CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DYNAMIC PROGRAMMING – PART II

Instructor: Abdou Youssef

WHAT YOU LEARNED LAST LECTURE A/B DP

Last lecture, you learned:

- How Dynamic Programming (DP) works, including its major design steps
- The *principle of optimality,* and how to state in a given problem
- How to prove the principle of optimality (by contradiction)
- How to apply DP to solve the Matrix Chain Problem, especially the derivation of a recurrence relation, and computing it in a table
- How to use the optimal splits recorded in the table to construct the actual optimal solution

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Apply Dynamic Programming (DP) more firmly
- Derive a well-known DP algorithm for the all-powerful *all-pairs shortest-paths problem*
- Formulate an optimal binary search tree approach to a practical problem of searching for items of different access-frequencies
- Develop a DP algorithm for the *Optimal Binary Search Tree problem*
- Conduct time complexity analysis on DP algorithms

OUTLINE

- Second application of DP: the all-pairs shortest-paths problem
- Formulation of searching for items of different access-frequencies, as an optimal-BST problem
- Third application of DP: the OBST problem

THE ALL-PAIRS SHORTEST PATH PROBLEM

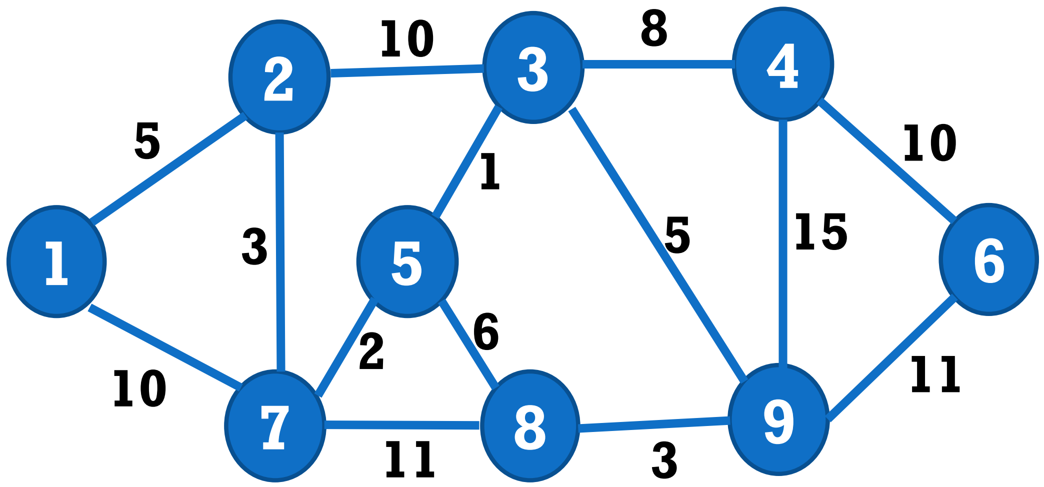
Problem statement:

- **Input:** A weighted graph G, represented by its weight matrix $W[1:n, 1:n]$, where n = number of nodes in G.
  - $W[i,i] = 0\ \forall i$,
  - $W[i,j] = \infty$ if there is no real edge between node i and node j;
  - Otherwise, $W[i,j]$ is a real number.
- **Output**: A distance matrix $A[1:n, 1:n]$, where $A[i,j]$ is the distance from node i to node j
- **Task**: Develop a DP algorithm for solving this problem

APPLYING DP TO THE APSP PROBLEM
-- (1) NOTATION (1/2) --

- **Definition**: A k-special path from node i to node j is a path from i to j where the label of every intermediary node is $\leq k$

- 2-special paths from 1 to 7
  - 1, 2, 7: Y/N? Yes
  - 1, 7: Y/N? Yes
  - 1, 2, 3, 5, 7: Y/N? NO. Why?



- 2-special path from 3 to 7:
  - 3, 7: Y/N? Yes, of length=∞
  - 3, 2, 7: Y/N? Yes, of length=13
- 5-special paths from 3 to 7: [3, 7]; [3, 2, 7]; [3, 5, 7]; [3, 2, 1, 7]

APPLYING DP TO THE APSP PROBLEM
-- (1) NOTATION (2/2) --

- For all nodes i and j, and all positive integers $k = 1, 2, \dots, n$, let
  - $A^{(k)}[i, j]$ = length of the shortest k-special path from node i to node j
  - Initial values: $A^{(0)}[i, j] = ??$
    - Note that any 0-special path from node i to node j is the single edge (i, j) because when k=0, no intermediate node can have a label ≤ 0 since all the labels are ≥ 1
    - Therefore, $A^{(0)}[i, j] = W[i, j]$
  - For what values of k is $A^{(k)}[i, j]$ the distance from i to j?
    - $k = n$. Why?

