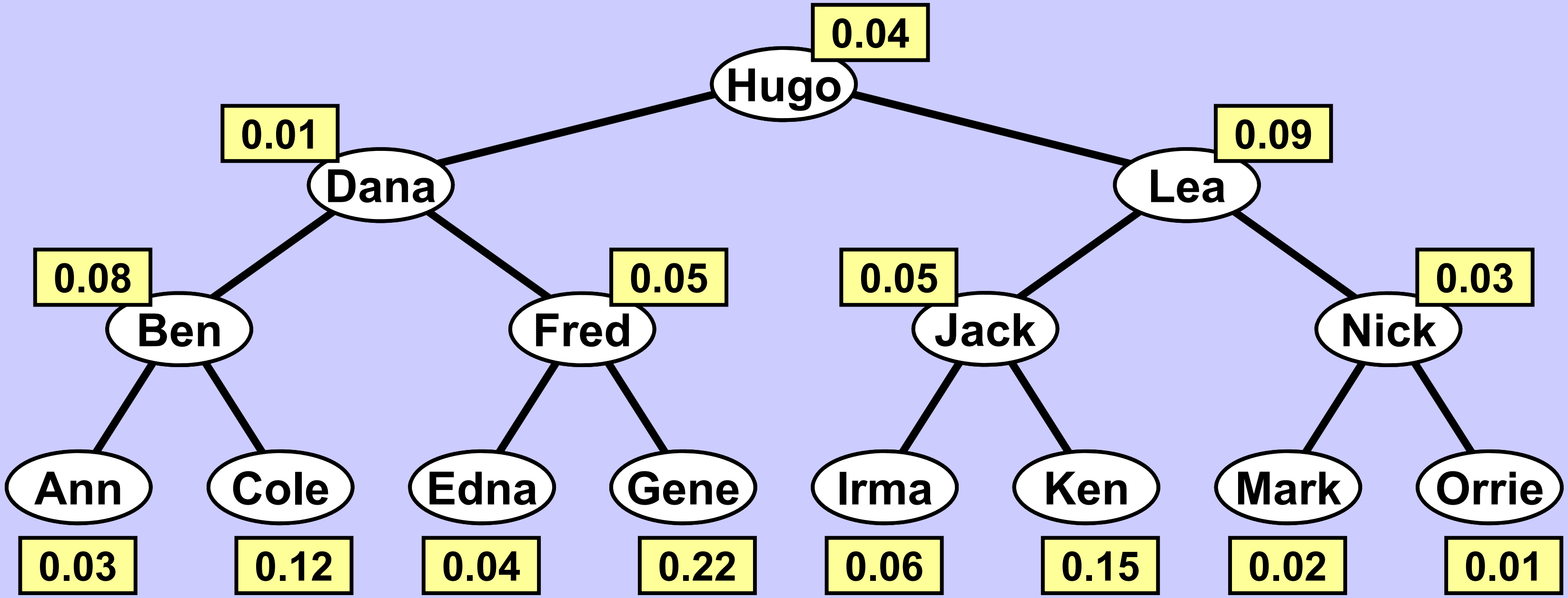
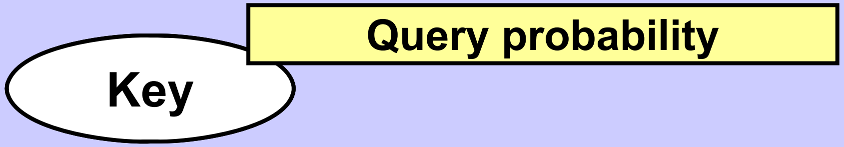
Dynamic programming

