otherwise} \end{cases}$$

Dynamic Programming over intervals

Main idea: problem Contains item 1 ... n

Optimal Solution of problem [1 ... n] = Subproblems of Smaller length.

Let $p[i, j]$ be subproblems Contains $i \ldots j$.

Base Case: subquestions of length 1: $p[1,1], p[2,2] \ldots p[n,n]$

Recursive Step: sub-problem of length 2: $p[1,2], p[2,3] \ldots$

sub-problem of length 3: $p[1,3], p[2,4] \ldots$

⋮

Until filling up the 2D table (n × n)

Longest Palindrome Substring

"level" "madam" ... (palindrome)

⇒ Let $p[i, j]$ be true if $X[i \ldots j]$ is a palindrome.

Initial Conditions

⇒ $p[i, i]$ is always true and $P[i, i+1] = (X[i] == X[i+1])$

Recurrence:

⇒ $p[i, j] = (X[i] = X[j])$ and $P[i+1, j-1]$ if $i \ne j$

Algorithm.

max ← 1

for $i \leftarrow 1$ to $n-1$ do:

  $p[i, i] \leftarrow$ true.

  if $x_i = x_{i+1}$ then $p[i, i+1] \leftarrow$ true.

  else $p[i, i+1] \leftarrow$ false

for $l \leftarrow 3$ to n do.

  for $i \leftarrow 1$ to $n - l + 1$ do.

    $j \leftarrow i + l - 1$

    if $p[i+1, j-1]$ = true and $x_i = x_j$ then

      $p[i, j] \leftarrow$ true, max ← l

    else $p[i, j] \leftarrow$ false

return max

Matrix-Chain Multiplication

$A_{p \times q} \times B_{q \times r} = C_{p \times r}$

Creating $C_{p \times r}$ needs pqr times multiplications

for three matrices $A_{10 \times 100}$, $B_{100 \times 5}$, $C_{5 \times 50}$

if $A(BC)$: # multiplications: $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$

if $(AB)C$: # multiplications: $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ Smaller!

Q: Given a Sequence or Chain $A_1, \ldots, A_n$ of n matrices

Determine the optimal way to parenthesize

(i.e. find the way which has minimum # of multiplications)

Optimal Structure

Values $p_0 \ldots p_n$ ⇒ Matrix A_i has size $p_{i-1} p_i$

$A_{i \ldots j}$: result of Matrix multiplication $A_i \times \cdots \times A_j$

Optimal Parenthesization

$A_{1 \ldots n} = A_{1 \ldots k} \times A_{k+1 \ldots n}$

Observation:

If Parenthesization is optimal

i.e. $A_{1 \ldots n} = A_{1 \ldots k} \times A_{k+1 \ldots n}$ is optimal

then parenthesization of $A_{1 \ldots k}$ & $A_{k+1 \ldots n}$ must be optimal

Since this is Solution of sub-problem

Let $m[i, j]$ be minimum number of multiplications to compute $A_{i \ldots j}$

Recurrence: $m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$

Algorithm:

for $i \leftarrow 1$ to n

  $m[i, i] = 0$

Time: $O(n^3)$

for $l \leftarrow 2$ to n

  for $i \leftarrow 1$ to $n - l + 1$

    $j \leftarrow i + l - 1$

    $m[i, j] \leftarrow \infty$

    for $k \leftarrow i$ to $j - 1$ do

      if $(m[i, k] + m[k+1, j] + p_{i-1} p_k p_j < m[i, j])$ then

        $m[i, j] \leftarrow m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

        $s[i, j] \leftarrow k$

return $s[1, n]$, $m[1, n]$

Binary Merge tree

A Binary merge tree is optimal if it minimize the weighted external path length

⇒ min $B(T) = \sum_{i=1}^{n} w(a_i) d(a_i)$

Algorithm == Huffman Encoding

Create a min-heap $T[1 \ldots n]$ base on n initial size

While (heap size ≥ 2) do:

  $x \leftarrow$ T. extract-min()

  $y \leftarrow$ T. extract-min()

  create node z whose children are a, b.

  $size(z) \leftarrow size(a) + size(b)$

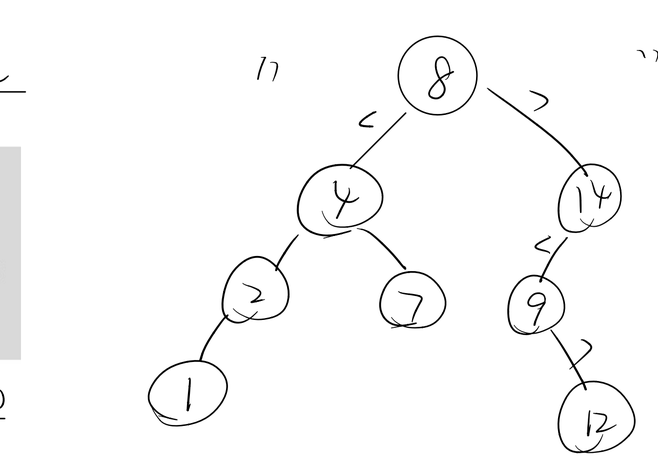
  T. insert(z)

return T. extract-min()