APPLYING DP TO THE APSP PROBLEM
-- (2) PRINCIPLE OF OPTIMALITY--

- We already saw last lecture that any sub-path of a shortest path is a shortest path between its end nodes

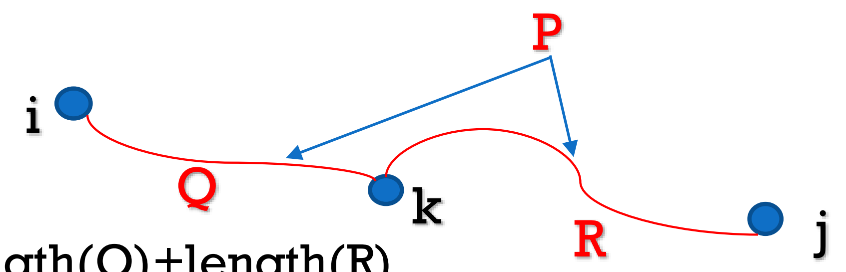
APPLYING DP TO THE APSP PROBLEM
-- (3) RECURRENCE RELATION --

- Divide the k-special paths from node i to node j into two groups:
  - Group A: Those k-special paths that do <u>not</u> go through node k
  - Group B: Those k-special paths that go through node k
- $A^{(k)}[i,j]$ =min(length of the k-special paths from node i to node j)
- $A^{(k)}[i,j]$ =min(min(length of k-special paths in Group A),

  min(length of k-special paths in Group B))
- Group A: if a k-special path doesn't go through node k, then every intermediary node on that path is $\leq k-1$, and thus the path is a $(k-1)$-special path
- Therefore: min(length of k-special paths in Group A) = min(length of $(k-1)$-special paths from node i to node j)= $A^{(k-1)}[i,j]$

APPLYING DP TO THE APSP PROBLEM
-- (3) RECURRENCE RELATION (CONTINUED)--

- Group B: k-special paths from node i to node j that go through node k
- The shortest k-special path P in Group B goes from node i to node k, then from node k to node j.
  - Call the first portion path Q
  - Call the second portion path R



  - min(length of k-special paths in Group B) = length(P) = length(Q)+length(R)
  - No intermediary node of Q can be node k because otherwise path P goes through node k multiple times, and that makes P having cycles and thus not the shortest in its group, contradicting the choice of P as the shortest in Group B
  - Therefore, every intermediary node of Q is $\leq k-1$
    - Thus, Q is a $(k-1)$-special path from node i to node k
  - $\therefore$ Q must be the <u>shortest</u> $(k-1)$-special path from node i to node k (by POO), and can be so proved by contradiction
  - $\therefore$ length(Q)=$A^{(k-1)}[i,k]$
  - We can prove similarly that: length(R)=$A^{(k-1)}[k,j]$
  - min(length of k-special paths in Group B) = length(Q)+length(R) = $A^{(k-1)}[i,k] + A^{(k-1)}[k,j]$

APPLYING DP TO THE APSP PROBLEM
-- (3) RECURRENCE RELATION (CONTINUED)--

- Recap:
  - min(length of k-special paths in Group A) = $A^{(k-1)}[i,j]$
  - min(length of k-special paths in Group B) = $A^{(k-1)}[i,k] + A^{(k-1)}[k,j]$
  - $A^{(k)}[i,j]$ =min(min(length of k-special paths in Group A),

    min(length of k-special paths in Group B))

    $=\min(A^{(k-1)}[i,j], A^{(k-1)}[i,k] + A^{(k-1)}[k,j])$

- So, the recurrence relation is:
  - $\boldsymbol{A^{(k)}[i,j] = \min(A^{(k-1)}[i,j], A^{(k-1)}[i,k] + A^{(k-1)}[k,j])}$**,** **where the recurrence index is $\boldsymbol{k}$**
  - $\boldsymbol{A^{(0)}[i,j] = W[i,j]}$

THE DP APSP ALGORITHM
-- ALSO KNOWN AS FLOYD-WARSHALL ALGORITHM --

```
Procedure APSP(input: W[1:n,1:n]; output: A[1:n,1:n])
begin
    for i=1 to n do
        for j=1 to n do
            A^(0)[i,j] = W[i,j];
        endfor
    endfor
    for k=1 to n do
        for i=1 to n do
            for j=1 to n do
                A^(k)[i,j]= min(A^(k-1)[i,j], A^(k-1)[i,k] + A^(k-1)[k,j]);
            endfor
        endfor
    endfor
end APSP
```

$A^{(0)}[i,j] = W[i,j]$

$A^{(k)}[i,j] = \min(A^{(k-1)}[i,j], A^{(k-1)}[i,k] + A^{(k-1)}[k,j])$

THE DP APSP ALGORITHM
-- SPACE COMPLEXITY --

Procedure APSP(**input**: W[1:n,1:n]; **output**: A[1:n,1:n])
begin
    for i=1 to n **do**
        for j=1 to n **do**
            $A^{(0)}[i,j] = W[i,j]$;
        endfor
    endfor
    for k=1 to n **do**
        for i=1 to n **do**
            for j=1 to n **do**
                $A^{(k)}[i,j]$= **min**($A^{(k-1)}[i,j]$, $A^{(k-1)}[i,k] + A^{(k-1)}[k,j]$);
            endfor
        endfor
    endfor
end APSP

Space Complexity Analysis:

- $A^{(k)}[i,j]$ must be written as $A[i,j,k]$, and so A must be an array $\mathrm{A}[1\!:\mathrm{n}, 1\!:\mathrm{n}, 1\!:\mathrm{n}]$
- This takes $O(n^3)$ memory.
- That is too costly!!
- Can we do better?