Optimal binary search tree

Optimal binary search tree

Balanced but not optimal

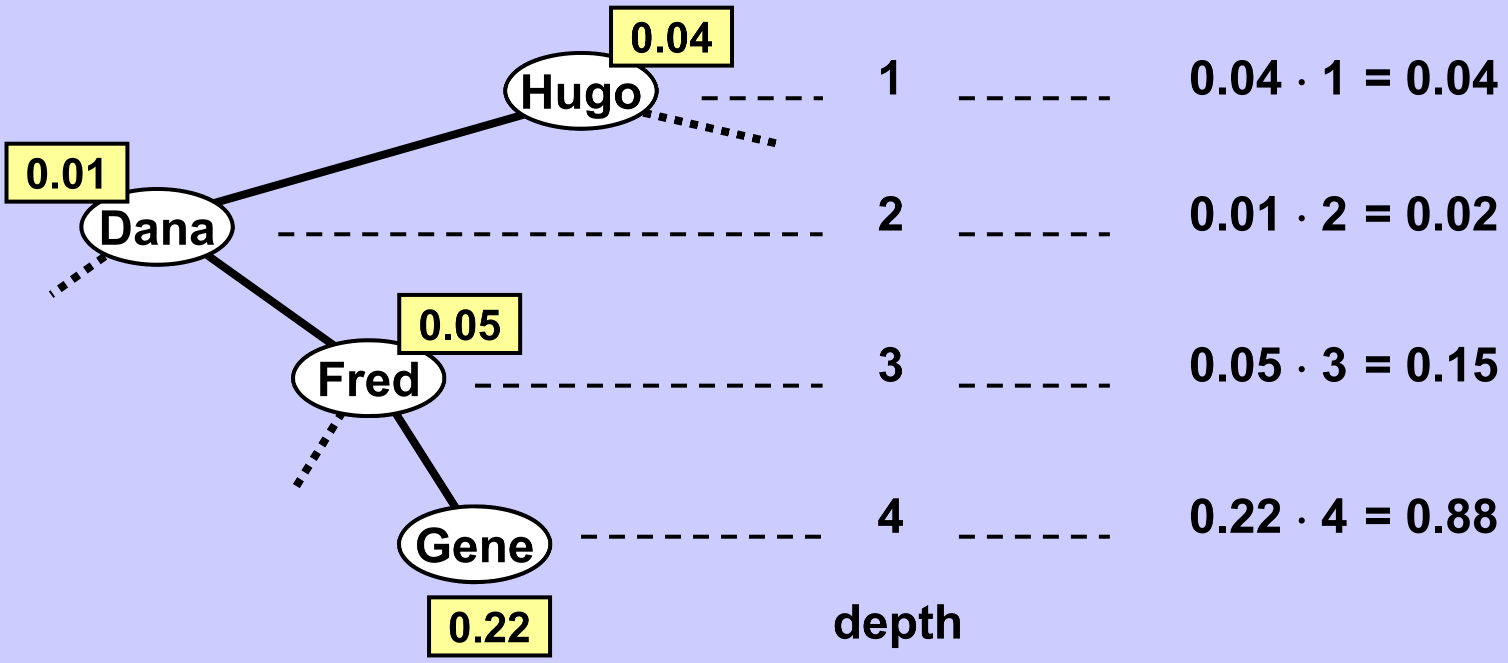




Optimal binary search tree

Node costs in a BST

Node cost = **probability** × depth



Node cost = average no. of tests in one operation Find.

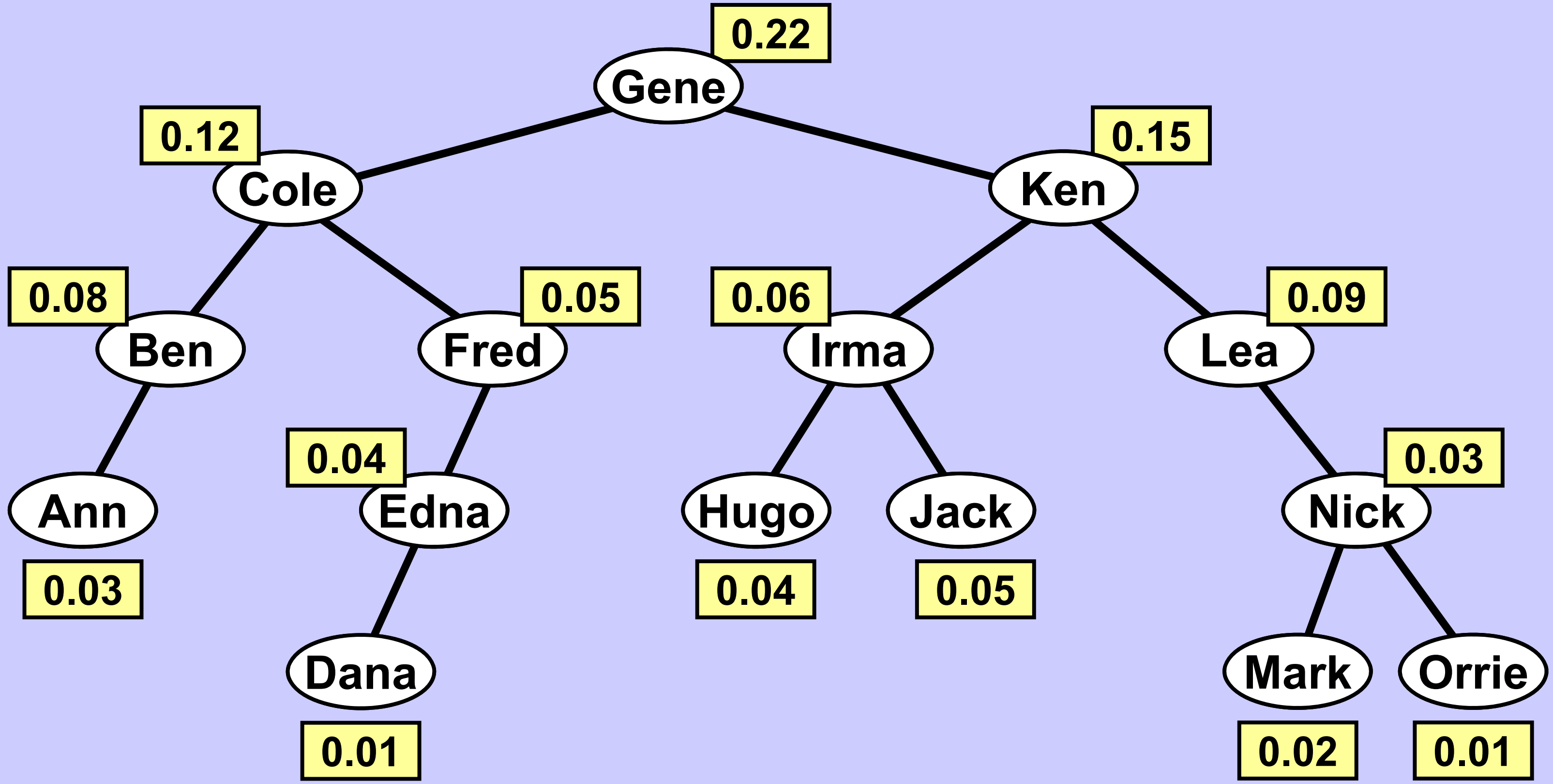
Cost of balanced search tree

key	probab. p_k	depth d_k	$p_k \cdot d_k$
Ann	0.03	4	0.03 · 4 = 0.12
Ben	0.08	3	0.08 · 3 = 0.24
Cole	0.12	4	0.12 · 4 = 0.48
Dana	0.01	2	0.01 · 2 = 0.02
Edna	0.04	4	0.04 · 4 = 0.16
Fred	0.05	3	0.05 · 3 = 0.15
Gene	0.22	4	0.22 · 4 = 0.88
Hugo	0.04	1	0.04 · 1 = 0.04
Irma	0.06	4	0.06 · 4 = 0.24
Jack	0.05	3	0.05 · 3 = 0.15
Ken	0.15	4	0.15 · 4 = 0.60
Lea	0.09	2	0.09 · 2 = 0.18
Mark	0.02	4	0.02 · 4 = 0.08
Nick	0.03	3	0.03 · 3 = 0.09
Orrie	0.01	4	0.01 · 4 = 0.04
		Total cost:	3.47

Total cost = avg. no. of tests in all operatios Find.

