CSE 548: Analysis of Algorithms

Prerequisites Review 5
(Dynamic Programming)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Fall 2019

The Rod Cutting Problem

Suppose you are given:

- a rod of length n inches, and
- a list of prices p_i for integer $i \in [1, n]$,
  where p_i is the selling price of a rod of length i inches.

Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

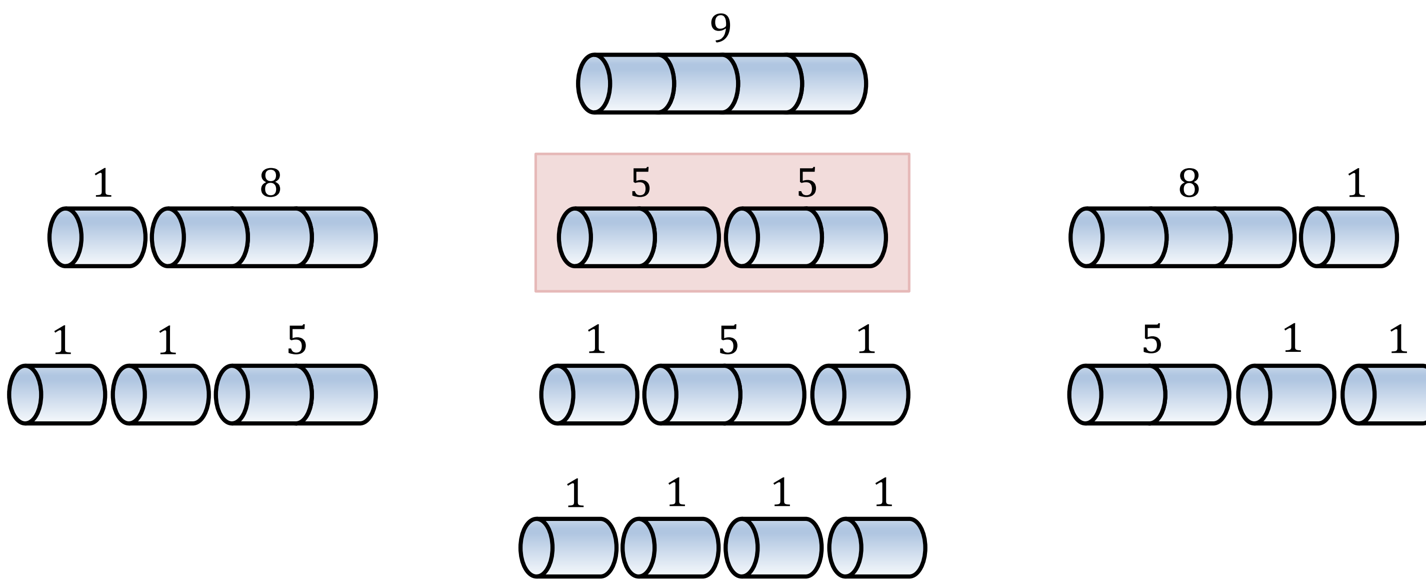
The Rod Cutting Problem

A sample price table for rods

length i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
price p_i	1	5	8	9	10	17	17	20	24	30

Solve the problem for $n = 4$ and the price table given above.

#pieces	**#ways**
1	$\binom{3}{0} = 1$
2	$\binom{3}{1} = 3$
3	$\binom{3}{2} = 3$
4	$\binom{3}{3} = 1$
Total:	$8 = 2^3$



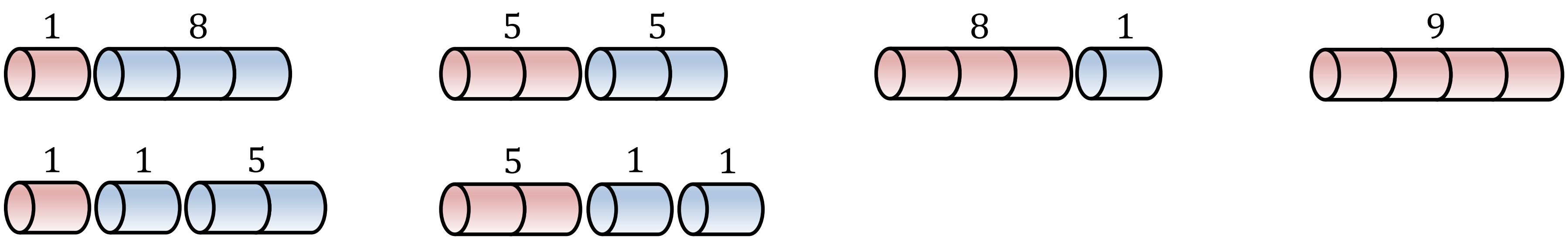
$r_n = 5 + 5 = 10$

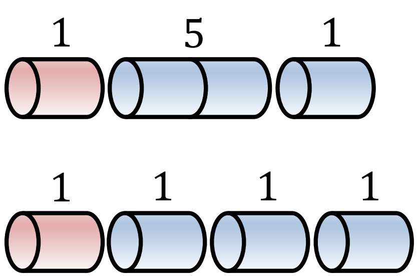
Rod Cutting: Standard Recursive Algorithm

A sample price table for rods

length i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
price p_i	1	5	8	9	10	17	17	20	24	30

There is a different way of looking at the cuts and thus computing r_n.





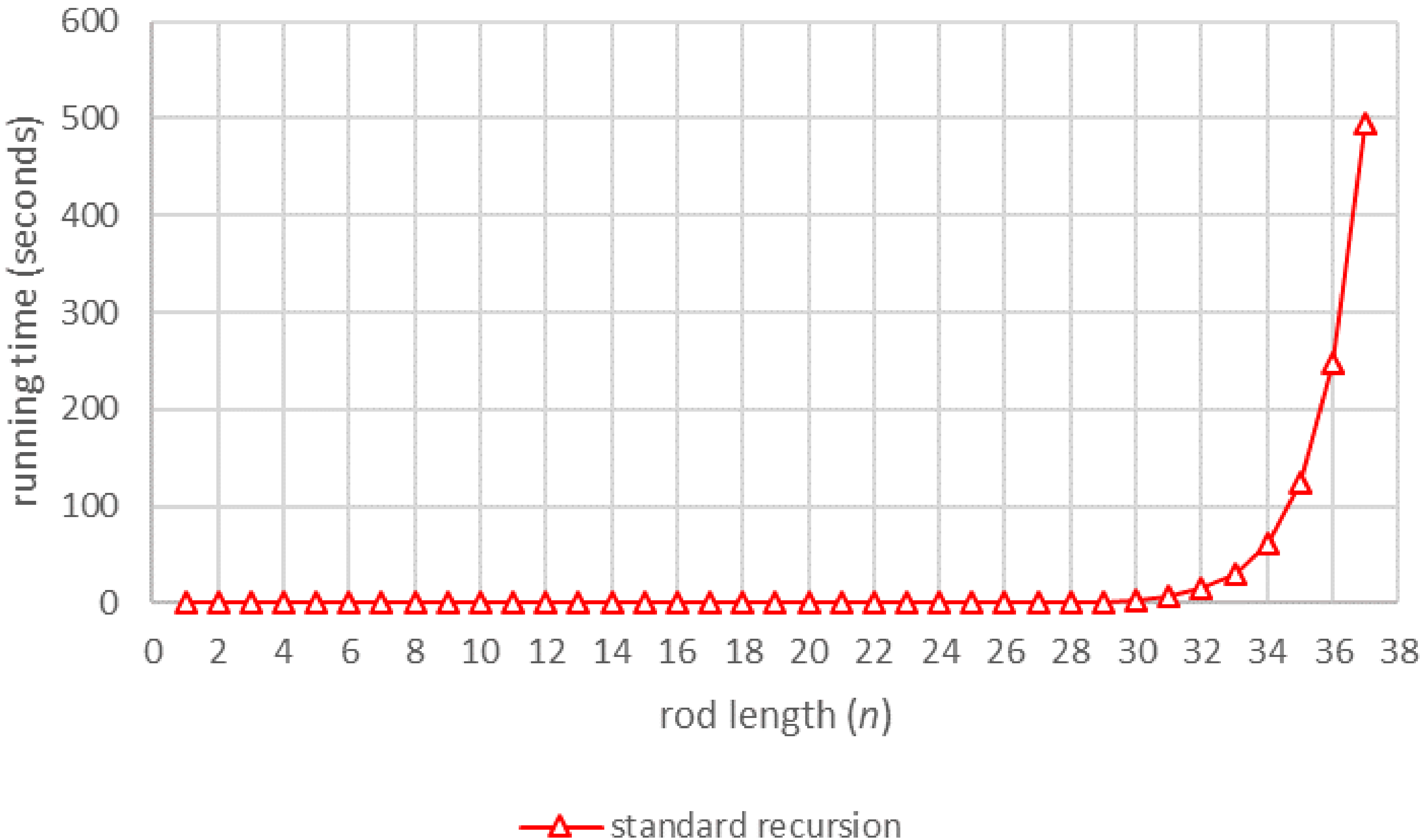
$$r_n = \begin{cases} 0, & if\ n = 0, \\ \max\limits_{1 \le i \le n} \{p_i + r_{n-i}\}, & if\ n > 0. \end{cases}$$