THE DP APSP ALGORITHM
-- REDUCED SPACE COMPLEXITY (1) --

- Note that once the matrix $\boldsymbol{A}^{(\boldsymbol{k})}$ has been computed, there is no need for matrix $\boldsymbol{A}^{(\boldsymbol{k}-\mathbf{1})}$
- Therefore, we don't need to keep the superscript
- By dropping it, the algorithm remains correct, and we save on space
- The algorithm becomes as presented next

THE DP APSP ALGORITHM
-- REDUCED SPACE COMPLEXITY (2) --

Procedure APSP(**input**: W[1:n,1:n]; **output**: A[1:n,1:n])
begin
    for i=1 to n **do**
        for j=1 to n **do**
            $\boldsymbol{A[i,j] = W[i,j]}$;
        endfor
    endfor
    for k=1 to n **do**
        for i=1 to n **do**
            for j=1 to n **do**
                $\boldsymbol{A[i,j] = \min(A[i,j],\ A[i,k] + A[k,j])}$;
            endfor
        endfor
    endfor
end APSP

if$(\boldsymbol{A[i,k] + A[k,j] < A[i,j]})$ **then**
$\boldsymbol{A[i,j] = A[i,k] + A[k,j]}$;

THE DP APSP ALGORITHM
-- TIME & SPACE COMPLEXITY --

```
Procedure APSP(input: W[1:n,1:n]; output: A[1:n,1:n])
begin
    for i=1 to n do
        for j=1 to n do
            A[i,j] = W[i,j];
        endfor
    endfor

    for k=1 to n do
        for i=1 to n do
            for j=1 to n do
                A[i,j]= min(A[i,j], A[i,k] + A[k,j]);
            endfor
        endfor
    endfor
end APSP
```

Time: $O(n^2)$

Time: $O(n^3)$

- Overall Time: $O(n^3)$
- Overall Space: $O(n^2)$ because we only need array A[1:n,1:n]

THE DP APSP ALGORITHM
-- EXERCISES--

- **Exercise 1**: Show that dropping the superscript k does not affect the correctness of the algorithm
- **Exercise 2**: Modify the ASAP algorithm so it computes the actual shortest paths, rather than just the distances
- **Exercise 3**:
  a) Could we have used the greedy single-source shortest-paths algorithm to compute the all-pairs shortest paths?
  b) If so, how?
  c) What would be the time complexity of that new algorithm?

3^{RD} APPLICATION OF DYNAMIC PROGRAMMING
-- OPTIMAL BINARY SEARCH TREES --

Problem Statement:

- **Input**:
  - A sorted array $a_1 < a_2 < \cdots < a_n$
  - Access probabilities $p_1, p_2, \ldots, p_n$, where $p_i = \Pr[a_i \text{ is searched for}]$
  - Probabilities $q_0, q_1, q_2, \ldots, q_n$ of accessing missing elements, where $q_0 = \Pr[\text{searching for } X, X < a_1]$, $q_i = \Pr[\text{searching for } X, a_i < X < a_{i+1}]$, and $q_n = \Pr[\text{searching for } X, X > a_n]$
- **Output**: A binary search tree (BST) for $a_1, a_2, \ldots, a_n$ such that the search time is minimized (depending on the probability of access). Assume that no insert or delete operations will be performed.
- **Task**: Develop a Dynamic Programming algorithm for this problem

OPTIMAL BINARY SEARCH TREES
-- AN EXAMPLE --

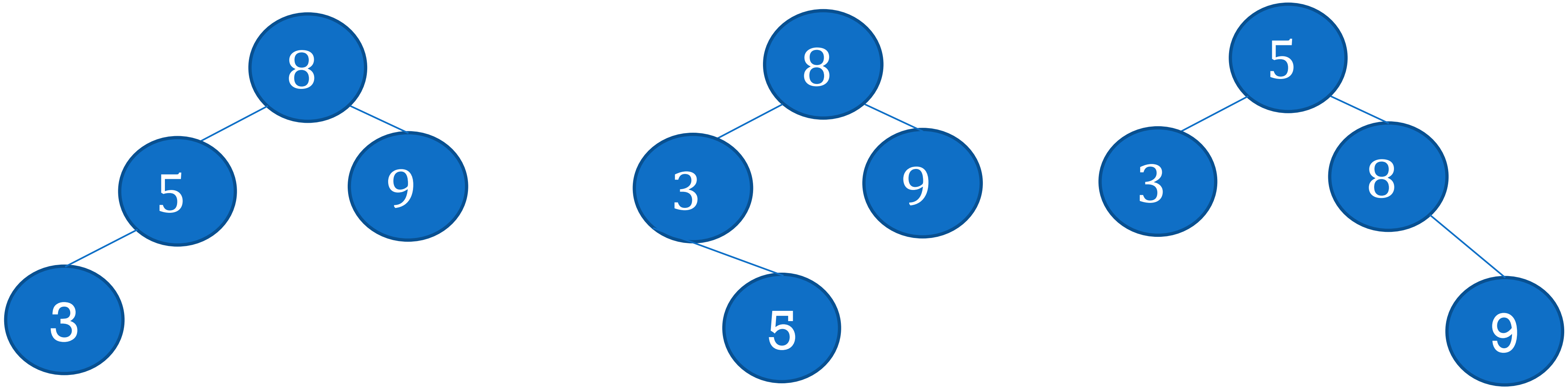
- An example to get a better sense
  - Suppose the $a_1, a_2, \ldots, a_n$ are phone numbers in a telephone directory
  - Some numbers belong to celebrities => are searched for more often
  - Some numbers belong to less well-known people, and so they're searched for less frequently
  - The phone company needs a **data structure for the phone** numbers so that
    - "hot" numbers are found fast (e.g., located close to the top of the BST),
    - while the less-frequently accessed numbers can take longer to find (i.e., can be placed in the middle or bottom of the BST)
  - The directory is fairly stable: few inserts/deletes, but many searches
  - Several data structures can be considered, by BSTs are good candidates
  - Question: Why can't we put all the numbers near the top so all are fast to find?