Optimal binary search tree

Optimal BST with specific query probabilities

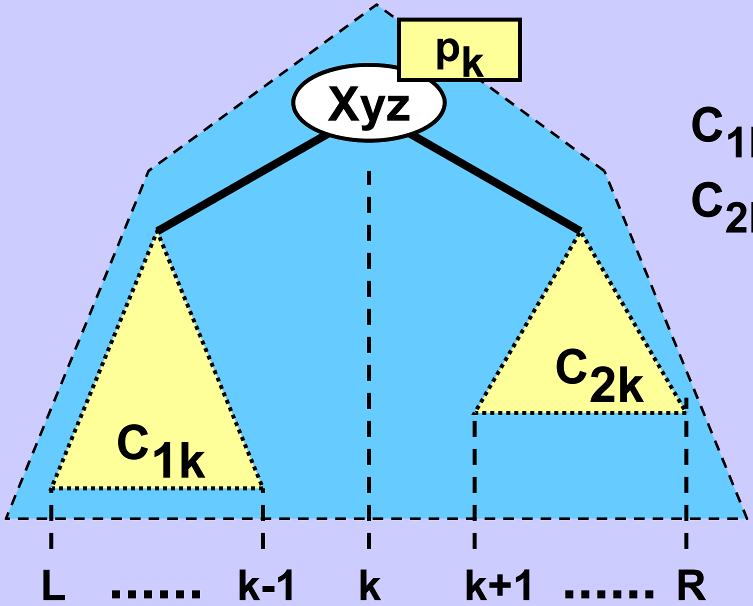


Cost of optimal BST

key	probab. p_k	depth d_k	$p_k \cdot d_k$
Ann	0.03	4	0.03 · 4 = 0.12
Ben	0.08	3	0.08 · 3 = 0.24
Cole	0.12	2	0.12 · 2 = 0.24
Dana	0.01	5	0.01 · 5 = 0.05
Edna	0.04	4	0.04 · 4 = 0.16
Fred	0.05	3	0.05 · 3 = 0.15
Gene	0.22	1	0.22 · 1 = 0.22
Hugo	0.04	4	0.04 · 4 = 0.16
Irma	0.06	3	0.06 · 3 = 0.18
Jack	0.05	4	0.05 · 4 = 0.20
Ken	0.15	2	0.15 · 2 = 0.30
Lea	0.09	3	0.09 · 3 = 0.27
Mark	0.02	5	0.02 · 5 = 0.10
Nick	0.03	4	0.03 · 4 = 0.12
Orrie	0.01	5	0.01 · 5 = 0.05
		Total cost	2.56

Speedup 3.47 : 2.56 = 1 : 0.74

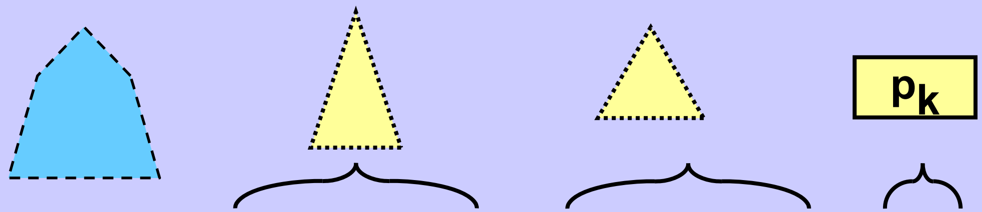
Computing the cost of optimal BST



C_{1k} ….. cost of the left subtree of node k

C_{2k} ….. cost of the right subtree of node k

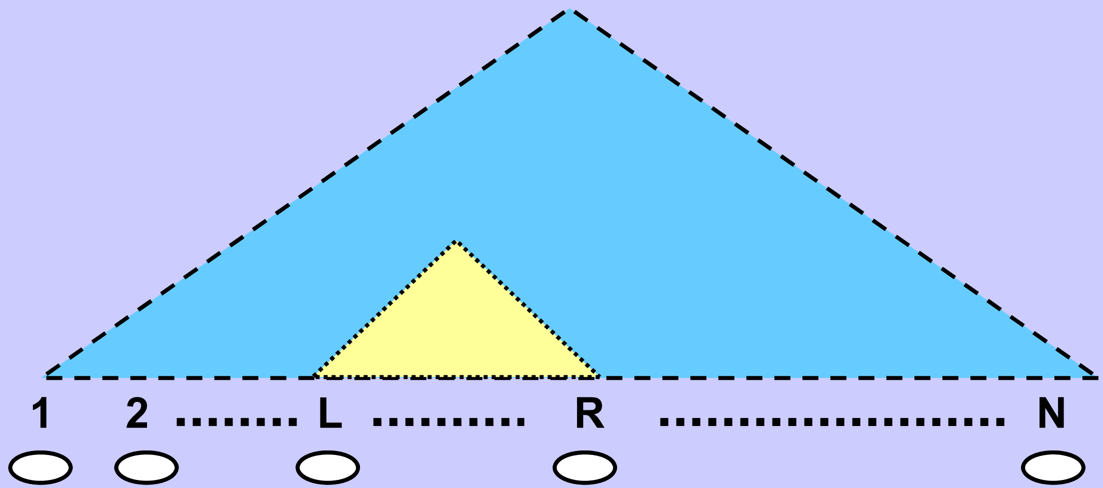
Recursive idea



$$\text{cost} = C_{1k} + \sum_{i=L}^{k-1} p_i + C_{2k} + \sum_{i=k+1}^{R} p_i + p_k$$