Binary Search Tree

```
Tree-Search(T,k):
x ← T.root
while x ≠ nil and k ≠ x.key do
    if k < x.key then x ← x.left
    else x ← x.right
return x
```

Worst time $\Theta(\log n)$



Same Question: Optimal BST

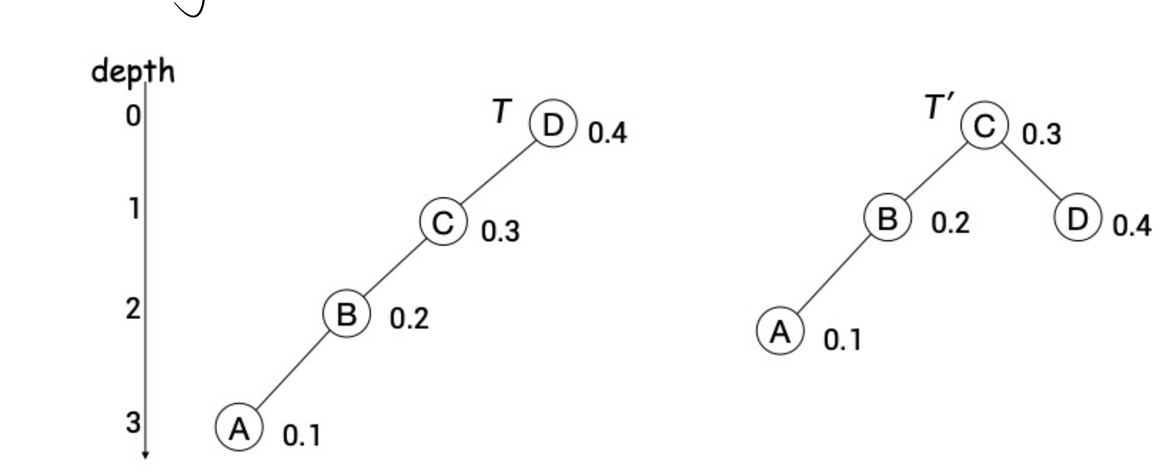
"Minimizing $B(T) = \sum_{i=1}^{n} w(a_i)(d(a_i) + 1)$"

Diffence: ① BST is Sorted!

② elements appears in both internal & leaf node

Motivation: if weight ⇒ probability then optimal BST will minimize total Search cost

Greedy don't work



(Greedy always choose heaviest node as root)

Let $B(T_{i,j}) = \sum_{l=i}^{j} w(a_l)(d(a_l) + 1)$

$w[i, j] = d(a_i) + d(a_{i+1}) + \cdots + d(a_j)$

if we choose k as root

$B(T_{i,j}) = [B(T_{i,k-1}) + w[i, k-1]] + d(a_k) + [B(T_{k+1,j}) + w[k+1, j]]$

$= B(T_{i,k-1}) + B(T_{k+1,j}) + w[i,j]$

Let $e[i, j]$ = minimum Cost of any BST on $a_i \ldots a_j$

$e[i,j] = \min_{i \le k \le j}\{e[i, k-1] + e[k+1, j]\} + w[i,j]$

$e[i, j] = 0$ for $i > j$ "Recurrence"

$e[i, i] = f(a_i)$ for all i

Algorithm

Let $e[1 \ldots n+1, 0 \ldots n]$, $w[1 \ldots n+1, 0 \ldots n]$, $root[1 \ldots n, 1 \ldots n]$ be new arrays of all 0

for $i \leftarrow 1$ to n:

  $w[i, i] \leftarrow f(a_i)$

  for $j \leftarrow i+1$ to n:

    $w[i, j] \leftarrow w[i, j-1] + f(a_j)$

Time: $O(n^3)$

for $l \leftarrow 1$ to n

  for $i \leftarrow 1$ to $n - l + 1$

    $j \leftarrow i + l - 1$

    $e[i, j] \leftarrow +\infty$

    for $k \leftarrow i$ to j do

      $t \leftarrow e[i, k-1] + e[k+1, j] + w[i, j]$

      if $t < e[i, j]$ then $e[i, j] \leftarrow t$, $root[i, j] \leftarrow k$

return Construct_BST(root, 1, n)

Construct_BST(root, i, j):

  if $i > j$ return nil

Time: $O(n)$

  Create a node z

  $z.key \leftarrow a[root[i, j]]$

  $z.left \leftarrow$ Construct_BST(root, i, root[i, j] − 1)

  $z.right \leftarrow$ Construct_BST(root, root[i, j] + 1, j)