OPTIMAL BINARY SEARCH TREES
-- EXAMPLE OF HOW TO GET THE PROBABILITIES --

- The phone company has a log of the number of times each phone number (real or missing) was searched for in recent months/years
- Say that the company received
  - N searched requests during that period
  - n_i of those requests were for number a_i
  - m_i of those requests were for missing numbers that fall between a_i and a_{i+1}.
- Then, $p_i = \frac{n_i}{N}$ and $q_i = \frac{m_i}{N}$ for all i

OPTIMAL BINARY SEARCH TREES
-- MANY SOLUTIONS EXIST--

- For the same input array $a_1 < a_2 < \cdots < a_n$, there can be many BSTs
- Example: for array 3<5<8<9, we can have



and many more

- Which is best?

OPTIMAL BINARY SEARCH TREES
-- CONCRETIZING THE COST OF A CANDIDATE BST (1) --

- The problem is asking for a best BST, with a vague notion of "best": minimizing the search time
- But the search time in a single BST varies: it takes O(d) where d is the depth (of the search element) in the tree, and that varies from node to node
- For optimization to work, we need to have a single cost value per solution (i.e., per candidate BST)
- And the cost should be smaller for BSTs where the "hot" elements can be found fast (i.e., close to the root)

OPTIMAL BINARY SEARCH TREES

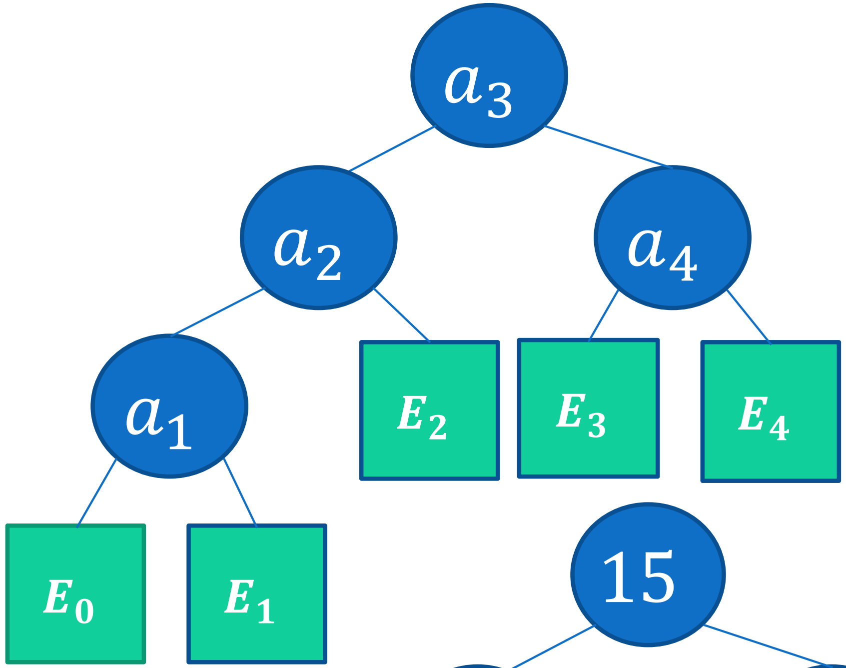
-- CONCRETIZING THE COST OF A CANDIDATE BST (2) --

- Strategy:
  - take the cost of a BST = the average search time in the tree
  - Since not all the nodes are equally important, take a weighted average
  - The weights are the access probabilities
- First attempt at formulating the cost C(T) of a BST T:
  - $C(T) = Avg(Search\ time\ in\ T) = \sum_{i=1}^{n} p_i \times search\_time_T(a_i)$
  - $C(T) = \sum_{i=1}^{n} p_i(depth_T(a_i) + 1)$
- But that ignores the searches for missing items