Computing the cost of optimal BST

Small optimal subtrees



Surely, there exists an optimal subtree containing keys with indices ranging from L to R.

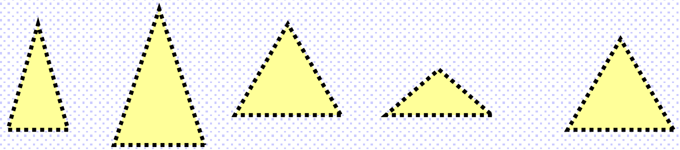
Subtree size
= no. of nodes
= L – R+1

There are	N	optimal subtrees of size	1
	N– 1		2
	N– 2		3
	⋮		⋮
	1	subtree	N

In total, there are N * (N+1) /2 different optimal subtrees.

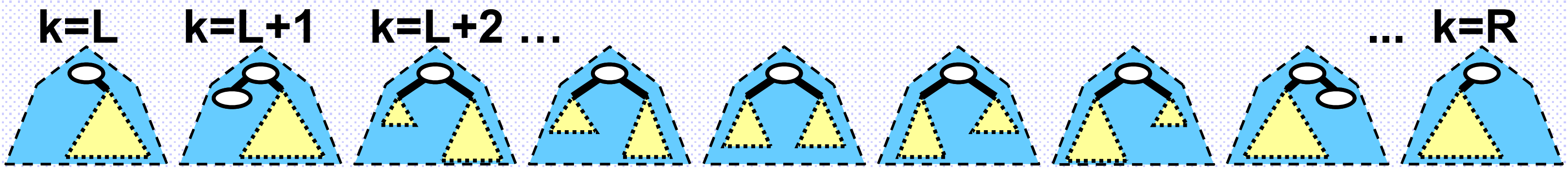
Minimizing the cost of optimal BST

Idea of recursive solution:



1. Assumption: All smaller optimal subtrees are known.

2. Try all possibilities: k = L, L+1, L+2, ..., R



3. Register the index k, which minimizes the cost expressed as

$$C_{1k} + \sum_{i=L}^{k-1} p_i + C_{2k} + \sum_{i=k+1}^{R} p_i + p_k$$

4. The key with index k is the root of the optimal subtree.

Minimizing the cost of optimal BST

C(L,R) ...... Cost of optimal subtree containig keys with indices: L, L+1, L+2, ..., R-1, R

$$C(L,R) = \min_{L \le k \le R} \{ C(L, k-1) + \sum_{i=L}^{k-1} p_i + C(k+1,R) + \sum_{i=k+1}^{R} p_i + p_k \} =$$

$$= \min_{L \le k \le R} \{ C(L, k-1) + C(k+1,R) + \sum_{i=L}^{R} p_i \} =$$

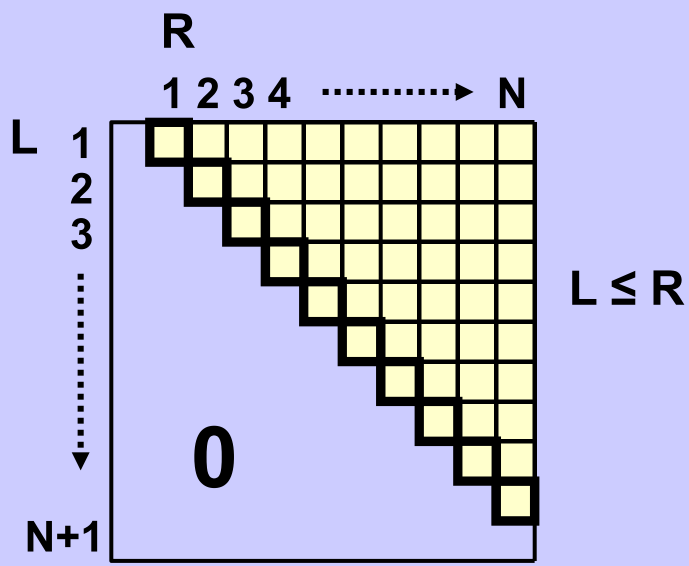
$$(*) \quad = \min_{L \le k \le R} \{ C(L, k-1) + C(k+1,R) \} + \sum_{i=L}^{R} p_i$$

