Dynamic Programming over Intervals

DP Over Intervals

Main idea of interval dynamic programming.

- Problem contains items $1,2,\dots,n$. Recurrence gives optimal solution of $\text{problem}[1,n]$ as function of optimal solution of subproblems of smaller length (length refers to the number of items in the problem)
  - Base case contains problems of length 1, i.e., $\text{problem}[1,1], \text{problem}[2,2], \dots, \text{problem}[n,n]$.
    - All solutions are stored in diagonals of a 2D array. Solutions of small problems are re-used by bigger ones.
  - Then, we solve $n-1$ problems of length 2, i.e., $\text{problem}[1,2], \text{problem}[2,3], \dots, \text{problem}[n-1,n]$.
  - ….
  - Then, we solve $n-l+1$ problems of length l, i.e., $\text{problem}[1,l]$, $\text{problem}[2,l+1], \dots, \text{problem}[n-l+1,n]$.
  - Finally, we solve 1 problem of size n: $\text{problem}[1,n]$

- Algorithm fills diagonals in a 2 D table from smallest to largest problem length. First the main diagonal (base case), then the one on top of that,….At the end, the solution of the $\text{problem}[1,n]$ is at the top right cell.

Longest Palindromic Substring

Def: A palindrome is a string that reads the same backward or forward.

Ex:

- **radar, level, racecar, madam**
- "A man, a plan, a canal – Panama!" (ignoring space, punctuation, etc.)

Problem: Given a string $X = x_1 x_2 \dots x_n$, find the longest palindromic substring.

Ex:

- $X =$ **ACCABA**
- Palindromic substrings: **CC, ACCA, ABA**
- Longest palindromic substring: **ACCA**

Note:

- Brute-force algorithm takes $O(n^3)$ time.
- Recall: A substring must be contiguous

Dynamic Programming Solution

Def: Let $p[i,j]$ be $true$ iff $X[i..j]$ is a palindrome.

The Recurrence:

Initial Conditions (subproblems of sizes 1 & 2)

- $p[i,i] = true$, for all i
  - **ACBBCABA**

- $p[i,i+1] = true$ if $x_i = x_{i+1}$
  - **ACBBCABA**

The Actual Recurrence

- $p[i,j] = true$

  if $x_i = x_j$ AND $p[i+1,j-1] = true$
  - **ACBBCABA**
  - **ACBBCABA**

i	1	2	3	4	5	6	7	8
	B	A	B	B	C	C	C	B

A Completed DP Table

Initial Condition
j=i; j=i+1

i/j	1	2	3	4	5	6	7	8
1	T	F						
2		T	F					
3			T	T				
4				T	F			
5					T	T		
6						T	T	
7							T	F
8								T

j=i+2

i/j	1	2	3	4	5	6	7	8
1	T	F	T					
2		T	F	F				
3			T	T	F			
4				T	F	F		
5					T	T	T	
6						T	T	F
7							T	F
8								T

j=i+4

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F	F			
2		T	F	F	F	F		
3			T	T	F	F	F	
4				T	F	F	F	T
5					T	T	T	F
6						T	T	F
7							T	F
8								T

j=i+3

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F				
2		T	F	F	F			
3			T	T	F	F		
4				T	F	F	F	
5					T	T	T	F
6						T	T	F
7							T	F
8								T

j>i+4

i/j	1	2	3	4	5	6	7	8
1	T	F	T	F	F	F	F	F
2		T	F	F	F	F	F	F
3			T	T	F	F	F	F
4				T	F	F	F	T
5					T	T	T	F
6						T	T	F
7							T	F
8								T

Largest is
BCCCB

The Algorithm

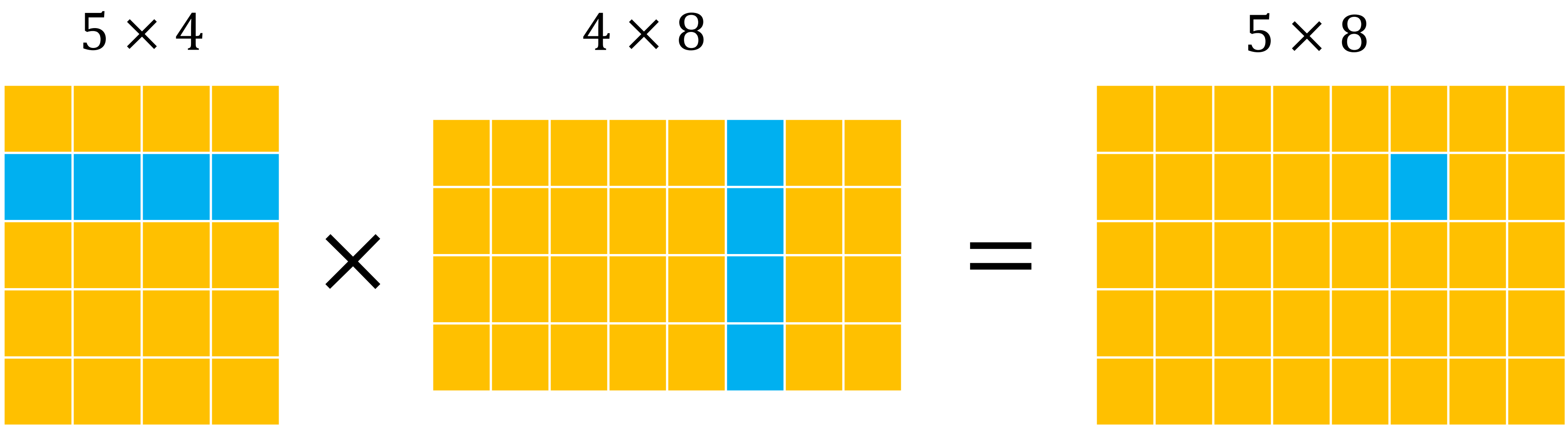
```
max ← 1
for i ← 1 to n − 1 do                          initial conditions
     p[i, i] ← true                                  length 1
     if x_i = x_{i+1} then                           length 2
          p[i, i + 1] ← true,  max ← 2
     else p[i, i + 1] ← false
for l ← 3 to n do                          for each length 3 to n
     for i ← 1 to n − l + 1 do             starting character
          j ← i + l − 1                     ending character
          if p[i + 1, j − 1] = true and x_i = x_j then
               p[i, j] ← true,  max ← l
          else p[i, j] ← false
return max
```