Rod Cutting: Standard Recursive Algorithm

$$r_n = \begin{cases} 0, & if\ n = 0, \\ \max\limits_{1 \le i \le n} \{p_i + r_{n-i}\}, & if\ n > 0. \end{cases}$$

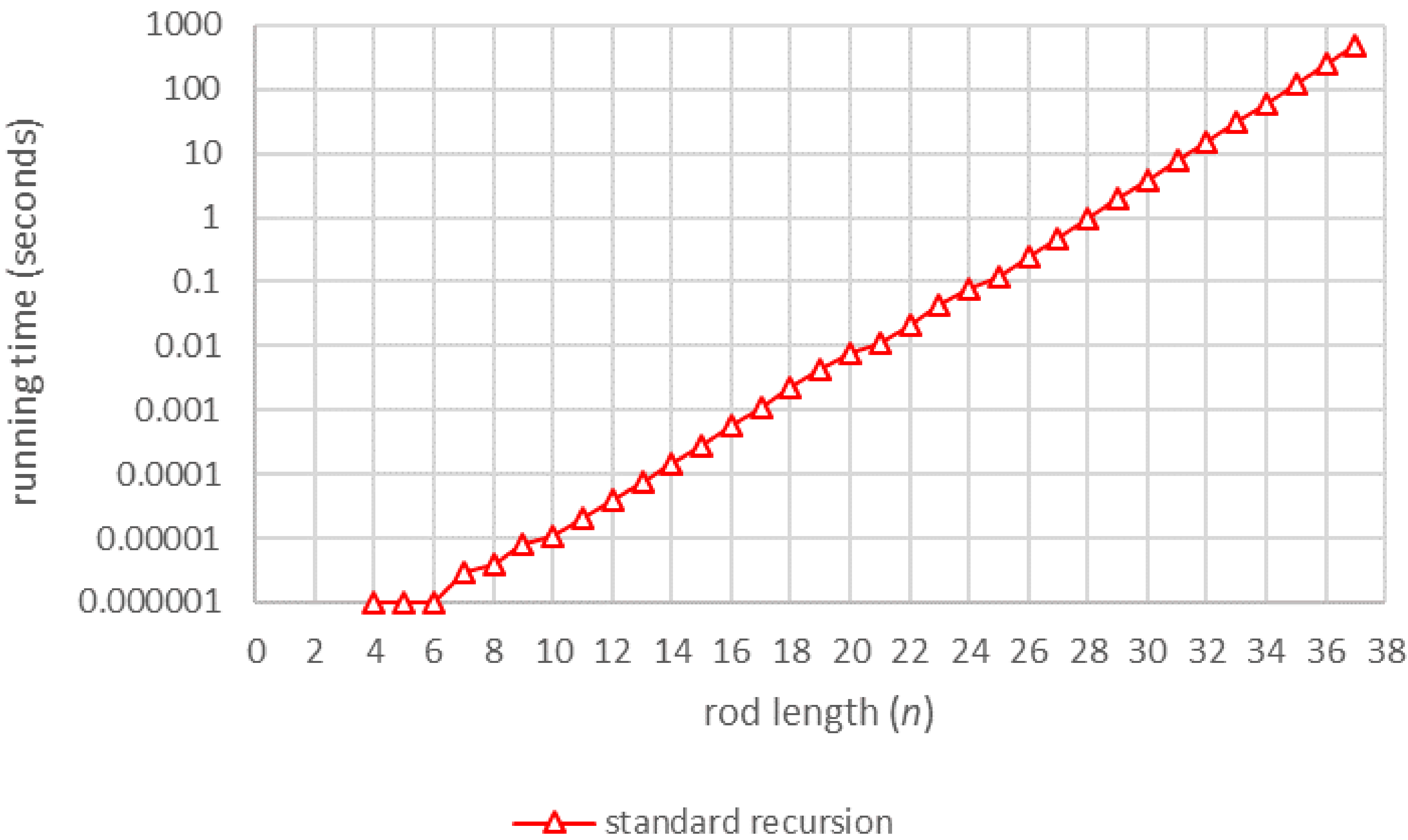
```
CUT-ROD ( p, n )
if n = 0 then
    return 0
q ← − ∞
for i ←1 to n do
    q ← max{ q, p[ i ] + CUT-ROD ( p, n − i ) }
return q
```

Rod Cutting: Standard Recursive Algorithm



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Standard Recursive Algorithm



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Standard Recursive Algorithm

```
CUT-ROD ( p, n )
  if n = 0 then
      return 0
  q ← −∞
  for i ← 1 to n do
      q ← max{ q, p[ i ] + CUT-ROD ( p, n − i ) }
  return q
```

Let $T(n)$ be the running time of the algorithm on an input of size n.

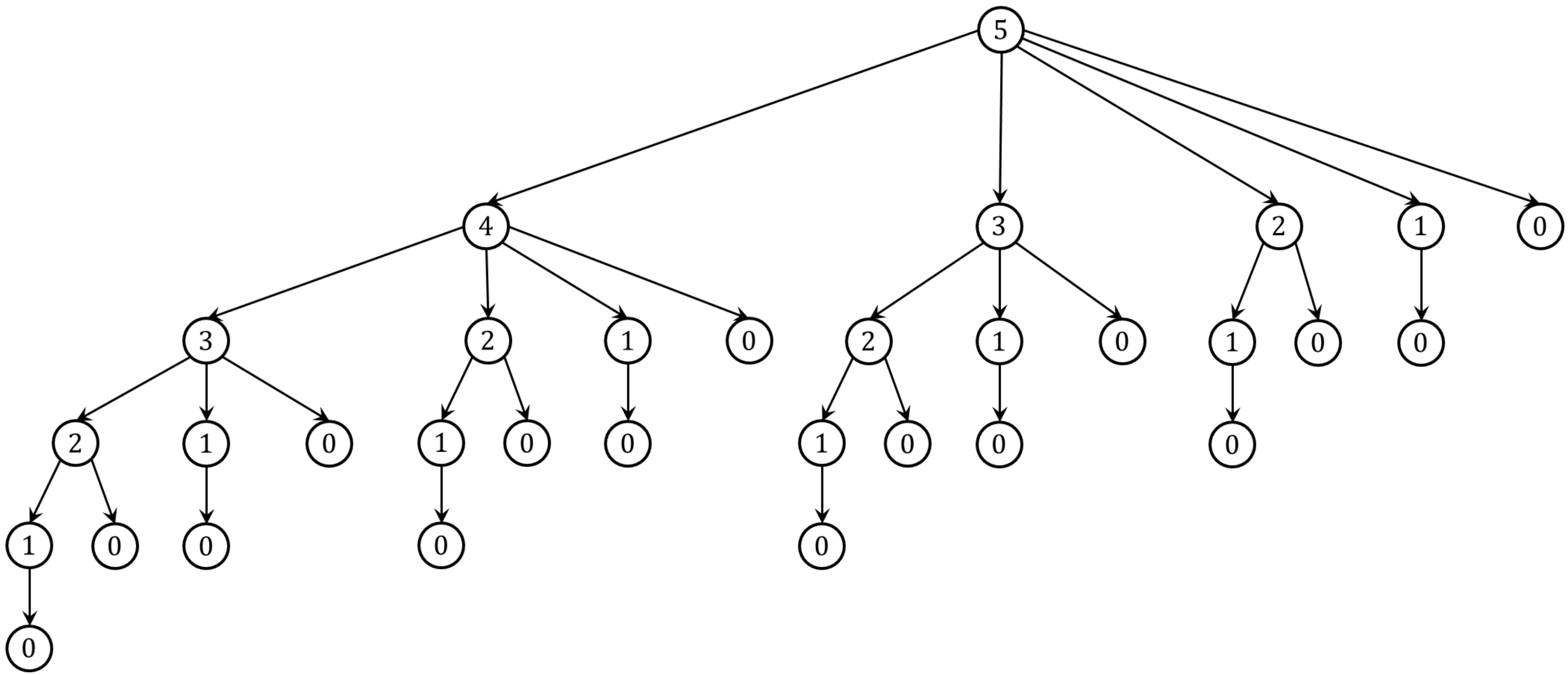
Then

$$T(n) = \begin{cases} \Theta(1), & if\ n = 0, \\ \sum_{i=1}^{n} T(n-i) + \Theta(1), & if\ n > 0. \end{cases}$$