The value minimizing (*) is the index of the root of the optimal subtree

Data structures for computing optimal BST

Costs of optimal subtrees

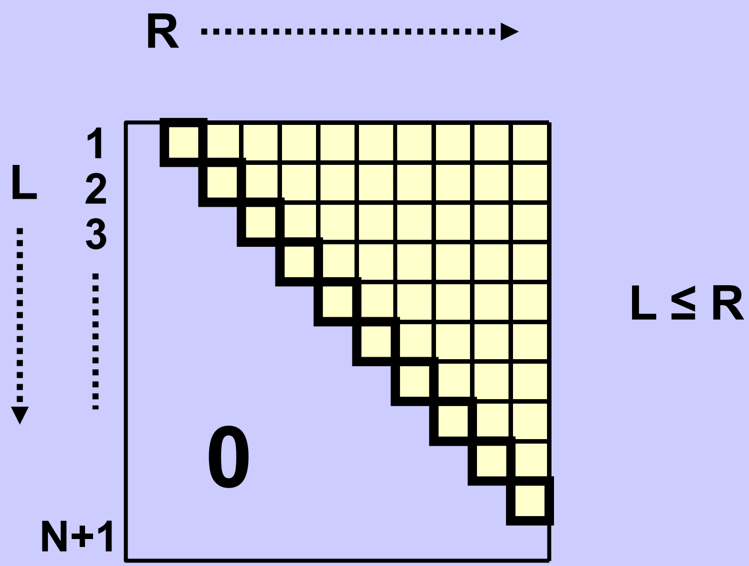
array C [L][R] (L ≤ R)



diagonal ... L= R

Roots of optimal subtrees

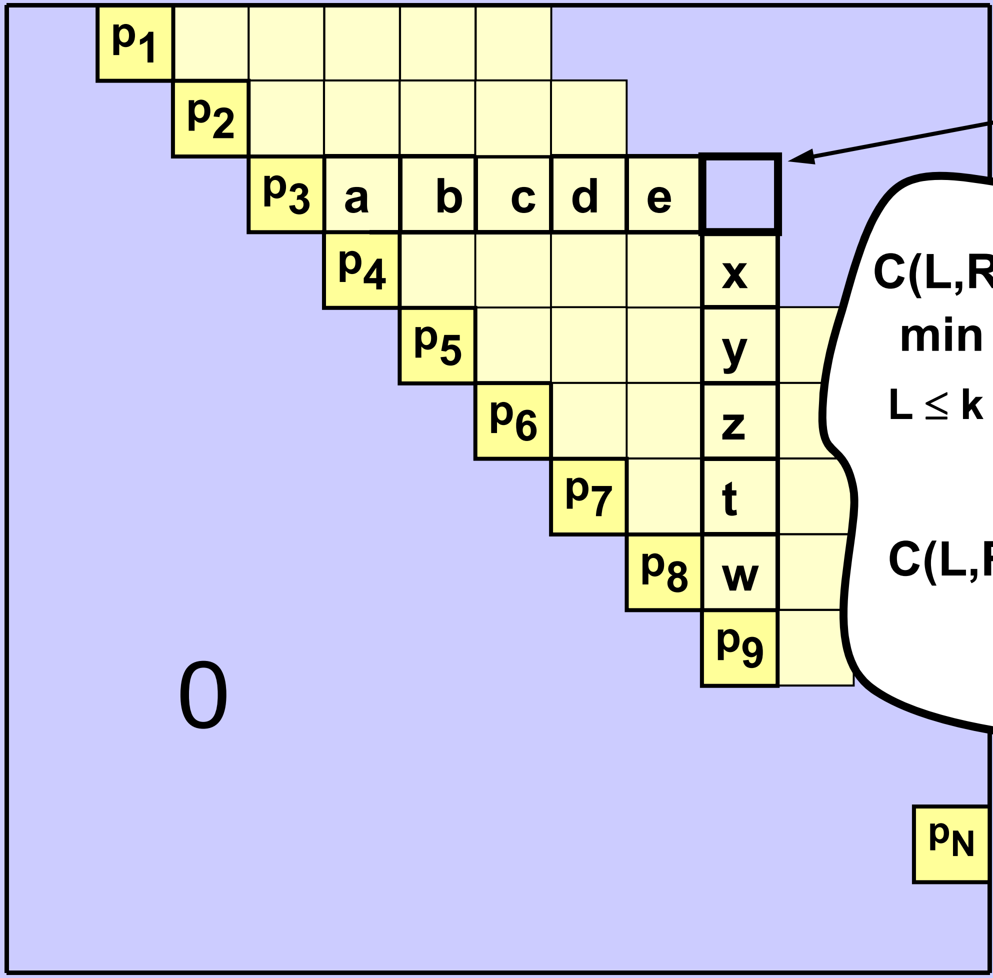
array roots [L][R] (L ≤ R)



