EECS 3101

Prof. Andy Mirzaian

Dynamic Programming

Those who cannot remember the past are doomed to repeat it.

— George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'. *I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt then about the term* 'mathematical'. *The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

— Richard Bellman, on the origin of his term 'dynamic programming' (1984)

STUDY MATERIAL:

- **[CLRS]** chapter 15
- **Lecture Note 9**
- **Algorithmics Animation Workshop:**
  - Optimum Static Binary Search Tree

TOPICS

- **Recursion Tree Pruning by Memoization**
- **Recursive Back-Tracking**
- **Dynamic Programming**

- **Problems:**
  - Fibonacci Numbers
  - Shortest Paths in a Layered Network
  - Weighted Event Scheduling
  - Knapsack
  - Longest Common Subsequence
  - Matrix Chain Multiplication
  - Optimum Static Binary Search Tree
  - Optimum Polygon Triangulation
  - More Graph Optimization problems considered later

Recursion Tree Pruning by Memoization

Alice: We have done this sub-instance already, haven't we?

Bob: Did you take any memo of its solution?
Let's re-use that solution and save time by not re-doing it.

Alice: That's a great time saver, since we have many many repetitions of such sub-instances!

Re-occurring Sub-instances

- In divide-&-conquer algorithms such as MergeSort, an instance is typically partitioned into “non-overlapping” sub-instances (e.g., two disjoint sub-arrays). That results in the following property:
  - For any two (sub) sub-instances, either one is a sub-instance of the other (a descendant in the recursion tree), or the two are disjoint & independent from each other.
  - We never encounter the same sub-instance again during the entire recursion.

- If the sub-instances “overlap”, further down the recursion tree we may encounter repeated (sub) sub-instances that are in their “common intersection”.
  - Example 1: The Fibonacci Recursion tree (see next slide).
  - Example 2: Overlapping sub-arrays as sub-instances.

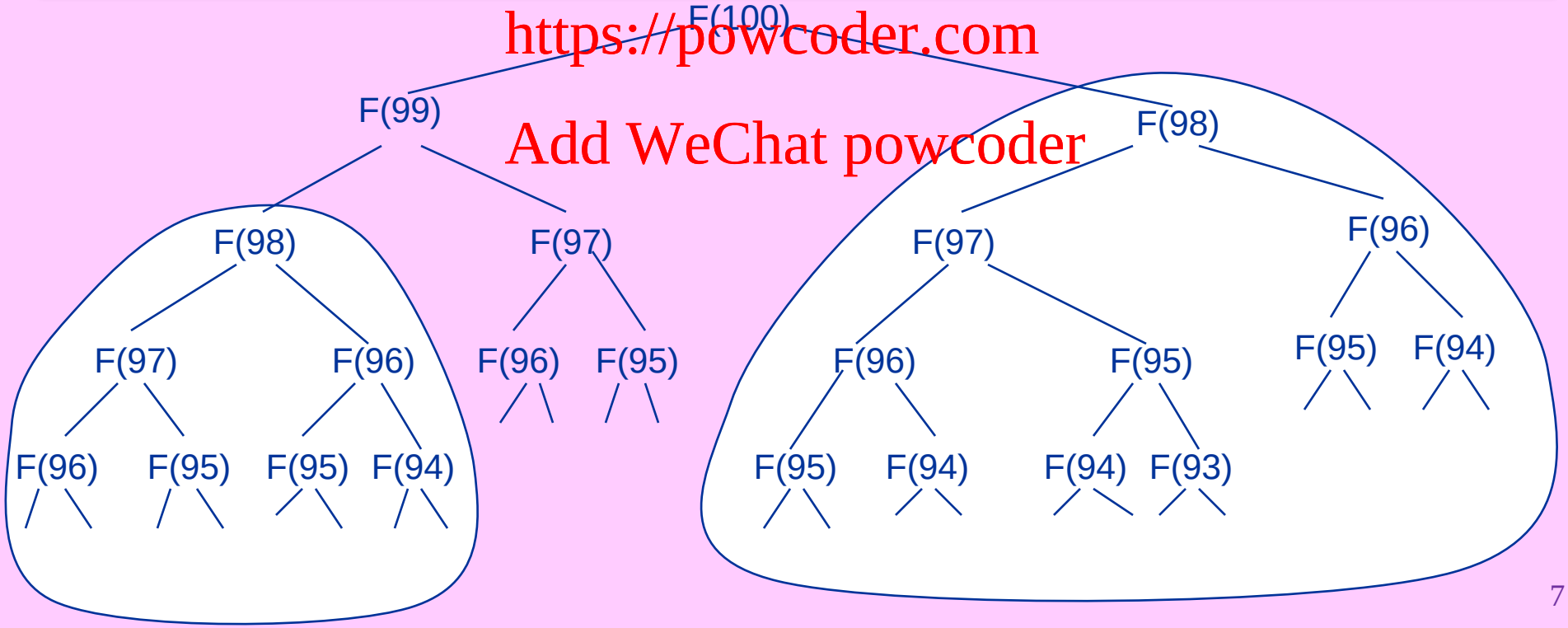
Fibonacci Recursion Tree

Algorithm **F**(n)
    if n∈{0,1} **then return** n
    return F(n-1) + F(n-2)
end

$$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(1) & n \geq 2 \\ \Theta(1) & n = 0,1 \end{cases}$$

$$\Rightarrow \quad T(n) = \Theta(F_n) = \Theta(\varphi^n).$$

But only n+1 distinct sub-instances are ever called:
F(0), F(1), …, F(n-2), F(n-1), F(n).
Why is it taking time exponential in n?



Pruning by Memoization

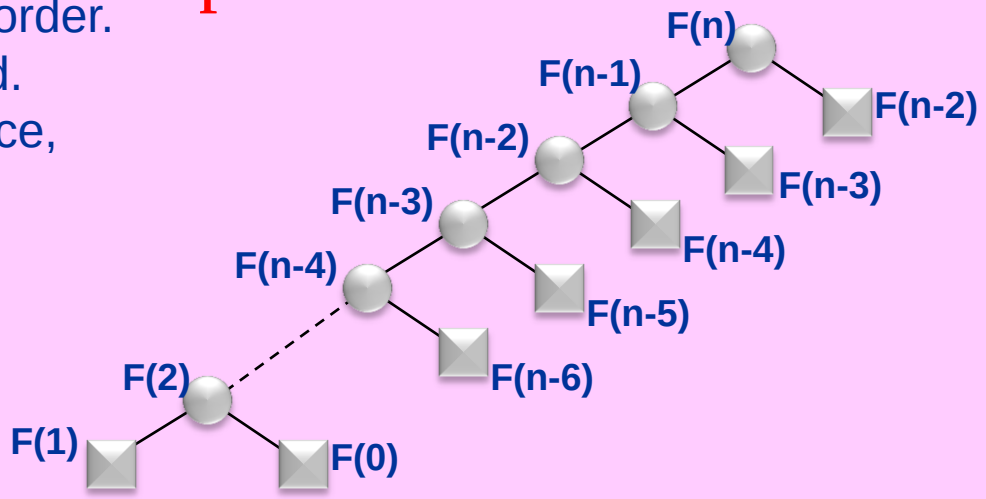
```
Algorithm  Fib(n)
    for t ← 0 .. 1 do Memo[t] ← t
    for t ← 2 .. n do Memo[t] ← null          § initialize Memo[0..n] table
    return  F(n)
end

Function  F(n)                                § recursion structure kept
    if Memo[n] = null then Memo[n] ← F(n-1) + F(n-2)   § memo feature added
    return  Memo[n]
end
```


Recursion tree evaluated in post-order.
Now all right sub-trees are pruned.
Instead of re-solving a sub-instance,
fetch its solution from
the memo table Memo[0..n].

$T(n) = \Theta(n)$.

Time-space trade off.



Memoization: Recursive vs Iterative

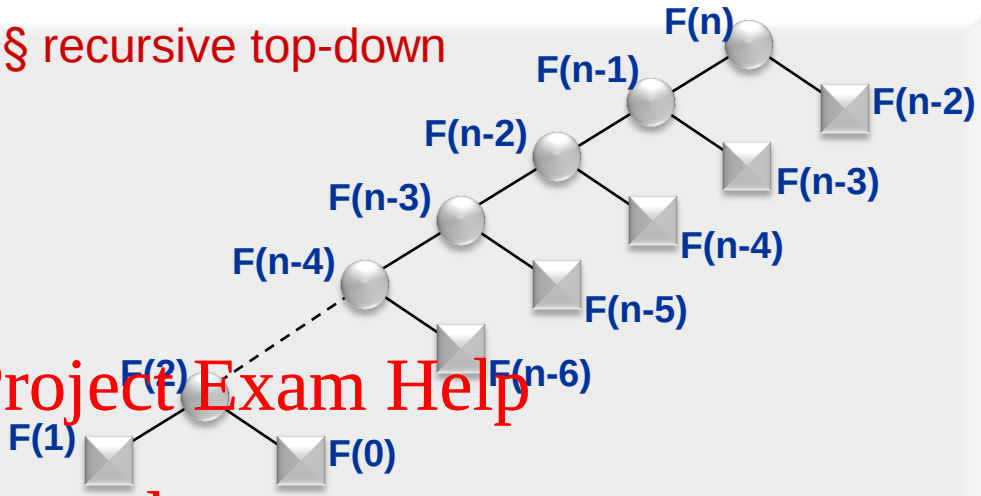
Algorithm Fib(n)

```
    for t ← 0 .. 1 do Memo[t] ← t
    for t ← 2 .. n do Memo[t] ← null
    return F(n)
end
```

Function F(n)

```
    if Memo[n] = null
        then Memo[n] ← F(n-1) + F(n-2)
    return Memo[n]
end
```

§ recursive top-down



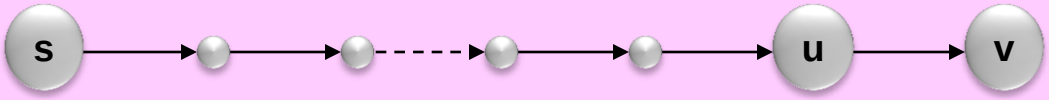
Algorithm Fib(n)

§ iterative bottom-up (from smallest to largest)

```
    for t ← 0 .. 1 do Memo[t] ← t
    for t ← 2 .. n do Memo[t] ← Memo[t-1] + Memo[t-2]
    return Memo[n]
end
```

Compact Memoization

- We want **compact** memos to save both **time** and **space**.
- Fibonacci case is simple; memoize a single number per instance.
- What if the solution to a sub-instance is a more elaborate structure?
- **Examples:**
  1. Sub-instance solution is subarray A[i..j]: Just store (i,j), its first and last index.
  2. Sub-matrix A[i..j][k..t]: Just store the 4 indices (i,j,k,t).
  3. Solutions to sub-instances are sequences. Furthermore, any prefix of such a sequence is also a solution to some "smaller" sub-instance.
     Then, instead of memoizing the entire solution sequence, we can memoize its next-to-last (as well as first and last) elements.
     We can recover the entire solution á posteriori by using the prefix property.
  4. Single-source shortest paths.
     If the last edge on the shortest path SP(s,v) from vertex s to v is edge (u,v), then memo ⟨s,u,v⟩ (and path length) for the solution to that sub-instance. We can consult the solution to sub-instance SP(s,u) to find "its" last edge, and so on. Á posteriori, working backwards, we can recover the entire sequence of vertices on the path from s to v.

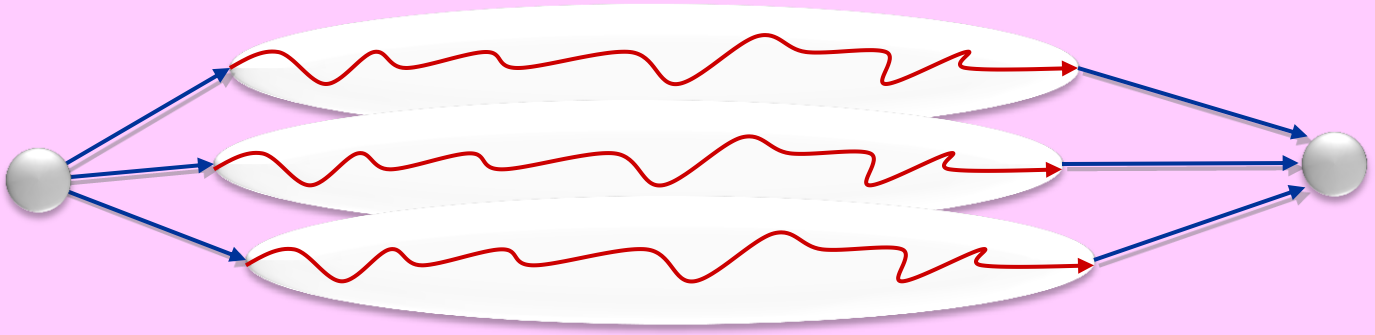


Recursive Back-Tracking

Alice: I know my options for the first step.
Can you show me the rest of the way?

Bob: OK. For each first step option,
I will show you the best way to finish the rest of the way.

Alice: Thank you. Then I can choose the option
that leads me to the overall best complete solution.



Solving Combinatorial Optimization Problems

- **Any combinatorial problem is a combinatorial optimization problem:**
  For each solution (valid or not) assign an objective value:
  1 if the solution is valid, 0 otherwise.
  Now ask for a solution that maximizes the objective value.

  Sorting: find a permutation of the input with minimum inversion count.

- **Greedy Method:**
  Fast & simple, but has limited scope of applicability to obtain exact optimum solutions.

- **Exhaustive Search:**
  A systematic brute-force method that explores the entire solution space.
  Wide range of applicability, but typically has exponential time complexity.

- . . .

The Optimum Sub-Structure Property
(repeated from "Greedy" Slide, p. 33)

- We just noticed an important property that will be used many times later:
- The optimum sub-structure property: any sub-structure of an optimum structure is itself an optimum structure (for the corresponding sub-instance).
- Problems with this property are usually amenable to more efficient algorithmic solutions than brute-force or exhaustive search methods.
- This property is usually shown by a "cut-&-paste" argument (see below).

- Example 1: The Coin Change Making Problem.
  Consider an optimum solution Sol ∈ OPT(S). Let G1 be a group of coins in Sol.
  Suppose G1 ∈ FEAS(U). Then we must have G1 ∈ OPT(U). Why?
  Because if G1 ∉ OPT(U), then we could cut G1 from Sol and paste in the optimum sub-structure G2 ∈ OPT(U) instead. By doing so, we would get a new solution Sol' ∈ FEAS(S) that has an even better objective value than Sol. But that would contradict Sol ∈ OPT(S).

- Example 2: The Shortest Path Problem.
  Let P be a shortest path from vertex A to vertex B in the given graph G.
  Let P' be any (contiguous) sub-path of P. Suppose P' goes from vertex C to D.
  Then P' must be a shortest path from C to D. If it were not, then there must be an even shorter path P'' that goes from C to D. But then, we could replace P' portion of P by P'' and get an even shorter path than P that goes from A to B.
  That would contradict the optimality of P.

Solving COP by Recursive Back-Tracking

- **Recursive Back-Tracking (RecBT):**
  a) Divide the solution space for a given instance into a number of sub-spaces.
  b) These sub-spaces must themselves correspond to one or a group of sub-instances of the same structure as the main instance.
  c) **Requirement:** the optimum sub-structure property.
  d) Recursively find the optimum solution to the sub-instances corresponding to each sub-space.
  e) Pick the best among the resulting sub-space solutions.
  This best of the best is the overall optimum structure solution for the main instance.

NOTE:

Divide-&-Conquer and Recursive BT vastly differ on what they divide.

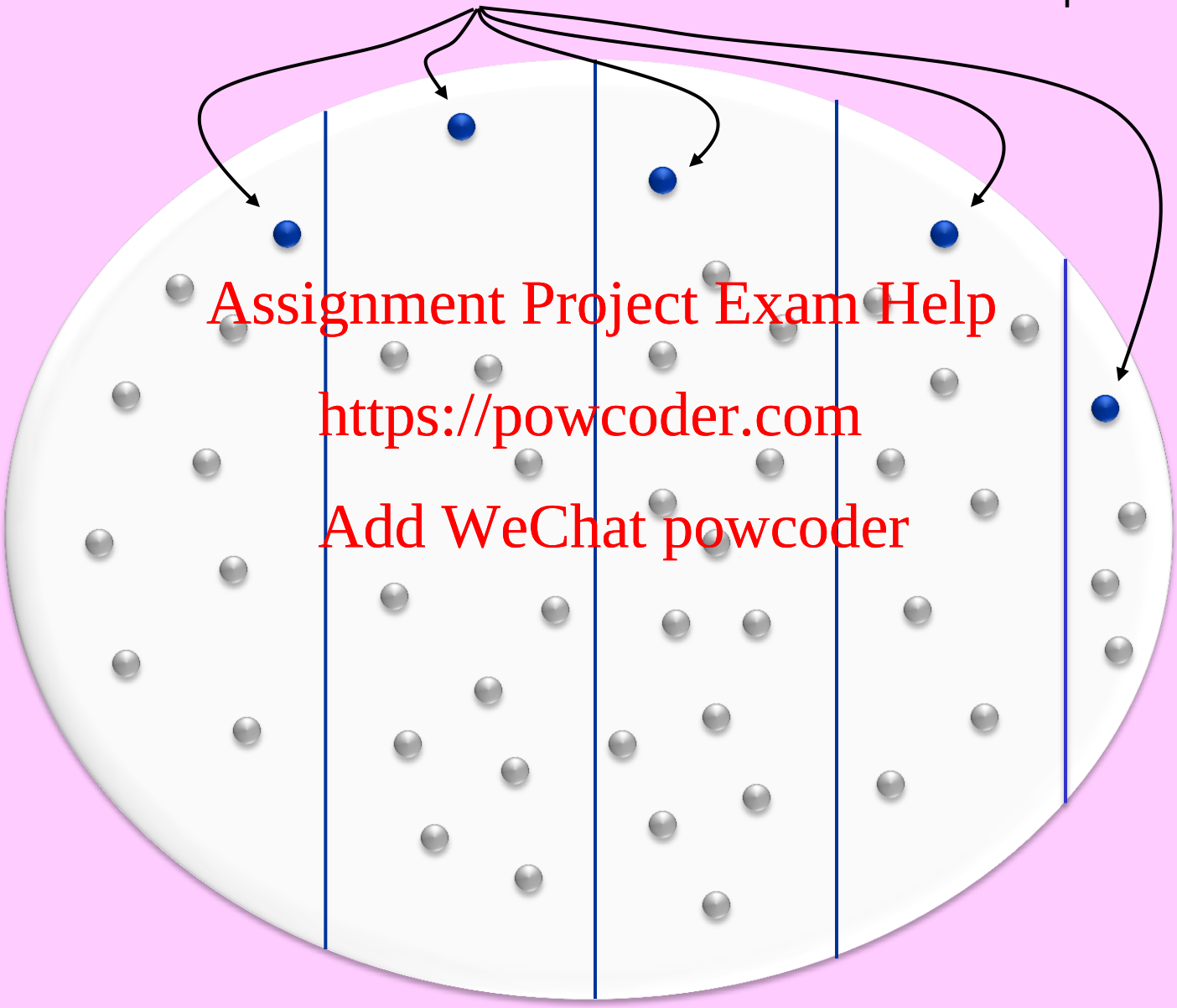
D&C divides an instance in one fixed way into a number of sub-instances.

RecBT divides the solution space into sub-spaces.

- This is done based on one or more sub-division options.
- Each option corresponds to one of the said sub-spaces.
- Each option generates one or more sub-instances whose complete solution corresponds to the solution within the corresponding sub-space.

RecBT Solution Sub-Spaces

Take best of these best solutions within each solution subspace



RecBT Recursion Tree

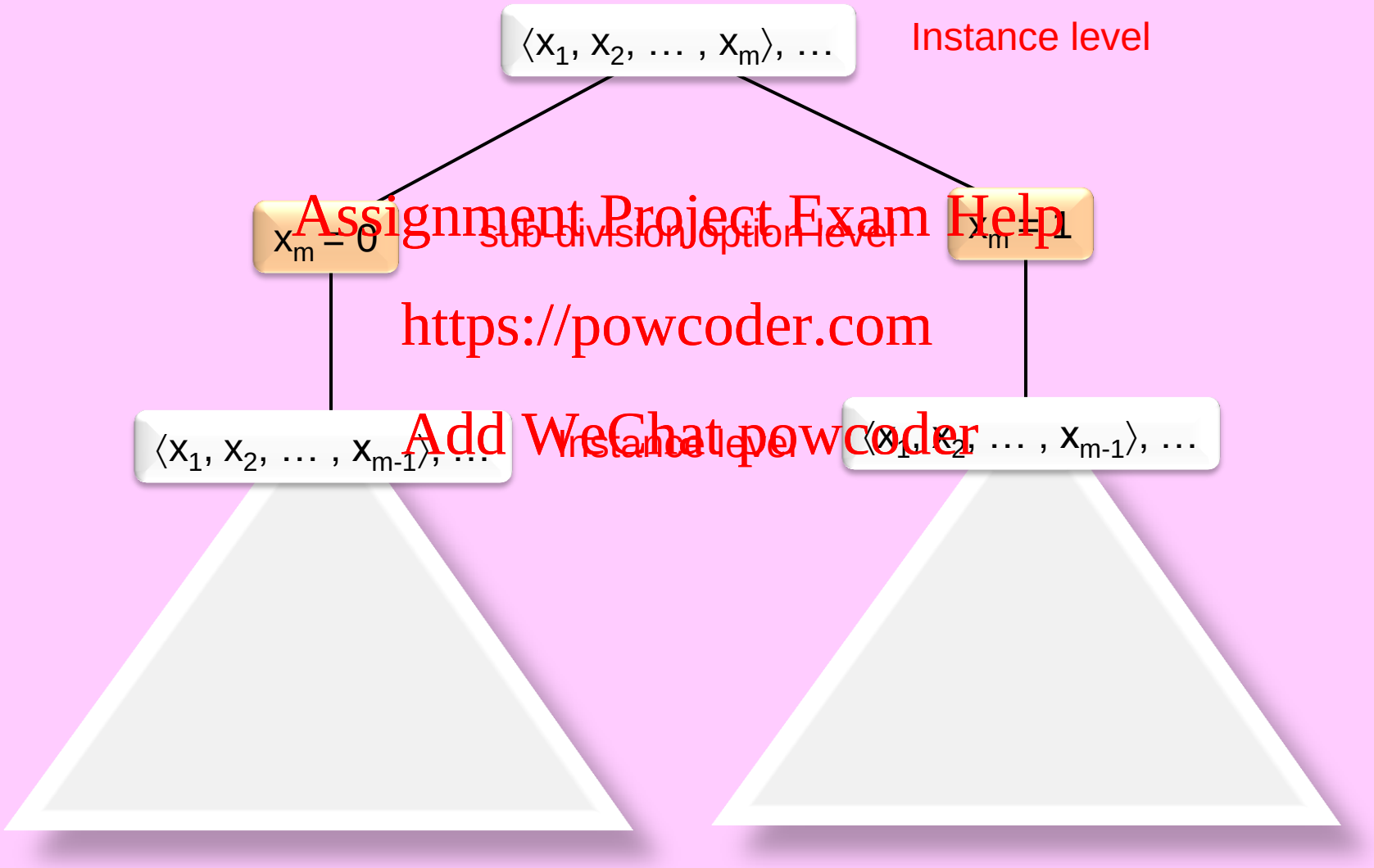
- How should we divide the entire solution space into sub-spaces, so that each sub-space corresponds to a (smaller) instance or a group of (smaller) instances of the same type as the main problem we are trying to solve?

- **The Recursion tree:**
  - Its root corresponds to the main instance given.
  - There are two types of nodes:
    - Sub-instance nodes appear at even depths of the recursion tree,
    - Sub-division option nodes appear at odd depths of the recursion tree.
  - For a specific (sub-) instance node, we may have a number of options on how to further divide it into groups of (sub-) sub-instances.
    For each such option, we have a sub-division option node as a child.
    Each option node has a group of children, each of which is a sub-instance node.
  - The leaves of the tree correspond to base case instances.
  - Examples follow …… P.T.O.

Example 1: RecBT Recursion Tree

0/1 binary decision on a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$.

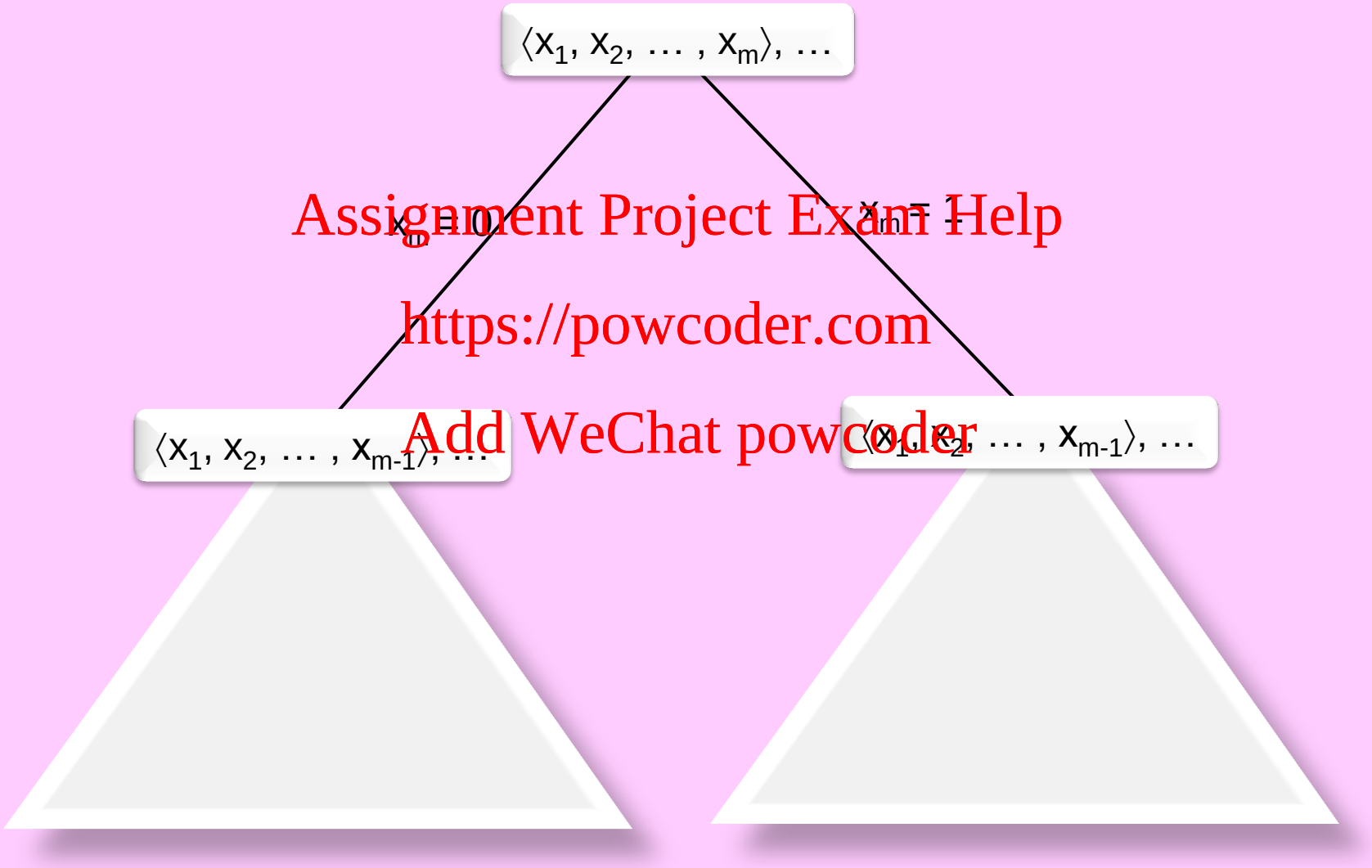
A simple situation: each division option node generates a single sub-instance child.