Solving: $T(n) = \Theta(2^n)$.

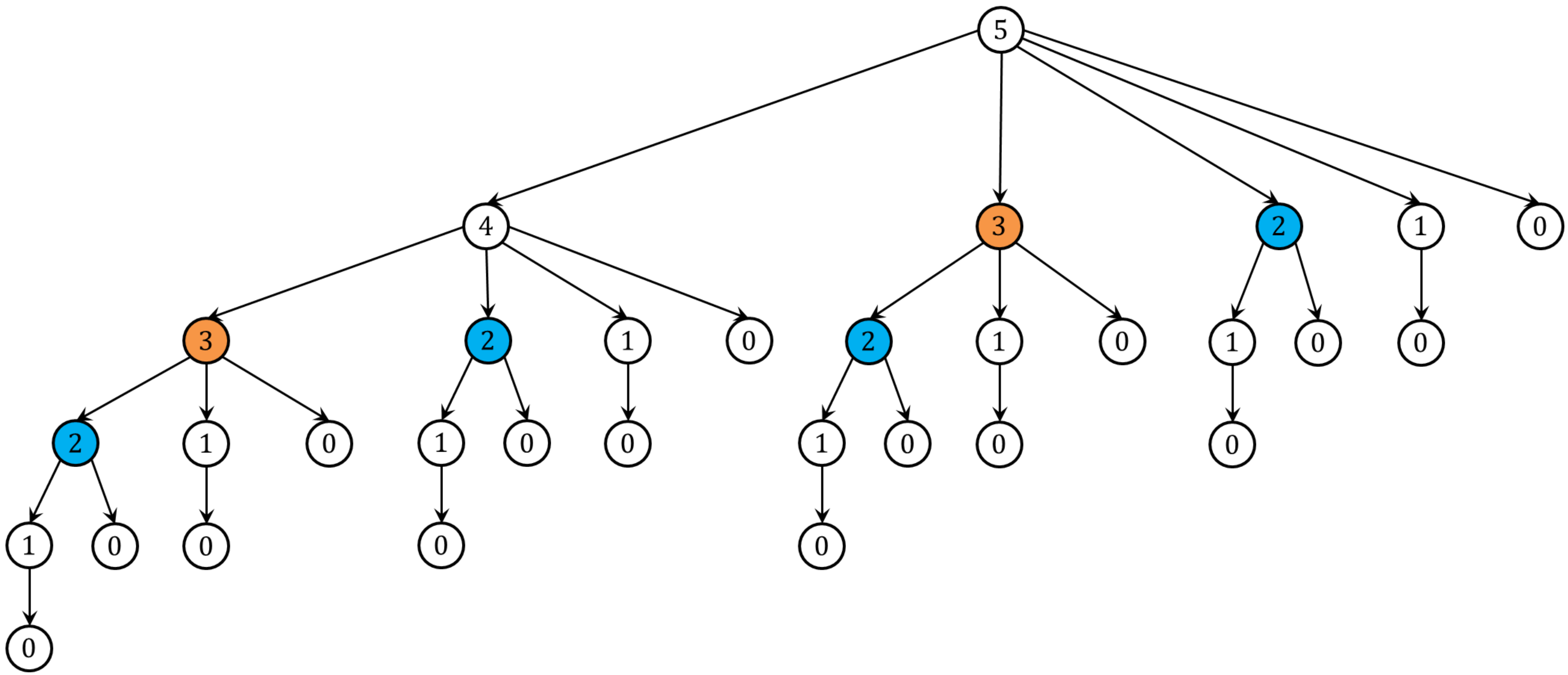
Rod Cutting: Standard Recursive Algorithm

When CUT-ROD(n) is called with $n = 5$, the values of n passed to the recursive function calls are shown below.



Rod Cutting: Standard Recursive Algorithm

When CUT-ROD(n) is called with $n = 5$, the values of n passed to the recursive function calls are shown below.



We are calling CUT-ROD(n) or solving the problem for the same value of n over and over again!

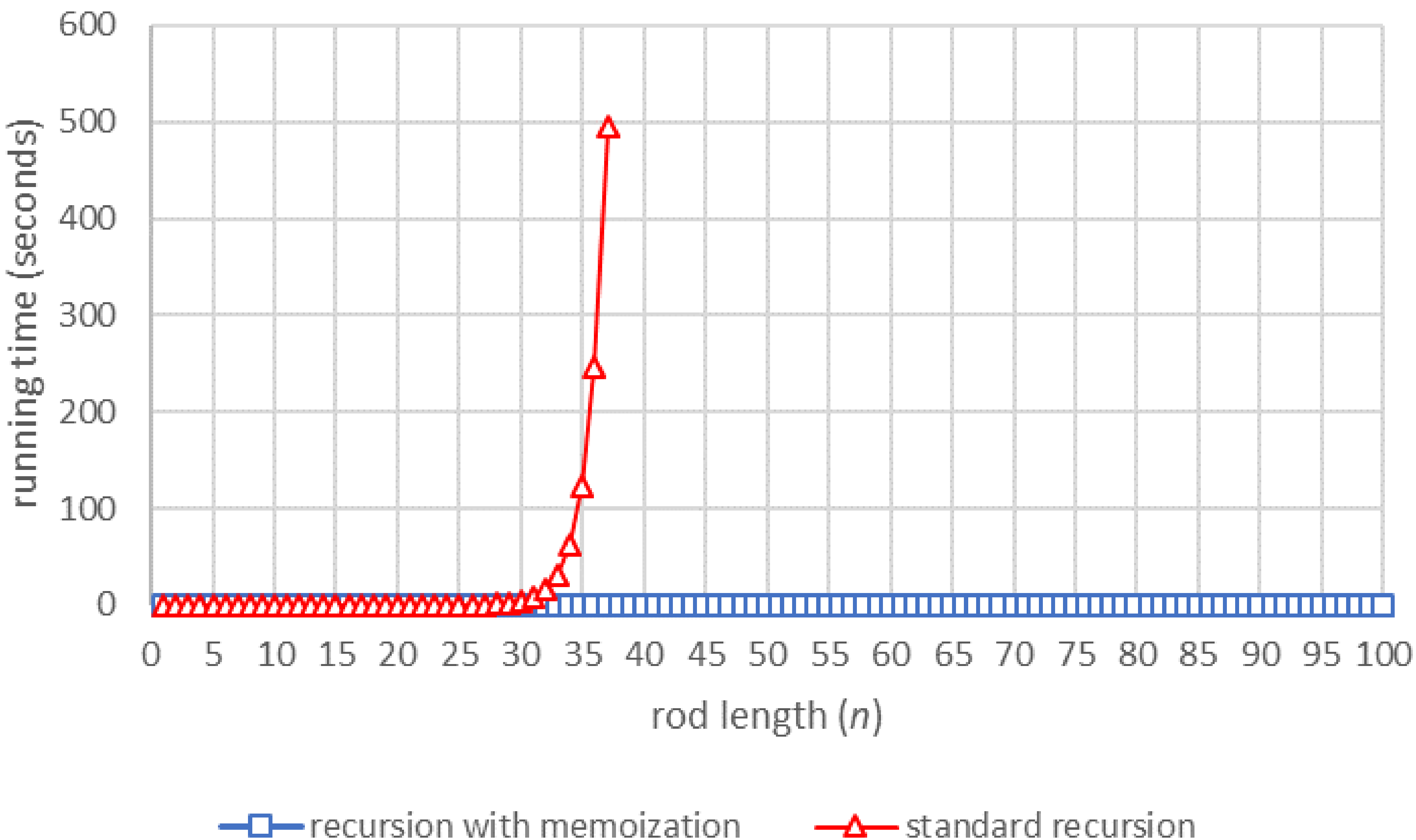
How about saving the solution when we solve the problem for any given value of n for the first time?