diagonal ... L= R

Computing optimal BST

The cost of a particular optimal subtree



L=3, R=9

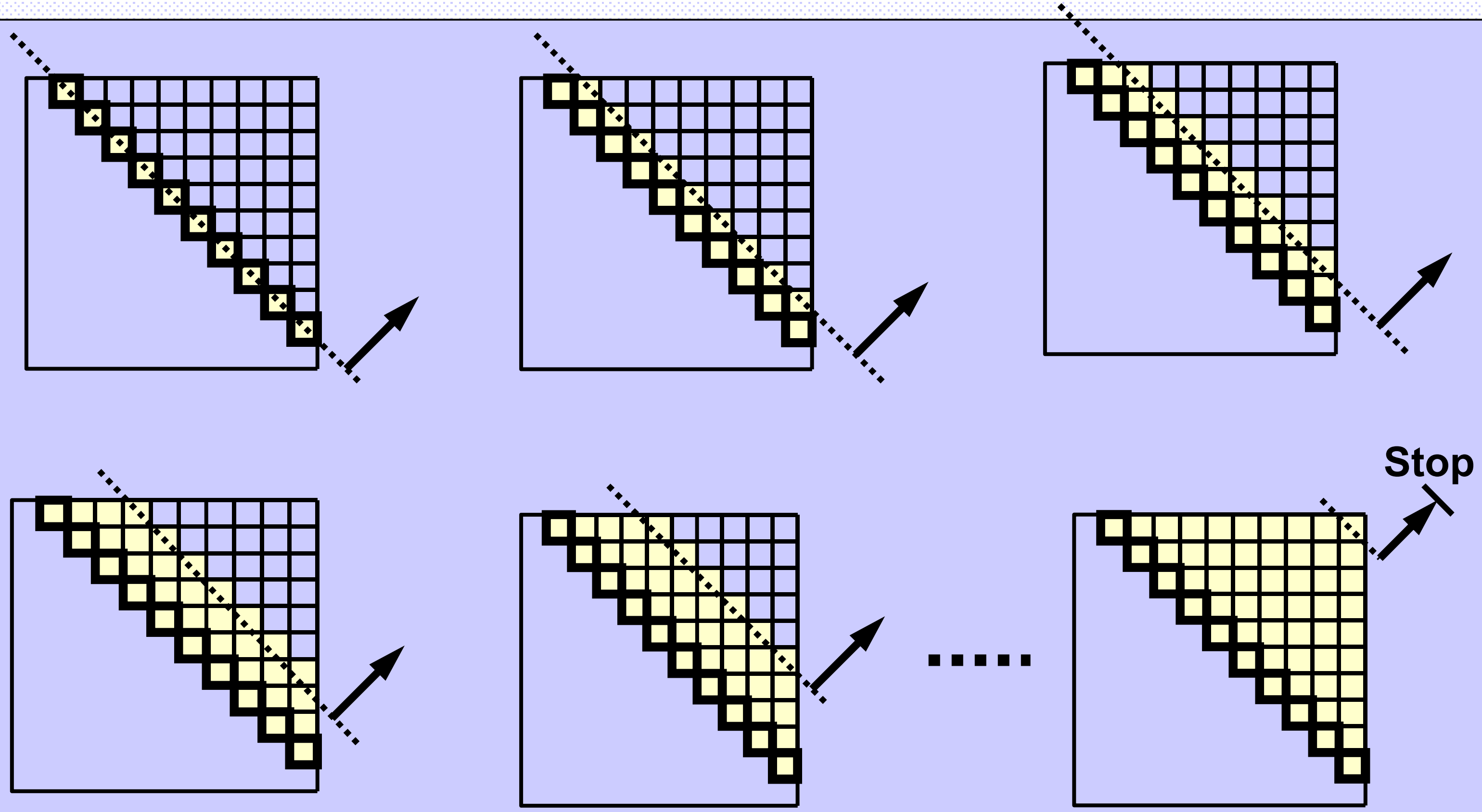
$$C(L,R) = \min_{L \le k \le R} \{ C(L, k-1) + C(k+1,R) \} + \sum_{i=L}^{R} p_i$$

$$C(L,R) = \min\{ 0+x,\ p_3+y,\ a+z,\ b+t,\ c+w,\ d+p_9,\ e+0 \}$$

Computing optimal BST

Dynamic programming strategy

– First process the smallest subtrees, then the bigger ones, then still more bigger ones, etc...



Computing optimal BST

Computing arrays of costs and roots

```
def optimalTree( Prob, N ):
  Costs = [[0]*N  for i in range(N)]
  Roots = [[0]*N  for i in range(N)]
  # size = 1, main diagonal
  for i in range(N):
    Costs[i][i] = Prob[i]; Roots[i][i] = i

  # size > 1, diagonals above main diagonal
  for size in range(1, N):
    L = 1; R = size
    while R < N:
      Costs[L][R] =
           min(Costs[L][k-1] + Costs[k+1][R], k = L..R)
      roots[L][R] = 'k minimizing previous line'
      Costs[L][R] += sum(Costs[L:R+1])
      L += 1; R += 1
   return Costs, Roots
```

Computing optimal BST

Building optimal BST using the array of subtree roots

```
# standard BST insert
def buildTree( Tree, L, R, Roots, Nodes ):
  if R < L: return
  rootindex = Roots[L][R]
  # standard BST insert
  # nodes in Nodes have to be sorted in increasing
  # order of their key values
  Tree.insert( Nodes[rootindex].key )
  buildTree( Tree, L, rootindex-1, Roots, Nodes )
  buildTree( Tree, rootindex+1, R, Roots, Nodes )
}
```