Example 1: RecBT Recursion Tree

0/1 binary decision on a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$.

A simple situation: each division option node generates a single sub-instance child.

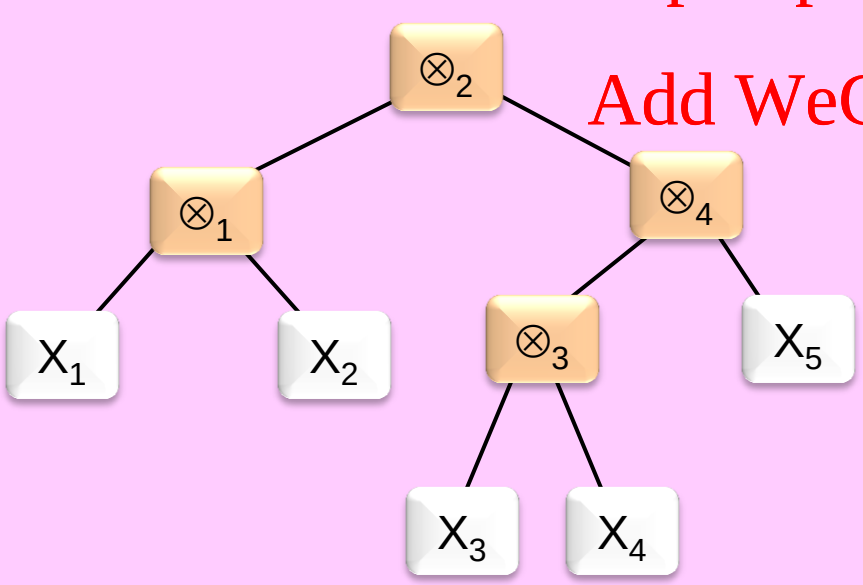


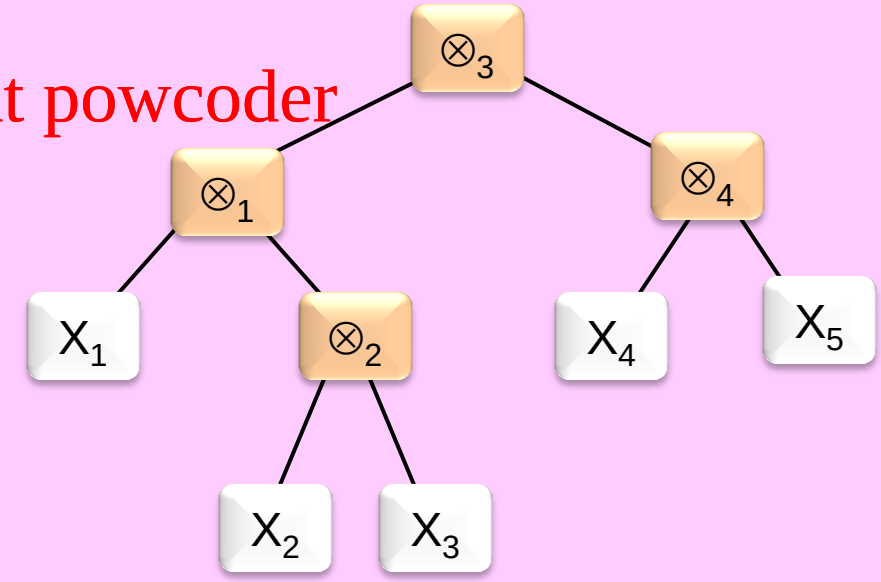
Example 2: RecBT Recursion Tree

Find optimum full parenthesization of $(X_1 \otimes X_2 \otimes \ldots X_k \otimes X_{k+1} \ldots \otimes X_n)$, where $\otimes$ is an associative binary operator and X_k's are operands.

First, let's see what a feasible solution looks like:
It can be viewed by the full parenthesization,
or by its parse tree. (Note: parse tree $\neq$ recursion tree.)

$((X_1 \otimes_1 X_2) \otimes_2 ((X_3 \otimes_3 X_4) \otimes_4 X_5))$ $(((X_1 \otimes_1 (X_2 \otimes_2 X_3)) \otimes_3 (X_4 \otimes_4 X_5))$



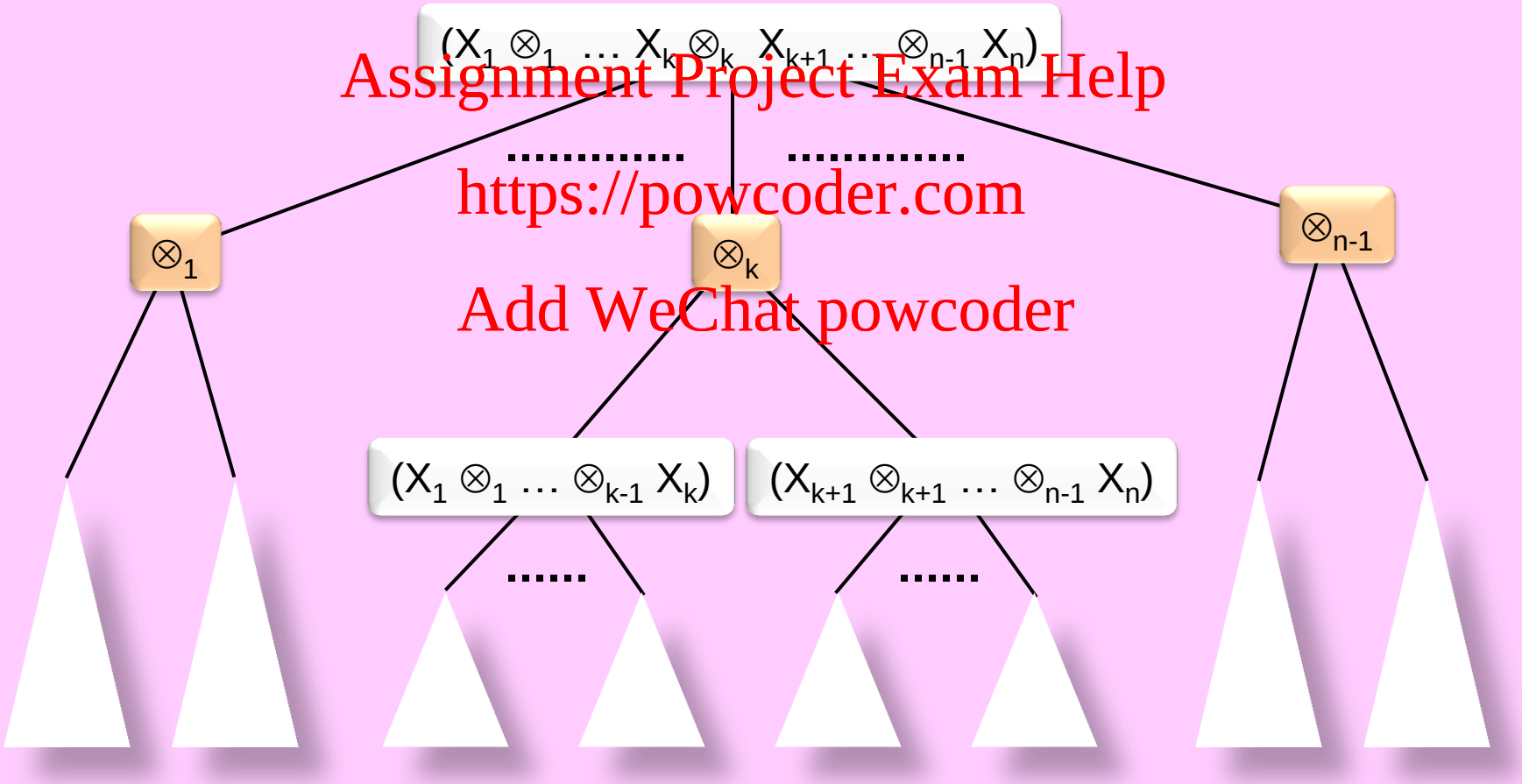


Example 2: RecBT Recursion Tree

Find optimum full parenthesization of $(X_1 \otimes_1 X_2 \otimes_2 \ldots X_k \otimes_k X_{k+1} \ldots \otimes_{n-1} X_n)$.

Decision: which of $\otimes_1, \ldots, \otimes_k, \ldots, \otimes_{n-1}$ should be performed LAST, i.e, form the root of the parse tree?

Induces n-1 top level sub-division option nodes.

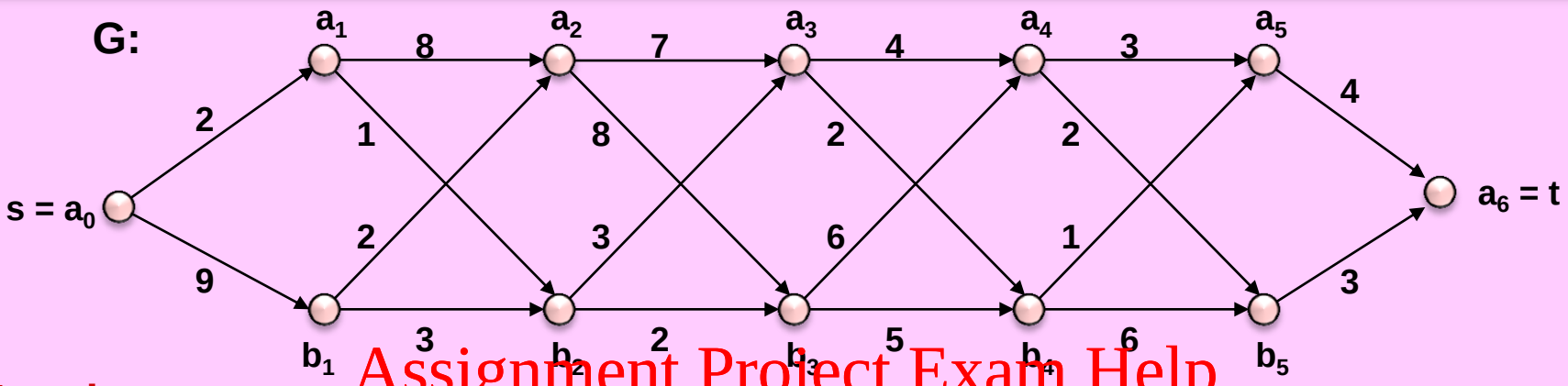


SHORTEST PATHS

in

LAYERED NETWORKS

Shortest s-t path in Layered Network



Notation:

- Layers 0..n in graph G. s = starting vertex at layer 0, t = terminal vertex at layer n.
  For now assume Θ(1) vertices per layer.

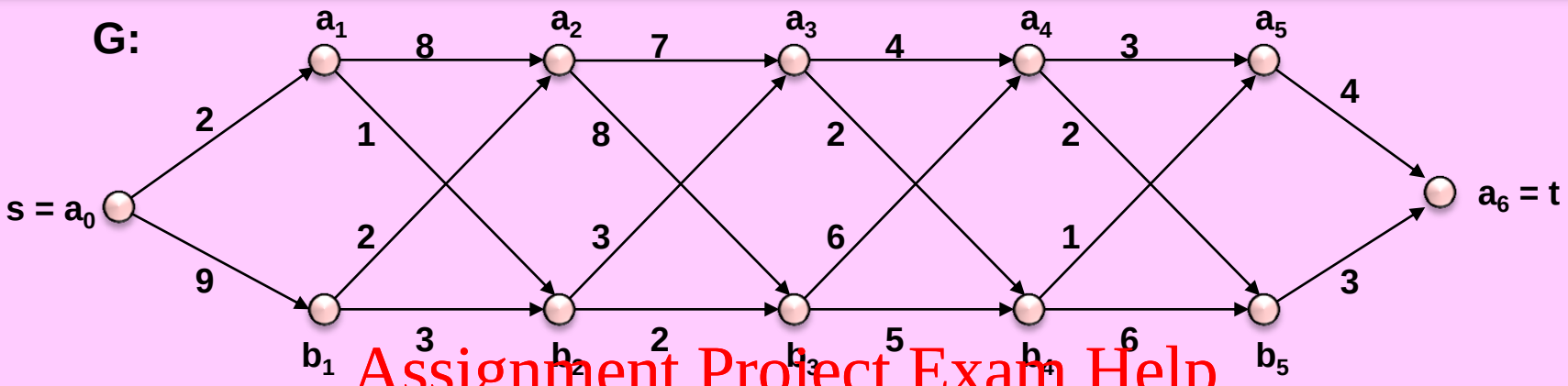
- w(u,v) = weight (or length) of edge (u,v) in G.

- SP(u,v) = shortest path in G from vertex u to v (as a sequence of vertices).
  CostSP(u,v) = cost (or length) of SP(u,v).

- Example: $SP(a_1,b_4) = \langle a_1, b_2, a_3, b_4 \rangle$
  $CostSP(a_1,b_4) = w(a_1,b_2) + w(b_2,a_3) + w(a_3,b_4) = 1+3+2 = 6.$

- 3 algorithmic solutions:
  - Recursive Back-Tracking without Pruning
  - Recursive Back-Tracking with Memoized Pruning
  - Iterative Memoized Algorithm

Shortest s-t path in Layered Network



- **Optimum sub-structure property:**
  - Any sub-path of a shortest path is a shortest path.
  - More specifically, removing the last edge of a shortest path results in a prefix path with one less edge that is itself a shortest path.

- Last edge on SP(s,t) is either (a_5, t) or (b_5, t).

  Option 1. (a_5, t): The rest of SP(s, t) must be SP(s, a_5).

  Option 2. (b_5, t): The rest of SP(s, t) must be SP(s, b_5).

- $SP(s, t) = \text{best of } \{ \langle SP(s, a_5), (a_5, t) \rangle, \langle SP(s, b_5), (b_5, t) \rangle \}$

  $CostSP(s, t) = \min\{ CostSP(s, a_5) + w(a_5, t), CostSP(s, b_5) + w(b_5, t) \}$

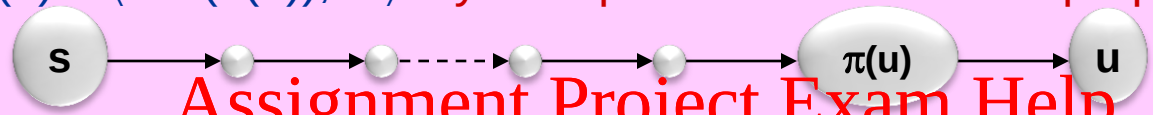
- **All sub-instances are of the form SP(s, u), where u is any vertex of G.**

Shortest s-t path in Layered Network

- Short-hand notation:

  SP(u) = SP(s,u), for any vertex u of G. (Starting vertex is always s.)

  CostSP(u) = CostSP(s,u)
- $\pi(u)$ = predecessor of vertex u on the shortest path from s to u.
- SP(u) = $\langle$ SP(π(u)), u $\rangle$ by the optimum sub-structure property.



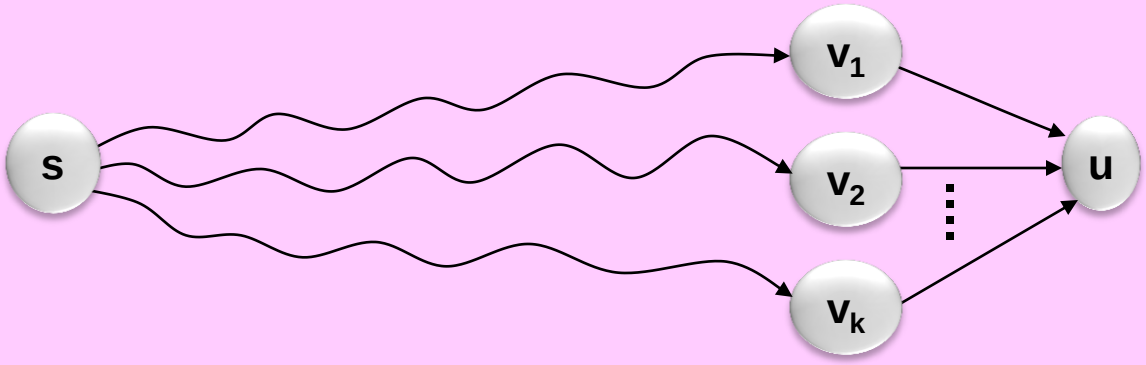
FACT: SP(u) = $\langle$ SP(π(u)), u $\rangle$, where

$\pi(u) \in \{ v \mid (v,u)$ is an edge in G $\}$

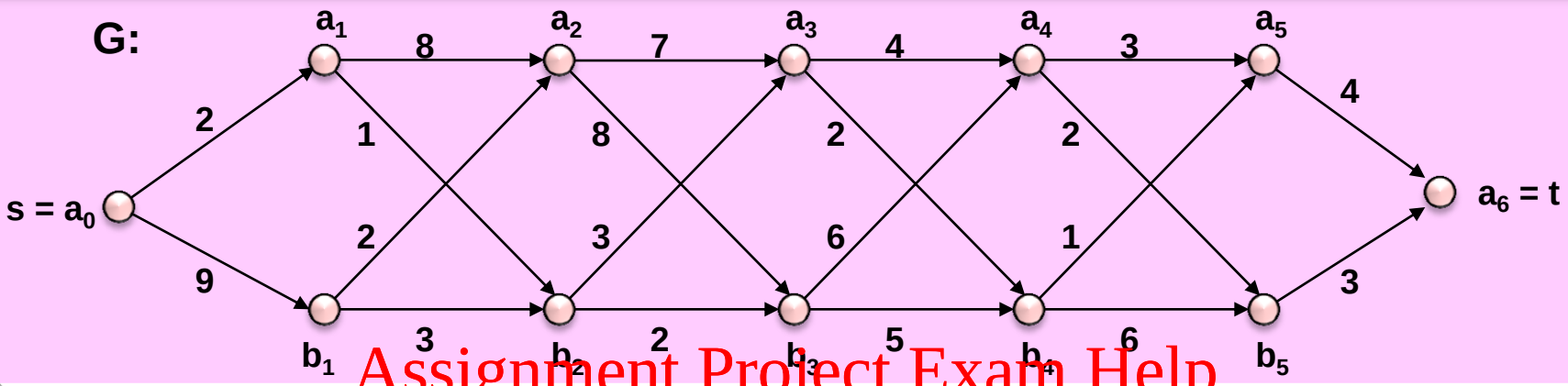
$\pi(u) = \text{argmin}_v \{ \text{CostSP}(v) + w(v,u) \mid (v,u)$ is an edge in G $\}$

$\text{CostSP}(u) = \min_v \{ \text{CostSP}(v) + w(v,u) \mid (v,u)$ is an edge in G $\}$

$= \text{CostSP}(\pi(u)) + w(\pi(u), u)$



Recursive Back-Tracking without Pruning



Algorithm **ShortestPath**(G, s, u)

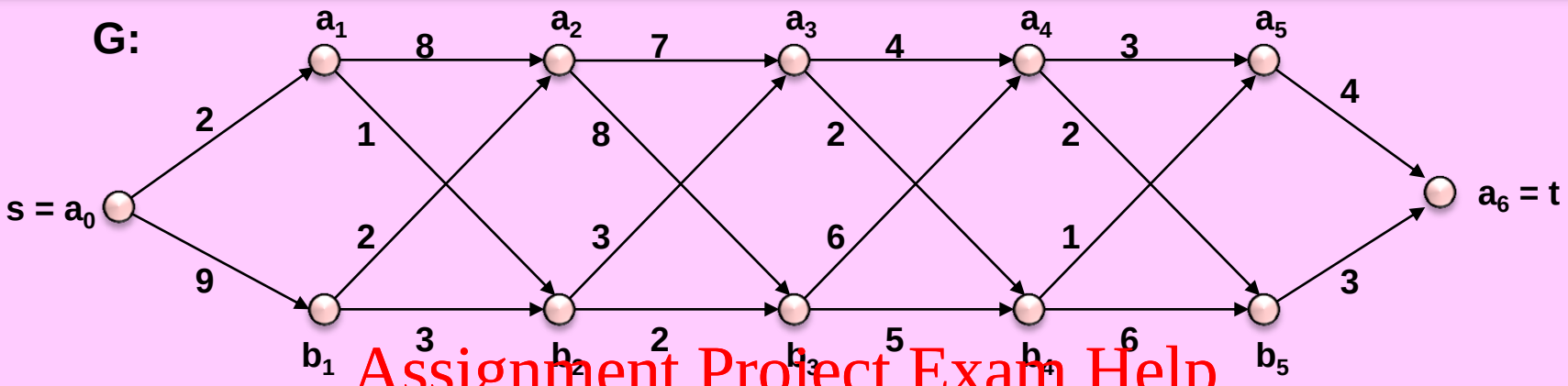
Pre-Cond: G = a weighted layered digraph, u = a vertex of G, s = source vertex.

Post-Cond: output is ⟨ SP(u), CostSP(u) ⟩

```
    if  u = s  then return ⟨ s, 0 ⟩    § base case
        § CostSP(u) = min_v { CostSP(v) + w(v,u) | (v,u) is an edge in G }
    cost ← ∞
    for  each vertex v∈V(G) :  (v,u)∈E(G)  do
        ⟨ prefixSP, prefixCost ⟩ ← ShortestPath(G, s, v)
        if  cost > prefixCost + w(v,u)  then  cost ← prefixCost + w(v,u);
                                              SP ← ⟨ prefixSP, u⟩
    end-for
    return ⟨ SP , cost ⟩
end
```

$2^{O(n)}$ time.

Á Posteriori Print Shortest s-t Path



⟨ SP , cost ⟩ ← **ShortestPath**(G, s, t)

$2^{O(n)}$ **time.**

Print SP

O(n) time.

Recursive Back-Tracking with Memoized Pruning

Memo[V(G)] = an array indexed by V(G).
$\forall u \in V(G)$: Memo[u] = ⟨Memo.π[u] , Memo.CostSP[u] ⟩ .

```
Algorithm ShortestPath(G, s, t)   § G = a layered graph …
    for each vertex u ∈ V(G) do
        Memo[u] ← ⟨ nil, ∞ ⟩          § Initialize memo table
    end-for
    Memo[s] ← ⟨ nil, 0 ⟩              § base case
    SP(G, s, t)                       § compute shortest paths
    PrintSP(G, s, t)                  § print shortest s-t path
end
```



```
Procedure SP(G, s, u)
    if u = s then return                    § base case
    for each vertex v∈V(G) : (v,u)∈E(G) do
        if Memo.CostSP[v] = ∞ then SP(G, s, v)
        if Memo.CostSP[u] > Memo.CostSP[v] + w(v,u)
            then Memo.CostSP[u] ← Memo.CostSP[v] + w(v,u);
                 Memo.π[u] ← v
    end-for
end
```

Recursive Back-Tracking with Memoized Pruning

Memo[V(G)] = an array indexed by V(G).
$\forall u \in V(G)$: Memo[u] = ⟨Memo.π[u] , Memo.CostSP[u] ⟩ .

```
Algorithm ShortestPath(G, s, t)      § G = a layered graph …
    for each vertex u ∈ V(G) do
        Memo[u] ← ⟨ nil, ∞ ⟩          § Initialize memo table
    end-for
    Memo[s] ← ⟨ nil, 0 ⟩              § base case
    SP(G, s, t)                       § compute shortest paths
    PrintSP(G, s, t)                  § print shortest s-t path
end
```

Θ(n) time


```
Procedure PrintSP(G, s, u)
Pre-Cond: G = a weighted layered digraph, u = a vertex of G, s = source vertex.
Post-Cond: output is SP(u), the shortest path from s to u in G.
    if u = nil then return           § base case
    PrintSP(G, s, Memo.π[u])
    print u
end
```

Iterative Memoized Algorithm

The closer the layer of u is to s, the "lower" the sub-instance is in the recursion tree, the earlier we need to solve it iteratively "bottom-up".
Solutions to larger sub-instances depend on the smaller ones.

```
Algorithm ShortestPath(G, s, t)          § G = a layered graph …
    for each vertex u ∈ V(G) do          § Initialize memo table
        ⟨ π[u], Cost[u] ⟩ ← ⟨ nil, ∞ ⟩
    Cost[s] ← 0                          § base case first
    for layer ← 1 .. n do                § in a reverse order of dependency
        for each vertex u in layer do
            for each vertex v∈V(G) : (v,u)∈E(G) do
                if   Cost[u] > Cost[v] + w(v,u)
                then Cost[u] ← Cost[v] + w(v,u);
                     π[u] ← v
    PrintSP(G, t)
end
```

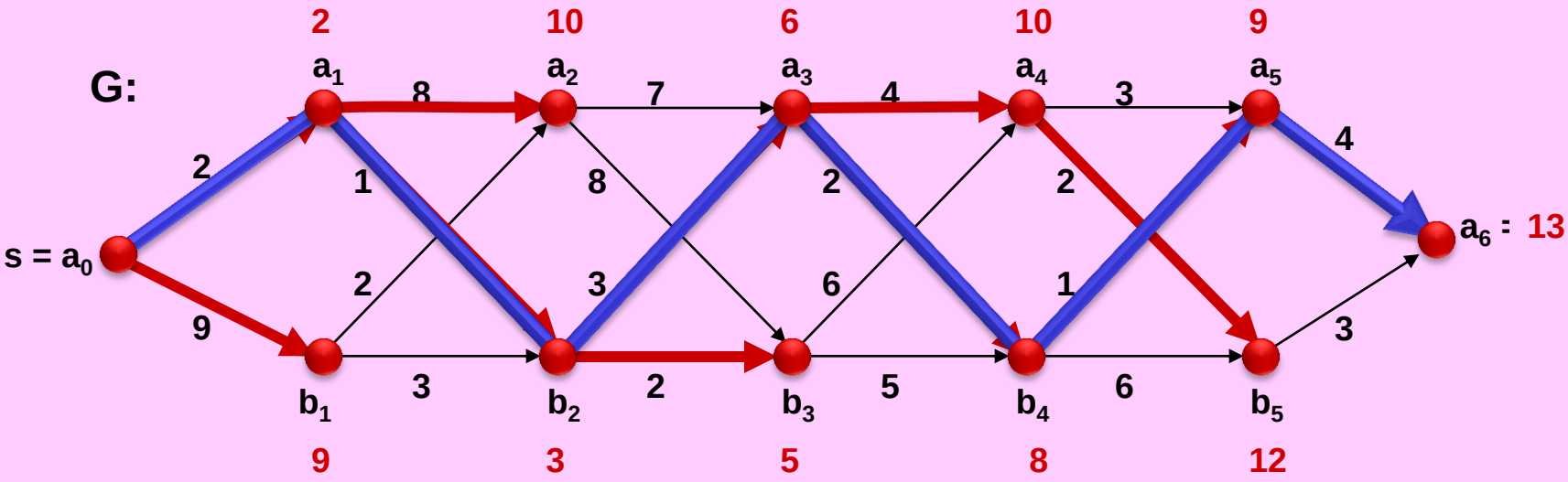

Θ(n) time

```
Procedure PrintSP(G, u)          § O(n) time
    if u = nil then return       § base case
    PrintSP(G, π[u]); print u
end
```

An Example Run

```
Algorithm ShortestPath(G, s, t)          § G = a layered graph …
    for each vertex u ∈ V(G) do          § Initialize memo table
        ⟨ π[u], Cost[u] ⟩ ← ⟨ nil, ∞ ⟩
    Cost[s] ← 0
    for layer ← 1 .. n do                § in a reverse order of dependency
        for each vertex u in layer do
            for each vertex v∈V(G) : (v,u)∈E(G) do
                if Cost[u] > Cost[v] + w(v,u)
                then Cost[u] ← Cost[v] + w(v,u);
                     π[u] ← v
    PrintSP(G, t)
end
```


Θ(n) time



Caution!