Rod Cutting: Recursion with Memoization

```
MEMOIZED-CUT-ROD ( p, n )

r[0..n] ← new array
for i ← 0 to n do
    r[i] ← −∞
return MEMOIZED-CUT-ROD-AUX ( p, n, r )
```

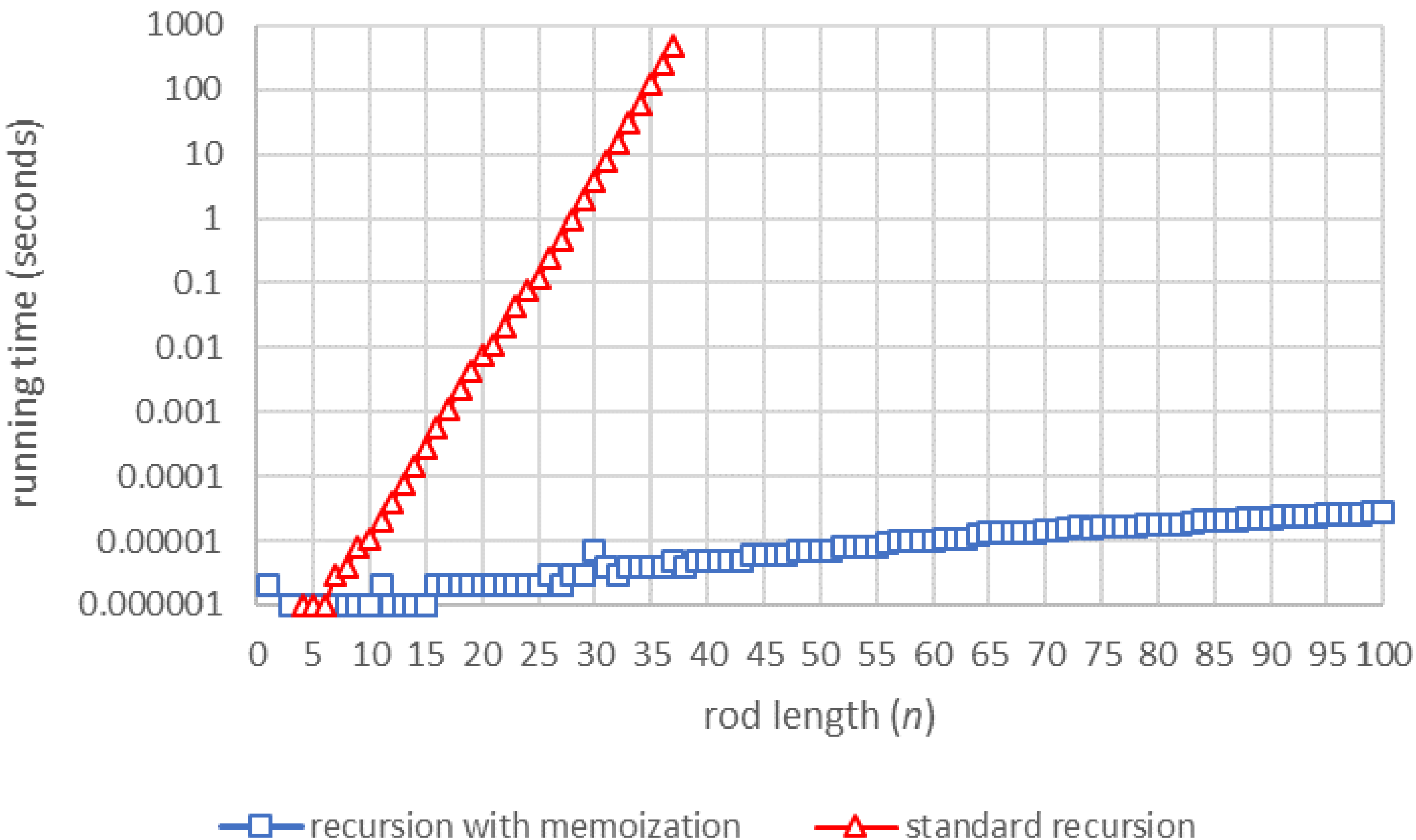
```
MEMOIZED-CUT-ROD-AUX ( p, n, r )

if r[n] ≥ 0 then
    return r[n]
if n = 0 then
    q ← 0
else q ← −∞
    for i ← 1 to n do
        q ← max{ q, p[ i ] + MEMOIZED-CUT-ROD-AUX ( p, n − i, r ) }
r[n] ← q
return q
```

Rod Cutting: Recursion with Memoization



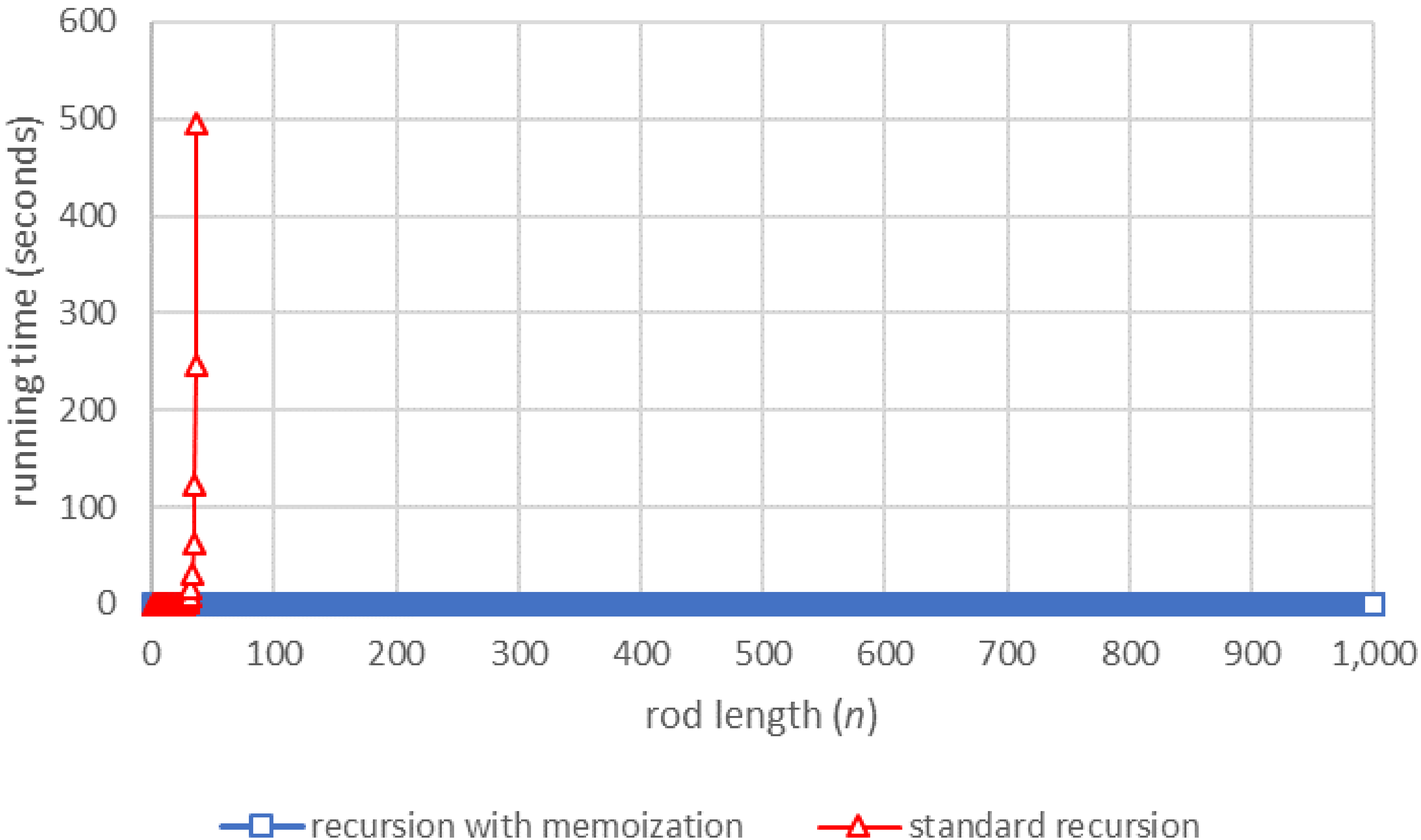
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