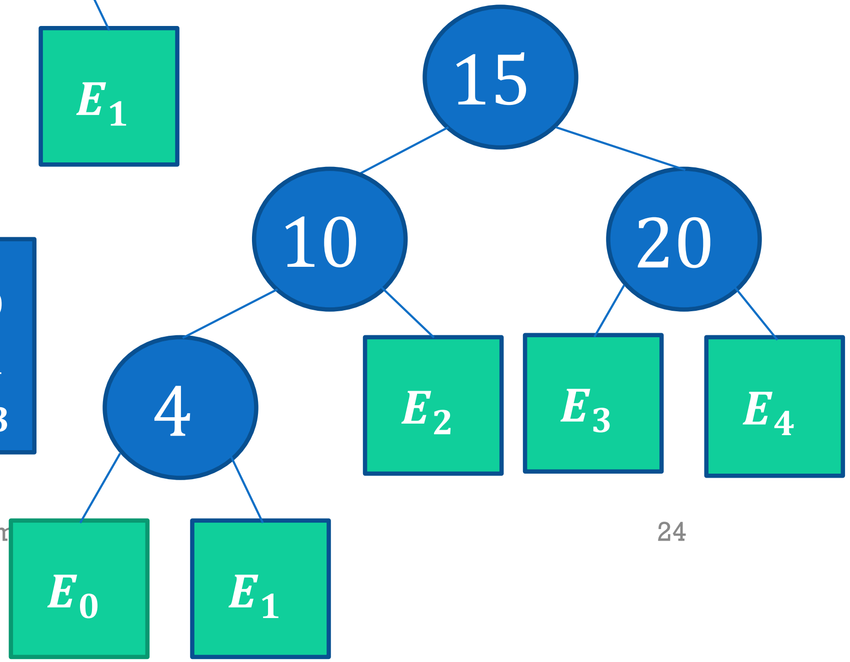
OPTIMAL BINARY SEARCH TREES

-- CONCRETIZING THE COST OF A CANDIDATE BST (3) --

- Some notation to help with access to missing numbers:
  - For every "missing leaf", create a new imaginary node, called an *external node* (in green squares)
  - Labels the external nodes $E_0, E_1, \ldots, E_n$ from left to right
  - In an n-node BST, there are n+1 external nodes
  - If $a_i < X < a_{i+1}$, search(X) takes us to external node E_i
  - Search_time(E_i)=depth$_T(E_i)$, and E_i is "search for" with a prob.ability q_i



Search(3) -> E_0
Search(7) -> E_1
Search(17)-> E_3



OPTIMAL BINARY SEARCH TREES
-- CONCRETIZING THE COST OF A CANDIDATE BST (4) --

- Final formulation of the cost C(T) of a BST T:

  - $C(T) = \sum_{s=1}^{n} p_s \times search\_time_T(a_s) + \sum_{s=0}^{n} q_s \times search\_time_T(E_s)$

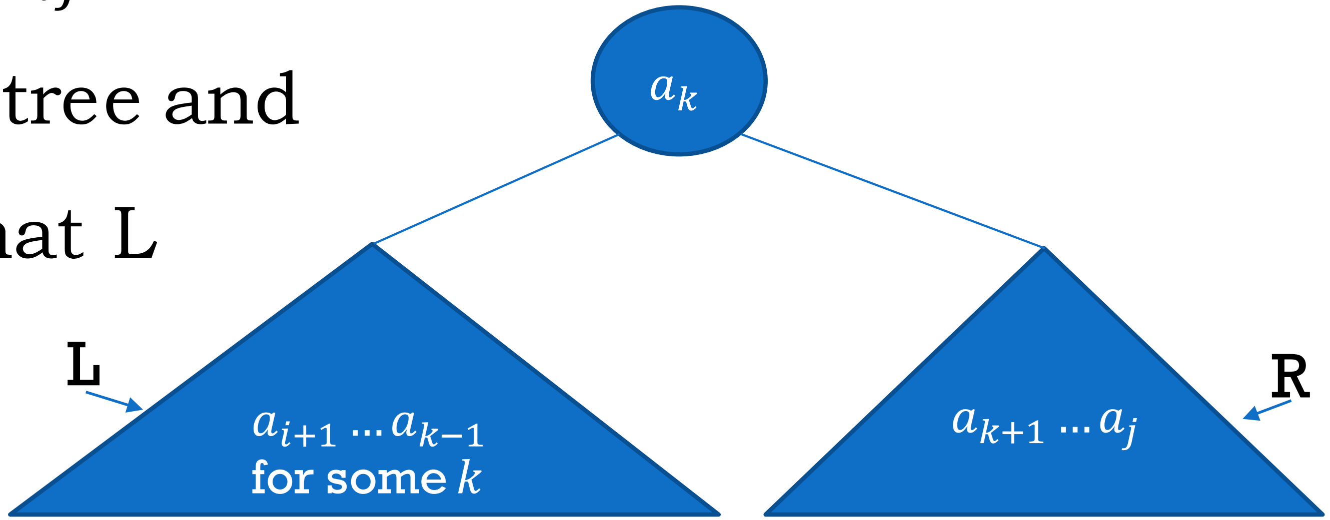
  - $\boldsymbol{C(T) = \sum_{s=1}^{n} p_s(depth_T(a_s) + 1) + \sum_{s=0}^{n} q_s depth_T(E_s)}$

OBST USING DYNAMIC PROGRAMMING -- (1) NOTATION --

- Let
  - $T_{ij} \stackrel{\text{def}}{=} OBST(a_{i+1}, \dots, a_j)$, that is, T_{ij} is the min-cost BST for elements $a_{i+1}, \dots, a_j$
  - $C_{ij} \stackrel{\text{def}}{=} C(T_{ij}) = \sum_{s=i+1}^{j} p_s(depth_{T_{ij}}(a_s) + 1) + \sum_{s=i}^{j} q_s depth_{T_{ij}}(E_s)$
  - $r_{ij} \stackrel{\text{def}}{=}$ the index of the root of T_{ij}, i.e., $a_{r_{ij}}$ is the root of T_{ij}
- Notice that
  - The final OBST (for the whole array) is T_{0n}
  - T_{ii} is empty for all i
  - $T_{i,i+1}$ is a single-node tree that has element a_{i+1}