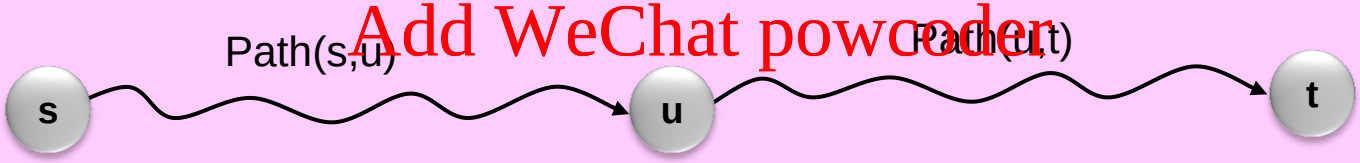
The Optimum Sub-Structure Property (OSSP) is vital.

Example: **Optimum** simple paths in a given **positively** weighted graph G.

1. **Longest** Simple Paths. (simplicity excludes cycles on the path)
2. **Shortest** Simple Paths.

The 1st one does NOT have the OSSP, but the 2nd does!

Path(s,t) = ⟨Path(s,u) , Path(u,t)⟩ (u is an intermediate vertex on the path)



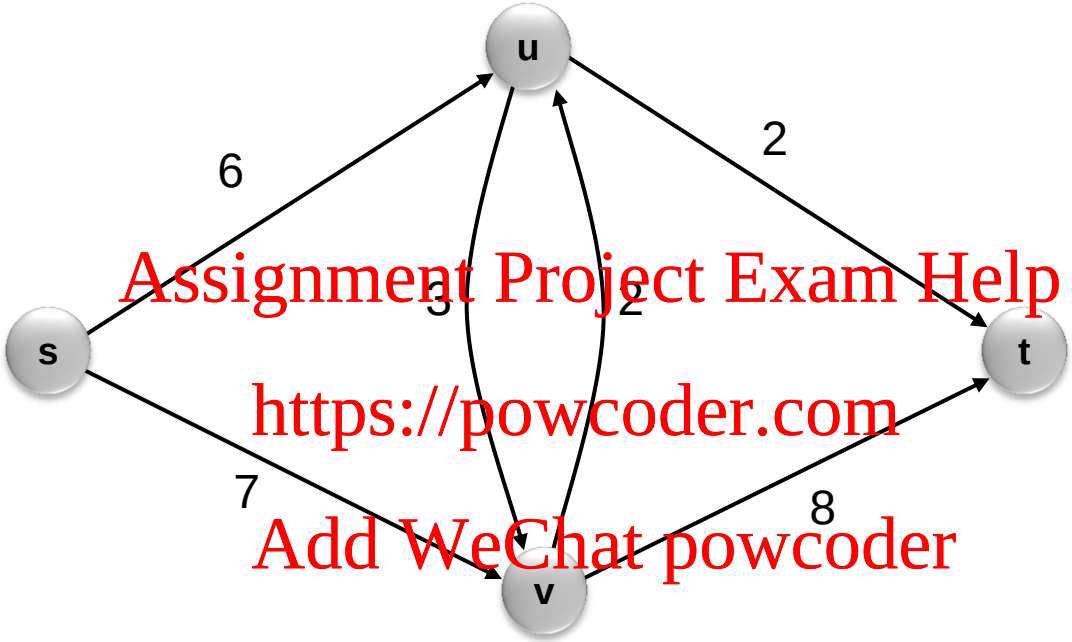
Path(s,t) is longest s-t path does NOT imply that Path(s,u) is longest s-u path, nor that Path(u,t) is longest u-t path.

Path(s,t) is shortest s-t path does imply that Path(s,u) is shortest s-u path, and that Path(u,t) is shortest u-t path.

Explain the discrepancy!

In the 1st case, sub-instances interfere with each other (not independent).

Caution!



⟨s, u, v, t⟩ is the longest simple s-t path,
but the sub-path ⟨s, u⟩ is not the longest simple s-u path!
Longest simple s-u path is ⟨s, v, u⟩.
Replacing ⟨s, u⟩ by ⟨s, v, u⟩ in ⟨s, u, v, t⟩ results in the non-simple path ⟨s, v, u, v, t⟩.
WITHOUT CYCLES IN G, THIS WON'T HAPPEN!

WEIGHTED EVENT SCHEDULING

A banquet hall manager has received a list of reservation requests for the exclusive use of her hall for specified time intervals. Each reservation also indicates how much they are willing to pay.

She wishes to make the most profit by granting a number of reservation requests that have no time overlap conflicts.

Help her select the maximum profitable conflict free time intervals.

Weighted Event Scheduling

Input: A set $S = \{ I_1, I_2, \cdots, I_n\}$ of n weighted event time-intervals $I_k = \langle s_k , f_k , w_k \rangle$, k =1..n, where $s_k < f_k$ are start and finishing times, and $w_k > 0$ is the weight of I_k.

Feasible Solutions: Any subset $C \subseteq S$ with no overlapping pair of intervals.

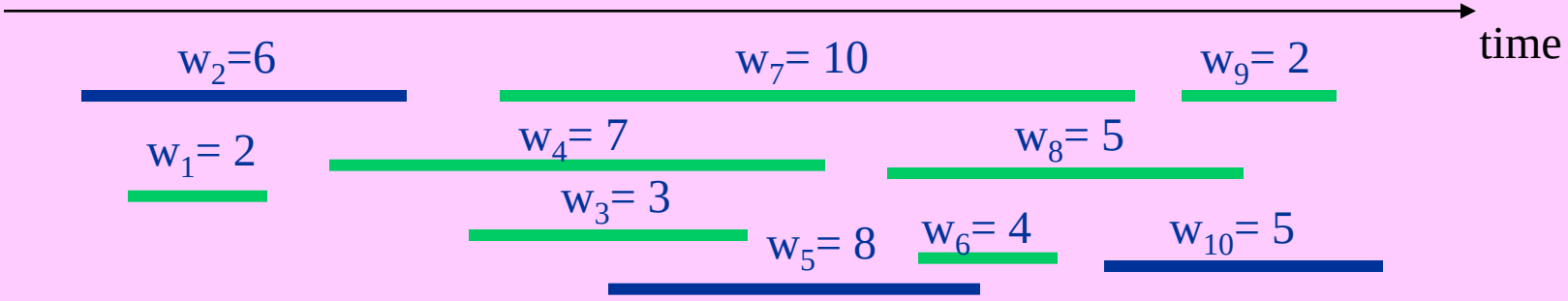
Objective Value: W(C) = sum of the weights of intervals in C.

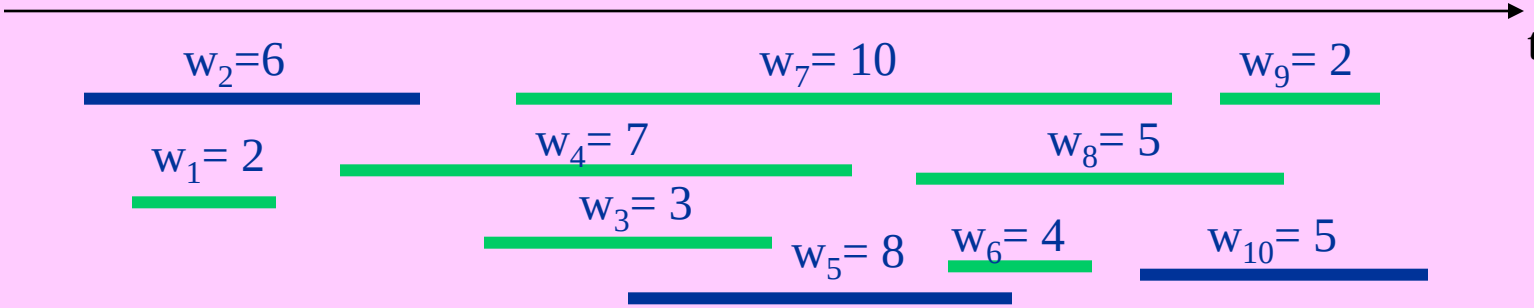
Goal: Output a feasible solution C with maximum W(C).

Reminder: We studied a greedy solution for the un-weighted case, i.e., $w_k = 1$, for k=1..n.

Example:

S = the weighted intervals shown below,
C = the set of blue intervals (happens to be the unique optimum),
W(C) = 6 + 8 + 5 = 19. (Note: the greedy strategy fails here.)



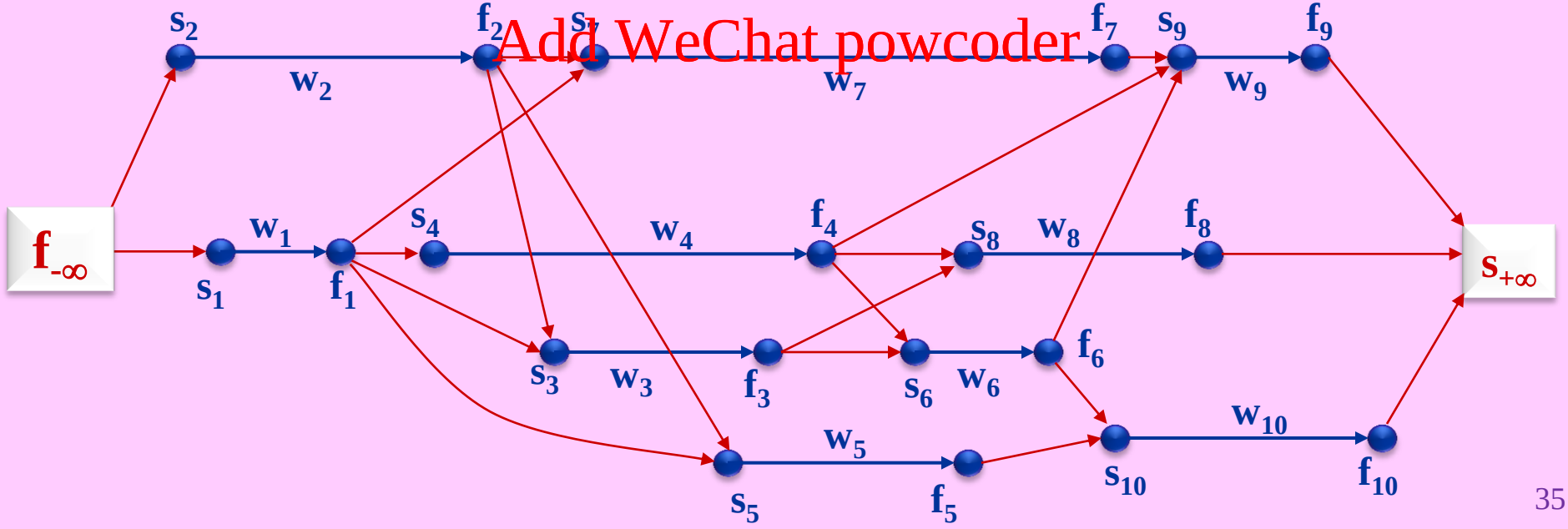


Implicit reduction to a special graph path problem:

$I_k = \langle s_k , f_k , w_k \rangle$ ⇒ s_k w_k f_k ; $f_k \le s_m$ ⇒ f_k **0** s_m

ignore transitive edges

Find longest path from $f_{-\infty}$ to $s_{+\infty}$:



Longest path in the graph

The digraph is acyclic ⇒ OSSP holds for longest paths.

LP(u) = Longest path from source to node u.
W(u) = weight (or length) of LP(u) [$W(f_{-\infty}) = 0$]
$\pi(u)$ = predecessor of node u on LP(u).

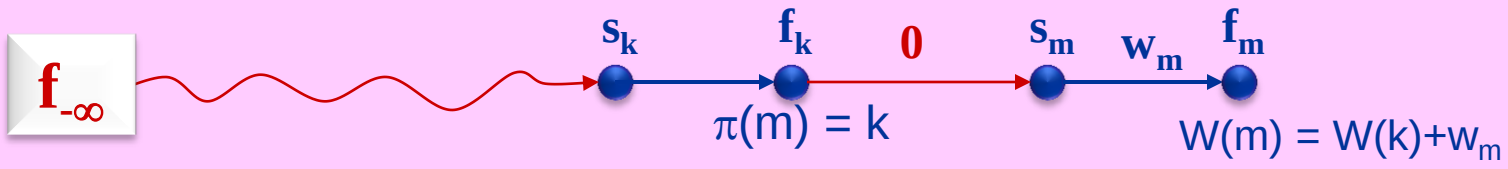
FACT: $LP(u) = \langle LP(\pi(u)), u \rangle$, where

$\pi(s_m) = \text{argmax}_k \{ W(f_k) \mid (f_k, s_m) \text{ is an edge in } G \}$

$W(s_m) = W(\pi(s_m)) = \max_k \{ W(f_k) \mid (f_k, s_m) \text{ is an edge in } G \}$

$W(f_m) = W(s_m) + w_m$

$\text{Memo}(I_m) = \langle \pi(m), W(m) \rangle = \langle \pi(s_m), W(f_m) \rangle$.



Scan Order: Sort starting & finishing times, and scan nodes in that order (as if you were working with a layered graph).

Algorithm **OptWeightEventSchedule**($S = \{I_k = \langle s_k , f_k , w_k \rangle \mid k = 1 \,..n\}$)

Pre-Cond: S is an instance of the weighted event scheduling problem.

Post-Cond: The optimum schedule C is returned.

EventNodes $\leftarrow \{\langle s_k , k\rangle \mid k = 1 \,..n\} \cup \{\langle f_k , k\rangle \mid k = 1 \,..n\}$

Sort **EventNodes** in ascending order of the first component

$\langle \pi node , \pi W\rangle \leftarrow \langle nil , 0\rangle$ § memo of opt path ending just before the next node

for each $\langle t , k\rangle \in$ **EventNodes**, in sorted order **do**

if $t = s_k$ **then** $\pi(k) \leftarrow \pi node$; $W(k) \leftarrow \pi W + w_k$

else § $t = f_k$

if $\pi W < W(k)$ **then** $\langle \pi node , \pi W\rangle \leftarrow \langle k , W(k)\rangle$

end-for

$C \leftarrow \varnothing$; $k \leftarrow \pi node$

while $k \neq nil$ **do** $C \leftarrow C \cup \{I_k\}$; $k \leftarrow \pi(k)$

return C

end

O(n log n) time

Dynamic Programming

Alice:

Now I see that I can save a great deal of time
by using some extra amount of memory space to take memos.
Using that, I can avoid reworking on repeated sub-instances.

Bob:

(1) Make sure the VIP herself, Mrs. OSSP, is present!
OSSP is needed to express solutions of larger sub-instances in terms of smaller ones.
(2) You can choose recursive top-down or iterative bottom up.
(3) If you choose iteration, solve and memoize all possible sub-instances
in reverse order of dependency.
(4) How many distinct sub-instances do you have, polynomial or exponential?
That determines the memo table size and affects time & space complexity.

Dynamic Programming

=

Recursive Back-Tracking + OSSP + Memoized Pruning

Dynamic Programming Design Steps:

1. Think about the recursion tree structure, its post-order evaluation, & how you will sub-divide an instance into (smaller) immediate sub-instances using sub-division option nodes. Identify base cases.
2. Make sure this sub-division into sub-instances satisfies the OSSP.
3. Determine the set of all possible distinct sub-instances that would be recursively generated in the recursion tree. (Moderate over-shooting is OK.)
   Make sure this set is closed under the sub-division in step 1.
4. Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.
5. Starting from the base cases, iteratively fill in the memo table in reverse order of dependency using the recurrence from step 4.
6. Á posteriori, use the memo table to recover & reconstruct the optimum solution to the given main instance.
7. Keep an eye on time & space efficiency.

PROBLEMS

- Fibonacci Numbers (done)
- Shortest Paths in a Layered Network (done)
- Weighted Event Scheduling (done)
- The Knapsack Problem
- Longest Common Subsequence
- Matrix Chain Multiplication
- Optimum Static Binary Search Tree
- Optimum Polygon Triangulation
- More graph problems later.

THE KNAPSACK PROBLEM

- Optimum subset of items that fit in the knapsack.
- Optimum subset of profitable investments with limited budget.

The Knapsack Problem

Input: Items 1..n with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, and knapsack weight capacity W (all positive integers).

Feasible Solutions: Any subset S of the items (to be placed in the knapsack) whose total weight does not exceed the knapsack capacity W.

Objective value: $V(S) = \sum_{k \in S} v_k$.

Goal: Output a feasible solution S with maximum objective value V(S).

Example:

k	1	2	3	4	5	
v_k	28	27	24	7	5	
w_k	9	7	6	6	4	W = 12

Greedy 1: Max v_k first: $S_1 = \{1\}$, $V(S_1) = 28$.

Greedy 2: Max v_k / w_k first: $S_2 = \{3, 5\}$, $V(S_2) = 24 + 5 = 29$.

Optimum: $S_{OPT} = \{2, 5\}$, $V(S_{OPT}) = 27 + 5 = 32$.

0/1 Knapsack Problem

maximize $\sum_{k=1}^{n} v_k x_k$

subject to : (1) $\sum_{k=1}^{n} w_k x_k \leq W$

(2) $x_k \in \{0, 1\}$ for $k = 1..n$.

Interpretation :

$$x_k = \begin{cases} 1 & \text{item } k \text{ selected} \\ 0 & \text{item } k \text{ not selected} \end{cases} \quad \text{for } k = 1..n.$$

WLOGA :

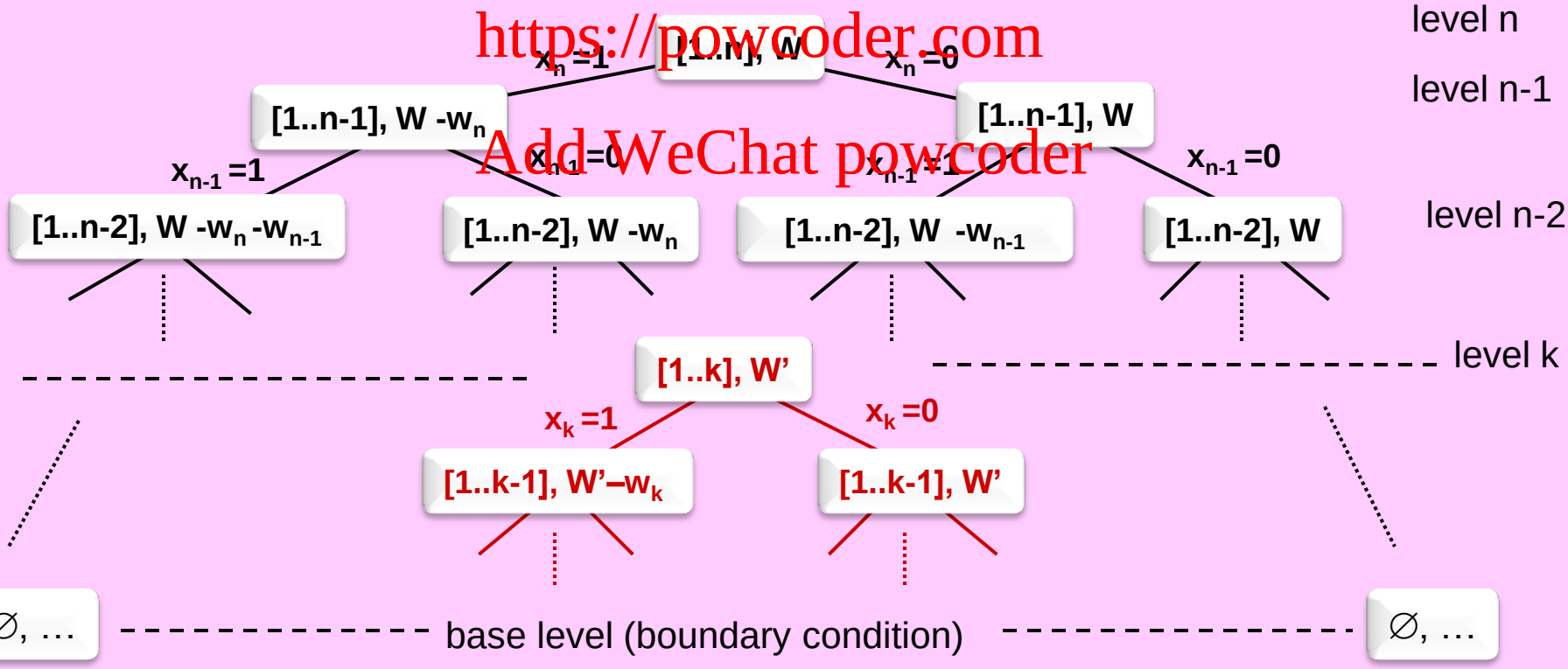
$$(a) \quad \forall k : w_k \leq W$$

$$(b) \quad \sum_{k=1}^{n} w_k > W$$

Dynamic Programming Design Step 1:

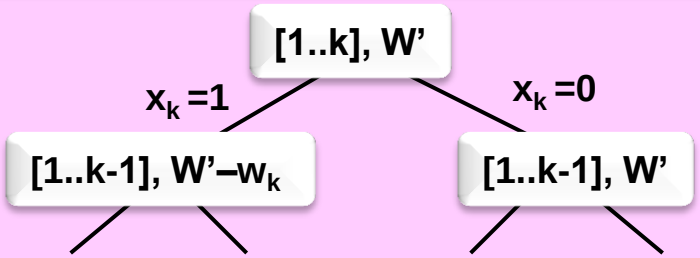
Think about the recursion tree structure, its post-order evaluation, & how you will sub-divide an instance into (smaller) immediate sub-instances using sub-division option nodes. Identify base cases.

- Root of the recursion tree corresponds to the main instance, involving all n items.
- Each node at the k^{th} level of the recursion tree corresponds to a sub-instance concerning the subset of the items [1 .. k]. (Objective value & Knapsack capacity … later!)
- At a node on the k^{th} level, we make a 0/1 decision on the k^{th} variable x_k.
- Each node at that level has 2-way branching:
  - 1-branch: include item k in the solution of that sub-instance;
  - 0-branch: exclude item k in the solution of that sub-instance.



Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.



- **OSSP is satisfied:** Consider an arbitrary sub-instance ([1..k], ⟨v_1, v_2, … , v_k⟩, ⟨w_1, w_2, … , w_k⟩ , W') at level k.

- Suppose ⟨x_1, x_2, … , x_k⟩ ∈ OPT ([1..k], ⟨v_1, v_2, … , v_k⟩, ⟨w_1, w_2, … , w_k⟩ , W')

- Case 0) x_k = 0. Then,

⟨x_1, x_2, …, x_{k-1}⟩ ∈ OPT ([1..k-1], ⟨v_1, v_2, … , v_{k-1}⟩ , ⟨w_1, w_2, … , w_{k-1}⟩ , W').

Case 1) x_k = 1. Then,

⟨x_1, x_2, …, x_{k-1}⟩ ∈ OPT ([1..k-1], ⟨v_1, v_2, … , v_{k-1}⟩ , ⟨w_1, w_2, … , w_{k-1}⟩ , W' – w_k).

- Why? Because if they were not, by cut-&-paste, we could replace them with a better feasible solution ⟨x_1, x_2, …, x_{k-1}⟩ for the respective knapsack capacity. But that would contradict the optimality of ⟨x_1, x_2, …, x_{k-1} , x_k ⟩ for the bigger instance.

Dynamic Programming Design Step 3:

Determine the set of all possible distinct sub-instances that would be recursively generated in the recursion tree. (Moderate over-shooting is OK.)
Make sure this set is closed under the sub-division in step 1.

- Any sub-instance that could possibly be generated in the recursion tree has the form:

  $([1..k], \langle v_1, v_2, \ldots, v_k \rangle, \langle w_1, w_2, \ldots, w_k \rangle, W')$

  for some $k = 0..n$, & some integer $W' \leq W$.

- If $W' < 0$, the corresponding sub-solution has conflict and will not result in any feasible solution. So, we prune the recursion sub-tree at that point immediately.

- Beyond this, we won't further pin-point which specific values of W' will show up. So, we consider any W' in the integer range 0..W.
  This is an over-shoot we will live with.

- So, all the sub-instances are of the form:

  $([1..k], \langle v_1, v_2, \ldots, v_k \rangle, \langle w_1, w_2, \ldots, w_k \rangle, W')$

  for $k = 0..n$, & $W' = 0 .. W$.

- k = 0 or W' = 0 form the boundary cases. Why?

Dynamic Programming Design Step 4:

Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.

$V_{OPT}(k, W')$ = the optimum solution value for the instance $([1..k], \langle v_1, v_2, \ldots, v_k \rangle, \langle w_1, w_2, \ldots, w_k \rangle, W')$, $k = 0..n$, $W' = 0..W$.

DP recurrence :

$$V_{OPT}(k, W') = \begin{cases} -\infty & \text{if } W' < 0 \\ 0 & \text{else if } k = 0 \text{ or } W' = 0 \\ \max\begin{Bmatrix} V_{OPT}(k-1, W'), \\ v_k + V_{OPT}(k-1, W'-w_k) \end{Bmatrix} & \text{for } k = 1..n \ \& \ W' = 1..W \end{cases}$$

Memo table: (n+1)×(W+1) matrix $V_{OPT}[0..n, 0..W]$.

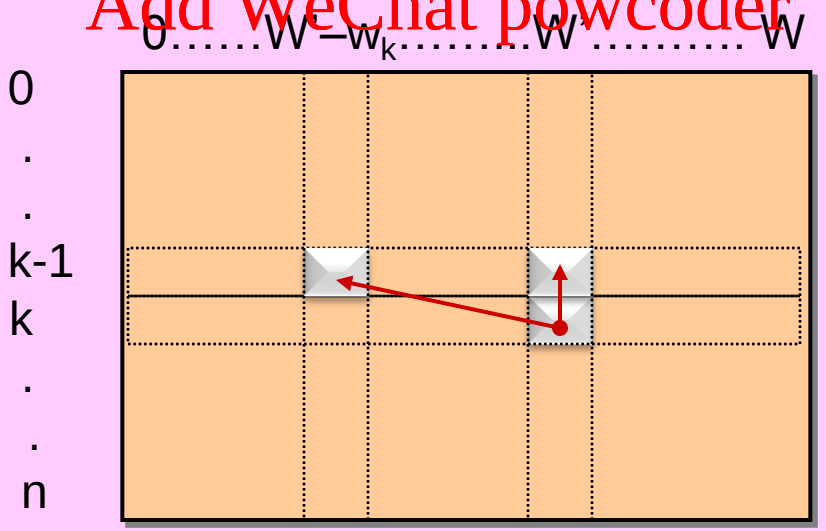
DP recurrence:

$$V_{OPT}(k,W') = \begin{cases} -\infty & \text{if } W' < 0 \\ 0 & \text{else if } k = 0 \text{ or } W' = 0 \\ \max\left\{ V_{OPT}(k-1,W'),\ v_k + V_{OPT}(k-1,W'-w_k) \right\} & \text{for } k = 1..n \ \&\ W' = 1..W \end{cases}$$

Boundary condition: row 0 and column 0

Memo table: $(n+1)\times(W+1)$ matrix $V_{OPT}[\,0..n,\ 0..W]$.

V_{OPT}:



the 2 arrows show dependence

Dynamic Programming Design Step 5:

Starting from the base cases, iteratively fill in the memo table in reverse order of dependency using the recurrence from step 4.