Running time: $O(n^2)$

Space: $O(n^2)$ but can be improved to $O(n)$

Interval DP: Matrix-Chain Multiplication

The product of two matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p \times q$ and $q \times r$) is a matrix $C_{p\times r}$. Generating $C_{p\times r}$ requires pqr scalar multiplications.



Given matrices A, B with entries $a_{i,j}, b_{i,j}$, the entries in the product matrix $C = A \times B$ are

$$c_{i,j} = \sum_{k=1}^{q} a_{i,k} b_{k,j}$$

Diagrams on this and the next few pages modified from
http://ramos.elo.utfsm.cl/~lsb/elo320/aplicaciones/aplicaciones/CS460AlgorithmsandComplexity/lecture17

Matrix-Chain Multiplication - 3 Matrices

The product of two matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p \times q$ and $q \times r$) is a matrix $C_{p\times r}$ with pr entries.
Calculating any one entry in $C_{p\times r}$ requires q scalar multiplications, so generating $C_{p\times r}$ requires pqr scalar multiplications (sm).

For three matrices (e.g., $A_{10\times 100}$, $B_{100\times 5}$ and $C_{5\times 50}$)
there are 2 ways to parenthesize:

$(A(BC)) = A_{10\times 100}\cdot E_{100\times 50}$

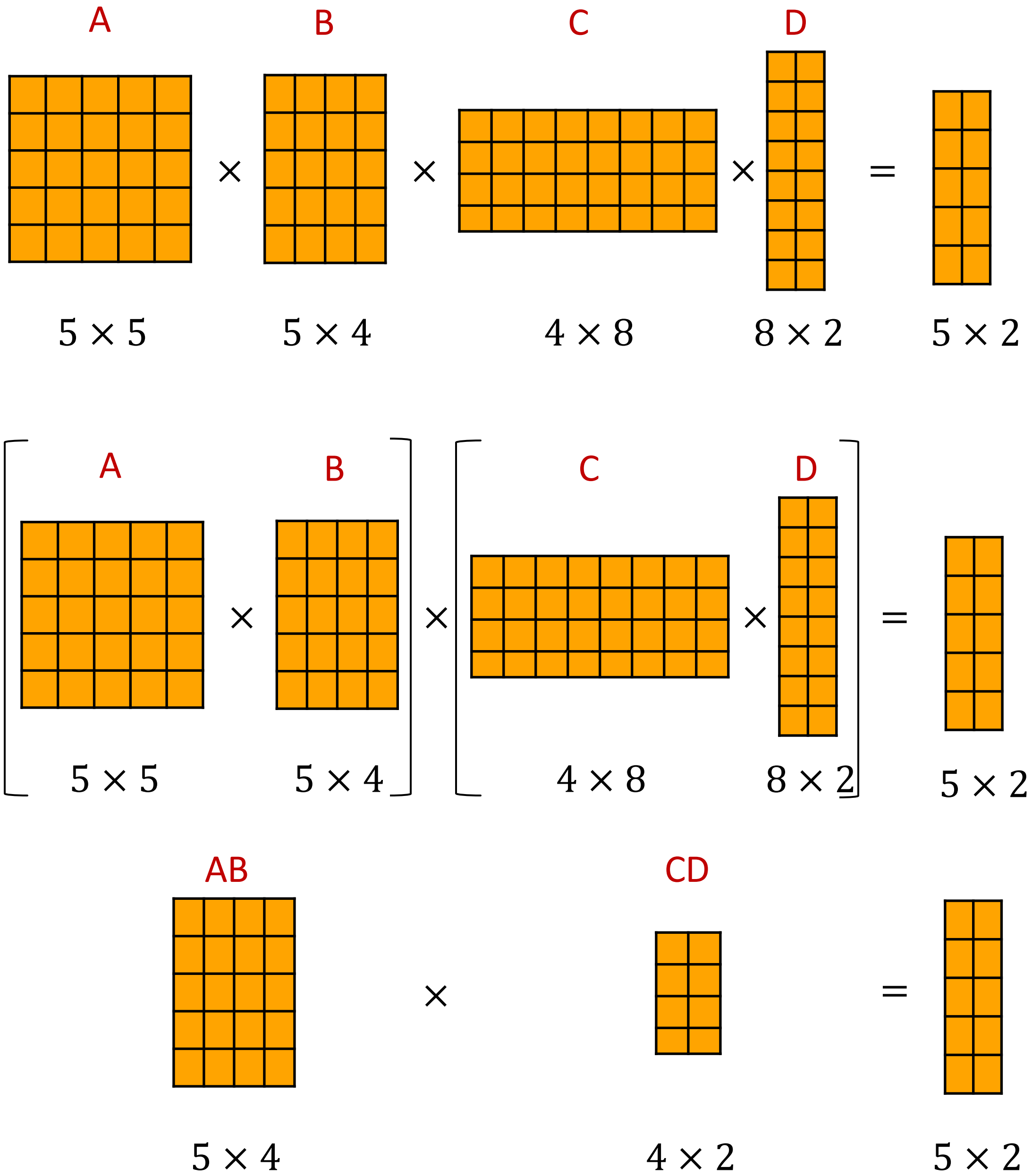
- $BC \Rightarrow 100\cdot 5\cdot 50 = 25{,}000$ sm
- $AE \Rightarrow 10\cdot 100\cdot 50 = 50{,}000$ sm
- Total $= 75{,}000$