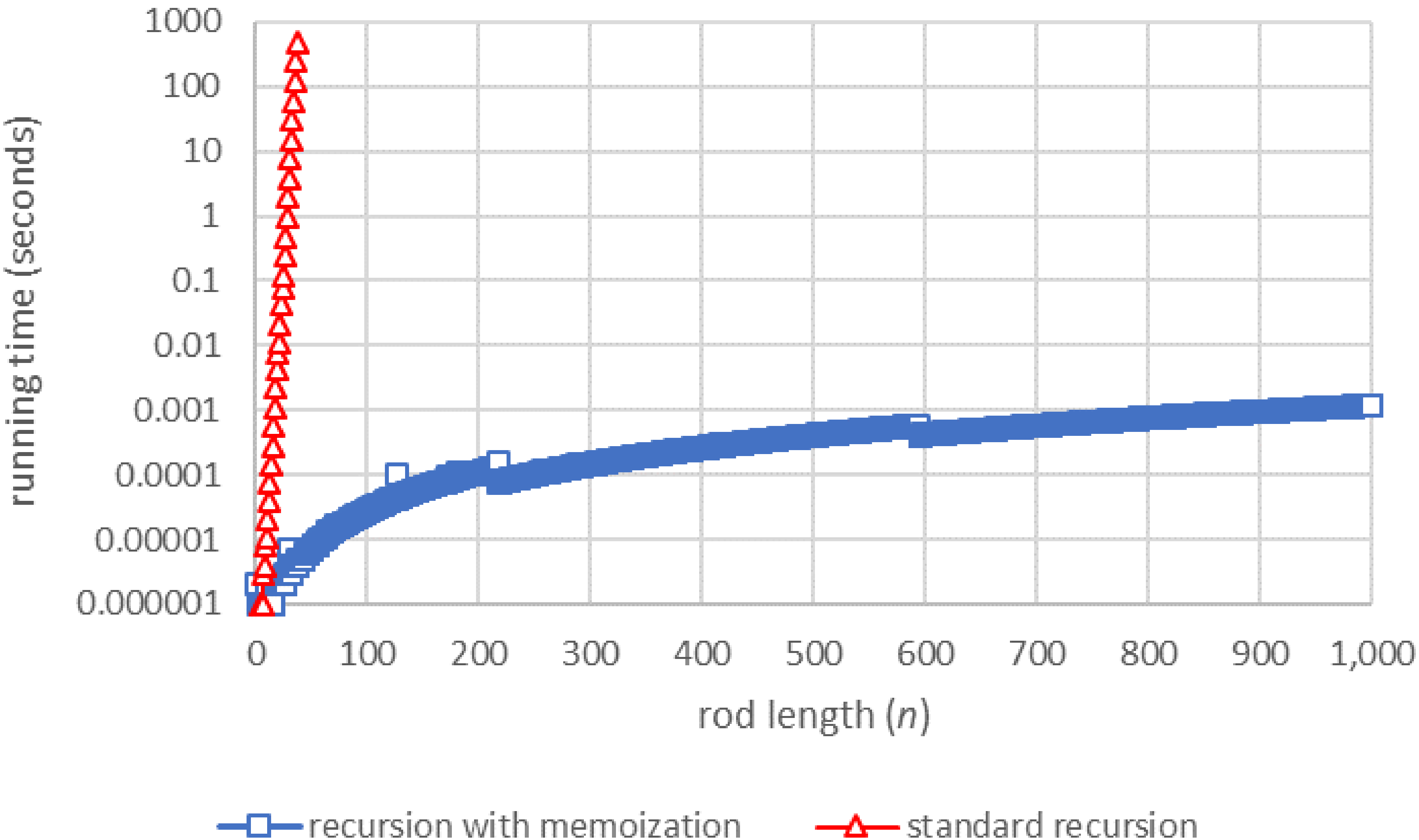
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



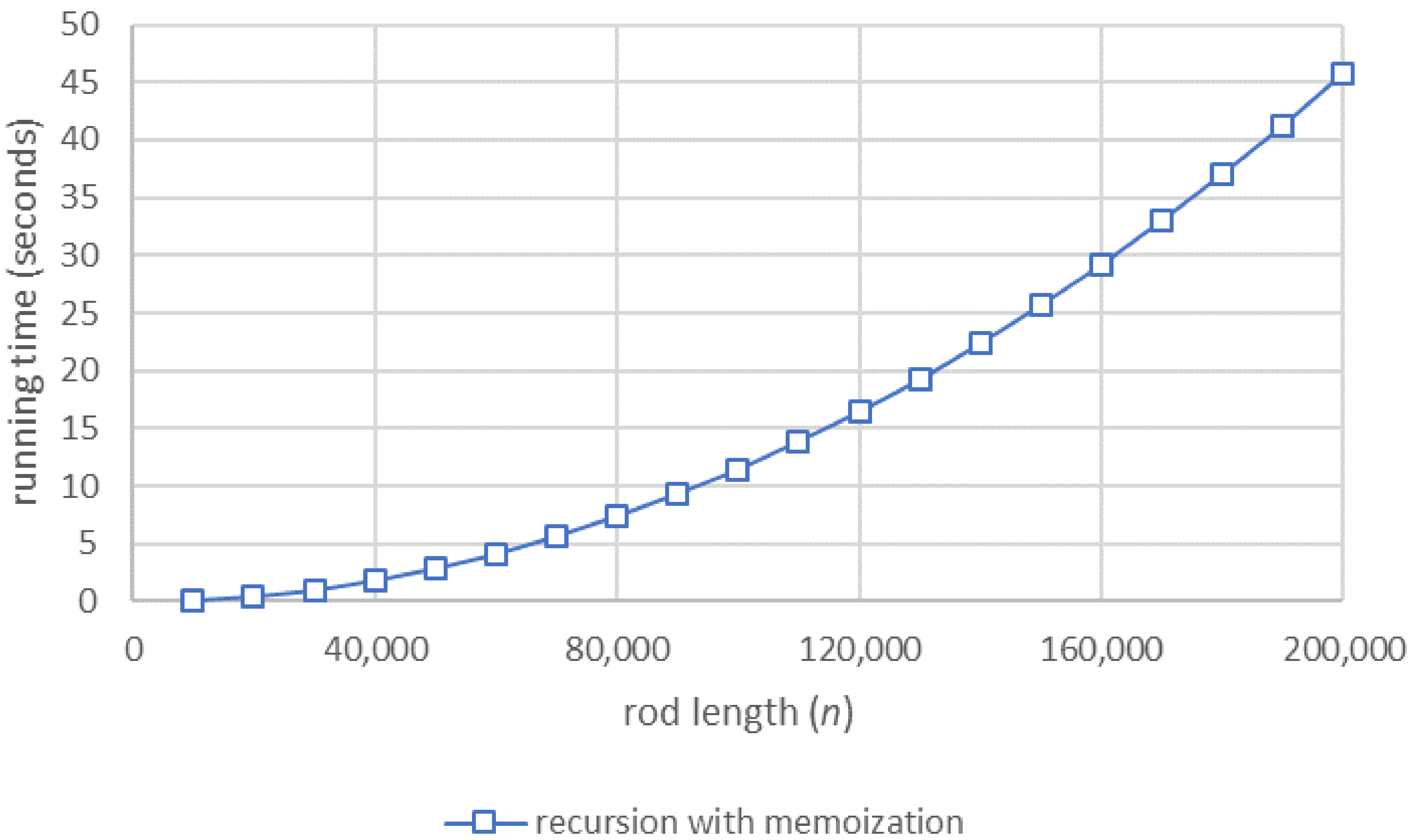
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