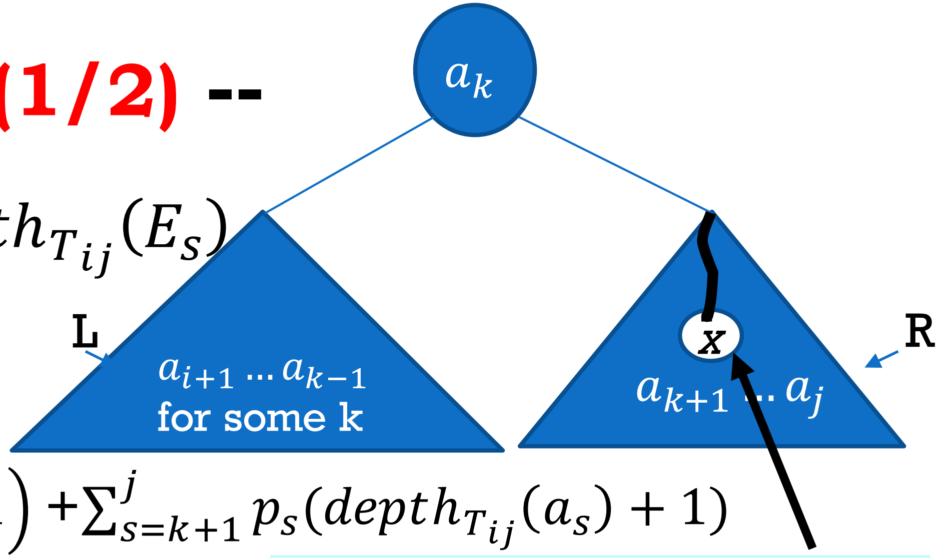
OBST USING DYNAMIC PROGRAMMING
-- (2) PRINCIPLE OF OPTIMALITY --

- **Statement of POO**: The left subtree of an OBST is an OBST, and the right subtree of an OBST is an OBST.
- **Proof**: Take an OBST T_{ij} and let L and R be its left subtree and right subtree. Prove that L and R are both OBST for their respective elements.



OBST

-- PROOF OF THE POO (1/2) --



- $C(T_{ij}) = \sum_{s=i+1}^{j} p_s(depth_{T_{ij}}(a_s) + 1) + \sum_{s=i}^{j} q_s depth_{T_{ij}}(E_s)$
- Break each sum along the two subtrees:
- $C(T_{ij}) = \sum_{s=i+1}^{k-1} p_s(depth_{T_{ij}}(a_s) + 1) + p_k\left(\cancel{depth_{T_{ij}}(a_k)} + 1\right) + \sum_{s=k+1}^{j} p_s(depth_{T_{ij}}(a_s) + 1)$

  $+ \sum_{s=i}^{k-1} q_s depth_{T_{ij}}(E_s) + \sum_{s=k}^{j} q_s depth_{T_{ij}}(E_s)$

$depth_{T_{ij}}(x) = depth_R(x) + 1$

- $C(T_{ij}) = \sum_{s=i+1}^{k-1} p_s(depth_L(a_s) + 1 + 1) + p_k + \sum_{s=k+1}^{j} p_s(depth_R(a_s) + 1 + 1) +$

  $+ \sum_{s=i}^{k-1} q_s(depth_L(E_s) + 1) \quad + \quad \sum_{s=k}^{j} q_s(dept_R(E_s) + 1)$

  $C(L)$ $\qquad$ $C(R)$

- $C(T_{ij}) = C(L) + C(R) + (p_{i+1} + \cdots + p_j) + (q_i + \cdots + q_j)$

Call this quantity: W_{ij}

$$C(T_{ij}) = C(L) + C(R) + W_{ij}$$

OBST
-- PROOF OF THE POO (2/2) --

- $\boldsymbol{C(T_{ij}) = C(L) + C(R) + W_{ij}}$
- If $\boldsymbol{L}$ is not optimal, then we can find a better BST tree $\boldsymbol{L'}$ for the same elements, i.e., $C(L') < C(L)$
- Replace $\boldsymbol{L}$ by $\boldsymbol{L'}$ in $\boldsymbol{T_{ij}}$, resulting a new tree $\boldsymbol{T'_{ij}}$ where
$$\boldsymbol{C(T'_{ij}) = C(L') + C(R) + W_{ij} < C(L) + C(R) + W_{ij} = C(T_{ij})}$$
- Thus, $\boldsymbol{C(T'_{ij}) < C(T_{ij})}$, making $\boldsymbol{T'_{ij}}$ better than optimal, contradiction.
- $\therefore$ The subtree L is optimal (and thus deserves the notation $\boldsymbol{T_{i,k-1}}$)
- Similarly, the subtree R is optimal (and thus deserves the notation $\boldsymbol{T_{kj}}$)
- Q.E.D. (End of proof of the POO)

OBST
-- (3) RECURRENCE RELATION --

- The RR was largely derived during the proof of the POO
- $C(T_{ij}) = C(L) + C(R) + W_{ij}$, where $L = T_{i,k-1}$ and $R = T_{kj}$
- Therefore, $C(T_{ij}) = C(T_{i,k-1}) + C(T_{kj}) + W_{ij}$

Recall that $C_{ij} = C(T_{ij})$, and so $C_{i,k-1} = C(T_{i,j-1})$ and $C_{kj} = C(T_{kj})$

- Leading to: $\boldsymbol{C_{ij} = C_{i,k-1} + C_{kj} + W_{ij}}$. But what is k?
- It is a split point (a_k is at the root), and can be: $i+1, i+2, \ldots, j$
- We try all possible values for k and pick the minimum.

$$\boldsymbol{C_{ij} = \min_{i+1 \le k \le j} (C_{i,k-1} + C_{kj} + W_{ij}); \qquad C_{ii} = 0}$$