$\big((AB)C\big) = D_{10\times 5}\cdot C_{5\times 50}$

- $AB \Rightarrow 10\cdot 100\cdot 5 = 5{,}000$ sm
- $DC \Rightarrow 10\cdot 5\cdot 50 = 2{,}500$ sm
- Total $= 7{,}500$

General problem: Given a sequence or chain $A_1, A_2, \ldots, A_n$, of n matrices,
determine the optimal way to parenthesize
(i.e., the solution with the minimum number of scalar multiplications).

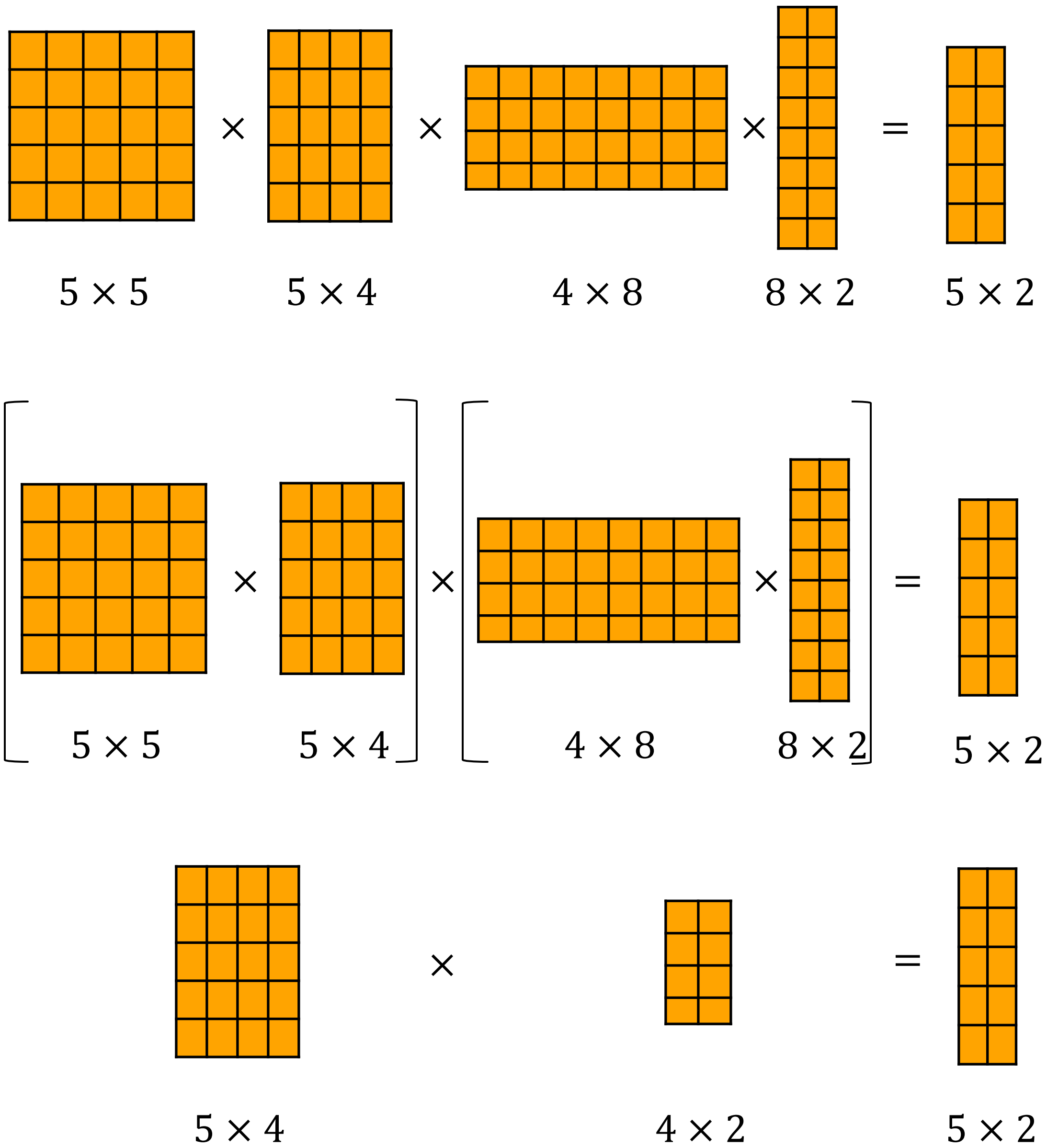
Matrix-Chain Multiplication - 4 Matrices



There are 5 different ways to multiply ABCD together

1. (A (B (CD)))
2. (A ((BC) D))
3. ((AB) (CD))
4. ((A (BC)) D)
5. (((AB) C) D)

Matrix-Chain Multiplication - 4 Matrices



There are 5 different ways to multiply ABCD together

1. (A (B (CD)))
2. (A ((BC) D))
3. ((AB) (CD))
4. ((A (BC)) D)
5. (((AB) C) D)