Algorithm Knapsack($\langle v_1, v_2, \ldots, v_n \rangle, \langle w_1, w_2, \ldots, w_n \rangle$, W)

Pre-Cond: input is a valid instance of the Knapsack Problem

Post-Cond: optimum 0/1 knapsack solution $\langle x_1, x_2, \ldots, x_n \rangle$ is returned

```
for W' ← 0 .. W   do  V_OPT[0,W'] ← 0                    § boundary cases first
for k ← 0 .. n    do  V_OPT[k,0] ← 0                     § boundary cases first

for k ← 1 .. n   do                                      § in a reverse order of dependency
    for W' ← 1 .. W  do
        V_OPT[k,W'] ← V_OPT[k-1,W']
        if w_k ≤ W' then if   V_OPT[k,W'] < v_k + V_OPT[k-1, W' – w_k ]
                         then V_OPT[k,W'] ← v_k + V_OPT[k-1, W' – w_k ]
    end-for
end-for

return OptSol(⟨v_1, v_2, ... , v_n⟩, ⟨w_1, w_2, ... , w_n⟩ , W, V_OPT)
end
```

Θ(nW) time & space

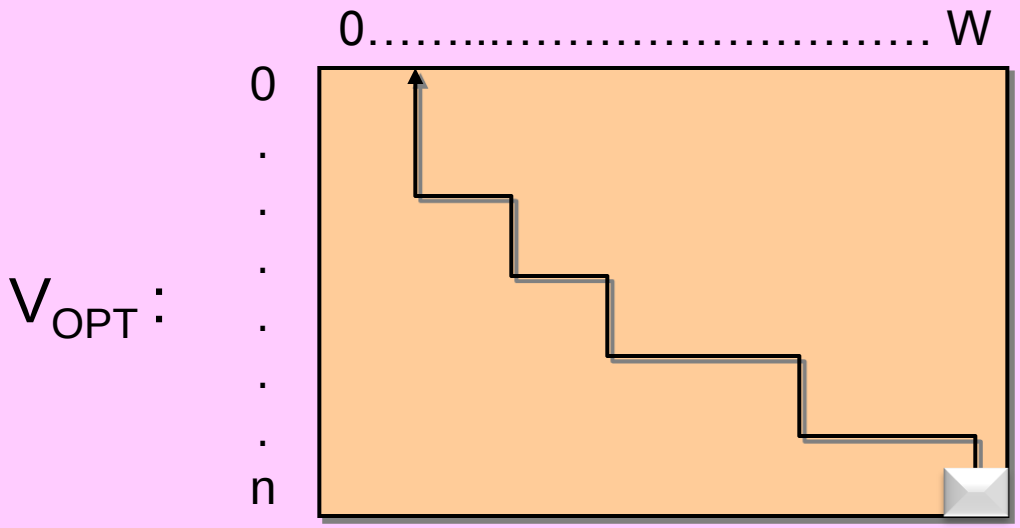
Dynamic Programming Design Step 6:

Á posteriori, use the memo table to recover & reconstruct the optimum solution to the given main instance.

Algorithm OptSol($\langle v_1, v_2, \ldots, v_n \rangle$, $\langle w_1, w_2, \ldots, w_n \rangle$, W, V_{OPT})

```
W' ← W
for k ← n downto 1 do                    § work backwards on the memo table
    if V_OPT[k,W'] = V_OPT[k-1,W']
        then x_k ← 0;
        else x_k ← 1; W' ← W' – w_k
end-for
return (⟨x_1, x_2, … , x_n⟩)
end
```


Θ(n) time



Dynamic Programming Design Step 7:

Keep an eye on time & space efficiency.

Time = Θ(nW), Space = Θ(nW).

This is not polynomial but exponential. Input number W requires log W bits.

We don't know of any polynomial time algorithm for the 0/1 Knapsack Problem.

We shall later see that 01KP is one of the so called NP-hard problems.

What other options do we have?

1. Find good approximations to the optimum solution (done in EECS 4101).

2. Consider relaxing the feasibility constraints to allow fractional values for the variables (not just 0/1 values).
   GREEDY 2 (already discussed) is correct for the fractional case!
   ……. P.T.O.

0/1 vs Fractional KP

(01KP):

$$\text{maximize} \quad \sum_{k=1}^{n} v_k x_k$$

$$\text{subject to :} \quad (1) \quad \sum_{k=1}^{n} w_k x_k \le W$$

$$(2) \quad x_k \in \{0,1\} \quad \text{for } k = 1..n.$$

(FKP) a constraint relaxation of 01KP :

$$\text{maximize} \quad \sum_{k=1}^{n} v_k x_k$$

$$\text{subject to :} \quad (1) \quad \sum_{k=1}^{n} w_k x_k \le W$$

$$(2') \quad 0 \le x_k \le 1 \quad \text{for } k = 1..n.$$

The Fractional KP

(FKP):

$$\text{maximize} \quad \sum_{k=1}^{n} v_k x_k$$

$$\text{subject to :} \quad (1) \quad \sum_{k=1}^{n} w_k x_k \le W$$

$$(2) \quad 0 \le x_k \le 1 \quad \text{for } k = 1..n.$$

FACT: FKP optimum solution can be obtained in O(n) time.

Proof (sketch):

- GREEDY 2: Consider items in non-increasing order of v_k / w_k.
  Place the items in the knapsack in that order until it is filled up.
- Only the last item placed in the knapsack may be fractionally selected.
- The first step can be done by sorting in O(n log n) time.
  However, instead of sorting we can use weighted selection in O(n) time
  to find out which item is placed in the knapsack last.
  (See our earlier lecture slides on the topic of selection & the exercises thereof.)

Exercise: Complete details of this algorithm & its proof of correctness.

LONGEST COMMON SUBSEQUENCE

The Longest Common Subsequence Problem

Input: Two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

Output: LCS(X,Y) = the longest common subsequence of X and Y.

Applications: DNA sequencing, DIFF in text editing, pattern matching, …

Example 1: ⟨ A A B ⟩ AAB is a sub-sequence of ABBACB

⟨ A B B A C B ⟩

Example 2: X = ⟨ A C B B A D ⟩ LCS(X,Y) = ⟨ A B A ⟩

Y = ⟨ A A A B C A C ⟩ ⟨A C A⟩ is also a solution

Easier Problem:

Is $X = \langle x_1, x_2, \ldots, x_m \rangle$ a subsequence of $Y = \langle y_1, y_2, \ldots, y_n \rangle$?

This can be checked in O(n) time. HOW?

Greedy & Brute-Force

GREEDY: To find LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$,
Match the first element x_1 of X with its earliest occurrence in Y.
Match the first element y_1 of Y with its earliest occurrence in X.
Take the better of these to options, the earliest match first.

Example:

$X = \langle D \quad A \quad B \rangle$

$Y = \langle A \quad B \quad B \quad A \quad C \quad D \rangle$

Choose the A-A match first.

End result: $\langle A\ B \rangle$ (which happens to be correct).

Counter-example:

$X = \langle D \quad A \quad D \quad A \quad D \quad B \rangle$

$Y = \langle B \quad B \quad A \quad A \quad C \quad D \rangle$

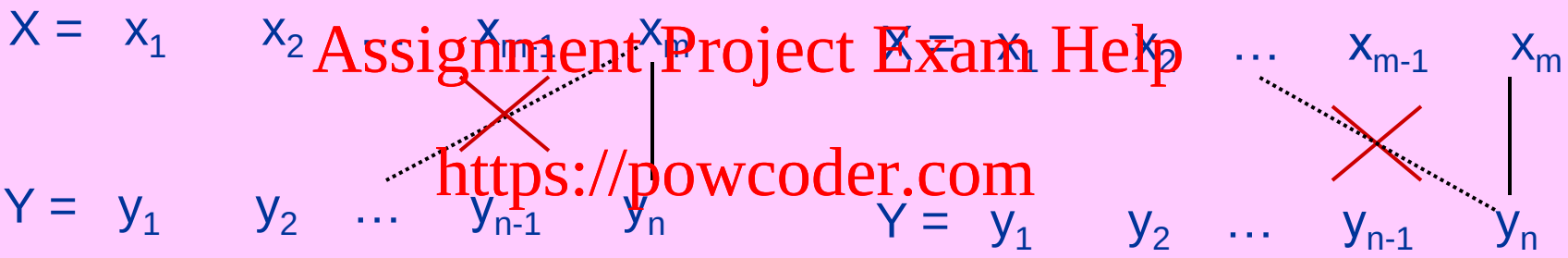
$LCS(X,Y) = \langle A\ A\ D \rangle$

Brute-Force Solution: WLOGA $m \leq n$.
Generate each of the 2^m sub-sequences of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and see if they are also a subsequence of Y, and remember the longest common subsequence.
This takes exponential time $O(n2^m)$.

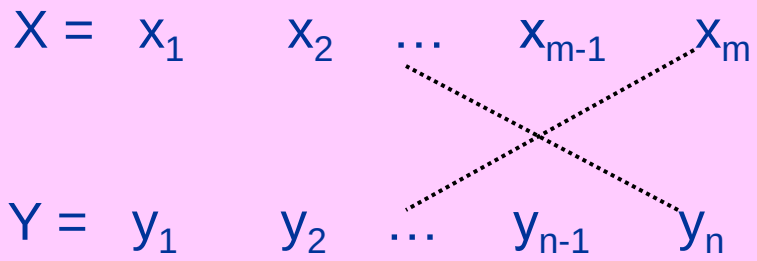
Key Observation

Need to decide on the last element of each sequence. Include or exclude it in the LCS solution?

Case 1: $[x_m = y_n]$: Then it's safe to match them (include both in LCS).



Case 2: $[x_m \neq y_n]$: Then at least one must be excluded from LCS.



Note: excluding one does not mean including the other!

Dynamic Programming Design Step 1:

Think about the recursion tree structure, its post-order evaluation, & how you will sub-divide an instance into (smaller) immediate sub-instances using sub-division option nodes. Identify base cases.

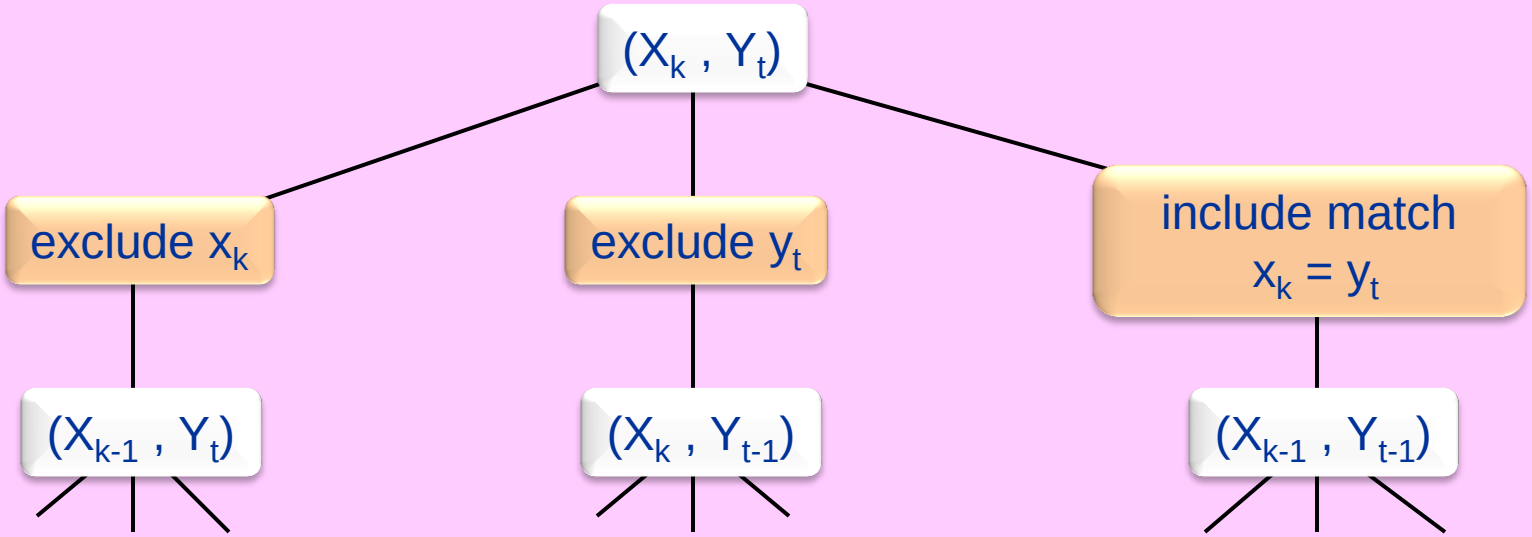
Main instance: LCS (X_m , Y_n), $X_m = \langle x_1, x_2, \dots , x_m \rangle$, $Y_n = \langle y_1, y_2, \dots , y_n \rangle$.

General Instance: LCS (X_k , Y_t), prefixes $X_k = \langle x_1, x_2, \dots , x_k \rangle$, $Y_t = \langle y_1, y_2, \dots , y_t \rangle$, for $k = 0..m$, $t = 0..n$.

3 sub-division options: should the last elements x_k and y_t be part of the LCS solution?

Immediate sub-instances: LCS (X_{k-1} , Y_t), LCS (X_k , Y_{t-1}), LCS (X_{k-1} , Y_{t-1}).

Leaves: Base cases LCS (X_0 , Y_t), $t = 0..n$, and LCS (X_k , Y_0), $k = 0..m$.



Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.

- **OSSP is satisfied** based on the Key Observation.

  Suppose $Z_p = LCS(X_k, Y_t)$.

- Case 1: $[x_k = y_t]$: Then $z_p = x_k = y_t$ and $Z_{p-1} = LCS(X_{k-1}, Y_{t-1})$.
- Case 2: $[x_k \neq y_t]$:

  Case 2a: $[z_p \neq x_k]$: Then $Z_p = LCS(X_{k-1}, Y_t)$.

  Case 2b: $[z_p \neq y_t]$: Then $Z_p = LCS(X_k, Y_{t-1})$.

[Note: cases 2a and 2b are not mutually exclusive.]

Dynamic Programming Design Step 3:

Determine the set of all possible distinct sub-instances that would be recursively generated in the recursion tree. (Moderate over-shooting is OK.)
Make sure this set is closed under the sub-division in step 1.

General Instance:

LCS (X_k, Y_t), prefixes $X_k = \langle x_1, x_2, \ldots, x_k \rangle$, $Y_t = \langle y_1, y_2, \ldots, y_t \rangle$,
$k = 0..m$, $t = 0..n$.

There are $(m+1)\times(n+1)$ sub-instances.

Dynamic Programming Design Step 4:

Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.

Define $C_{LCS}[k,t]$ = length of LCS(X_k , Y_t), and

$O_{LCS}[k,t]$ = last element option in LCS(X_k , Y_t) ∈ { "↑", "←" , "↖" }

$$C_{LCS}[k,t] = \begin{cases} 0 & \text{if } k = 0 \text{ or } t = 0 \\ 1 + C_{LCS}[k-1, t-1] & \text{if } k > 0, t > 0, x_k = y_t \\ \max\{ C_{LCS}[k-1,t], C_{LCS}[k,t-1] \} & \text{if } k > 0, t > 0, x_k \neq y_t \end{cases}$$

$$O_{LCS}[k,t] = \begin{cases} nil & \text{if } k = 0 \text{ or } t = 0 \\ "\nwarrow" & \text{if } k > 0, t > 0, x_k = y_t \\ "\uparrow" \text{ , } "\leftarrow" & \text{if } k > 0, t > 0, x_k \neq y_t \end{cases}$$

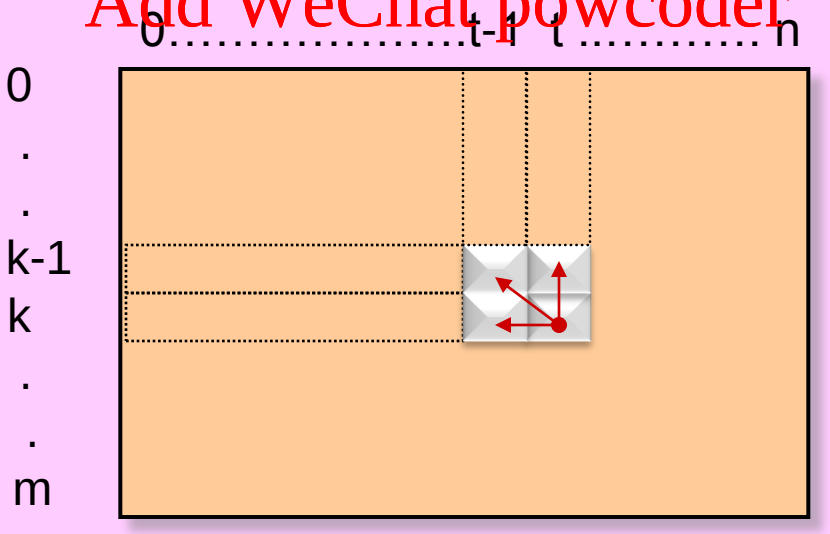
Memo table: 2 (m+1)×(n+1) matrices C_{LCS}[0..m, 0..n] & O_{LCS}[0..m, 0..n] .

$$C_{LCS}[k,t] = \begin{cases} 0 & \text{if } k=0 \text{ or } t=0 \\ 1 + C_{LCS}[k-1,t-1] & \text{if } k>0, t>0, x_k = y_t \\ \max\{C_{LCS}[k-1,t], C_{LCS}[k,t-1]\} & \text{if } k>0, t>0, x_k \neq y_t \end{cases}$$

$$O_{LCS}[k,t] = \begin{cases} nil & \text{if } k=0 \text{ or } t=0 \\ "\nwarrow" & \text{if } k>0, t>0, x_k = y_t \\ "\uparrow" \text{ or } "\leftarrow" & \text{if } k>0, t>0, x_k \neq y_t \end{cases}$$

Memo table: 2 (m+1)×(n+1) matrices C_{LCS}[0..m, 0..n] & O_{LCS}[0..m, 0..n] .

C_{LCS} & O_{LCS} :



3 arrows show dependence

Dynamic Programming Design Step 5:

Starting from the base cases, iteratively fill in the memo table in reverse order of dependency using the recurrence from step 4.

Algorithm LCS($X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$)
Pre-Cond: input is a valid instance of the LCS Problem
Post-Cond: LCS(X,Y) is printed

```
for k ← 0 .. m   do  C_LCS[k,0] ← 0          § boundary cases first
for t ← 1 .. n   do  C_LCS[0,t] ← 0          § boundary cases first

for k ← 1 .. m   do
    for t ← 1 .. n  do                       § in a reverse order of dependency
        if x_k = y_t
        then  C_LCS[k,t] ← 1+C_LCS[k-1,t-1];   O_LCS[k,t] ← "↖"
        else if  C_LCS[k-1,t] ≥ C_LCS[k,t-1]
              then C_LCS[k,t] ← C_LCS[k-1,t];   O_LCS[k,t] ← "↑"
              else  C_LCS[k,t] ← C_LCS[k,t-1];   O_LCS[k,t] ← "←"
    end-for
end-for
PrintLCS(O_LCS , X , m, n)
end
```


Θ(nm) time & space

C_{LCS} / O_{LCS}		Y	a	a	b	a	c	b	a
		0	1	2	3	4	5	6	7
X	0	0	0	0	0	0	0	0	0
a	1	0	↖ 1	↖ 1	← 1	↖ 1	← 1	← 1	↖ 1
b	2	0	↑ 1	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2
c	3	0	↑ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3	← 3
c	4	0	↑ 1	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↑ 3
a	5	0	↖ 1	↖ 2	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4

Dynamic Programming Design Step 6:

Á posteriori, use the memo table to recover & reconstruct the optimum solution to the given main instance.

Algorithm **PrintLCS**(O_{LCS} , X , k, t)

if k = 0 or t = 0 **then return**

if $O_{LCS}[k,t]$ = "↖" **then** **PrintLCS**(O_{LCS} , X , k-1, t-1); print x_k

else if $O_{LCS}[k,t]$ = "↑" **then** **PrintLCS**(O_{LCS} , X , k-1, t)

else **PrintLCS**(O_{LCS} , X , k, t-1)

Θ(m+n) time

LCS(X,Y) = a b c a

C_{LCS} / O_{LCS}		Y	a	a	b	a	c	b	a
		0	1	2	3	4	5	6	7
X	0	0	0	0	0	0	0	0	0
a	1	0	1	1	← 1	1	← 1	←1	1
b	2	0	1	1	2	←2	← 2	2	← 2
c	3	0	1	1	2	2	3	←3	← 3
c	4	0	1	1	2	2	3	3	3
a	5	0	1	2	2	3	3	3	4

Dynamic Programming Design Step 7:

Keep an eye on time & space efficiency.

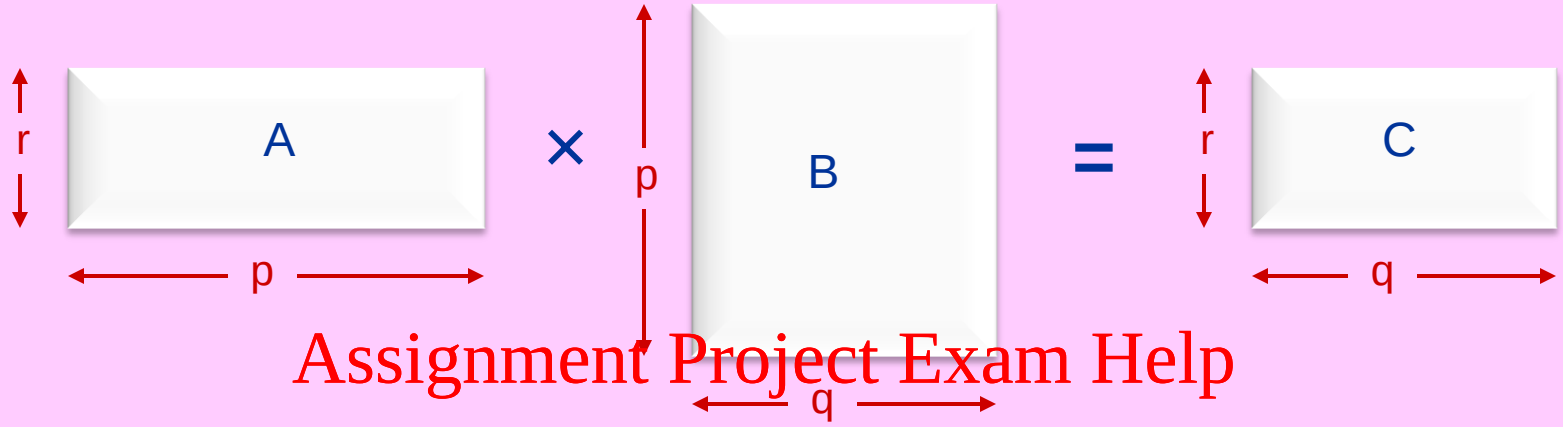
Time = Θ(mn), Space = Θ(mn).

Any improvement?

Sub-quadratic solutions exit for certain special cases of the problem.
See the **Bibliography** & **Exercise** sections.

MATRIX CHAIN MULTIPLICATION

The Problem



Cost of matrix multiplication = rpq (scalar multiplications).

Matrix multiplication is associative (but not commutative).

PROBLEM: What is the minimum cost multiplication order for:

$$\underset{p_0 \times p_1}{(A_1} \times \underset{p_1 \times p_2}{A_2} \times \ldots \underset{p_{k-1} \times p_k}{A_k} \times \underset{p_k \times p_{k+1}}{A_{k+1}} \ldots \times \underset{p_{n-1} \times p_n}{A_n)} \quad ?$$

That is, find the minimum cost full parenthesization of this expression.

The Matrix Chain Multiplication Problem

Input: The dimension sequence $\langle p_0, p_1, \ldots, p_n \rangle$ of the matrix chain multiplication expression $(A_1 \times A_2 \times \ldots A_k \times A_{k+1} \ldots \times A_n)$, where A_k is a $p_{k-1} \times p_k$ matrix, k=1..n.

Output: An optimum full parenthesization of the input expression.

Goal: Minimize total scalar multiplication cost.

Task: a (recursive) sequence of decisions.

Top level decision: which multiplication should be done **LAST**?

If it is the k^{th} multiplication, then we have:

$$((A_1 \times A_2 \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_n)).$$

Cost of that single **LAST** multiplication will be $\mathbf{p_0 \times p_k \times p_n}$.

Then, recursively we have to decide on the left and right sub-expressions.

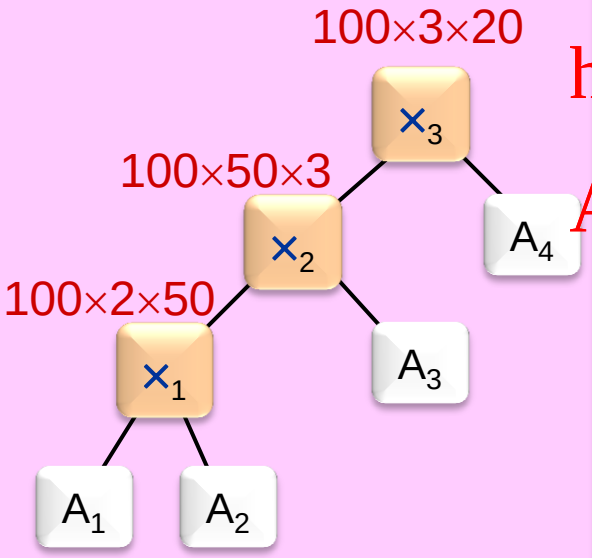
Example

$A_1 \times_1 A_2 \times_2 A_3 \times_3 A_4$

100×2 2×50 50×3 3×20

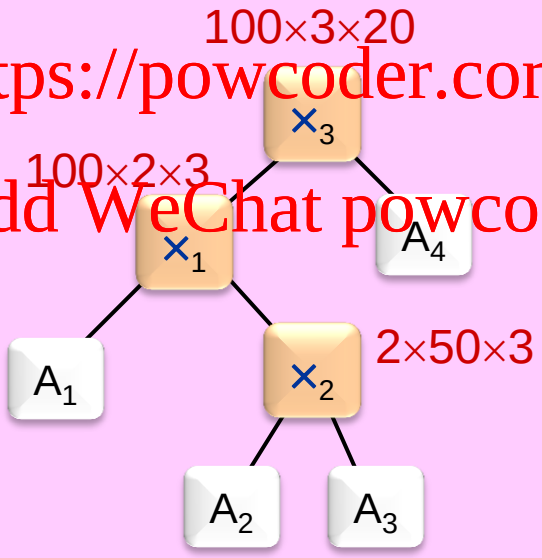
$(((A_1 \times A_2) \times A_3) \times A_4)$

cost = 31,000



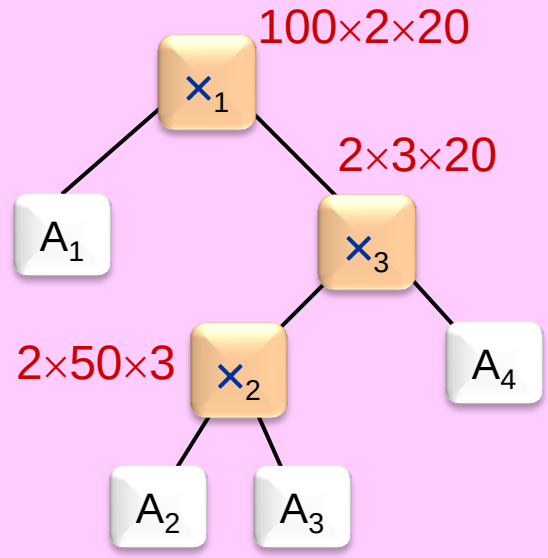
$((A_1 \times (A_2 \times A_3)) \times A_4)$

cost = 6,900



$(A_1 \times ((A_2 \times A_3) \times A_4))$

cost = 4,420



Size of Solution Space

Q: In how many ways can we fully parenthesize $(A_1 \times A_2 \times \ldots A_k \times A_{k+1} \ldots \times A_n)$?

A: The Catalan Number: $P(n) = \frac{1}{n}\binom{2n-2}{n-1} = \Omega\left(\frac{4^n}{n^{1.5}}\right)$.