$\boldsymbol{r_{ij}}$ **= the $\boldsymbol{k}$ that gives the minimum**

$$W_{ij} = (p_{i+1} + \cdots + p_j) + (q_i + \cdots + q_j)$$
$$W_{i,j-1} = (p_{i+1} + \cdots + p_{j-1}) + (q_i + \cdots + q_{j-1})$$
$$\boldsymbol{W_{ij} = W_{i,j-1} + p_j + q_j;\ W_{ii} = q_i}$$

OBST
-- (4) ALGORITHM FOR THE RR --

// Compute the weights W_{ij}'s first

Procedure weights (**In**: $p[1{:}n], q[0{:}n]$;
Out: $W[0{:}n, 0{:}n]$)

```
Begin
    for i = 1 to n do
        W[i,i] = q(i);
    endfor
    for l = 1 to n do    // l = j − i = size of T_ij
        for i = 0 to n − l do
            j = i + l;
            W[i,j] = W[i,j − 1] + p[j] + q[j];
        endfor
    endfor
end
```

Time: $O(n^2)$
Reason: the double-for loop, whose body takes O(1) time, takes $O(n \times n)O(1) = O(n^2)$

Proc OBST(**In**: $p[1{:}n], q[0{:}n], W[0{:}n, 0{:}n]$;
Out: $C[0{:}n, 0{:}n], r[0{:}n, 0{:}n]$)

```
begin
    for i=0 to n do: C[i,i] := 0; endfor
    for l=1 to n do
        for i=0 to n − l do
            j = i + l;
            C[i,j] :=∞;
            m := i+1; // index of the min
            for k=i + 1 to j do  //compute the min
                if C[i,j] > C[i,k − 1] + C[k,j] then
                    C[i,j] = C[i,k − 1] + C[k,j] ;
                    m := k;
                endif
            endfor
            C[i,j] := C[i,j] + W[i,j];
            r[i,j] := m;
        endfor
    endfor
end
```

Time:
$\sum_{l=1}^{n} \sum_{i=0}^{n-1} cl = \sum_{l=1}^{n} cln = cn \sum_{l=1}^{n} l = cn\left[\frac{n(n+1)}{2}\right] = O(n^3)$

OBST
-- (4) ALGORITHM (CONTINUED) --

```
// Compute the tree T_ij
Proc create-tree(In: r[0:n, 0:n], a[1:n], i, j;
                                        Out: T)
begin
    if (i == j) then
        T=null;
        return;
    endif
    T := new(node);  // the root of T_ij
    k :=  r[i,j];
    T --> data := a[k];
    if (j == i + 1)
        return;
    endif
    create-tree(r, a, i, k − 1; T --> left);
    create-tree(r, a, k, j; T --> right);
end create-tree
```

Time: O(n)
Proof: By induction

```
// Overall algorithm
Proc final-tree(In: a[1:n], p[1:n], q[1:n]; Out: T)
begin
  weights(p[1:n], q[0:n];  W[0:n, 0:n]);
  OBST(p[1:n], q[0:n], W[0:n, 0:n];
              C[0:n, 0:n], r[0:n, 0:n]);
  create-tree(r[0:n, 0:n], a[1:n], 0, n; T);
End
```

Time: $O(n^2) + O(n^3) + O(n) = O(n^3)$
Space: $O(n^2)$ for C[ ], W[ ], and r[ ]

OBST DP ALGORITHM
-- AN EXAMPLE (1/3) --

- Input: $n = 4, \quad a_1 < a_2 < a_3 < a_4;$ $\quad p_1 = \frac{1}{10}, p_2 = \frac{2}{10}, p_3 = \frac{3}{10}, p_4 = \frac{1}{10}$

$$q_0 = 0, q_1 = \frac{1}{10}, q_2 = \frac{1}{20}, q_3 = \frac{1}{20}, q_4 = \frac{1}{10}$$

- Weights:

$W_{ii} = q_i$

$W_{ij} = W_{i,j-1} + p_j + q_j$

Table is computed one row after another, top to bottom

$W_{00} = 0$	$W_{11} = \frac{1}{10}$	$W_{22} = \frac{1}{20}$	$W_{33} = \frac{1}{20}$	$W_{44} = \frac{1}{10}$
$W_{01} = \frac{2}{10}$	$W_{12} = \frac{3.5}{10}$	$W_{23} = \frac{4}{10}$	$W_{34} = \frac{2.5}{10}$	
$W_{02} = \frac{4.5}{10}$	$W_{13} = \frac{7}{10}$	$W_{24} = \frac{6}{10}$		
$W_{03} = \frac{8}{10}$	$W_{14} = \frac{9}{10}$			
$W_{04} = \frac{10}{10}$				

OBST DP ALGORITHM
-- AN EXAMPLE (2/3) --

- The C's and r's:

$C_{ii} = 0$

$C_{ij} = \min_{i+1 \le k \le j} \left(C_{i,k-1} + C_{kj} + W_{ij} \right)$

r_{ij} = the k that gives the minimum

$C_{00} = 0$	$C_{11} = 0$	$C_{22} = 0$	$C_{33} = 0$	$C_{44} = 0$
$C_{01} = \frac{2}{10}$ $r_{01} = 1$	$C_{12} = \frac{3.5}{10}$ $r_{12} = 2$	$C_{23} = \frac{4}{10}$ $r_{23} = 3$	$C_{34} = \frac{2.5}{10}$ $r_{34} = 4$	
$C_{02} = \frac{6.5}{10}$ $r_{02} = 2$	$C_{13} = \frac{10.5}{10}$ $r_{13} = 3$	$C_{24} = \frac{8.5}{10}$ $r_{24} = 3$		
$C_{03} = \frac{14}{10}$ $r_{03} = 2$	$C_{14} = \frac{15}{10}$ $r_{14} = 3$			
$C_{04} = \frac{19}{10}$ $r_{04} = 3$				