Costs are

1. 5(5)2 + 5(4)2 + 4(8)2 = 154
2. 5(5)2 + 5(4)8 + 5(8)2 = 290
3. 5(5)4 + 4(8)2 + 5(4)2 = 204
4. 5(5)8 + 5(4)8 + 5(8)2 = 440
5. 5(5)4 + 5(4)8 + 5(8)2 = 340

Recall: Multiplying p×q and q×r matrices requires p×q×r multiplications And yields a p×r matrix

Problem Definition

- Input: Values $p_0\, p_1 \cdots p_{n-1}\, p_n$
- These represent sizes of n matrices $A_1\, A_2 \cdots A_n$
  Matrix A_i has dimensions $p_{i-1} \times p_i$

- $A_{i \ldots j}$: matrix that is the product of $A_i\, A_{i+1} \cdots A_j$
  By construction $A_{i \ldots j}$ has dimensions $p_{i-1} \times p_j$

- Goal: To find a minimum cost way of multiplying $A_1 A_2 \cdots A_n$ to get the final result $A_{1 \ldots n}$.
  cost = # of total scalar multiplications performed

This is known as an optimal parenthesization of $A_1 A_2 \cdots A_n$ because the parentheses denote how to perform the multiplications e.g., ((AB) (CD)) means first calculate $X = AB$, then calculate $Y = CD$ and finally calculate XY.

Optimal Solution Structure

- Given: Values $p_0\, p_1 \cdots p_{n-1}\, p_n$ s.t. Matrix A_i has size $p_{i-1} \times p_i$
- $A_{i\cdots j}$: denotes matrix that results from the product $A_i A_{i+1} \cdots A_j$

- An (optimal) parenthesization of $A_1 A_2 \cdots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$.
  $A_{1\cdots n} = (A_1 A_2 \cdots A_k) \cdot (A_{k+1} A_{k+2} \cdots A_n) = A_{1\cdots k} \cdot A_{k+1\cdots n}$

- In the optimal parenthesization,
  1st : compute matrices $A_{1\cdots k}$ and $A_{k+1\cdots n}$;
  2nd : multiply $A_{1\cdots k}$ and $A_{k+1\cdots n}$ together to get final matrix $A_{1\cdots n}$

- **Observation**: If parenthesization of $A_1 A_2 \cdots A_n$ is optimal
  => parenthesizations of subchains $A_1 A_2 \cdots A_k$ and $A_{k+1} \cdots A_n$
  must also be optimal (why?)
  => The optimal solution to the problem contains within it the optimal solution to subproblems

Recurrence

- $m[i,j]$ =minimum number of scalar multiplications necessary to compute $A_{i\ldots j}$

- Suppose the optimal parenthesization of $A_{i\ldots j}$ splits product between A_k and A_{k+1}, for some integer k, $\mathrm{i} \leq k < \mathrm{j}$
  - $A_{i\ldots j}$ = $(A_i A_{i+1} \cdots A_k) \cdot (A_{k+1} A_{k+2} \cdots A_j) = A_{i\ldots k} \cdot A_{k+1\ldots j}$
  - min cost of computing $A_{i\ldots j}$ = min cost of computing $A_{i\ldots k}$ + min cost of computing $A_{k+1\ldots j}$ + cost of multiplying $A_{i\ldots k}$ and $A_{k+1\ldots j}$
  - Cost of multiplying $A_{i\ldots k}$ and $A_{k+1\ldots j}$ is $p_{i-1} p_k p_j$

- But… optimal parenthesization occurs at *some* value of *k*. Check all possible values of *k* and select the best one.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

DP Algorithm

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s
(s records values of k at which minima occurred)

MATRIX-CHAIN-ORDER($p[], n$)
for $i = 1$ **to** n
    $m[i,i] = 0$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$ **then**
                $m[i,j] = q$
                $s[i,j] = k$
return $m[\,]$ and $s[\,]$

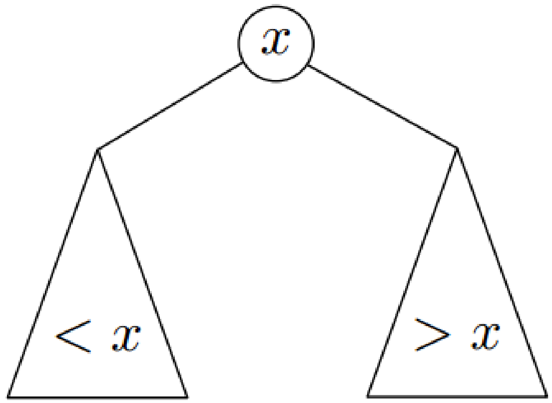
Initial Conditions

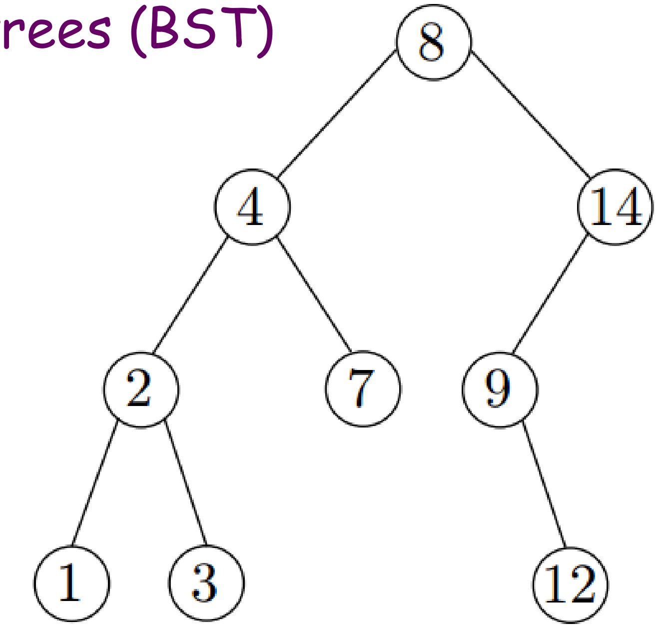
Recurrence Relation

Location of Minimum

Time is $O(n^3)$, space is $O(n^2)$

Binary search trees (BST)