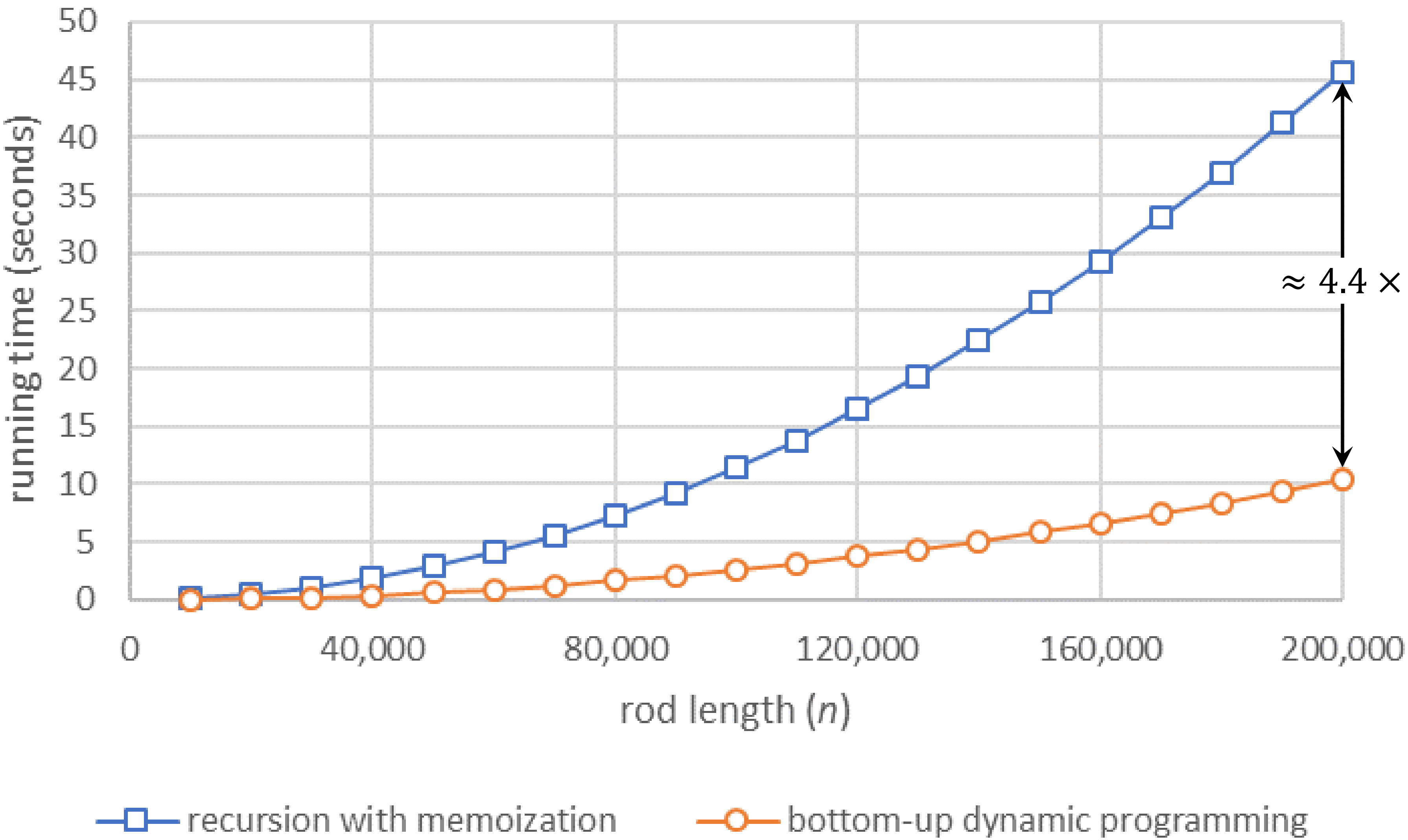
Rod Cutting: Bottom-up Dynamic Programming

```
BOTTOM-UP-CUT-ROD ( p, n )

r[0..n] ← new array
r[0] ← 0
for j ← 1 to n do
    q ← −∞
    for i ← 1 to j do
        q ← max{ q, p[ i ] + r[j − i] }
    r[j] ← q
return r[n]
```

Rod Cutting: Bottom-up Dynamic Programming

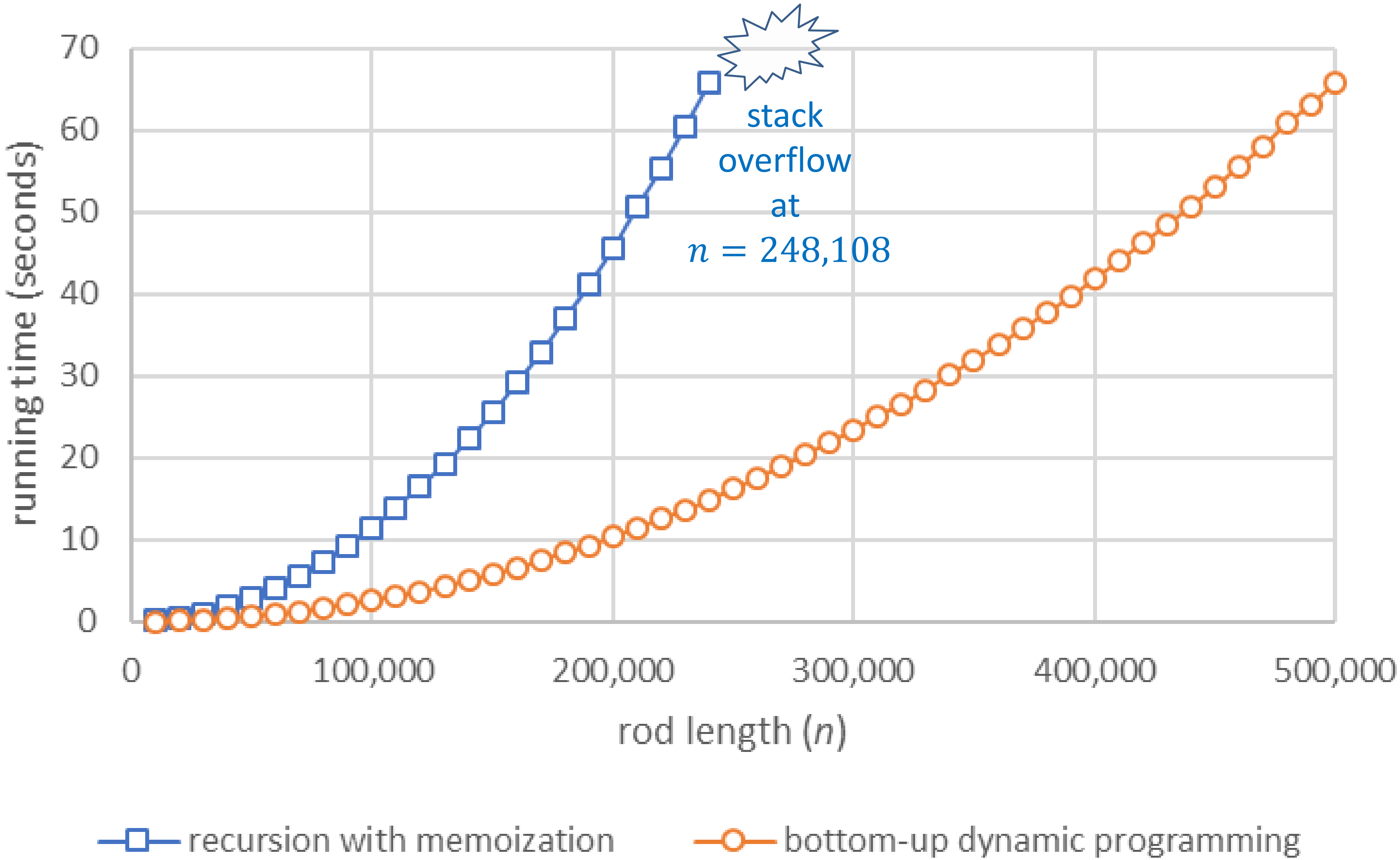
```
BOTTOM-UP-CUT-ROD ( p, n )

r[0..n] ← new array
r[0] ← 0
for j ← 1 to n do
    q ← − ∞
    for i ← 1 to j do
        q ← max{ q, p[ i ] + r[j − i] }
    r[j] ← q
return r[n]
```