Table is computed one row after another, top to bottom

$i = 1$
$j = 3$

Example of computing the C's

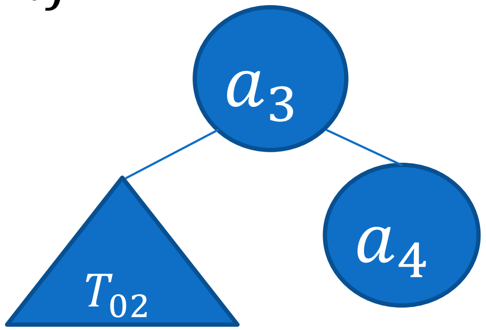
C_{13} after top 2 rows have been computed:

$$C_{13} = \min_{2 \le k \le 3} \left(C_{1,k-1} + C_{k3} + W_{13} \right) = \min\left(C_{11} + C_{23} + \frac{7}{10}, C_{12} + C_{33} + \frac{7}{10} \right) = \min\left(0 + \frac{4}{10} + \frac{7}{10}, \frac{3.5}{10} + 0 + \frac{7}{10}\right)$$

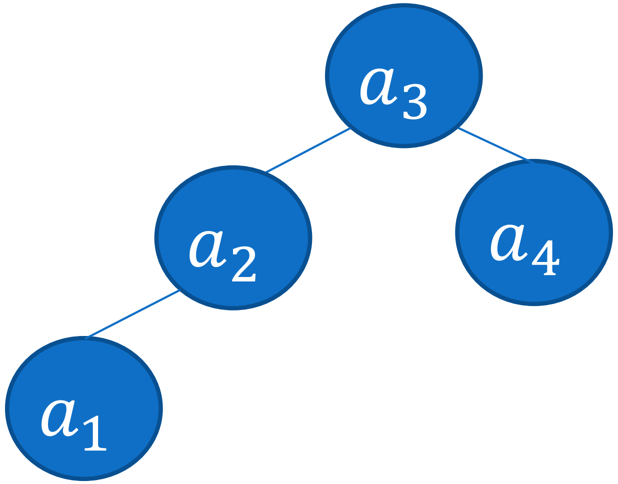
$$C_{13} = \min\left(\frac{11}{10}, \frac{10.5}{10} \right) = \frac{10.5}{10}$$ corresponding to $k = 3$, thus $r_{13} = 3$

OBST DP ALGORITHM
-- AN EXAMPLE (3/3) --

- Constructing the actual OBST from the r_{ij}'s in the table:



- T=T_{04} has root $a_{r_{04}} = a_3$
  - $\therefore$ left subtree has $\{a_1, a_2\}$, i.e., T_{02}
  - and right subtree has a the single node a_4
- Let's construct T_{02}
  - T_{02} has root $a_{r_{02}} = a_2$
  - T_{02} has a left subtree T_{01}: a single node a_1
  - T_{02} has a right subtree T_{22}, which is empty
- The tree is done



EXERCISES

- **Exercise 4**: If all the q_i's are equal to 0, and all the p_i's are all equal (to 1/n), prove that the cost C(T) of any binary search tree T is equal to (the average depth of T)+1, where

  (the average depth of T) = $\frac{1}{n}(depth_T(1) + depth_T(2) + \cdots + depth_T(n))$

- **Exercise 5**: Give a greedy algorithm for matrix chain problem, and prove/disprove that your greedy method guarantees or does not guarantee optimality

- **Exercise 6**: Give a greedy algorithm for OBST problem, and prove/disprove that your greedy method guarantees or does not guarantee optimality

WRAP-UP

- Done with Dynamic Programming
- Let's review the lessons learned
- And point to other applications of DP

LESSONS LEARNED SO FAR

- DP is an optimization design technique
- It has a litmus test (the principle of optimality): DP applies if OOP holds
- Proof of the POO is almost always by contradiction
- Solving the recurrence relation is non-recursive, bottom up, from smallest subsolutions to the final solution, where the subsolutions are usually recorded by filling a table
- The actual optimal solution is constructed by using the optimal split points recorded in the table
- Time complexity of DP algorithms tend to be cubic (i.e., $O(n^3)$), so they're slower than greedy algorithms, but still reasonably fast
- DP is more powerful than greedy: in many situations where greedy solutions are not optimal, DP applies and gives optimal solutions
- Still, DP does not always apply: when POO fails, don't bother with DP

OTHER APPLICATIONS OF DYNAMIC PROGRAMMING

- Hidden Markov Models (used in machine learning, statistics, Natural Language Processing, Bioinformatics, etc.)
- Viterbi algorithm (in communications/networking, e.g., modulation/demodulation)
- Sequence alignment in genetics (Needleman–Wunsch algorithm)
- Longest common subsequence
- DP is used in Reinforcement Learning (click here)
- Scheduling (and time sharing)
- Query optimization in databases
- Robot control
- Flight control
- Many more