Tree-Search(T, k):
$x \leftarrow T.root$
while $x \neq nil$ **and** $k \neq x.key$ **do**
  if $k < x.key$ **then** $x \leftarrow x.left$
  else $x \leftarrow x.right$
return x

The (worst-case) search time in a balanced BST is $\Theta(\log n)$

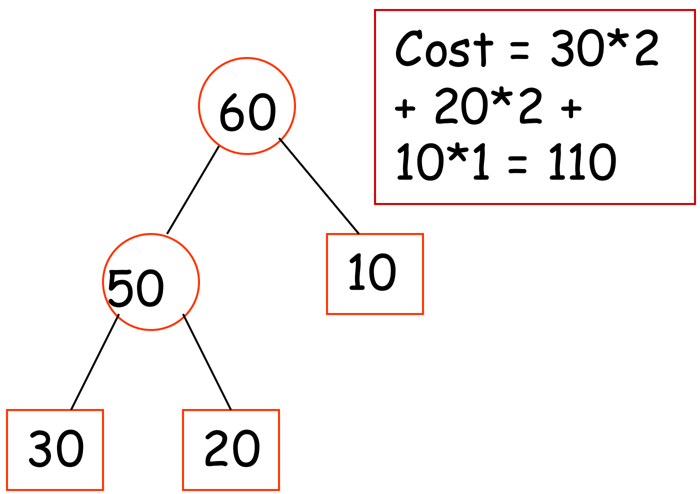
Q: If we know the probability of each key being searched for, can we design a (possibly unbalanced) BST to optimize the expected search time?

Similar Problem seen in Huffman Codes: Binary Merge Tree

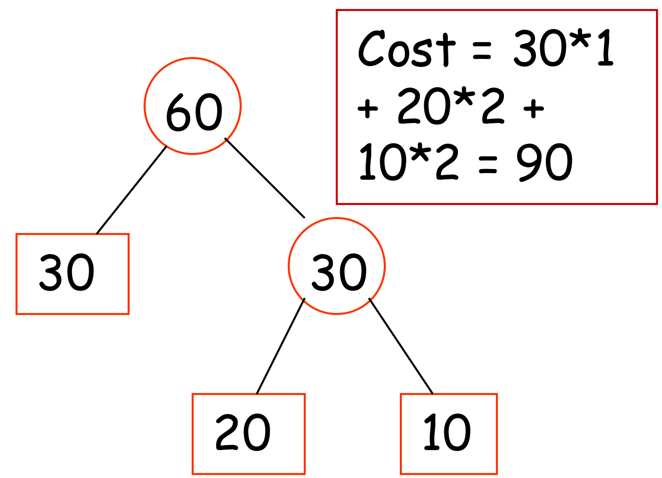
You are given a set of leaf nodes $a_1, \ldots, a_n$ and associated leaf weights $w(a_1), \ldots, w(a_n)$

Create a binary tree from the leaf nodes towards the root, in which the size of each node is the sum of the sizes of the two children.

A binary merge tree is **optimal** if it minimizes the weighted external path length. The *weighted external path length* of the tree is $B(T) = \sum_{i=1}^{n} w(a_i) d(a_i)$



Merge L_1 and L_2, then with L_3



Merge L_2 and L_3, then with L_1

Optimal Binary Merge Tree Greedy Algorithm - Same as Huffman Coding

Input: $n \geq 2$ leaf nodes, each with a size (i.e., # list elements) .

Output: a binary tree with the given leaf nodes which has a minimum total weighted external path lengths

Algorithm:

Create a min-heap $T[1..n]$ based on the n initial sizes.

While (the heap size $\geq$ 2) do

- extract from the heap two smallest values a and b
- create intermediate node of size $a+b$ whose children are a and b
- insert the value $a+b$ into the heap

Time complexity $= O(n \log n)$

It can be shown that the Binary Merge Tree is optimal

The Optimal Binary Search Tree Problem

Problem Definition (simpler than the version in textbook):

Given n keys $a_1 < a_2 < \cdots < a_n$, with weights $f(a_1), \ldots, f(a_n)$, find a binary search tree T on these n keys such that

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

is minimized, where $d(a_i)$ is the depth of a_i.