This is also the number of binary trees with n external nodes.
It satisfies the following recurrence relation [CLRS (15.6), p. 372]:

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n \geq 2 \end{cases}$$

Example: $P(21) = 131{,}282{,}408{,}400.$

A Greedy Heuristic

Notation: $A_{i..j} = A_i \times A_{i+1} \times \ldots A_k \times A_{k+1} \ldots \times A_{j-1} \times A_j$

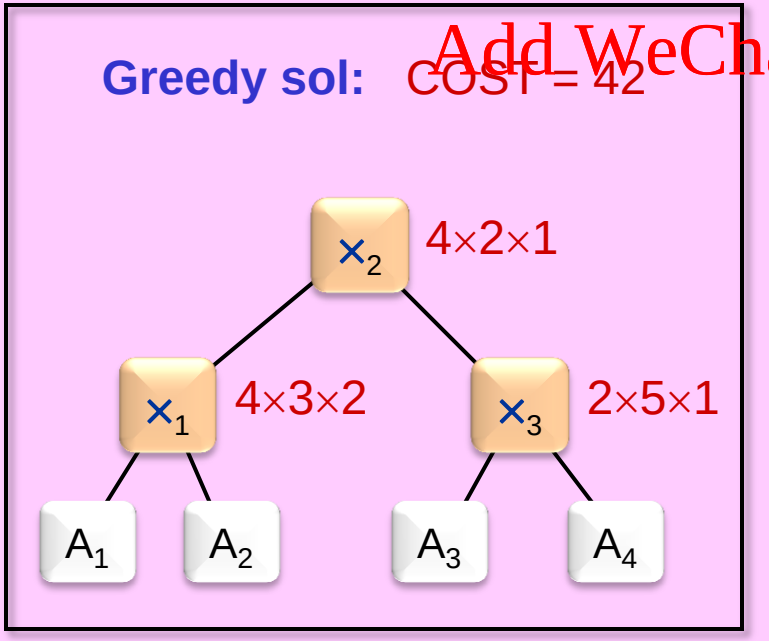
Decision: Which multiplication $\times_k$, for $k \in \{i..j-1\}$, done **LAST**?

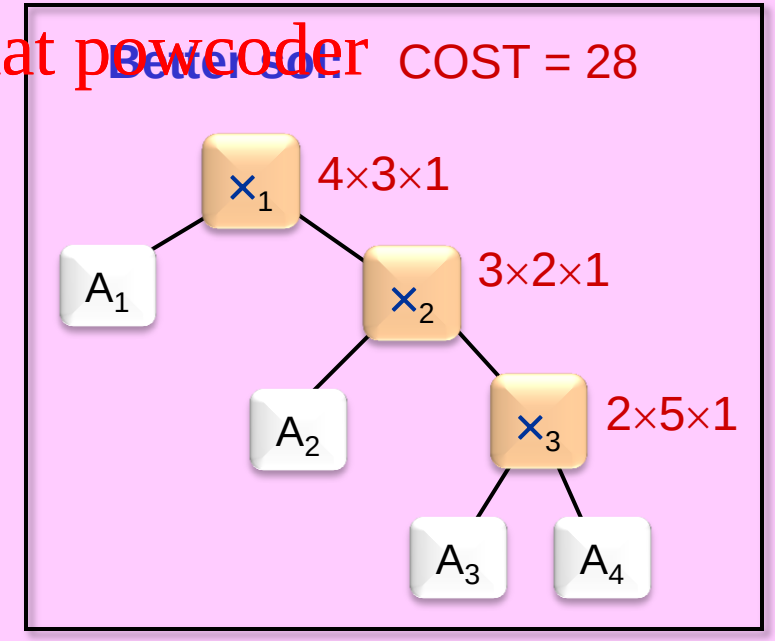
$$A_{i..j} = A_{i..k} \times_k A_{k+1..j}$$

Greedy: Choose $\times_k$ with **min** root cost $p_{i-1} \times p_k \times p_j$, i.e., min $\mathbf{p_k}$.

Counter-example: $A_1 \times_1 A_2 \times_2 A_3 \times_3 A_4$

4×3 $\quad$ 3×2 $\quad$ 2×5 $\quad$ 5×1

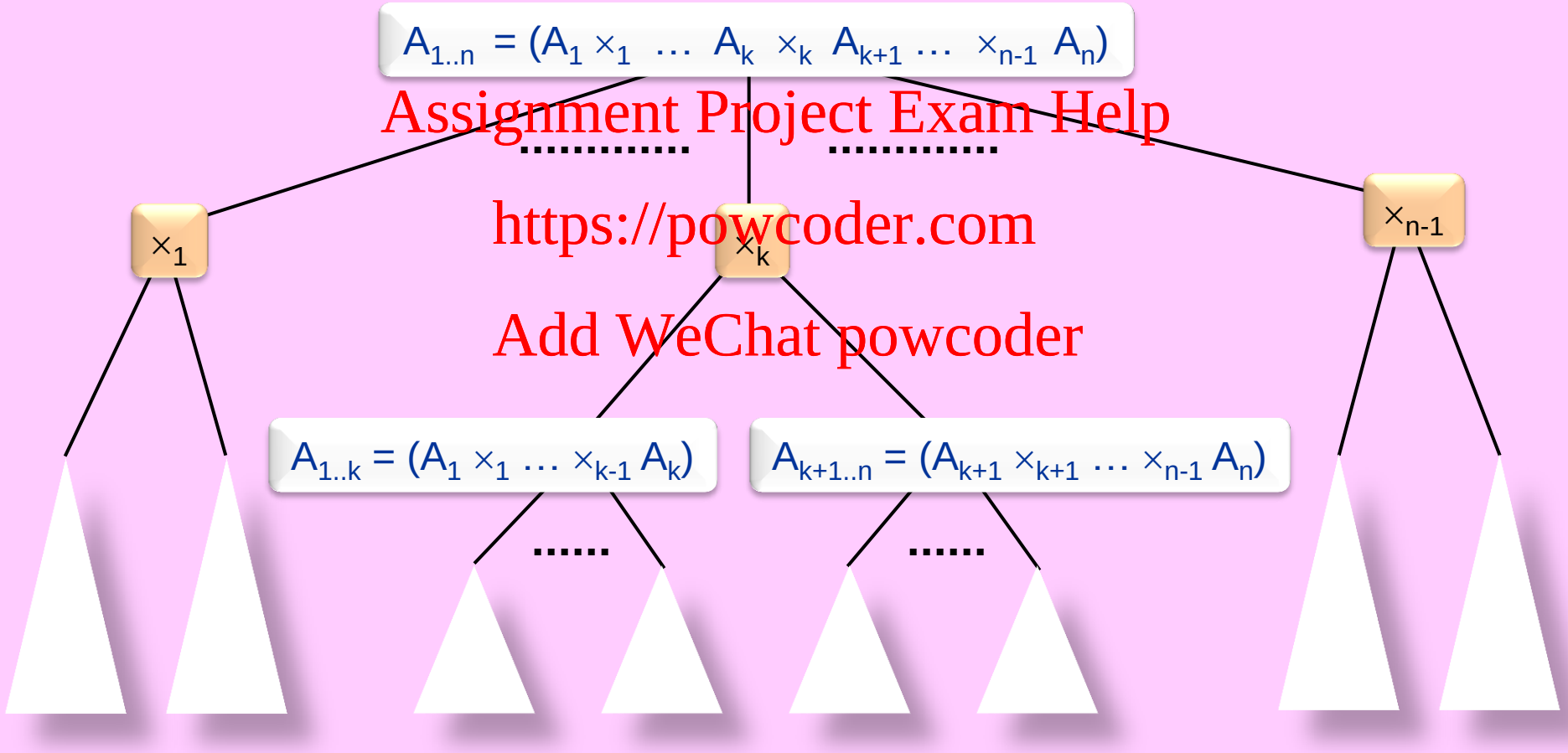




Dynamic Programming Design Step 1:

Think about the recursion tree structure, its post-order evaluation, & how you will sub-divide an instance into (smaller) immediate sub-instances using sub-division option nodes. Identify base cases.

Sub-division decision: which multiplication $\times_k$ should form root of the parse tree?



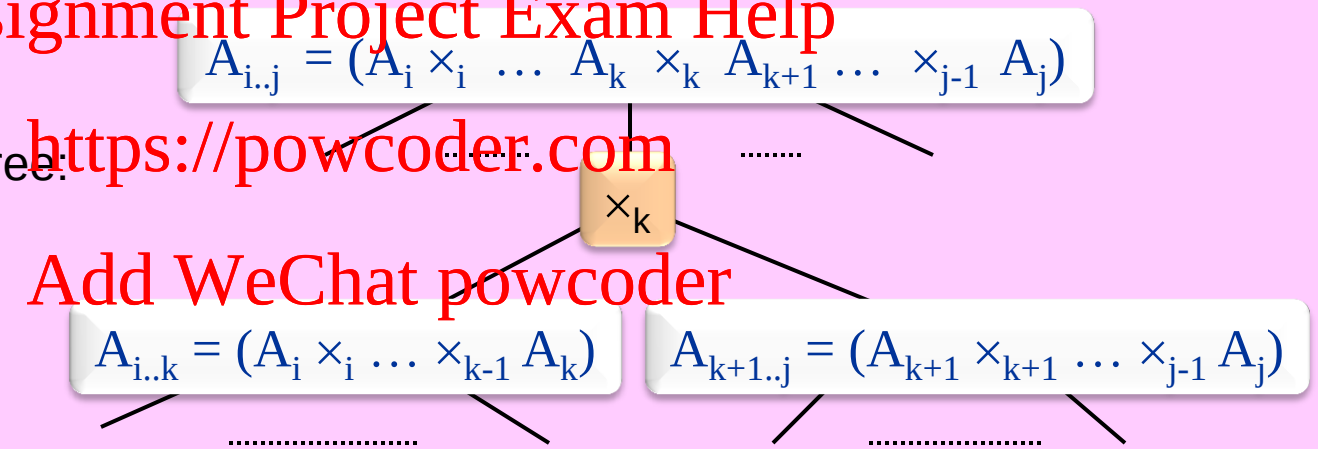
Leaves are expressions of the form $A_{i..i}$ (single operand, no operator).

Dynamic Programming Design Step 2:

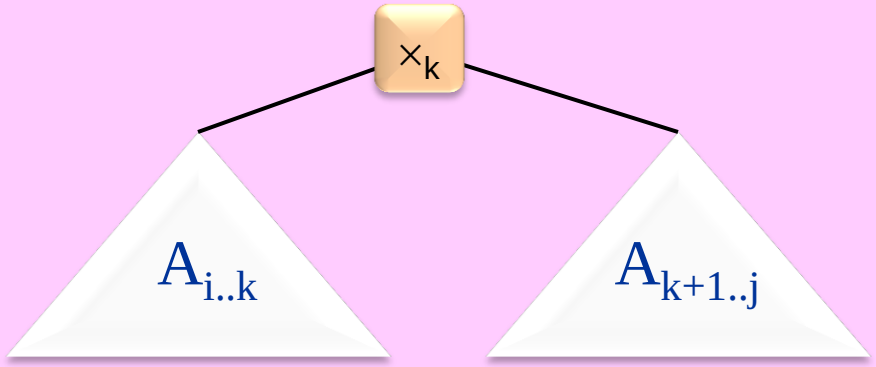
Make sure this sub-division into sub-instances satisfies the OSSP.

- OSSP is satisfied:
  Regardless of which multiplication $\times_k$ forms the root of the parse tree,
  its left and right sub-trees form independent sub-expressions,
  and therefore, must be done in optimum way.
  Otherwise, we can use the cut-&-paste argument to reach contradiction.

Part of recursion tree:



Parse tree for $A_{i..j}$:



Dynamic Programming Design Step 3:

Determine the set of all possible distinct sub-instances that would be recursively generated in the recursion tree. (Moderate over-shooting is OK.)
Make sure this set is closed under the sub-division in step 1.

The main instance: $A_{1..n} = (A_1 \times \ldots \times A_n)$.

General sub-instance:

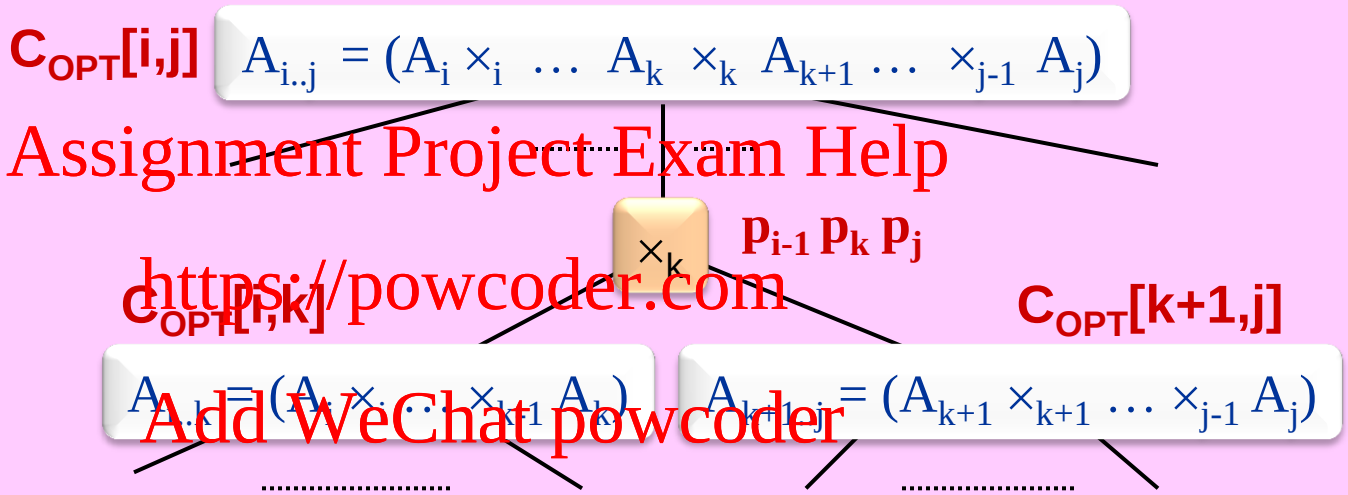
$$A_{i..j} = (A_i \times \ldots \times A_j), \text{ for } 1 \le i \le j \le n.$$

There are $n \times (n+1)/2$ such sub-instances.

Dynamic Programming Design Step 4:

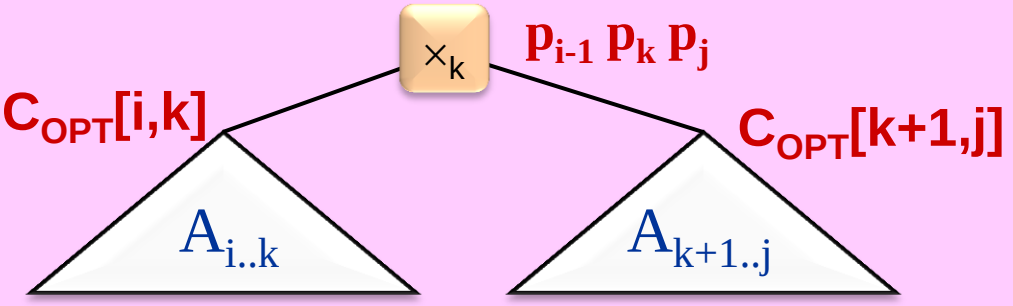
Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.

part of Recursion tree:



$$C_{OPT}[i,j] = \min_k \{ C_{OPT}[i,k] + C_{OPT}[k+1,j] + p_{i-1}\, p_k\, p_j \}$$

OPT Parse tree for $A_{i..j}$:



Dynamic Programming Design Step 4:

Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.

Memo Tables for sub-instance $A_{i..j} = (A_i \times \ldots \times A_j)$, $1 \le i \le j \le n$:

$C_{OPT}[i,j]$ = cost of optimum parse tree of $A_{i..j}$

$R_{OPT}[i,j]$ = root (i.e., index k of last multiplication) of opt parse tree for $A_{i..j}$

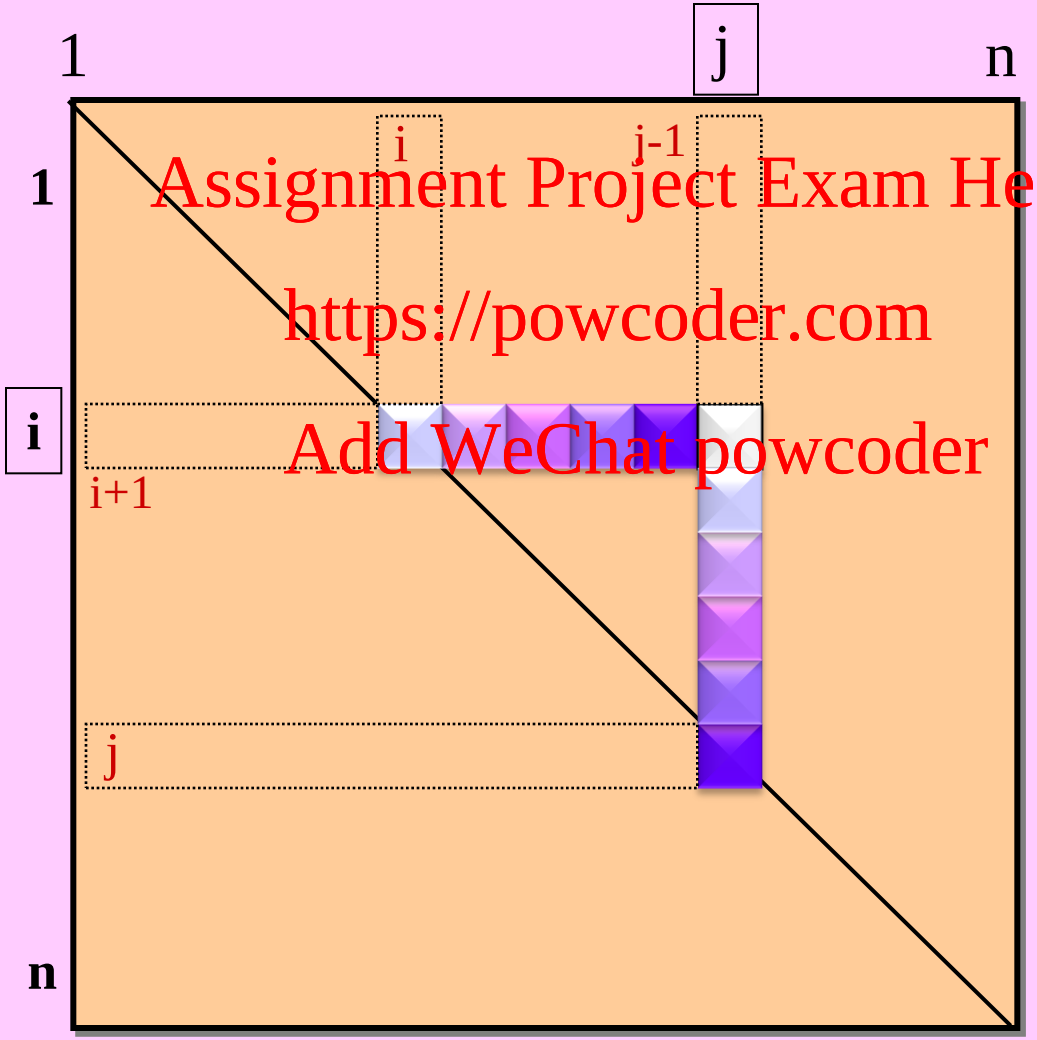
$$C_{OPT}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k=i..j-1} \{ C_{OPT}[i, k] + C_{OPT}[k+1, j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

$$R_{OPT}[i, j] = \begin{cases} \text{nil} & \text{if } i = j \\ \operatorname{argmin}_{k=i..j-1} \{ C_{OPT}[i, k] + C_{OPT}[k+1, j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

sub-instance $A_{i..j} = (A_i \times \ldots \times A_j)$, $1 \le i \le j \le n$

$$C_{OPT}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k=i..j-1} \{ C_{OPT}[i, k] + C_{OPT}[k+1, j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

DEPENDENCE:



A reverse order of dependency for table fill-in order:

j increase major
i decrease minor

Dynamic Programming Design Step 5:

Starting from the base cases, iteratively fill in the memo table in reverse order of dependency using the recurrence from step 4.

Algorithm **OptMatrixChainMultOrder**($\langle p_0, p_1, \ldots, p_n \rangle$)
Pre-Cond: input is a valid instance of the MCM Problem
Post-Cond: Memo Tables C_{OPT} and R_{OPT} returned

```
for j ← 1 .. n   do                    § in a reverse order of dependency
    C_OPT[j,j] ← 0                     § boundary case first
    for i ← j-1  downto 1  do          § in a reverse order of dependency
        C_OPT[i,j] ← ∞                 § instance A_i..j
        for k ← i .. j – 1  do         § option: split A_i..j at k
            newCost ← C_OPT[i,k] + C_OPT[k+1,j] + p_i-1 p_k p_j
            if C_OPT[i,j] > newCost  then  C_OPT[i,j] ← newCost; R_OPT[i,j] ← k
        end-for
    end-for
end-for
return ( C_OPT , R_OPT )
end
```


$\Theta(n^3)$ time
$\Theta(n^2)$ space

Dynamic Programming Design Step 6:

Á posteriori, use the memo table to recover & reconstruct the optimum solution to the given main instance.

Initial call: **MatrixChainMultiply**(A , R_{OPT} , 1 , n)

where $A = \langle A_1 , A_2 , \ldots, A_n \rangle$

Algorithm **MatrixChainMultiply**(A, R_{OPT}, i , j) § $1 \le i \le j \le n$

if $i = j$ **then return** A_i

$k \leftarrow R_{OPT}[i,j]$

$X \leftarrow$ **MatrixChainMultiply**(A, R_{OPT}, i , k)

$Y \leftarrow$ **MatrixChainMultiply**(A, R_{OPT}, k+1 , j)

return MatrixMultiply(X,Y)

end

Dynamic Programming Design Step 7:

Keep an eye on time & space efficiency.

Algorithm OptMatrixChainMultOrder

Time = $\Theta(n^3)$, Space = $\Theta(n^2)$.

Any improvement?

See the Bibliography section.

[T.C. Hu 1982] shows an O(n log n) time solution.

OPTIMUM STATIC BINARY SEARCH TREE

The Problem Setting

- T = A Static Binary Search Tree (static ≡ searches only, no inserts or deletes):
- n nodes holding n keys $K_1 < K_2 < \ldots < K_n$.
- n+1 external nodes $E_0, E_1, \ldots, E_n$ left-to-right.
- $p(i)$ = successful search probability for key K_i, i =1..n.
- $q(i)$ = unsuccessful search probability ending up at external node E_i, i=0..n.
- $\sum_{i=1..n} p(i) + \sum_{i=0..n} q(i) = 1$.
- $Cost(T) =$ expected # key comparisons to search for a given key in T.

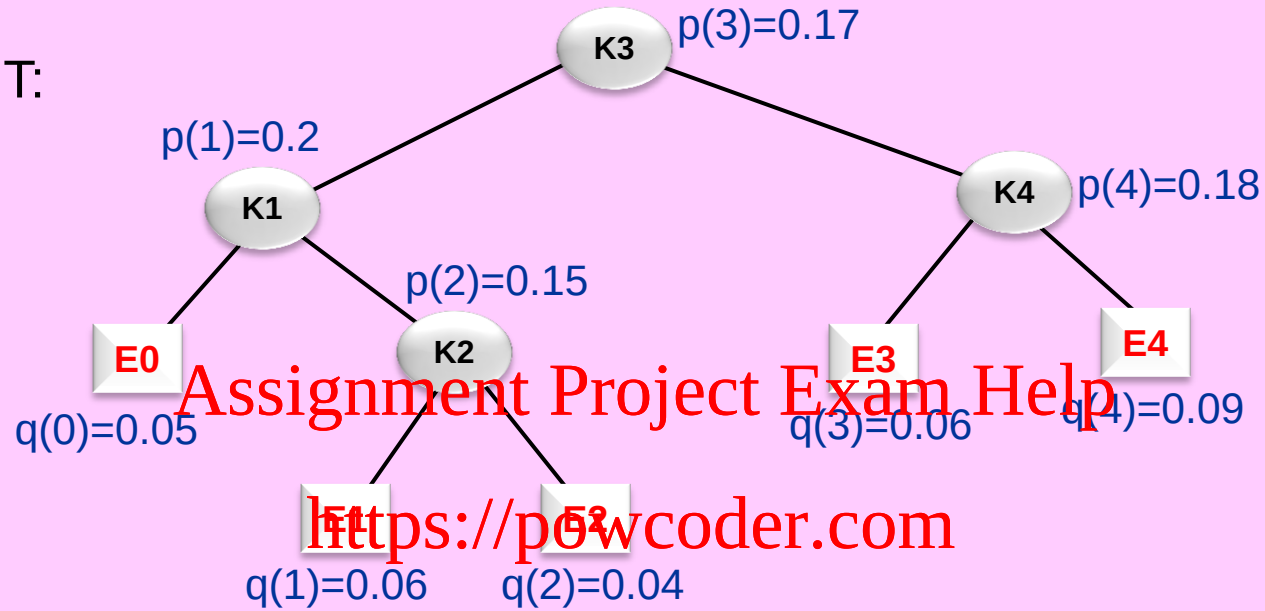
$$Cost(T) = \sum_{t=1}^{n} p(t) \cdot \left(1 + depth_T(K_t)\right) + \sum_{t=0}^{n} q(t) \cdot depth_T(E_t) .$$

- Given p(i)'s, q(i)'s, and K_i's, our goal is to find & construct the BST T that **minimizes** this expected search cost.

- Recall that the # of possible BST's is the Catalan # $P(n+1) = \frac{1}{n+1}\binom{2n}{n}$.

Example

T:



$$\text{Cost}(T) = \sum_{t=1}^{n} p(t) \cdot (1 + \text{depth}_T(K_t)) + \sum_{t=0}^{n} q(t) \cdot \text{depth}_T(E_t)$$

$= (0.2\times2 + 0.15\times3 + 0.17\times1 + 0.18\times2) +$
$(0.05\times2 + 0.06\times3 + 0.04\times3 + 0.06\times2 + 0.09\times2)$

$= 2.08$

The Optimum Static BST Problem

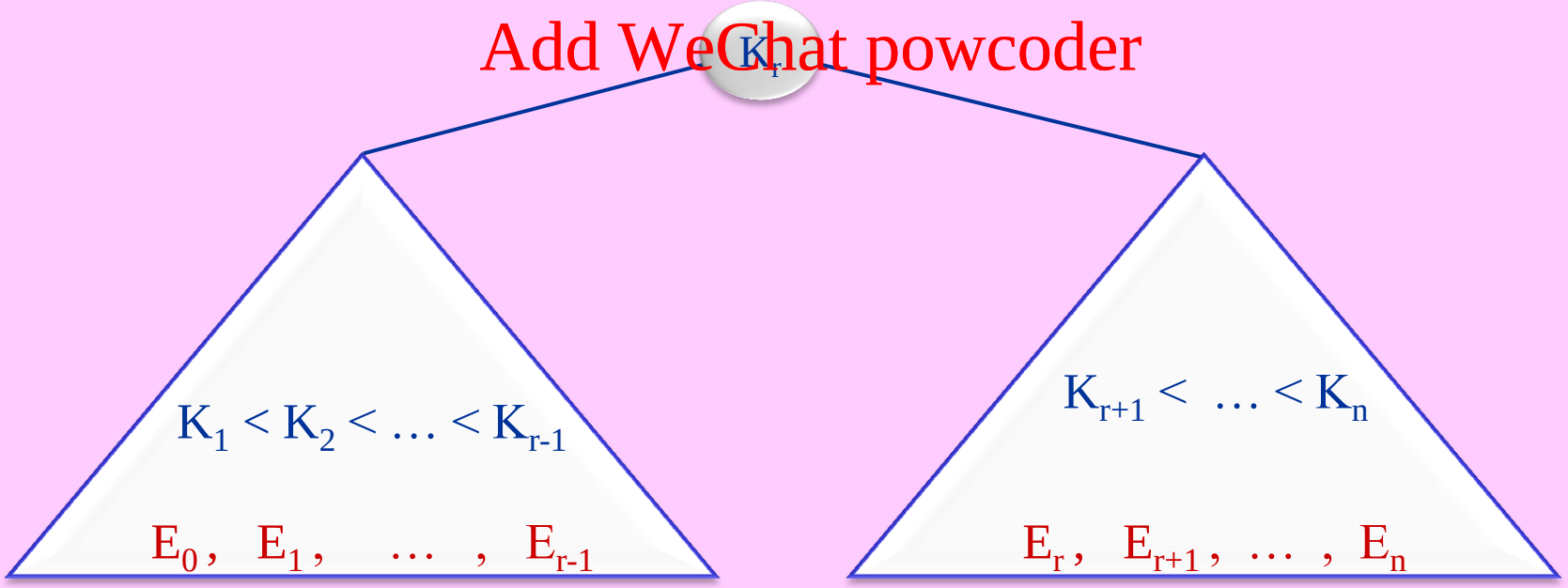
Input: The keys $K_1 < K_2 < \ldots < K_n$ and successful & unsuccessful search probabilities p[1..n] & q[0..n].