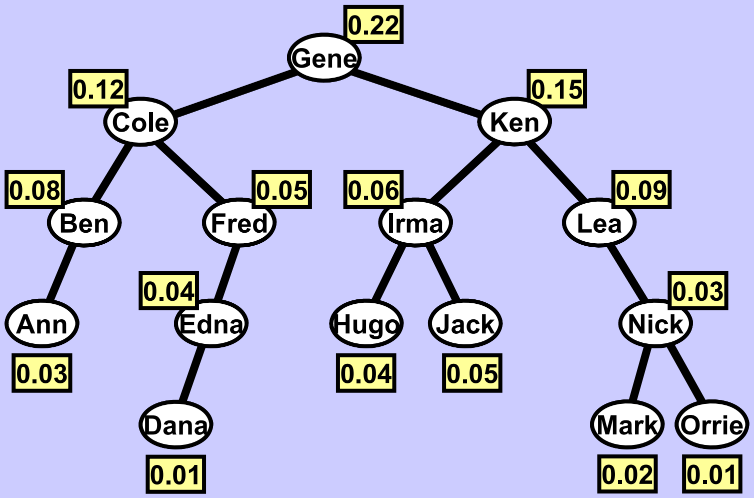
Computing optimal BST

Roots of optimal subtrees

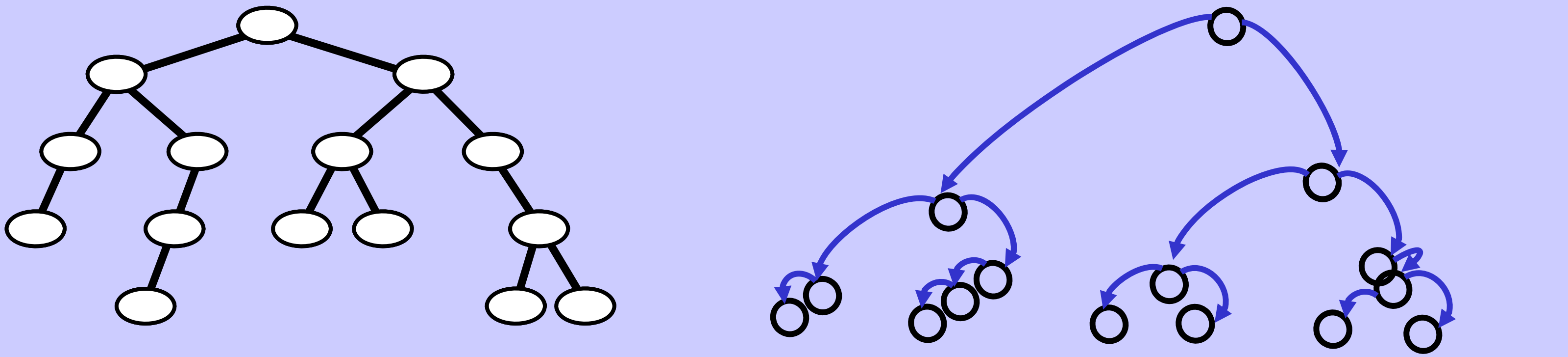
0	1	2	3	3	3	3	3	7	7	7	7	7	7	7	7
0	0	2	3	3	3	3	7	7	7	7	7	7	7	7	7
0	0	0	3	3	3	3	7	7	7	7	7	7	7	7	7
0	0	0	0	4	5	6	7	7	7	7	7	7	7	7	7
0	0	0	0	0	5	6	7	7	7	7	7	7	7	7	7
0	0	0	0	0	0	6	7	7	7	7	7	7	7	11	11
0	0	0	0	0	0	0	7	7	7	7	7	11	11	11	11
0	0	0	0	0	0	0	0	8	9	9	11	11	11	11	11
0	0	0	0	0	0	0	0	0	9	9	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	10	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	0	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	0	0	12	12	12	12
0	0	0	0	0	0	0	0	0	0	0	0	0	13	14	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Computing optimal BST

Tree and array correspondence



0	1	2	3	3	3	3	3	7	7	7	7	7	7	7	7
0	0	2	3	3	3	3	7	7	7	7	7	7	7	7	7
0	0	0	3	3	3	3	7	7	7	7	7	7	7	7	7
0	0	0	0	4	5	6	7	7	7	7	7	7	7	7	7
0	0	0	0	0	5	6	7	7	7	7	7	7	7	7	7
0	0	0	0	0	0	6	7	7	7	7	7	7	7	11	11
0	0	0	0	0	0	0	7	7	7	7	7	11	11	11	11
0	0	0	0	0	0	0	0	8	9	9	11	11	11	11	11
0	0	0	0	0	0	0	0	0	9	9	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	10	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	0	11	11	11	11	11
0	0	0	0	0	0	0	0	0	0	0	0	12	12	12	12
0	0	0	0	0	0	0	0	0	0	0	0	0	13	14	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



Computing optimal BST

Costs of optimal subtrees

	1-A	2-B	3-C	4-D	5-E	6-F	7-G	8-H	9-I	10-J	11-K	12-L	13-M	14-N	15-O
1-A	0.03	0.14	0.37	0.39	0.48	0.63	1.17	1.26	1.42	1.57	2.02	2.29	2.37	2.51	2.56
2-B	0	0.08	0.28	0.30	0.39	0.54	1.06	1.14	1.30	1.45	1.90	2.17	2.25	2.39	2.44
3-C	0	0	0.12	0.14	0.23	0.38	0.82	0.90	1.06	1.21	1.66	1.93	2.01	2.15	2.20
4-D	0	0	0	0.01	0.06	0.16	0.48	0.56	0.72	0.87	1.32	1.59	1.67	1.81	1.86
5-E	0	0	0	0	0.04	0.13	0.44	0.52	0.68	0.83	1.28	1.55	1.63	1.77	1.82
6-F	0	0	0	0	0	0.05	0.32	0.40	0.56	0.71	1.16	1.43	1.51	1.63	1.67
7-G	0	0	0	0	0	0	0.22	0.30	0.46	0.61	1.06	1.31	1.37	1.48	1.52
8-H	0	0	0	0	0	0	0	0.04	0.14	0.24	0.54	0.72	0.78	0.89	0.93
9-I	0	0	0	0	0	0	0	0	0.06	0.16	0.42	0.60	0.66	0.77	0.81
10-J	0	0	0	0	0	0	0	0	0	0.05	0.25	0.43	0.49	0.60	0.64
11-K	0	0	0	0	0	0	0	0	0	0	0.15	0.33	0.39	0.50	0.54
12-L	0	0	0	0	0	0	0	0	0	0	0	0.09	0.13	0.21	0.24
13-M	0	0	0	0	0	0	0	0	0	0	0	0	0.02	0.07	0.09
14-N	0	0	0	0	0	0	0	0	0	0	0	0	0	0.03	0.05
15-O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.01

Dynamic programming

Longest common subsequence (LCS)

Longest common subsequence

Two sequences

A: C B E A D D E A — |A| = 8

B: D E C D B D A — |B| = 7

Common subsequence

A: **C** B E A **D** D E **A**

B: D E **C** D B **D** **A**

C: **C D A** — |C| = 3

Longest common subsequence (LCS)

A: C B **E** A **D** **D** E **A**

B: D **E** C **D** B **D** **A**

C: **E D D A** — |C| = 4

Longest common subsequence

A_n: $(a_1, a_2, \ldots, a_n)$

B_m: $(b_1, b_2, \ldots, b_m)$

C_k: $(c_1, c_2, \ldots, c_k)$

....