Rod Cutting: Bottom-up Dynamic Programming



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Bottom-up Dynamic Programming



*Run on a dual-socket (2×8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursive Divide-&-Conquer

Divide-and-Conquer-Cut-Rod (p, n)

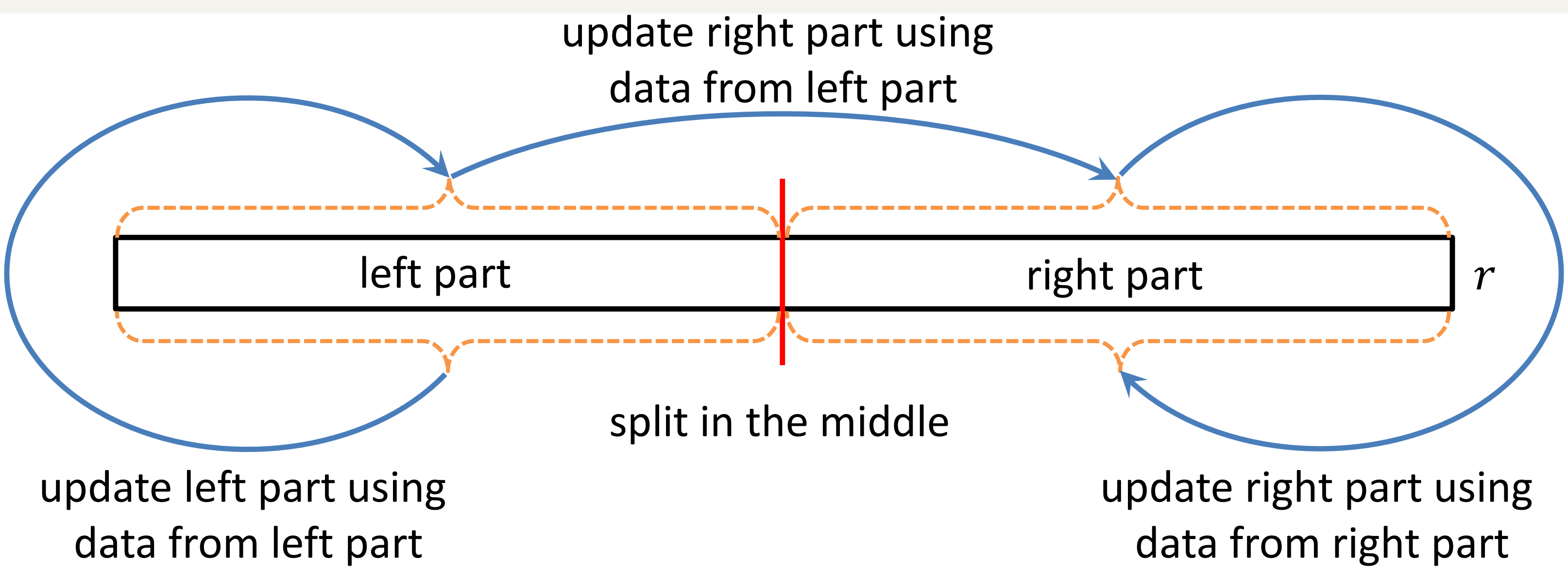
1. $r[0..n] \leftarrow$ new array
2. $r[0] \leftarrow 0$
3. *for* $i \leftarrow 1$ *to* n *do*
4. $\quad r[i] \leftarrow -\infty$
5. *DC-Cut-Rod-A* ($p, r, 1, n$)
6. *return* $r[n]$

DC-Cut-Rod-Solve-Base (p, r, k_1, n_1, k_2, n_2)

1. *for* $j \leftarrow k_2$ *to* $k_2 + n_2 - 1$ *do*
2. $\quad q \leftarrow r[j]$
3. $\quad$ *for* $i \leftarrow k_1$ *to* $\min\{\, j,\ k_1 + n_1 - 1 \,\}$ *do*
4. $\quad\quad q \leftarrow \max\{\, q, p[\, i \,] + r[j - i] \,\}$
5. $\quad r[j] \leftarrow q$

Rod Cutting: Recursive Divide-&-Conquer

```
DC-CUT-ROD-A ( p, r, k, n )
    if n ≤ BASE_SIZE then
        DC-CUT-ROD-SOLVE-BASE ( p, r, k, n, k, n )
    else
        m ← ⌊n/2⌋
        DC-CUT-ROD-A ( p, r, k, m )                      // update left part using left part
        DC-CUT-ROD-B ( p, r, k, m, k + m, n − m )        // update right part using left part
        DC-CUT-ROD-A ( p, r, k + m, n − m )              // update right part using right part
```



Rod Cutting: Recursive Divide-&-Conquer

DC-Cut-Rod-B (p, r, k_1, n_1, k_2, n_2)

if $n \leq BASE\_SIZE$ *then*

  DC-Cut-Rod-Solve-Base (p, r, k_1, n_1, k_2, n_2)

else

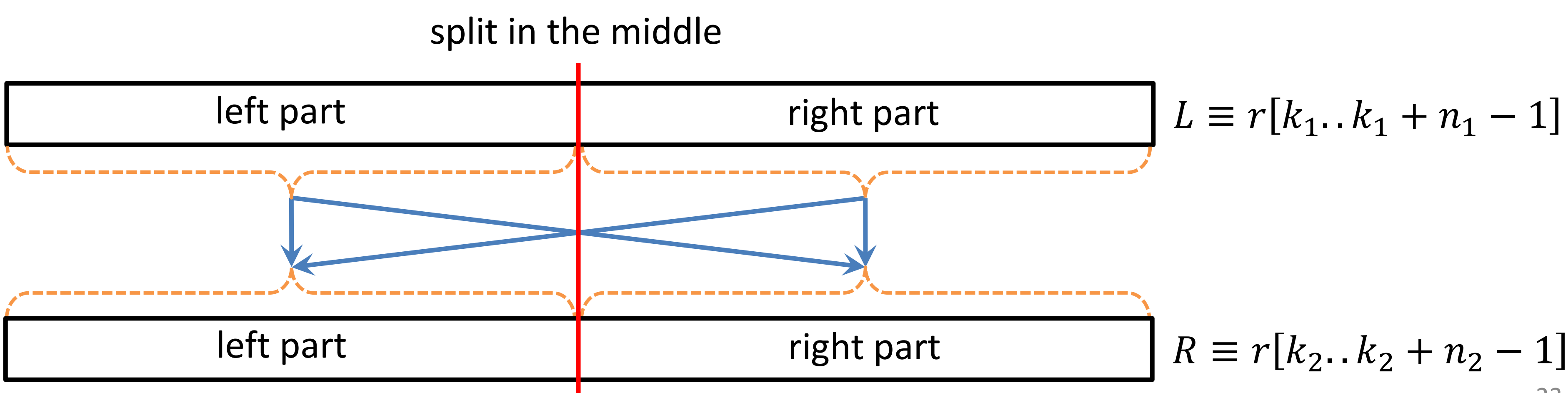
  $m_1 \leftarrow \lfloor n_1/2 \rfloor,\ m_2 \leftarrow \lfloor n_2/2 \rfloor$ // let $L \equiv [k_1..k_1 + n_1 - 1]$ and $R \equiv [k_2..k_2 + n_2 - 1]$

  DC-Cut-Rod-B (p, r, k_1, m_1, k_2, m_2) // left of L updates left of R

  DC-Cut-Rod-B ($p, r, k_1 + m_1, n_1 - m, k_2, m_2$) // right of L updates left of R

  DC-Cut-Rod-B ($p, r, k_1, m_1, k_2 + m_2, n_2 - m_2$) // left of L updates right of R