Output: An optimum BST T holding the input keys.

Goal: Minimize expected search cost Cost(T).

Task: a (recursive) sequence of decisions.

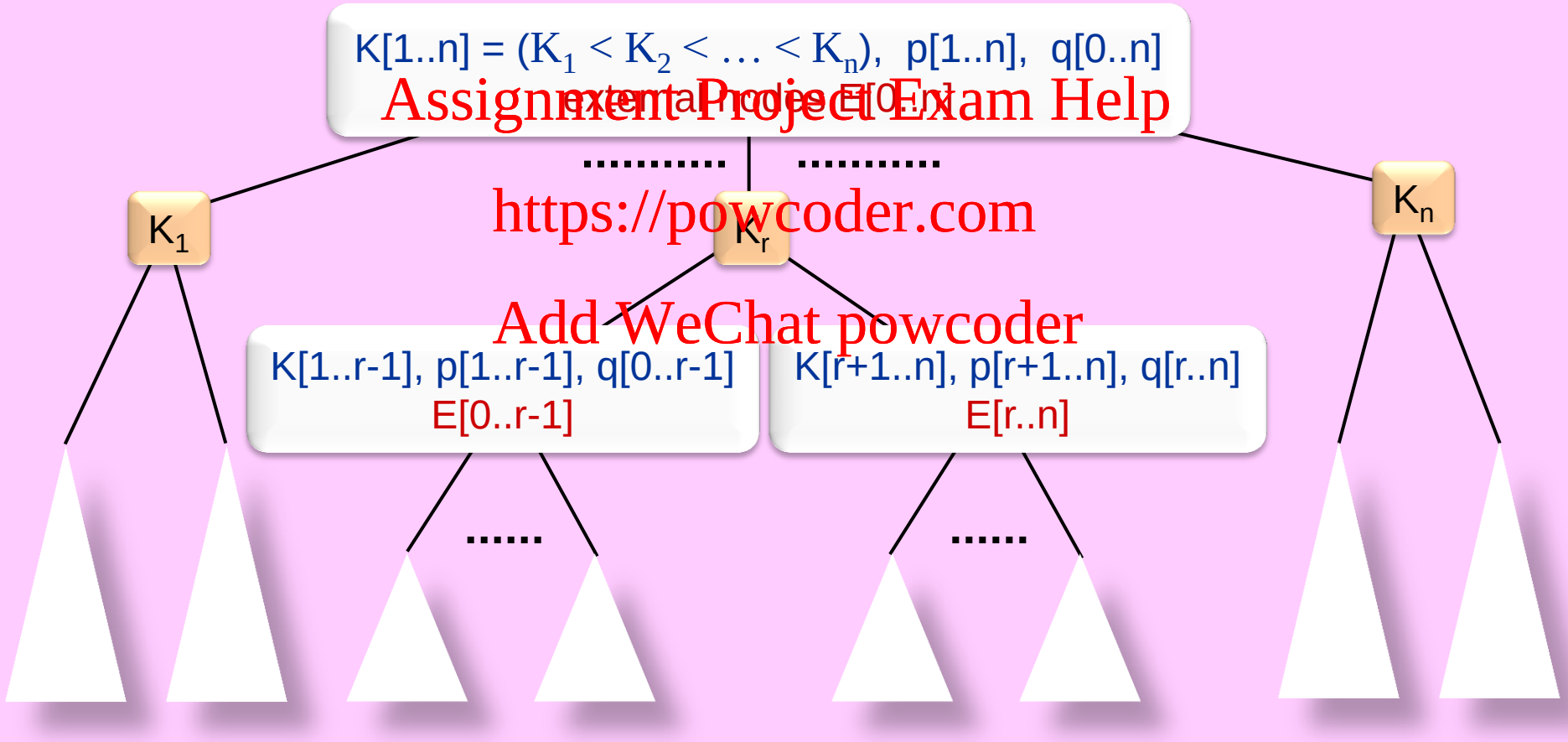
Top level decision: which key k_r should be at the root of T? n options.



Dynamic Programming Design Step 1:

Think about the recursion tree structure, its post-order evaluation, & how you will sub-divide an instance into (smaller) immediate sub-instances using sub-division option nodes. Identify base cases.

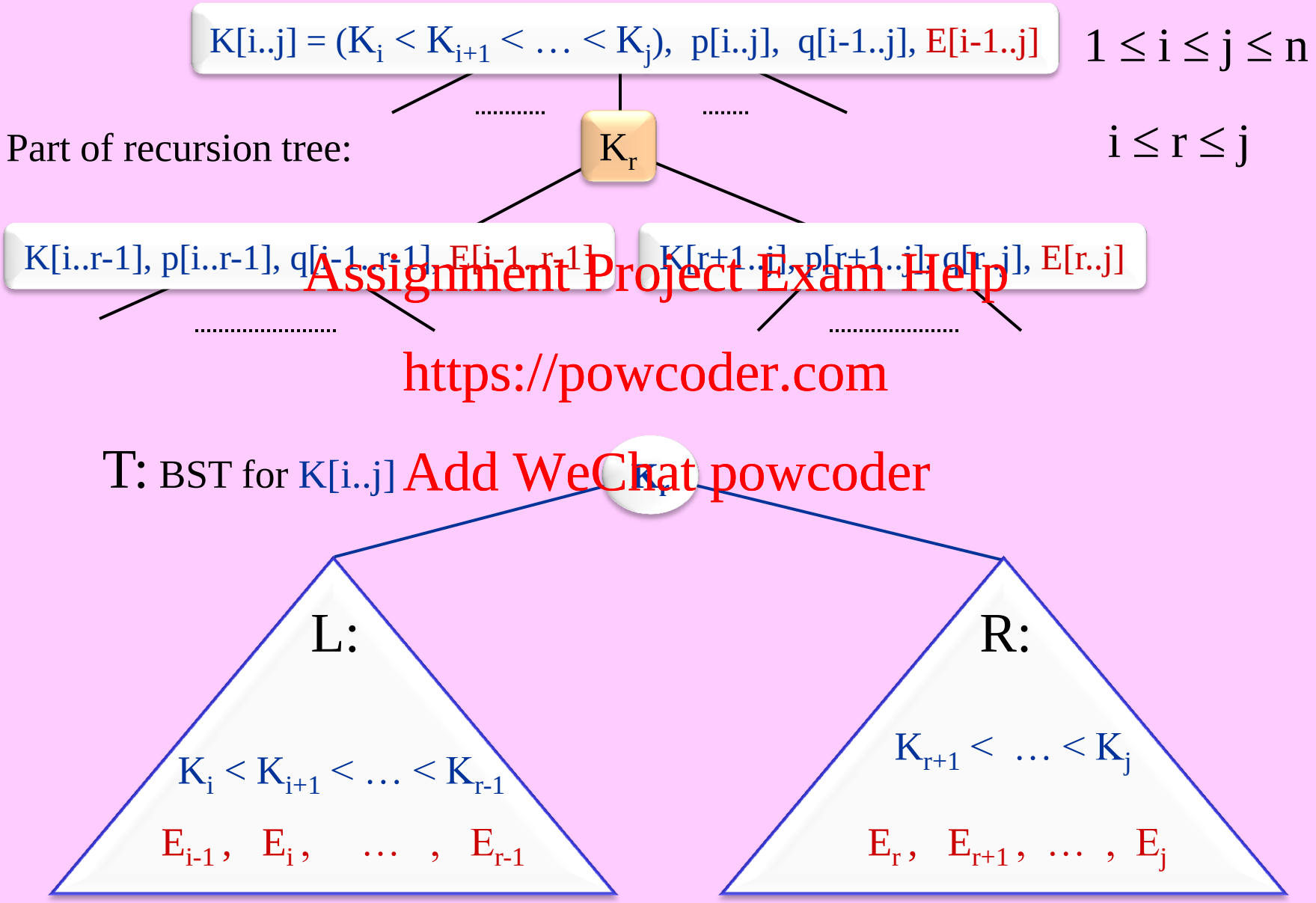
Sub-division decision: which key K_r should form root of the BST?



Leaves are empty trees of the form E[i] (no key, a single external node).

Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.



Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.

$$\text{Cost}(T) = \sum_{t=i}^{j} p(t)\cdot(1+\text{depth}_T(K_t)) + \sum_{t=i-1}^{j} q(t)\cdot\text{depth}_T(E_t)$$

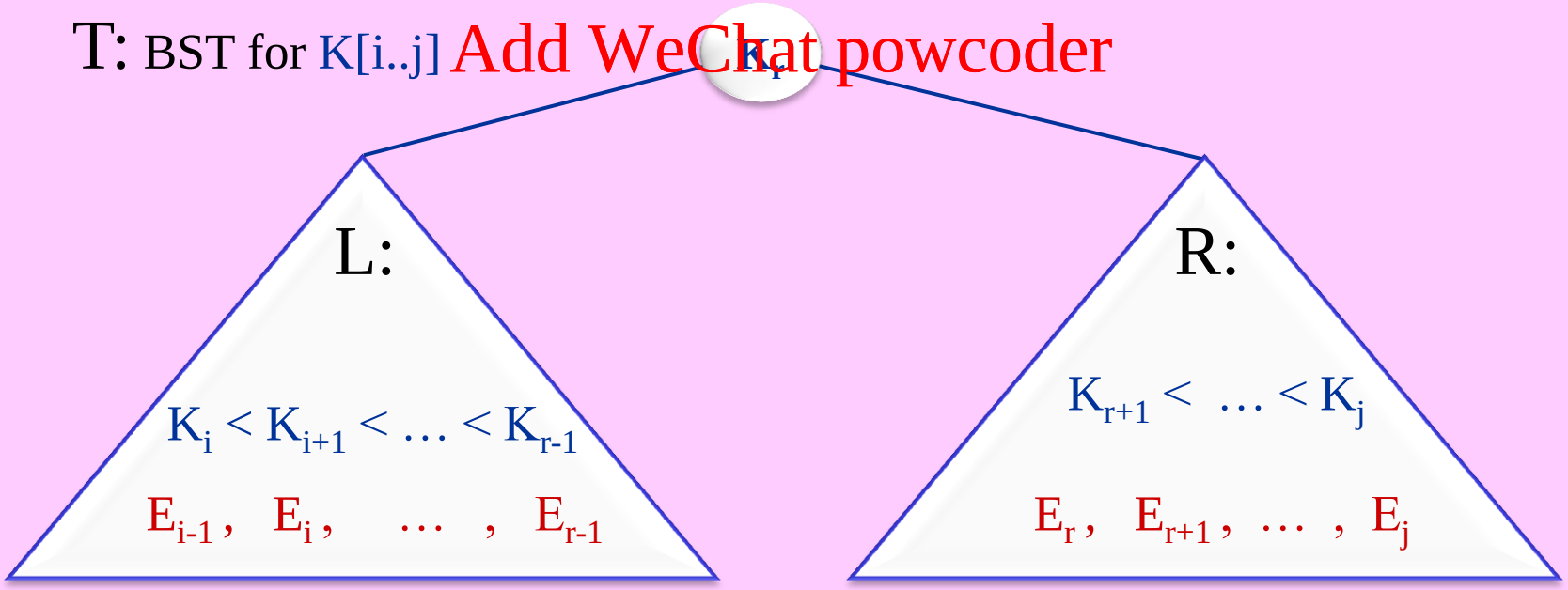
$$= \left(\sum_{t=i}^{r-1} p(t)\cdot(1+\text{depth}_T(K_t)) + p(r) + \sum_{t=r+1}^{j} p(t)\cdot(1+\text{depth}_T(K_t))\right) +$$

$$\left(\sum_{t=i-1}^{r-1} q(t)\cdot\text{depth}_T(E_t) + \sum_{t=r}^{j} q(t)\cdot\text{depth}_T(E_t)\right)$$

$$= \left(\sum_{t=i}^{r-1} p(t)\cdot(2+\text{depth}_L(K_t)) + p(r) + \sum_{t=r+1}^{j} p(t)\cdot(2+\text{depth}_R(K_t))\right) +$$

$$\left(\sum_{t=i-1}^{r-1} q(t)\cdot(1+\text{depth}_L(E_t)) + \sum_{t=r}^{j} q(t)\cdot(1+\text{depth}_R(E_t))\right)$$

T: BST for K[i..j]



Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.

$$\text{Cost}(T) = \sum_{t=i}^{j} p(t)\cdot(1+\text{depth}_T(K_t)) + \sum_{t=i-1}^{j} q(t)\cdot\text{depth}_T(E_t)$$

$$= \left(\sum_{t=i}^{r-1} p(t)\cdot(1+\text{depth}_T(K_t)) + p(r) + \sum_{t=r+1}^{j} p(t)\cdot(1+\text{depth}_T(K_t))\right) +$$
$$\left(\sum_{t=i-1}^{r-1} q(t)\cdot\text{depth}_T(E_t) + \sum_{t=r}^{j} q(t)\cdot\text{depth}_T(E_t)\right)$$

$$= \left(\sum_{t=i}^{r-1} p(t)\cdot(2+\text{depth}_L(K_t)) + p(r) + \sum_{t=r+1}^{j} p(t)\cdot(2+\text{depth}_R(K_t))\right) +$$
$$\left(\sum_{t=i-1}^{r-1} q(t)\cdot(1+\text{depth}_L(E_t)) + \sum_{t=r}^{j} q(t)\cdot(1+\text{depth}_R(E_t))\right)$$

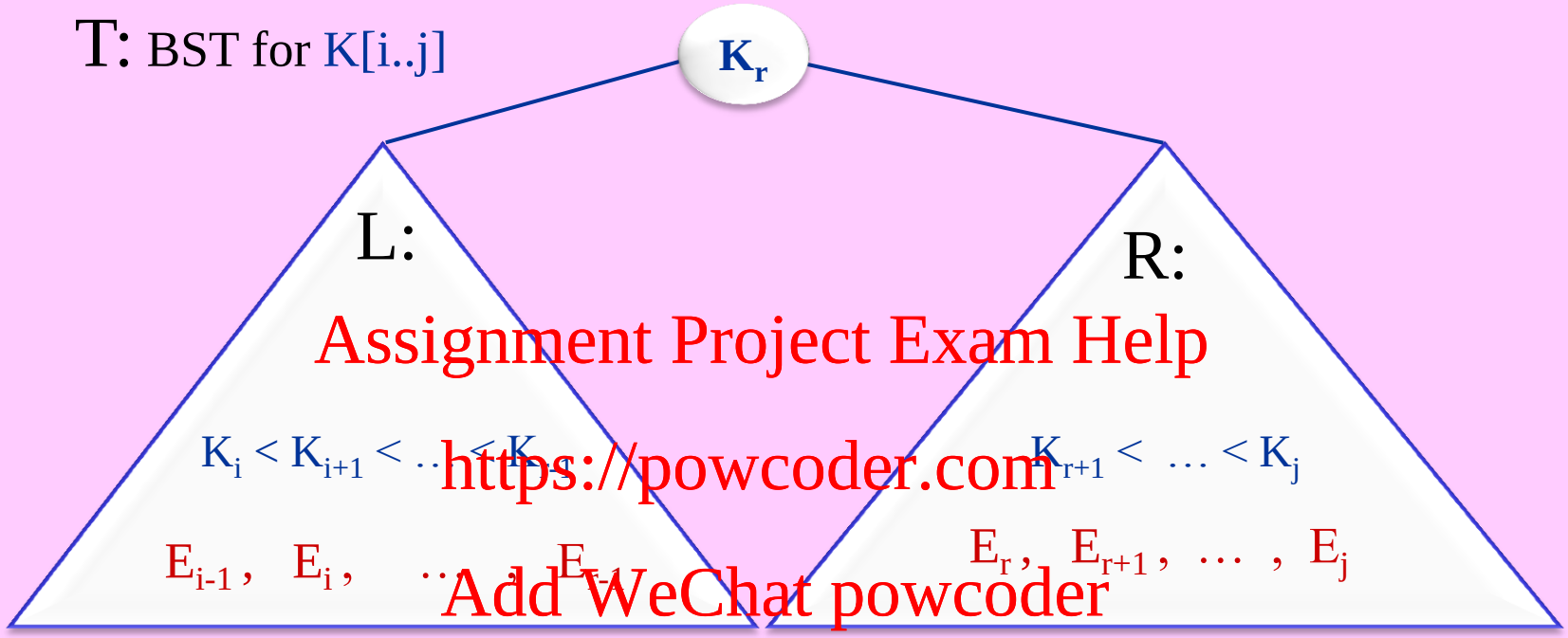
$$= \left(\sum_{t=i-1}^{r-1} p(t)\cdot(1+\text{depth}_L(K_t)) + \sum_{t=i-1}^{r-1} q(t)\cdot\text{depth}_L(E_t)\right) +$$
$$\left(\sum_{t=r+1}^{j} p(t)\cdot(1+\text{depth}_R(K_t)) + \sum_{t=r}^{j} q(t)\cdot\text{depth}_R(E_t)\right) +$$
$$\left(\sum_{t=i}^{j} p(t) + \sum_{t=i-1}^{j} q(t)\right)$$

$$= \text{Cost}(L) + \text{Cost}(R) + \left(\sum_{t=i}^{j} p(t) + \sum_{t=i-1}^{j} q(t)\right)$$

$$= \text{Cost}(L) + \text{Cost}(R) + PQ(i, j)\ .$$

Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.



$$\text{Cost}(T) = \text{Cost}(L) + \text{Cost}(R) + PQ(i, j)\ .$$

Instance[i..j]:
K[i..j],
p[i..j],
q[i-1..j],
E[i-1..j]

Instance[i..r-1]:
K[i..r-1],
p[i..r-1],
q[i-1..r-1],
E[i-1..r-1]

Instance[r+1..j]:
K[r+1..j],
p[r+1..j],
q[r..j],
E[r..j]

Independent of choice of r

Dynamic Programming Design Step 2:

Make sure this sub-division into sub-instances satisfies the OSSP.

$$\text{Cost}(T) = \underbrace{\text{Cost}(L) + \text{Cost}(R)} + PQ(i, j) .$$

Instance [i..j]:

No interference between instances [i..r-1] & [r+1..j]

OSSP is satisfied:

Regardless of which key K_r is chosen as the root of T,
If T is a minimum cost BST for instance [i..j],
Then,
L must be a minimum cost BST for instance [i..r-1], and
R must be a minimum cost BST for instance [r+1..j].

Proof: By the cut-&-paste argument.

Dynamic Programming Design Step 3:

Determine the set of all possible distinct sub-instances that would be recursively generated in the recursion tree. (Moderate over-shooting is OK.)
Make sure this set is closed under the sub-division in step 1.

The main instance:

$K[1..n] = (K_1 < K_2 < \ldots < K_n)$, $p[1..n]$, $q[0..n]$, external nodes $E[0..n]$.

General sub-instance:

$K[i..j] = (K_i < K_{i+1} < \ldots < K_j)$, $p[i..j]$, $q[i-1..j]$, external nodes $E[i-1..j]$,

for $1 \le i \le j \le n$, and boundary cases $0 \le i-1 = j \le n$.

[The boundary case j=i-1, corresponds to the empty BST: E_{i-1}.]

There are $(n+1)\times(n+2)/2$ sub-instances.

Dynamic Programming Design Step 4:

Using OSSP, develop the DP Recurrence Formula that expresses the memo of the optimum solution of each (sub-)instance in terms of the memos of optimum solutions to its (smaller) immediate (sub-)sub-instances. This formula has to be general enough to be valid for each sub-instance in the set of sub-instances determined in step 3. Express base cases by some direct method.

Memo Tables for sub-instance K[i..j], p[i..j], q[i-1..j], external nodes E[i-1..j]:

$C_{OPT}[i,j]$ = cost of optimum BST
$R_{OPT}[i,j]$ = root index r of optimum BST

$$C_{OPT}[i,j] = \begin{cases} 0 & \text{if } 0 \le i-1 = j \le n \\ \min_{r=i..j} \{C_{OPT}[i,r-1] + C_{OPT}[r+1,j]\} + PQ(i,j) & \text{if } 1 \le i \le j \le n \end{cases}$$

$$R_{OPT}[i,j] = \begin{cases} \text{nil} & \text{if } 0 \le i-1 = j \le n \\ \text{argmin}_{r=i..j} \{C_{OPT}[i,r-1] + C_{OPT}[r+1,j]\} & \text{if } 1 \le i \le j \le n \end{cases}$$

Dynamic Programming Design Step 4:

$$C_{OPT}[i, j] = \begin{cases} 0 & \text{if } 0 \le i - 1 = j \le n \\ \min_{r=i..j} \{C_{OPT}[i, r-1] + C_{OPT}[r+1, j]\} + PQ(i, j) & \text{if } 1 \le i \le j \le n \end{cases}$$

$$R_{OPT}[i, j] = \begin{cases} \text{nil} & \text{if } 0 \le i - 1 = j \le n \\ \text{argmin}_{r=i..j} \{C_{OPT}[i, r-1] + C_{OPT}[r+1, j]\} & \text{if } 1 \le i \le j \le n \end{cases}$$

$$PQ(i, j) = \left(\sum_{t=i}^{j} p(t) + \sum_{t=i-1}^{j} q(t)\right)$$

In O(n) time pre-compute PQ[j] (incrementally on j=0..n) :

$$PQ[j] = PQ(1, j) = \left(\sum_{t=1}^{j} p(t) + \sum_{t=0}^{j} q(t)\right), \text{ for } j = 0..n.$$

$$\left(PQ[0] = q(0), \quad PQ[j] = PQ[j-1] + p(j) + q(j), \text{ for } j = 1..n. \right)$$

Now each PQ(i,j) can be obtained in O(1) time:

$$PQ(i, j) = PQ[j] - PQ[i-1] + q(i-1).$$

We will use this in our main algorithm.

Dynamic Programming Design Step 5:

Starting from the base cases, iteratively fill in the memo table in reverse order of dependency using the recurrence from step 4.

Algorithm **OptBSTMemo**(p[1..n], q[0..n])
Pre-Cond: input is a valid instance of the Opt Static BST Problem
Post-Cond: Memo Tables C_{OPT} and R_{OPT} returned

```
PQ[0] ← q[0];  C_OPT[1,0] ← 0                          § base case first
for j ← 1 .. n do
    PQ[j] ← PQ[j-1]+p[j]+q[j]
    C_OPT[j+1,j] ← 0                                   § base case first
    for i ← j downto 1 do                              § in a reverse order of dependency
        C_OPT[i,j] ← ∞                                 § instance [i..j]
        pq ← PQ[j] – PQ[i-1] + q[i-1]                  § PQ(i,j)
        for r ← i .. j do                              § option: opt BST [i..j] with root r
            newCost ← C_OPT[i,r-1] + C_OPT[r+1,j] + pq
            if C_OPT[i,j] > newCost then C_OPT[i,j] ← newCost;   R_OPT[i,j] ← r
        end-for
    end-for
end-for
return ( C_OPT , R_OPT )
end
```


$\Theta(n^3)$ time
$\Theta(n^2)$ space

Dynamic Programming Design Step 6

Á posteriori, use the memo table to recover & reconstruct the optimum solution to the given main instance.

Initial call: **OptBST**(R_{OPT} , 1 , n) , keys K[1..n].

Algorithm **OptBST**(R_{OPT}, i , j)
Pre-Cond: input instance [i..j], R_{OPT} is the opt root memo table.
Post-Cond: OptBST for instance [i..j] constructed & its root node returned.

```
if  i = j-1 then return nil
r ← R_OPT[i,j]
x ←  a new node
key[x]  ← K[r]
left[x]   ← OptBST(R_OPT, i , r-1)
right[x] ← OptBST(R_OPT, r+1 , j)
return x
end
```


Θ(n) time

AAW has a nice animation of the algorithm.
An example follows …

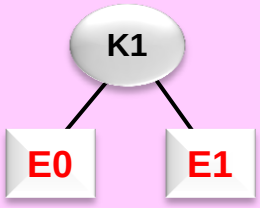
$C_{ij} = C_{OPT}([i..j])$	K_1	K_2	K_3	K_4	K_5
	p[1] = 0.1	p[2] = 0.1	p[3] = 0.05	p[4] = 0.05	p[5] = 0.05
q[0] = 0.05	q[1] = 0.05	q[2] = 0.2	q[3] = 0.2	q[4] = 0.1	q[5] = 0.05

Base:

$C_{10}=0$ E0 $C_{21}=0$ E1 $C_{32}=0$ E2 $C_{43}=0$ E3 $C_{54}=0$ E4 $C_{65}=0$ E5

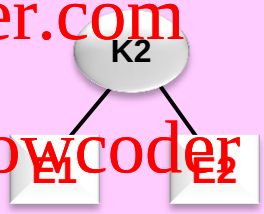
j = 1:

$C_{11}=0.2$

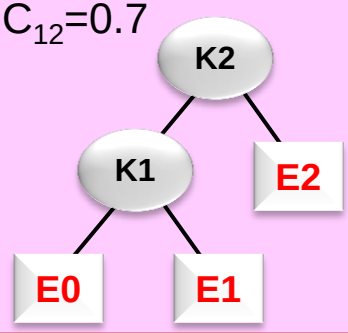


j = 2:

$C_{22}=0.35$

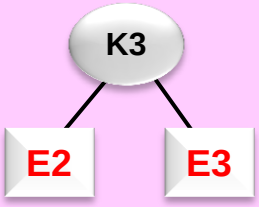


$C_{12}=0.7$

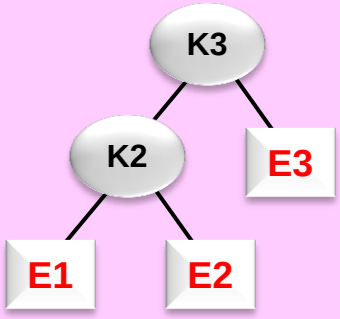


j = 3:

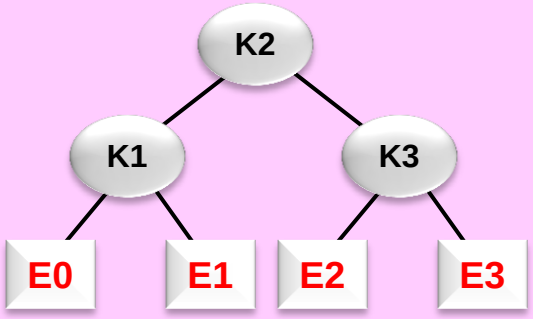
$C_{33}=0.45$



$C_{23}=0.95$



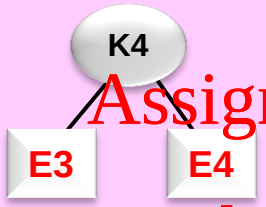
$C_{13}=1.4$



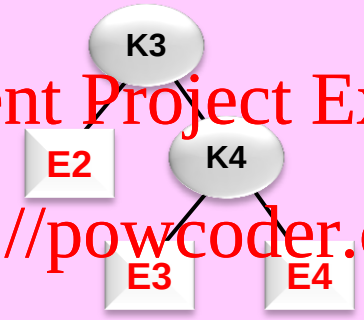
$C_{ij} = C_{OPT}([i..j])$	K_1	K_2	K_3	K_4	K_5
	p[1] = 0.1	p[2] = 0.1	p[3] = 0.05	p[4] = 0.05	p[5] = 0.05
q[0] = 0.05	q[1] = 0.05	q[2] = 0.2	q[3] = 0.2	q[4] = 0.1	q[5] = 0.05

j = 4:

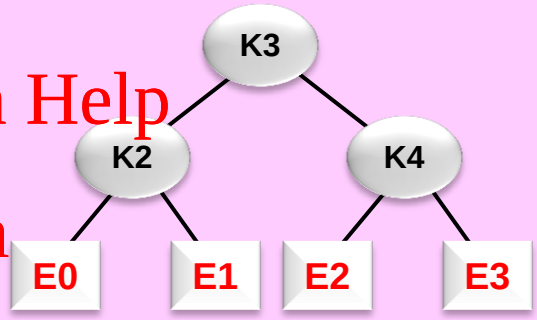
C_{44}=0.35



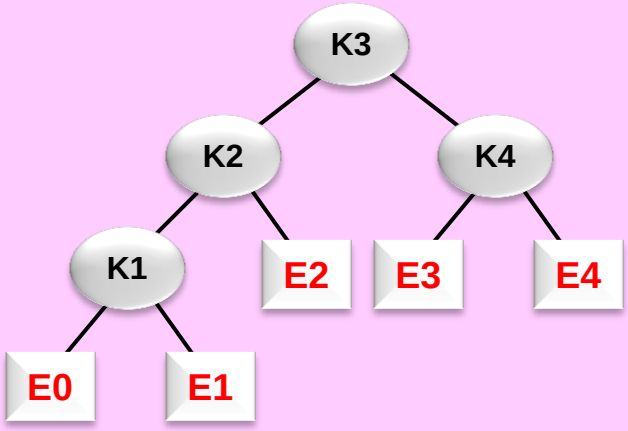
C_{34}=0.95



C_{24}=1.45



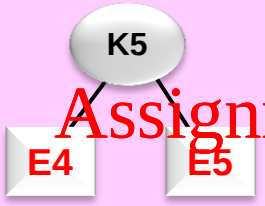
C_{14}=1.95



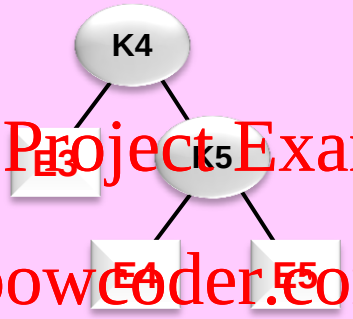
$C_{ij} = C_{OPT}([i..j])$	K_1	K_2	K_3	K_4	K_5
	p[1] = 0.1	p[2] = 0.1	p[3] = 0.05	p[4] = 0.05	p[5] = 0.05
q[0] = 0.05	q[1] = 0.05	q[2] = 0.2	q[3] = 0.2	q[4] = 0.1	q[5] = 0.05

j = 5:

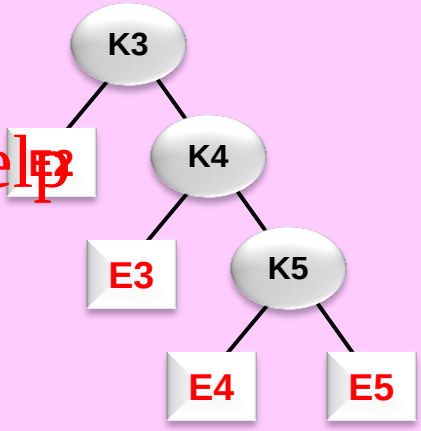
C_{55}=0.2



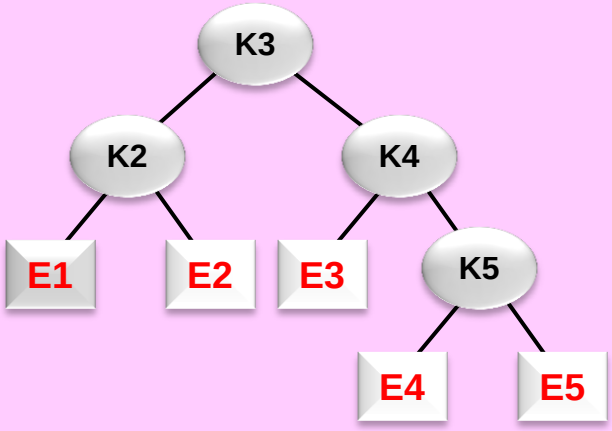
C_{45}=0.65



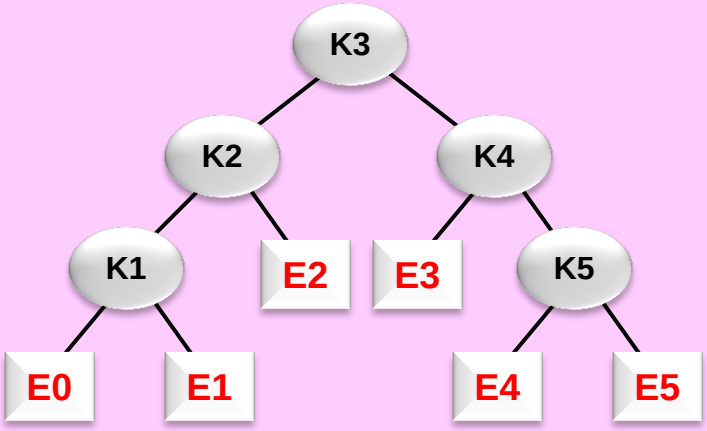
C_{35}=1.35



C_{25}=1.85

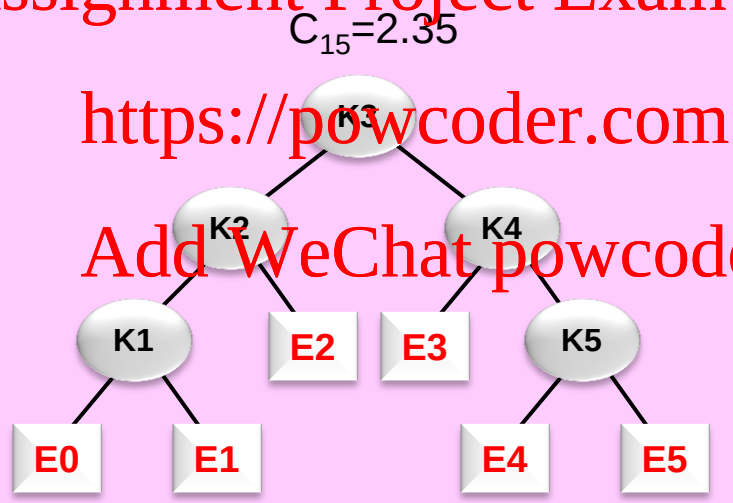


C_{15}=2.35



$C_{ij} = C_{OPT}([i..j])$	K_1	K_2	K_3	K_4	K_5
	p[1] = 0.1	p[2] = 0.1	p[3] = 0.05	p[4] = 0.05	p[5] = 0.05
q[0] = 0.05	q[1] = 0.05	q[2] = 0.2	q[3] = 0.2	q[4] = 0.1	q[5] = 0.05

Optimum Static BST T: Cost(T) = 2.35



Dynamic Programming Design Step 7:

Keep an eye on time & space efficiency.

Algorithm Optimum Static Binary Search Tree

Time = $\Theta(n^3)$, Space = $\Theta(n^2)$.

Any improvement?

- Knuth vol III, page 436:
  observed that the root option range: $i \le R_{OPT}(i,j) \le j$
  can be narrowed down to: $R_{OPT}(i,j-1) \le R_{OPT}(i,j) \le R_{OPT}(i+1,j)$.

  This cuts down the # of iterations of the 3rd nested loop in the algorithm.

  The algorithm complexity improves to: **Time = $\Theta(n^2)$, Space = $\Theta(n^2)$.**

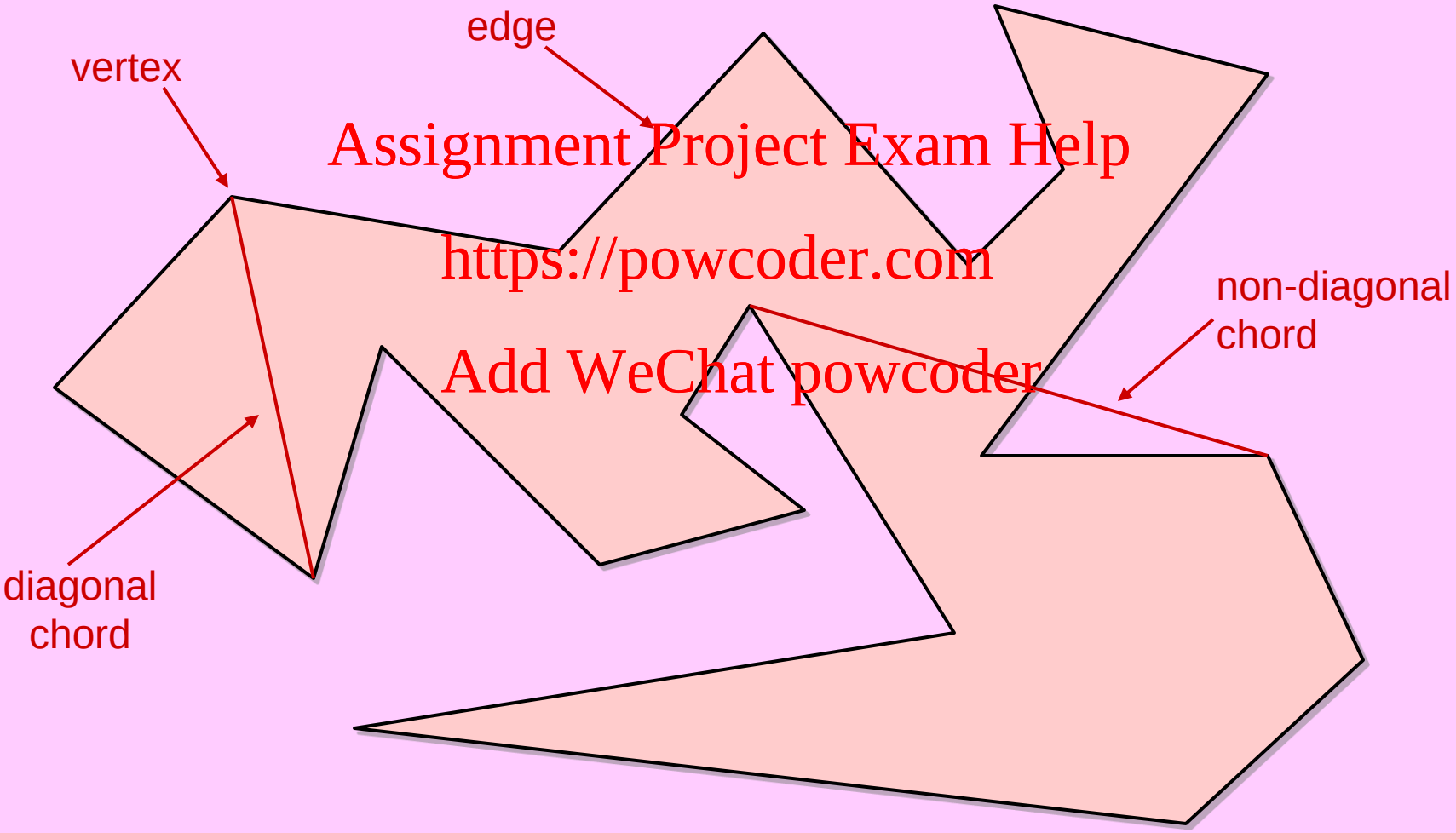
- Hu-Tucker have a slick algorithm for the special case when all p[1..n] = 0, with time complexity O(n log n).

- In EECS 4101 we study a competitive self-adjusting BST called Splay Tree.

OPTIMUM POLYGON TRIANGULATION

Simple Polygon P

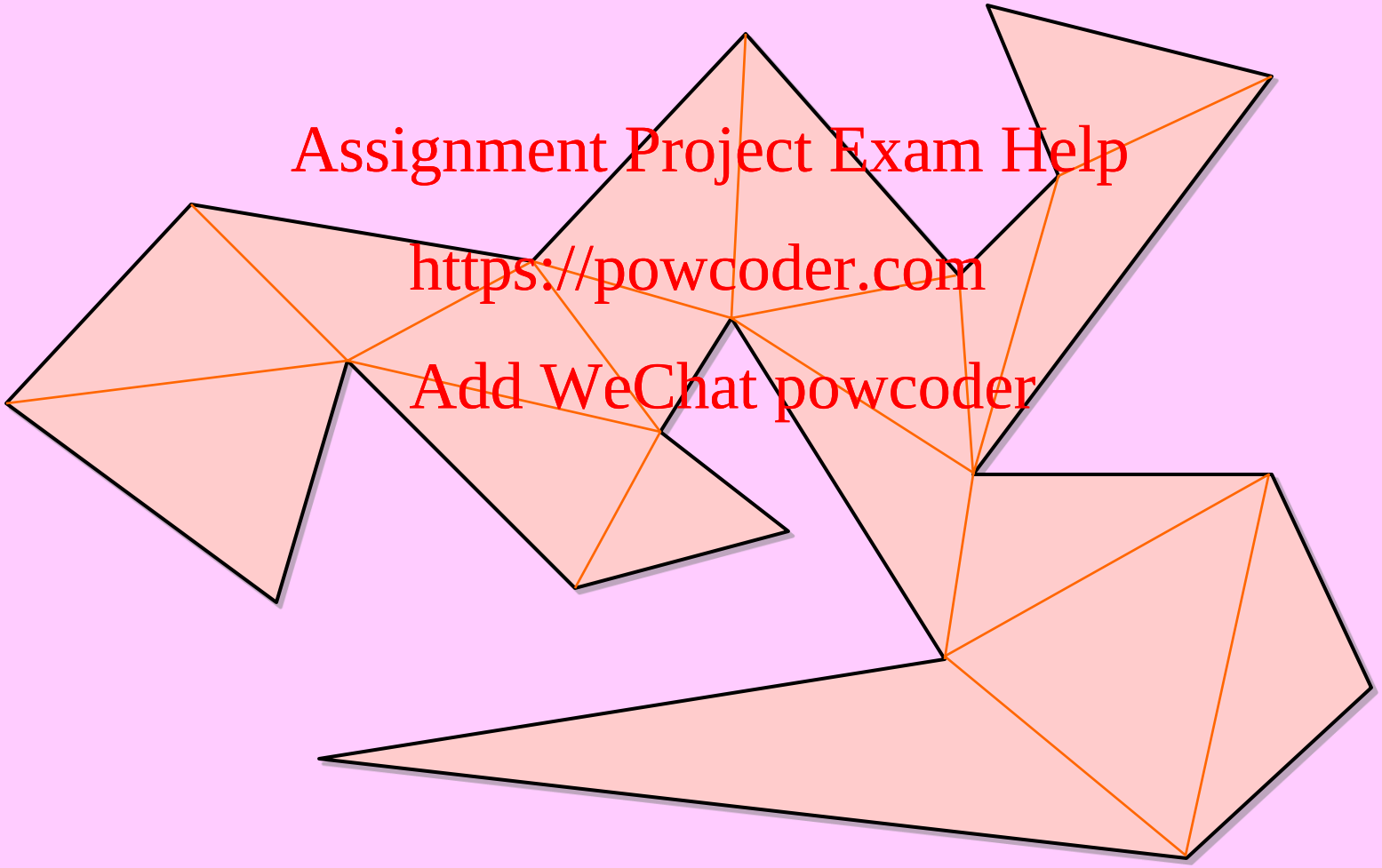
Simple Polygon P: A non-self-crossing closed chain of finitely many line-segments.
Partitions the plane into: **Boundary, Interior, Exterior.**



Simple Polygon P

A Triangulation T of P: Any maximal set of pair-wise non-crossing diagonals. T partitions the interior of P into triangles.

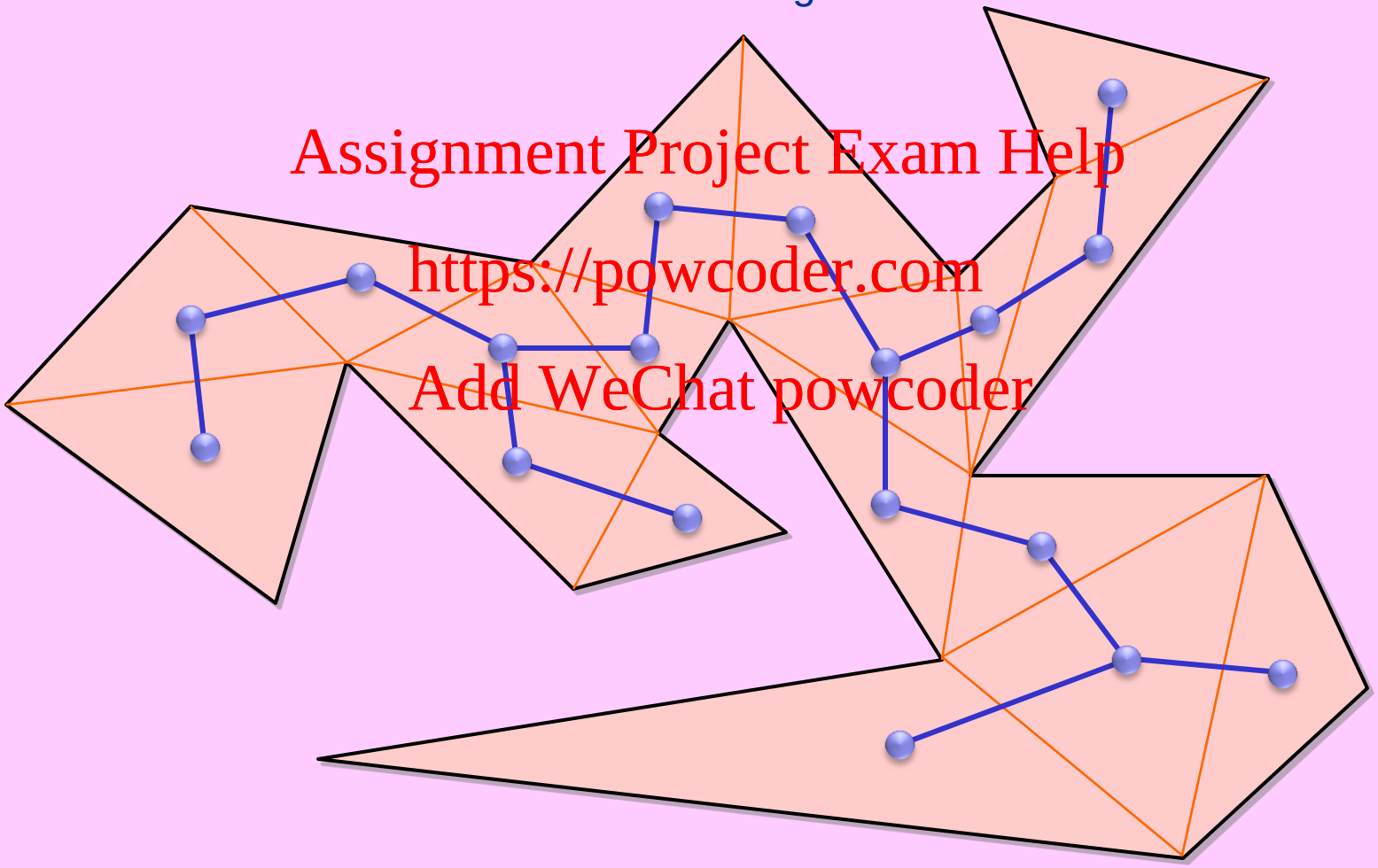
FACT: Any simple polygon admits at least one triangulation.



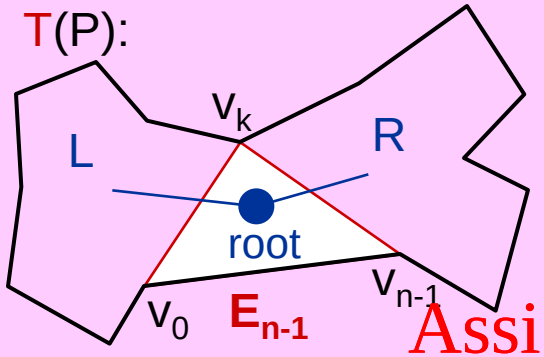
Simple Polygon P

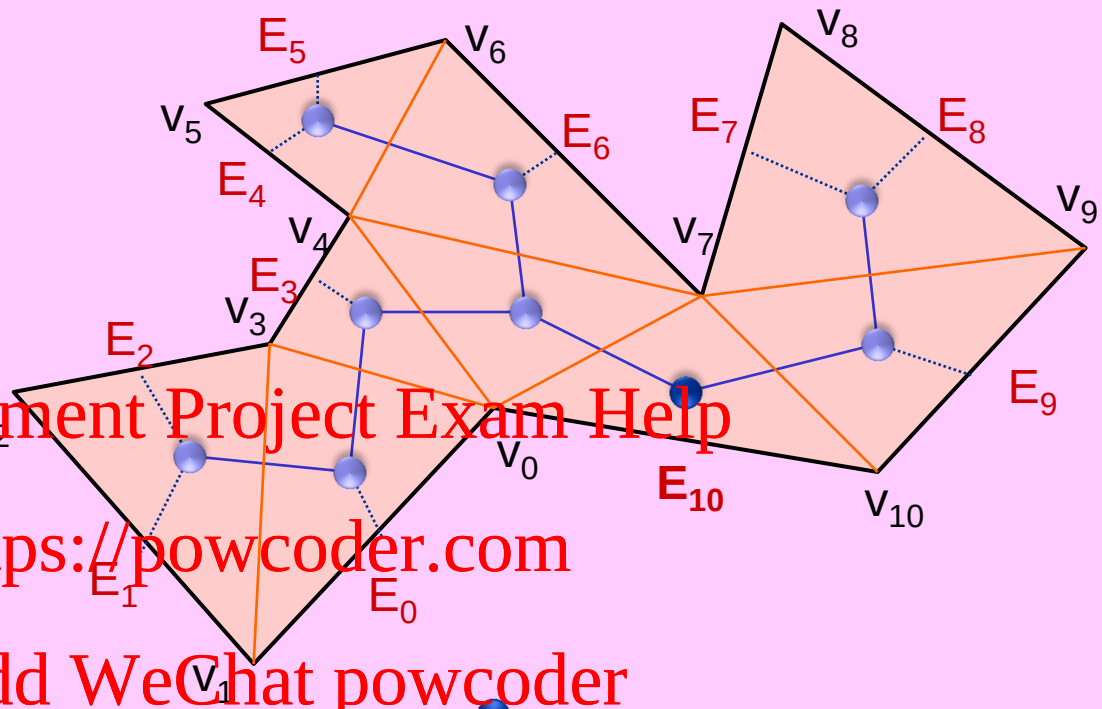
Dual of Triangulation T: A graph whose nodes correspond to triangles of T, and two such nodes are connected by a dual edge if the corresponding pair of triangles share a diagonal.

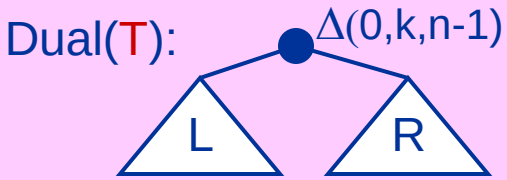
FACT: The dual is a tree of maximum node degree 3.

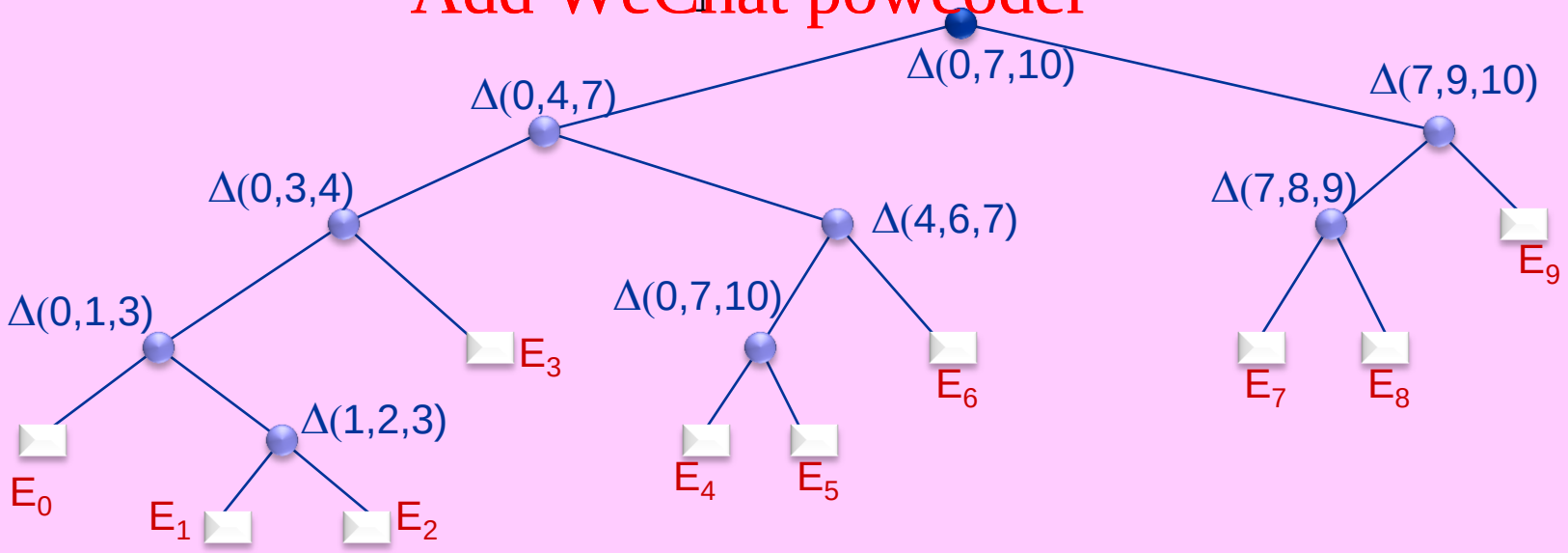


FACT: The dual becomes a Binary Tree if we designate as root its non-trivalent vertex corresponding to the triangle incident to the **last edge of P**.









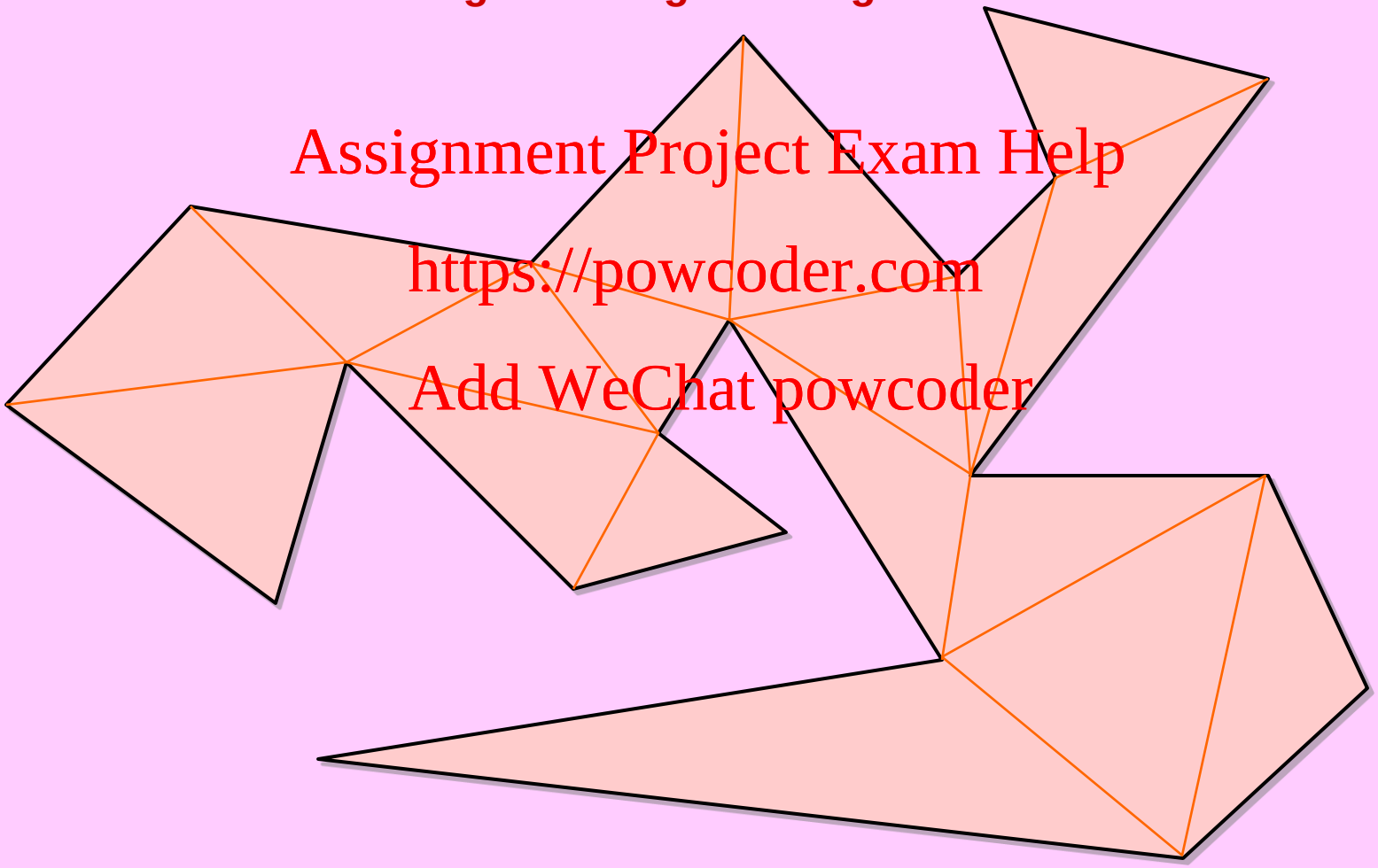
Simple Polygon P

PROBLEM:

Among many possible triangulations of P, we wish to find the optimum triangulation.