$C_k = LCS(A_n, B_m)$

	1	2	3	4	5	6	7	8
A_8:	C	B	E	A	D	D	E	A
B_7:	D	E	C	D	B	D	A	
C_4:	E	D	D	A				

Recursive rules:

$$(a_n = b_m) \implies (c_k = a_n = b_m) \ \& \ (C_{k-1} = LCS(A_{n-1}, B_{m-1}))$$

	1	2	3	4	5	6	7	8
A_8:	C	B	E	A	D	D	E	A
B_7:	D	E	C	D	B	D	A	
C_4:	E	D	D	A				

	1	2	3	4	5	6	7	8
A_7:	C	B	E	A	D	D	E	A
B_6:	D	E	C	D	B	D	A	
C_3:	E	D	D	A				

Longest common subsequence

$(a_n \mathrel{!}= b_m) \;\&\; (c_k \mathrel{!}= a_n) \implies (C_k = LCS(A_{n-1}, B_m))$

	1	2	3	4	5	6	7	8
A_7:	C	B	E	A	D	D	E	
B_6:	D	E	C	D	B	D		
C_3:	E	D	D					

	1	2	3	4	5	6	7	8
A_6:	C	B	E	A	D	D	E	
B_6:	D	E	C	D	B	D		
C_3:	E	D	D					

$(a_n \mathrel{!}= b_m) \;\&\; (c_k \mathrel{!}= b_m) \implies (C_k = LCS(A_n, B_{m-1}))$

	1	2	3	4	5	6	7	8
A_5:	C	B	E	A	D			
B_5:	D	E	C	D	B			
C_2:	E	D						

	1	2	3	4	5	6	7	8
A_5:	C	B	E	A	D			
B_4:	D	E	C	D	B			
C_2:	E	D						

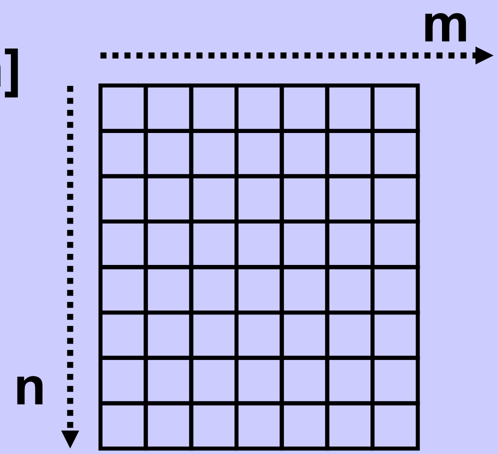
Longest common subsequence

Recursive function c(m, n) computes LCS length

$$C(n,m) = \begin{cases} 0 & n = 0 \text{ or } m = 0 \\ C(n-1, m-1) + 1 & n > 0, m > 0, a_n = b_m \\ \max\{ C(n-1, m), C(n, m-1) \} & n > 0, m > 0, a_n \neq b_m \end{cases}$$

Dynamic programming strategy

C[n][m]




```
for a in range( 1, n+1 ):
  for b in range( 1, m+1 ):
    C[a][b] = ....
}
```

Longest common subsequence

Construction of 2D LCS array

```
def findLCS():
  for a in range( 1, n+1 ):
    for b in range( 1, m+1 ):
      if A[a] == B[b]:
         C[a][b] = C[a-1][b-1]+1
         arrows[a][b] = DIAG ↖

      else:
        if C[a-1][b] > C[a][b-1]:
           C[a][b] = C[a-1][b];
           arrows[a][b] = UP ↑
        else:
          C[a][b] = C[a][b-1];
          arrows[a][b] = LEFT ←
```

Longest common subsequence

LCS array for "CBEADDEA" and "DECDBDA"

C

			0	1	2	3	4	5	6	7
		B:		D	E	C	D	B	D	A
	0		0	0	0	0	0	0	0	0
	1	C	0	← 0	← 0	↖ 1	← 1	← 1	← 1	← 1
	2	B	0	← 0	← 0	↑ 1	← 1	↖ 2	← 2	← 2
	3	E	0	← 0	↖ 1	← 1	← 1	↑ 2	← 2	← 2
A:	4	A	0	← 0	↑ 1	← 1	← 1	↑ 2	← 2	↖ 3
	5	D	0	↖ 1	← 1	← 1	↖ 2	← 2	↖ 3	← 3
	6	D	0	↖ 1	← 1	← 1	↖ 2	← 2	↖ 3	← 3
	7	E	0	↑ 1	↖ 2	← 2	← 2	← 2	↑ 3	← 3
	8	A	0	↑ 1	↑ 2	← 2	← 2	← 2	↑ 3	↖ 4

Longest common subsequence

LSC printout -- recursively :)

```
def outLCS(a, b) {
if a == 0 or b == 0 return

if arrows[a][b] == DIAG:
    outLCS(a-1, b-1);       // recursion ...
    print(A[a])             //... reverses the sequence!

else:
    if arrows[a][b] == UP:
         outLCS(a-1,b);
    else:
         outLCS(a,b-1);
}
```