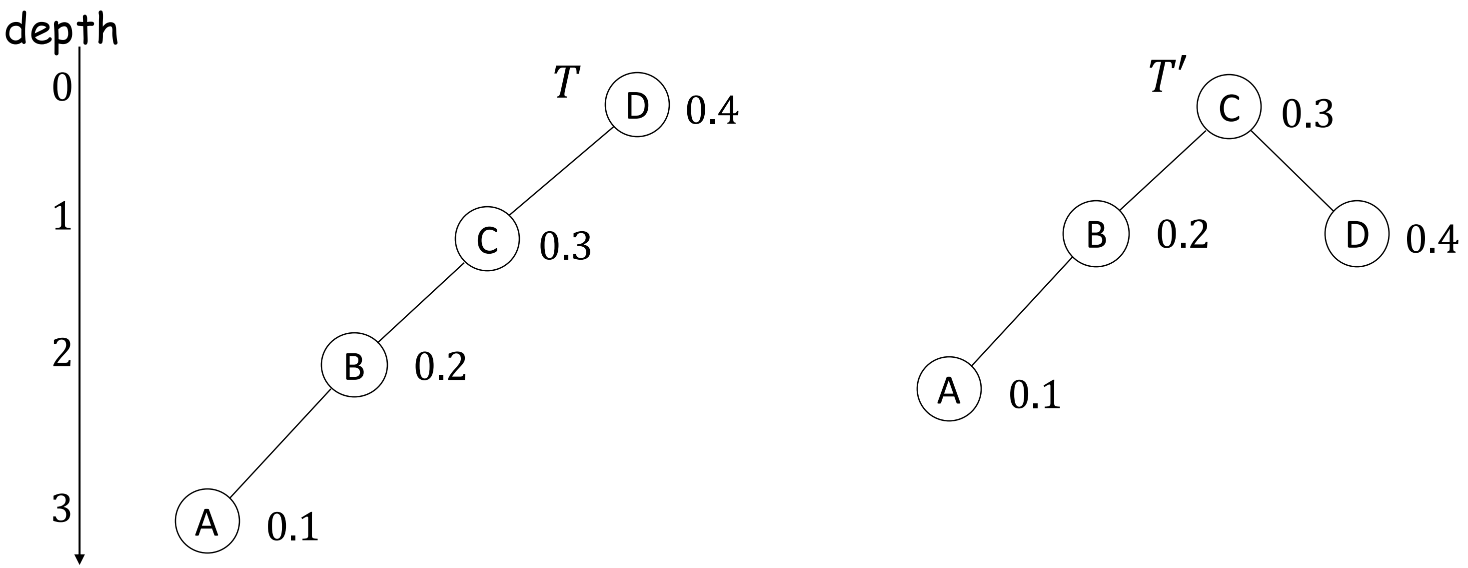
Note: Similar to the Binary Merge Tree problem but with 2 key differences:

- The tree has to be a BST, i.e., the keys are stored in sorted order. In a Binary Merge Tree, there is no ordering among the leaves.
- Keys appear as both internal and leaf nodes. In a Binary Merge Tree, keys (characters) appear only at the leaf nodes.

Motivation: If the weights are the probabilities of the elements being searched for, such a BST will minimize the expected search cost.

Greedy Won't Work

Cannot apply Huffman algorithm because it assumes that all keys must be at leaves. Alternative greedy algorithm: Always pick the heaviest key as root, then recursively build the tree top-down.



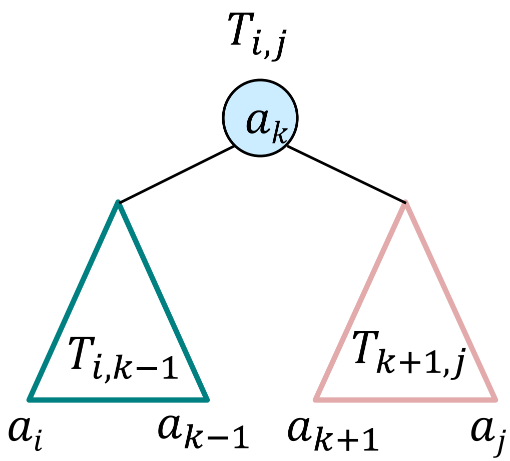
T was built using greedy strategy and has cost $B(T) = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 4 = 2$

T' has a smaller cost $B(T') = 0.4 \cdot 2 + 0.3 \cdot 1 + 0.2 \cdot 2 + 0.1 \cdot 3 = 1.8$

Dynamic Programming: The Recurrence

Let $T_{i,j}$ be some tree on the subset of nodes $a_i < a_{i+1} < \cdots < a_j$.
The cost is well defined as $B(T_{i,j}) = \sum_{t=i}^{j} f(a_t)(d(a_t) + 1)$

Let $w[i,j] = f(a_i) + \cdots + f(a_j)$



Suppose we ***knew*** root of $T_{i,j}$ was a_k.

$T_{i,j}$ is a BST, so left and right sub-tree children of a_k
are some tree $T_{i,k-1}$ on $a_i < \cdots < a_{k-1}$
and some tree $T_{k+1,j}$ on $a_{k+1} < \cdots < a_j$

Nodes in $T_{i,k-1}$ and $T_{k+1,j}$ are one level deeper in $T_{i,j}$ than in their original trees. So the cost of $T_{i,j}$ *is*

$$\begin{aligned}
B(T_{i,j}) &= \left(B(T_{i,k-1}) + w[i,k-1]\right) + f(a_k) + \left(B(T_{k+1,j}) + w[k+1,j]\right) \\
&= B(T_{i,k-1}) + B(T_{k+1,j}) + w[i,k-1] + f(a_k) + w[k+1,j] \\
&= B(T_{i,k-1}) + B(T_{k+1,j}) + w[i,j]
\end{aligned}$$