Optimum may be defined by a variety of objectives.

Here we consider **minimizing total diagonal length**.



Minimum Length Triangulation of Simple Polygon

Input: A simple polygon P given by its sequence of n vertices $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ in clockwise order around its boundary. Each vertex is given by its x and y coordinates.

Feasible Solutions: Any triangulation T of P.

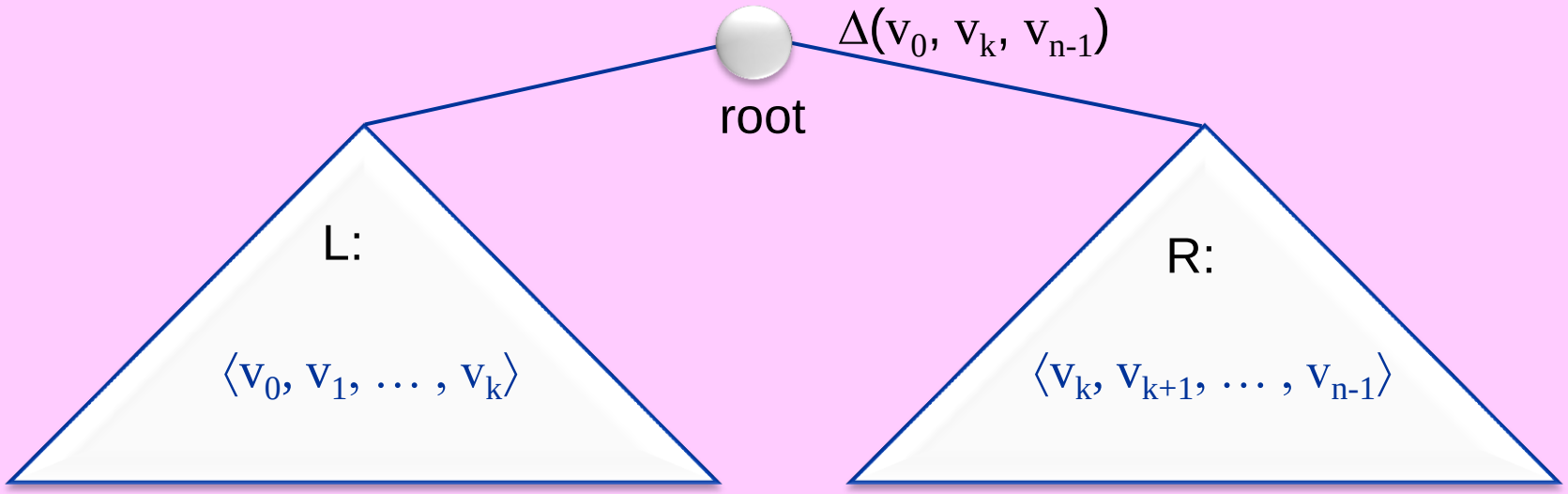
Objective Value: Cost(T) = total length of the diagonals in T.

Goal: Output a triangulation T of P with minimum objective value Cost(T).

Top level decision:

Which triangle $\Delta(v_0, v_k, v_{n-1})$ should form the root of Dual(T)?

Up to n-2 options $k \in \{1 \ldots n-2\}$.

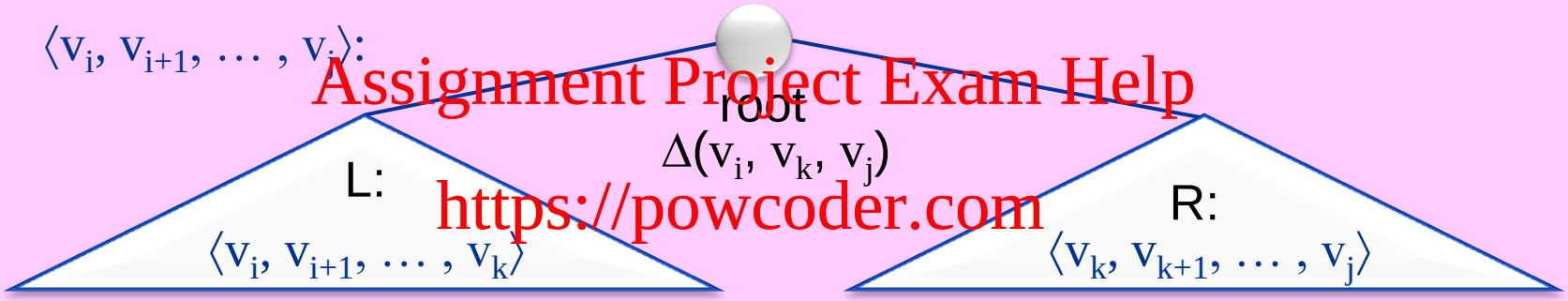


Exercise: similar to previous 2 problems

Dynamic Programming Design Steps 1-7: Work out each step.

General sub-instance: simple sub-polygon $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, $0 \le i < j \le n-1$.

Decision option: Which triangle $\Delta(v_i, v_k, v_j)$, $k \in [i+1 .. j-1]$, should be "root"?



An option k is valid (conflict-free) if (besides "last edge" $\langle v_i , v_j \rangle$) the 2 sides $\langle v_i , v_k \rangle$ and $\langle v_k , v_j \rangle$ of the "root" triangle are either edges or diagonals (i.e., do not cross any edge) of the sub-polygon $\langle v_i, v_{i+1}, \ldots, v_j \rangle$.

Exercise: develop an O(j-i) time algorithm to decide whether an option k is valid.

Write the DP recurrence, expressing OPT sol of a general instance in terms of its immediate sub-instances, taking into account all its possible valid options of how to divide the "sub-polygon" into "root, left subtree, right subtree".

What is the complexity of your resulting algorithm?

Bibliography

- M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, T. Pitassi, *"Toward a Model for Backtracking and Dynamic Programming,"* 2007. (pdf)

The Longest Common Subsequence Problem:

- T.G. Szymanski, *"A special case of the maximal common subsequence problem,"* Technical Report TR-170, CS Lab., Princeton, 1975.

  Gives an O((m+n)log(m+n)) time algorithm for the special case when no symbol appears more than once in a sequence.

- W.J. Masek, M.S. Paterson, *"A faster algorithm computing string edit distances,"* Journal of Computer and System Sciences, 20(1):18-31, 1980.

  Gives an O(mn/log n) time algorithm, where $n \leq m$, for the special case in which sequences are drawn from a constant-size alphabet.

The Matrix Chain Multiplication Problem:

- T.C. Hu, *"Combinatorial Algorithms,"* 1982. Has an O(n log n) time solution.

Optimum Static BST Problem:

- Knuth, vol III, page 436: Improves the DP algorithm to $O(n^2)$ time.

- T.C. Hu, A.C. Tucker *"Optimal Computer Search Trees and Variable-Length Alphabetical Codes,"* SIAM J. Applied Math, 21(4):514-532, 1971.
  Has an O(n log n) time solution for the special case p[1..n] = 0.

Exercises

1. Weighted Interval Point Cover Problem:

We studied a greedy solution for the un-weighted Interval Point Cover Problem.
Here we want to study the weighted version.
We are given a set $P = \{ p_1, p_2, \ldots , p_n\}$ of n points, and a set $I = \{ I_1 = \langle s_1, f_1, w_1 \rangle, I_2 = \langle s_2, f_2, w_2 \rangle, \ldots , I_m = \langle s_m, f_m, w_m \rangle\}$ of m weighted intervals, all on the real line, where $w_t > 0$ is the weight of interval I_t.
The problem is to find out whether or not I collectively covers P, and if yes, then report a minimum weight subset $C \subseteq I$ of (possibly overlapping) intervals that collectively cover P.

(a) Can you find a greedy strategy that would solve this weighted version? Explain.
(b) Design & analyze a dynamic programming algorithm to solve this problem.

2. Cover Fixed Points with Moveable Sticks:

We are given a set $P = \{p_1 , p_2 , \ldots , p_n\}$ of n fixed points on the real line,
as well as a set $L = \{l_1 , l_2 , \ldots , l_m\}$ of m stick lengths.
We can place the m sticks, with the given lengths, anywhere on the real line.
The problem is to determine whether or not there is any placing of the m sticks that collectively cover every point in P, and if yes, output one such placing.

(a) Develop a greedy algorithm to solve the special case when all sticks have equal length.
(b) Design & analyze a dynamic programming algorithm to solve the general case.

3. **Largest 1-block in a 0/1 matrix:**
We are given an n×n matrix A with 0/1 entries. The problem is to find the largest size square block (i.e, contiguous sub-matrix) of A, all of whose entries are 1.
Design and analyze an $O(n^2)$-time dynamic programming algorithm for this problem.
[Hint: for each 1 entry, find the largest square 1-block with the said entry at its lower right corner.]

4. **Fractional Knapsack Problem:**
Complete the proof of our claim that FKP can be solved in O(n) time.
Give the details of the algorithm and prove its correctness.

5. **Special Cases of the Knapsack Problem:**
Investigate better(?) greedy or dynamic programming solutions
to the following special cases of the 0/1 Knapsack Problem.

(a) All items have equal weight but arbitrary values.
(b) All items have equal value but arbitrary weights.
(c) All items have the same value-to-weight ratio.
(d) Each item has weight 1 or 2, but arbitrary value.

6. **Optimum Knapsack Packing Problem:**
We are given n items with positive integer weights $w_1, w_2, \ldots, w_n$.
We have an unlimited number of knapsacks at our disposal,
each with the same positive integer weight capacity W (also given).
Assume that any single item can fit in a knapsack (i.e., $\max\{w_1, w_2, \ldots, w_n\} \le W$).
We wish to pack all n items in as few knapsacks as possible.
Design and analyze an algorithm for this problem.
[Is there a greedy solution? Is there a DP solution? Does OSSP hold?]

7. **[CLRS, Exercise 15.2-1, p. 378]:** Find an optimum parenthesization of a matrix-chain multiplication whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

8. **[CLRS, Exercise 15.3-3, p. 389]:** Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does OSSP (Optimum Sub-Structure Property) hold in this case?

9. **Greedy Matrix-Chain Multiplication:** We gave a counter-example to the greedy strategy that fixes the **LAST** multiplication to be the one with minimum scalar cost. How about the greedy strategy that fixes the **FIRST** multiplication to be the one with minimum scalar cost? Either prove that this one works, or give a counter-example.

10. **[CLRS, Exercise 15.4-1, p. 396]:** Run the LCS algorithm on $X = \langle 1, 0, 0, 1, 0, 1, 0 \rangle$ and $Y = \langle 0, 1, 0, 1, 1, 0, 1, 0 \rangle$. Show the memo table and how you extract the LCS from it.

11. **The Binary LCS Problem:** Design and analyze an O(n log n) time algorithm to determine LCS(X,Y) where X and Y are two given 0/1 sequences of length n each.

12. **[CLRS, Exercises 15.4-5 & 15.4-6, p. 397]:** We want to find the longest monotonically-increasing subsequence of a given sequence of n numbers.
    (a) Show an $O(n^2)$-time algorithm for this problem.
    (b) Show an O(n log n)-time algorithm for this problem.
    [Hint: Observe that the last element of a candidate subsequence of length k is at least as large as the last element of a candidate subsequence of length k-1. Maintain candidate subsequences by linking them through the input sequence.]

13. **[CLRS, Exercise 15.4-4, p. 396]:**
Show how to compute the length of an LCS using only $2 \cdot \min\{m,n\}$ entries in the C_{LCS} table plus O(1) additional memory cells. Then show how to do this using $\min\{m,n\}$ entries plus O(1) additional memory cells.

14. **Shortest Common Super-sequence Problem (SCSP):**
A sequence X is said to be a **super-sequence** of sequence Y, if Y is a sub-sequence of X.
A variation of a problem in the Genome Project is: given two sequences X and Y, compute a shortest length sequence that is super-sequence of both X and Y.
Design and analyze an efficient algorithm for SCSP.

15. **[Knuth's improvement]:** As we mentioned, Knuth has shown that there are always roots of optimum BSTs such that $R_{OPT}(i,j-1) \le R_{OPT}(i,j) \le R_{OPT}(i+1,j)$ for all $1 \le i < j \le n$.
Use this fact to modify the OptBSTMemo algorithm to run in $\Theta(n^2)$ time.

16. **[CLRS Exercise 15.5-2, p. 404]:** Determine the cost and structure of an optimum binary search tree for a set of n=7 keys with the following access probabilities:

k	0	1	2	3	4	5	6	7
p_k		.04	.06	.08	.02	.1	.12	.14
q_k	.06	.06	.06	.06	.05	.05	.05	.05

17. A Greedy Binary Search Tree Heuristic:

Since often only the approximate values of the search probabilities p[1..n] & q[0..n] are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal, i.e., its expected search cost is almost minimal for the given search probabilities. This exercise explores an O(n log n) time greedy heuristic that results in nearly optimal binary search trees. This greedy strategy first fixes the root index to be

$$\text{root} = \text{argmin}_{k} \mid PQ(0,k-1) - PQ(k,n) \mid .$$

That is, it chooses the root so that the aggregate search probabilities of its left and right subtrees are as balanced as possible.

a) Generalize this greedy choice to be applicable to any sub-instance.
b) Apply this greedy heuristic to the probability table given in exercise 16, and compare the expected search cost of the resulting BST with that of the optimum obtained in that exercise.
   Does the greedy heuristic also give you the optimum? If not, by what percentage is it off?
c) Write an O(n log n) time implementation of this greedy heuristic.
d) **[Hard:]** What is the worst ratio of the solution cost obtained by this greedy heuristic compared with the optimum cost? Express the answer as a function of n, if not exactly, then at least asymptotically in Θ(.) notation.

18. Improved Binary Search Tree by Local Search:

Local Search is another iterative design technique that works as follows: You start with some (easily constructed) feasible solution S. Then, with each iteration, you improve S by modifying it according to some simple local change rule. If and when the iterations terminate, you have a local optimum based on the chosen rule. Of course this local optimum may or may not be the global optimum. See Fig (a) below. (In some settings this is also called the hill climbing method.)

Here is a proposed Local Search strategy for the Optimum Static BST Problem: You start with an arbitrary BST that holds the given set of keys. The local change is based on rotating a link of the BST. See Fig (b) below. Rotation preserves the in-order sequence of keys, and hence, it maintains a valid BST. If there is any link in the BST whose rotation lowers the cost, you may perform that rotation (but one rotation per iteration).

Study this strategy. Evaluate the change in the expected search cost caused by a specified rotation. You may keep extra auxiliary info at the nodes to help a speedy reevaluation & update. Since there are only finitely many BSTs of the given n keys, the iterations will eventually terminate at some locally optimum BST, one in which no single rotation can further improve the expected search cost.

Is this locally optimum BST guaranteed to be globally optimum? Either prove that it is, or give a counter-example by demonstrating a small BST that is locally optimum for your chosen search probabilities p[1..n] and q[0..n], but there is a lower cost BST for these same search probabilities.

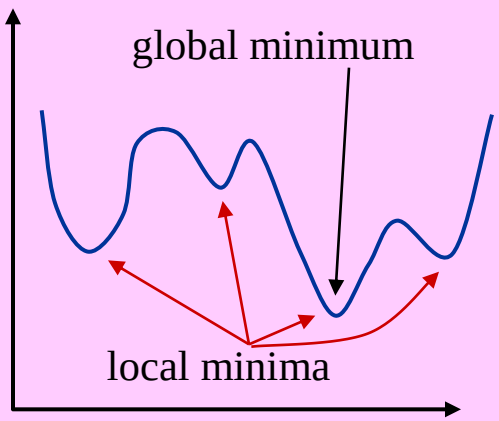


Fig (a)

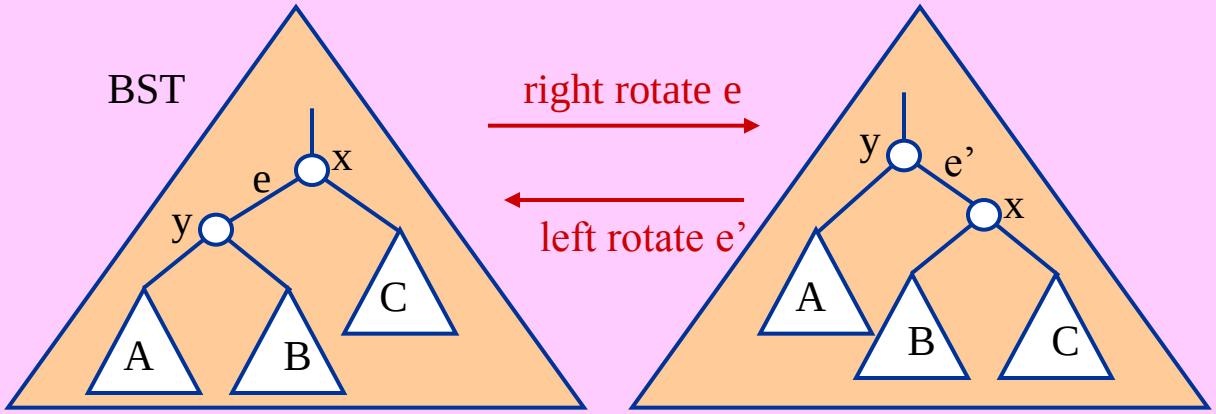


Fig (b)

19. Minimum Length Triangulation of a Polygon:
Complete the design and analysis of this problem as set out in these lecture slides.

20. Max-Min Aspect-Ratio Triangulation of a Polygon:
Consider triangulating a given **simple** polygon P with n vertices. In some scientific/industrial applications it is important to find a triangulation that avoids, as much as possible, triangles that are long and skinny looking. (For instance, in finite-element methods and mesh generation, long and skinny triangles cause numerical instability due to round-off errors.)
To this end, let us define the **aspect ratio** of a triangle to be the length of its shortest edge divided by the length of its longest edge. We notice that the aspect ratio is a real number between 0 and 1, and it is 1 if and only if the triangle is equilateral (the closer to 1 the better). We define the aspect ratio of a triangulation to be the **minimum aspect ratio of its member triangles** (i.e., its worst triangle). We define an **optimum triangulation** to be one whose aspect ratio is maximum among all triangulations of the given polygon (i.e., we want to maximize the minimum aspect ratio of all triangles in the triangulation). Design & analyze a dynamic programming algorithm to compute an optimum triangulation (as defined above) for P, when
(a) P is a given **convex** polygon. (In this special case every chord of P is a diagonal.)
(b) P is a given **simple** polygon.

21. We are given as input a sequence L of *n* numbers, and must partition it into as few contiguous subsequences as possible such that the numbers within each subsequence add to at most 100. For instance, for the input [80,-40,30,60,20,30] the optimal partition has two subsequences, [80] and [-40,30,60,20,30]. Let C[*i*] denote the number of subsequences in the optimum solution of the sub-instance consisting of the first *i* numbers, and suppose that the last subsequence in the optimum solution for all of L has *k* terms; for instance, for the same example as above, C[3] = 1 (the sub-instance [80,-40,30] needs only one subsequence) and *k* = 5 (the optimum solution for L has five numbers in the last subsequence).
Write a DP recurrence formula showing how to compute C[*n*] from *k* and from earlier values of C.

22. [CLRS, Problem 15-3, p. 405]: Bitonic Euclidean Traveling Salesman Problem:

The Euclidean Traveling Salesman Problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane. Fig (a) below shows the solution to a 7-point instance of the problem. This problem is NP-hard, and its solution is therefore believed to require more than polynomial time. (Refer to our upcoming topic on NP-hardness.)

J.L. Bentley has suggested that we simplify the problem by restricting our attention to **bitonic tours**, that is, tours that start at the leftmost point, go strictly left-to-right to the rightmost point, and then go strictly right-to-left back to the starting point. Fig (b) below shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time dynamic programming algorithm for determining an optimum bitonic tour. You may assume that no two points have the same x-coordinate.

[Hint: Top level DP decision: which edges connect to the rightmost point? What is the resulting sub-instance?]

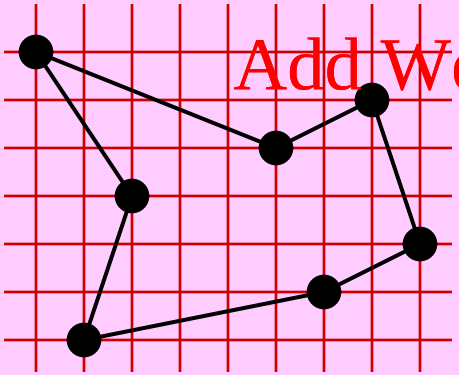


Fig (a)

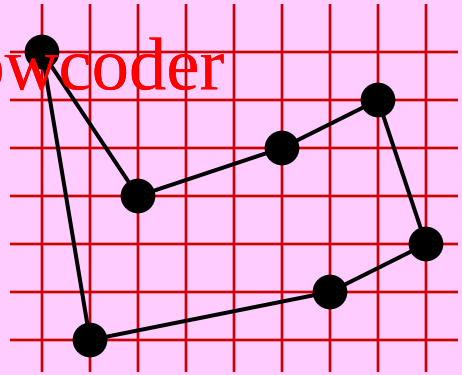


Fig (b)

23. Optimum Shipment of Supplies from Warehouses to Destinations:

Supplies are to be shipped from two warehouses W_1 and W_2 to n destinations D_k, k=1..n.
Input:

s_j = the supply inventory at W_j, j = 1,2.
d_k = the demand at D_k, k = 1..n, (assume $\Sigma s_j = \Sigma d_k$).
c_{jk} = the cost of shipping one unit of supply from warehouse W_j to destination D_k, j=1,2, k=1..n.

Let x_{jk} denote the number of supply units that will be shipped from warehouse W_j to destination D_k.

A feasible solution to the problem is an assignment of integer values to x_{jk}, such that

(i) all demands are met, i.e., $x_{1k} + x_{2k} = d_k$, for all k=1..n, and
(ii) total units shipped from each warehouse does not exceed its inventory, i.e.,

for j=1,2: $x_{j1} + x_{j2} + \cdots + x_{jn} \leq s_j$.

The objective is to minimize the total shipment cost $\Sigma_{jk}\, c_{jk}x_{jk}$.

Consider a sub-instance restricted to only the first t destinations k =1..t.
Let $C_t(x)$ be the cost incurred when W_1 has an inventory of x, and supplies are sent to D_k, k=1..t, in an optimal manner (the inventory at W_2 is $\Sigma_{k=1..t}\, d_k - x$).
The cost of an optimum solution is $C_n(s_1)$.

a) Use the OSSP to obtain a recurrence relation for $C_t(x)$.
b) Develop an algorithm to obtain an optimum solution to the problem.

24. [CLRS, 2nd edition, Problem 15-7, p. 369]: Scheduling to maximize Profit:
Suppose you have one machine and a set of n jobs $\{J_1, J_2, \ldots, J_n\}$ to process on that machine. Each job J_k has a processing time t_k, a profit p_k, and a deadline d_k. The machine can process only one job at a time, and job J_k must run uninterruptedly for t_k consecutive time units. If job J_k is completed by its deadline d_k, you receive a profit p_k, but if it is completed after its deadline, you receive a profit of 0. Give an algorithm to find the schedule that obtains the maximum amount of profit, assuming that all processing times are integers between 1 and n. What is the running time of your algorithm?

25. Two Machine Job Processing:
You are given a set of n jobs $\{J_1, J_2, \ldots, J_n\}$, each to be processed on only one of two available machines A and B. Jobs cannot be split between machines.
If job J_k is processed on machine A, then A_k units of processing time are needed.
If it is processed on machine B, then B_k units of processing time are needed.
Because of the peculiarities of the jobs and the machines, it is quite possible that $A_k \geq B_k$ for some k, while $A_k < B_k$ for some other k.
A feasible solution is an assignment S of each job to a unique machine.
The objective function Cost(S) is the amount of time units (starting from time 0) required to process all n jobs according to assignment S.
The goal is to find such an assignment S that minimizes Cost(S).
Design & analyze a dynamic programming solution to this problem.

26. A Stock Storage Problem:

You are running a company that sells trucks, and predictions tell you the quantity of sales to expect over the next n months. Let d_k denote the number of sales you expect in month k. We will assume that all sales happen at the beginning of the month, and trucks that are not sold are **stored** until the beginning of the next month.

You can store at most S trucks at any given time, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks.

The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands d_k, and minimize the costs.

In summary: There are two parts to the cost:

(i) **storage** - it costs C for each truck on hand that is not needed that month;

(ii) **ordering fees** - it costs K for each order placed.

In each month you need enough trucks to satisfy the demand d_k, but the number left over after satisfying the demand for the month should not exceed the inventory limit S.

(a) Define Opt(k,t) to be the minimum cost for the first k months, assuming there are t trucks left in the inventory at the end of month k.
Give the complete recurrence equations for Opt(k,t) using the dynamic programming sub-structure optimality property (OSSP).

(b) Using part (a), design and analyze a dynamic programming algorithm that solves this inventory problem by outputting the schedule of placing orders, and runs in time polynomial in n and S.

27. Consider the alphabet $\Sigma = \{a,b,c\}$. The elements of Σ have the multiplication table shown below, where the row is the left-hand symbol and the column is the right-hand symbol. For example **ab = b** and **ba = a**. Observe that the multiplication given by this table is neither associative nor commutative.

Your task is to design an efficient algorithm that when given a string $\mathbf{w = w_1 w_2 \ldots w_n}$ of characters in Σ^*, determines whether or not it is possible to parenthesize **w** such that the resulting expression is **b**. You do not have to give the parenthesization itself.
For example, if **w=abba**, your algorithm should return YES since either **a((bb)a) = b** or **((ab)b)a = b**.

Design and analyze an efficient dynamic programming algorithm for this problem.

	a	b	c
a	c	b	a
b	a	c	a
c	b	b	c

28. A card game against Elmo: Greedy vs. DP:

You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns (starting with you) removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Elmo follows the obvious greedy strategy when it's his turn; Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

a) Prove that you should not also use the greedy strategy. That is, show that there is an instance of the game that you can win, but *lose* with the same greedy strategy as Elmo.

b) Describe and analyze an efficient algorithm to determine, given the initial sequence of n cards, the maximum number of points that you can collect playing against Elmo.

c) Five years later, Elmo has become a *much* stronger player. Describe and analyze an efficient algorithm to determine, given the initial sequence of n cards, the maximum number of points that you can collect playing against a *perfect* opponent.

END