Dynamic Programming: The Recurrence

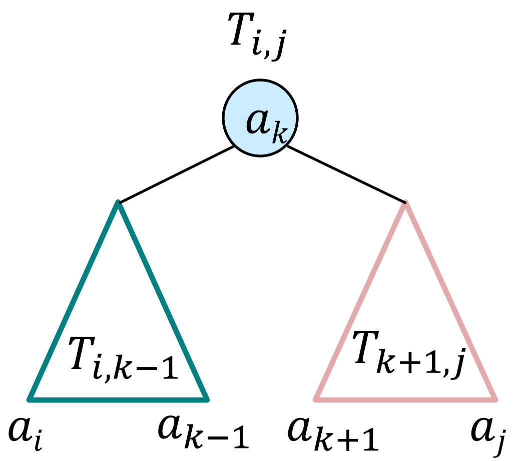
Let $T_{i,j}$ be some tree on the subset of nodes $a_i < a_{i+1} < \cdots < a_j$.
The cost is well defined as $B(T_{i,j}) = \sum_{t=i}^{j} f(a_t)(d(a_t)+1)$

Let $w[i,j] = f(a_i) + \cdots + f(a_j)$

Suppose we ***knew*** root of $T_{i,j}$ was a_k.

The cost of $T_{i,j}$ is

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



In particular, suppose $T_{i,j}$ has root a_k and is a minimum cost tree over all trees with nodes $a_i, \ldots, a_j$
⇒ its left subtree $T_{i,k-1}$ must be a minimum cost tree with nodes $a_i, \ldots, a_{k-1}$

If it wasn't, we could replace $T_{i,k-1}$ by a lesser cost subtree with nodes $a_i, \ldots, a_{k-1}$.
By (*), this would reduce $B(T_{i,j})$, contradicting that $T_{i,j}$ is a minimum cost tree.

Similarly, the right subtree $T_{k+1,j}$ must be minimum cost for $a_{k+1}, \ldots, a_j$

Dynamic Programming: The Recurrence

Def: $e[i,j]$ = the minimum cost of any BST on $a_i, \ldots, a_j$

Idea: The root of the BST can be any of $a_i, \ldots, a_j$. We try each of them.

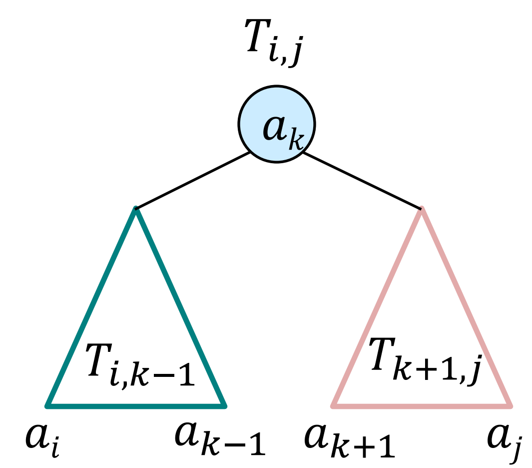
Recurrence:

Let $w[i,j] = f(a_i) + \cdots + f(a_j)$

Suppose we knew min-cost BST $T_{i,j}$ for $[i,j]$ and that its root was a_k.

Its left subtree $T_{i,k-1}$ must be optimal for $[i,k-1]$
and its right subtree $T_{k+1,j}$ must be optimal for $[k+1,j]$

$$
\begin{aligned}
\Rightarrow \quad e[i,j] &= B(T_{i,j}) \\
&= B(T_{i,k-1}) + B(T_{k+1,j}) + w[i,j] \\
&= e[i,k-1] + e[k+1,j] + w[i,j]
\end{aligned}
$$



To find $T_{i,j}$ we can try out every possible value of k and return the one which minimizes tree cost!

Dynamic Programming: The Recurrence

Def: $e[i,j]$ = the minimum cost of any BST on $a_i, \dots, a_j$

Idea: The root of the BST can be any of $a_i, \dots, a_j$. We try each of them.

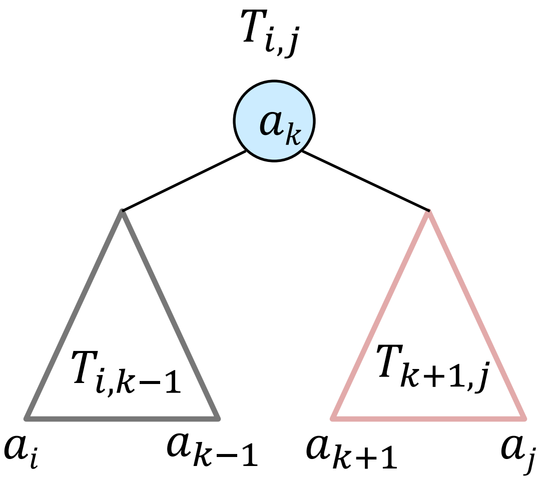
Recurrence:

Let $w[i,j] = f(a_i) + \cdots + f(a_j)$

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

$e[i,j] = 0$ for $i > j$.

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



Note: All $w[i,j]$'s can be pre-computed in $O(n^2)$ time.

The Algorithm

Idea: We will do the bottom-up computation by the increasing order of the problem size.

```
let e[1..n, 1..n], w[1..n, 1..n], root[1..n, 1..n] be new arrays of all 0
for i = 1 to n                                          computation of w[i, j]
    w[i, i] ← f(a_i)
    for j = i + 1 to n
        w[i, j] ← w[i, j − 1] + f(a_j)
for l ← 1 to n                                          length of [i,j]
    for i ← 1 to n − l + 1                              First node
        j ← i + l − 1                                   Last node
        e[i, j] ← ∞
        for k ← i to j                                  Root k that
            t ← e[i, k − 1] + e[k + 1, j] + w[i, j]     minimizes
            if t < e[i, j] then                         e[i, k − 1] + e[k + 1, j] + w[i, j]
                e[i, j] ← t
                root[i, j] ← k
return Construct-BST(root, 1, n)
```

Running time: $O(n^3)$

Space: $O(n^2)$

Construct the Optimal BST

Construct-BST($root, i, j$):
if $i > j$ **then return** nil
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($root, root[i,j] + 1, j$)
return z

Running time of this part: $O(n)$

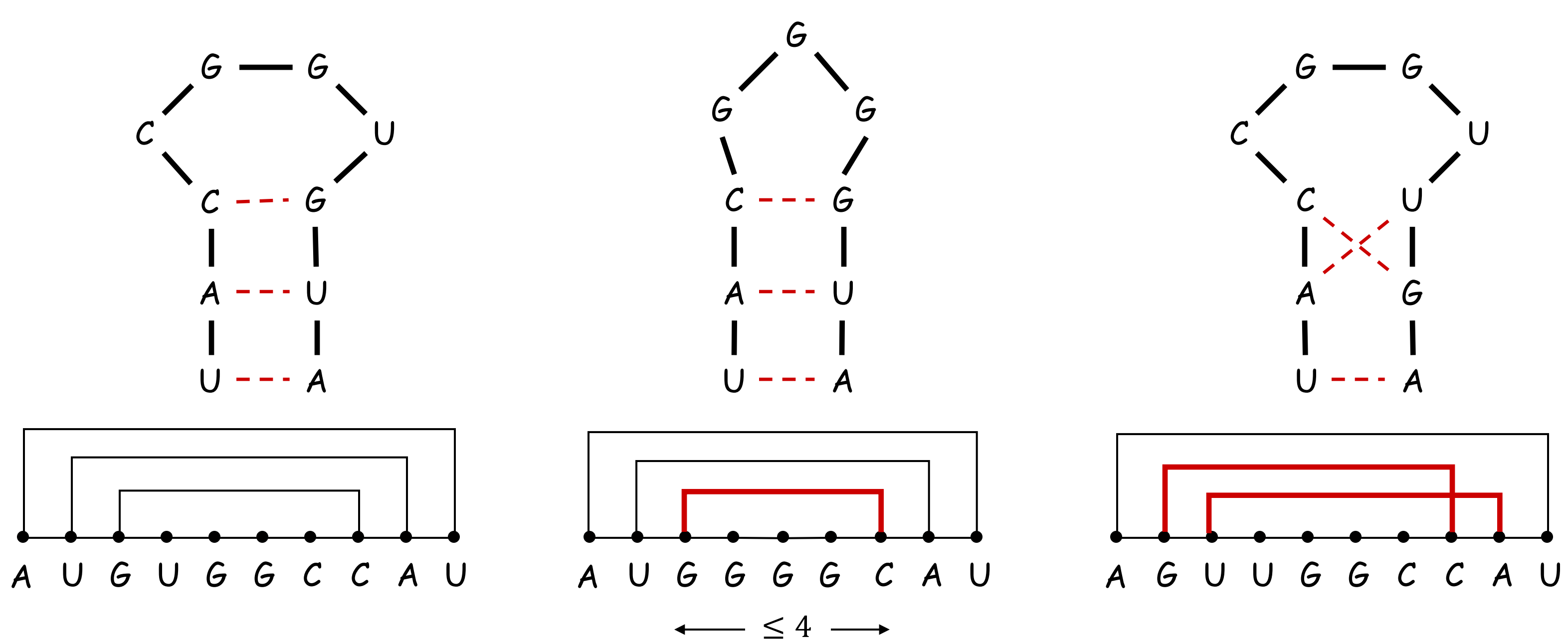
RNA Secondary Structure

Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.

- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases: If $(b_i, b_j) \in S$, then $i < j - 4$.

- [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.



ok sharp turn crossing

The Problem

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy, which is proportional to the number of base pairs.

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

That is, find the maximum number of base pairs that can be matched satisfying the no sharp turns and no crossing constraints

The Recurrence

Def. $M[i,j]$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Recurrence.

- Case 1. If $i \geq j - 4$.
  - $M[i,j] = 0$ by no-sharp turns condition.

- Case 2. $i < j - 4$
  - Case 2a: Base b_j is not matched in optimal solution for $[i,j]$
    - $M[i,j] = M[i,j-1]$
  - Case 2b: Base b_j pairs with b_k for some $i \leq k \leq j - 5$.
    - Try matching b_j to all possible b_k.
    - non-crossing constraint decouples problem into sub-problems
    - $M[i,j] = 1 + \max_k \{M[i,k-1] + M[k+1,j-1]\}$

Take max over k such that $i \leq k \leq j - 5$ and b_k and b_j are Watson-Crick complements

The Algorithm

```
let M[1..n, 1..n], s[1..n, 1..n] be new arrays of all 0
for l ← 1 to n
    for i ← 1 to n − l + 1
        j ← i + l − 1
        M[i, j] ← M[i, j − 1]
        for k ← i to j − 5
            if b_k and b_j are not complements then continue
            t ← 1 + M[i, k − 1] + M[k + 1, j − 1]
            if t > M[i, j] then
                M[i, j] ← t
                s[i, j] ← k
Construct-RNA(s, 1, n)
```

Running time: $O(n^3)$

Space: $O(n^2)$

```
Construct-RNA(s, i, j):
if i ≥ j − 4 then return
if s[i, j] = 0 then Construct-RNA(s, i, j − 1)
print s[i, j], "-", j
Construct-RNA(s, i, s[i, j] − 1)
Construct-RNA(s, s[i, j] + 1, j − 1)
```