  DC-Cut-Rod-B ($p, r, k_1 + m_1, n_1 - m_1, k_2 + m_2, n_2 - m_2$) // right of L updates right of R



Rod Cutting: Recursive Divide-&-Conquer

Let $T(n)$, $T_A(n)$ and $T_B(n)$ be the running times of *DIVIDE-AND-CONQUER-CUT-ROD*, *DC-CUT-ROD-A* and *DC-CUT-ROD-B*, respectively, on an input of size n. Then

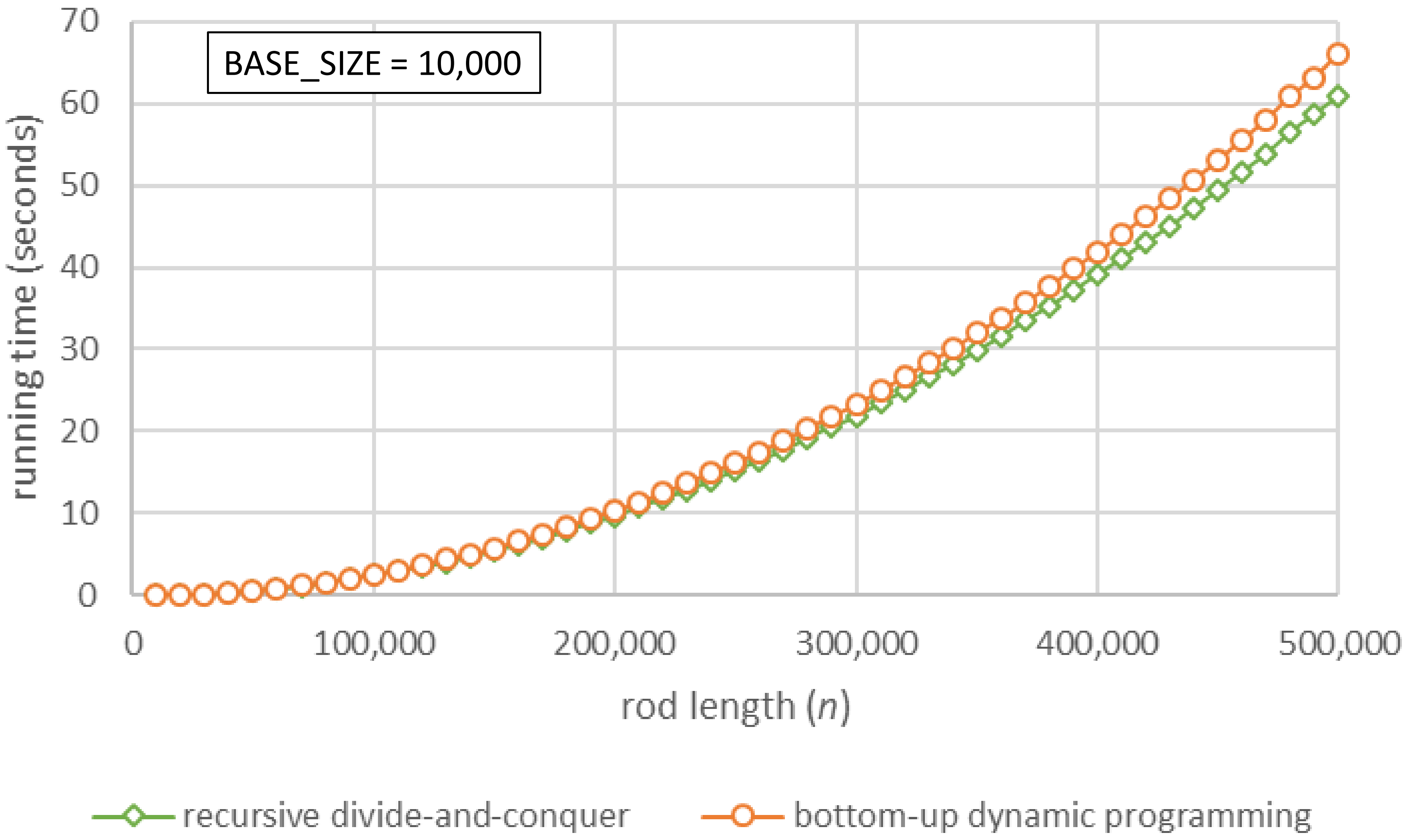
$$T(n) = T_A(n) + \Theta(n).$$

$$T_A(n) = \begin{cases} \Theta(1), & if\ n \leq BASE\_SIZE, \\ 2T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + \Theta(1), & otherwise. \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1), & if\ n \leq BASE\_SIZE, \\ 4T_B\left(\frac{n}{2}\right) + \Theta(1), & otherwise. \end{cases}$$

Solving: $T(n) = \Theta\left(n^2\right)$.

Rod Cutting: Recursive Divide-&-Conquer



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursive Divide-&-Conquer

*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Extracting the Solution

```
EXTENDED-BOTTOM-UP-CUT-ROD ( p, n )

r[0..n] ← new array, s[0..n] ← new array
r[0] ← 0
for j ← 1 to n do
    q ← − ∞
    for i ← 1 to j do
        if q < p[i] + r[j − i] then
            q ← p[ i ] + r[j − i]
            s[j] ← i
    r[j] ← q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION ( p, n )

(r, s) ← EXTENDED-BOTTOM-UP-CUT-ROD ( p, n )
while n > 0 do
    print s[n]
    n ← n − s[n]
```

Rod Cutting: Extracting the Solution

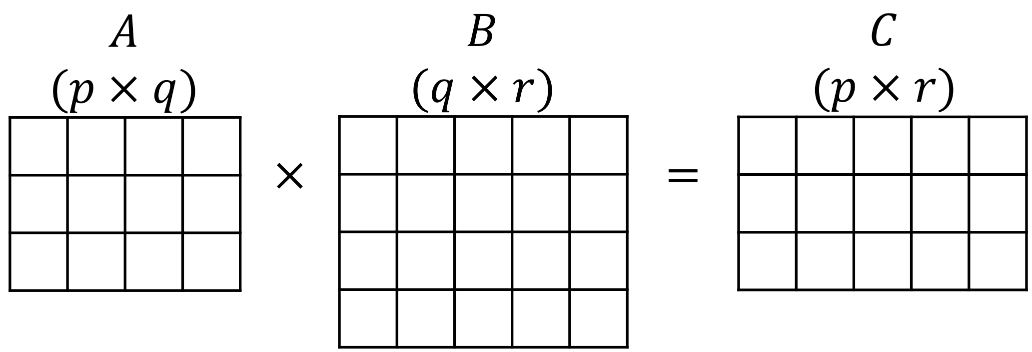
A sample price table for rods

length i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
price p_i	1	5	8	9	10	17	17	20	24	30

EXTENDED-BOTTOM-UP-CUT-ROD(p, n) returns the following arrays:

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

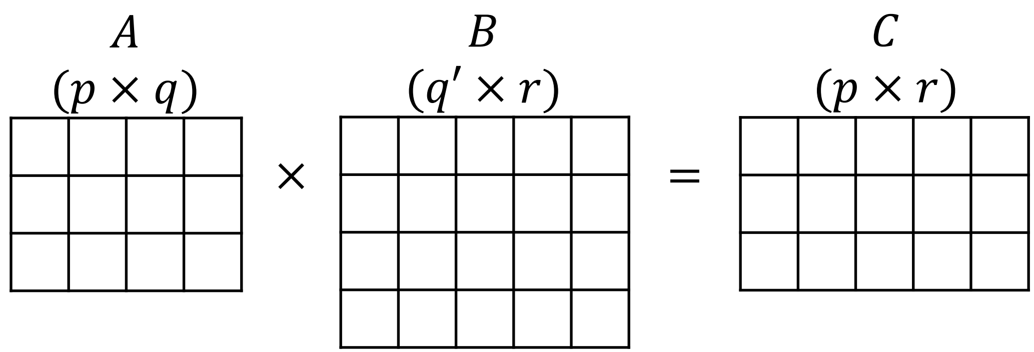
Matrix-Chain Multiplication



A $p \times q$ matrix A and a $q' \times r$ matrix B can be multiplied provided $q = q'$.

The result will be a $p \times r$ matrix C.

Matrix-Chain Multiplication



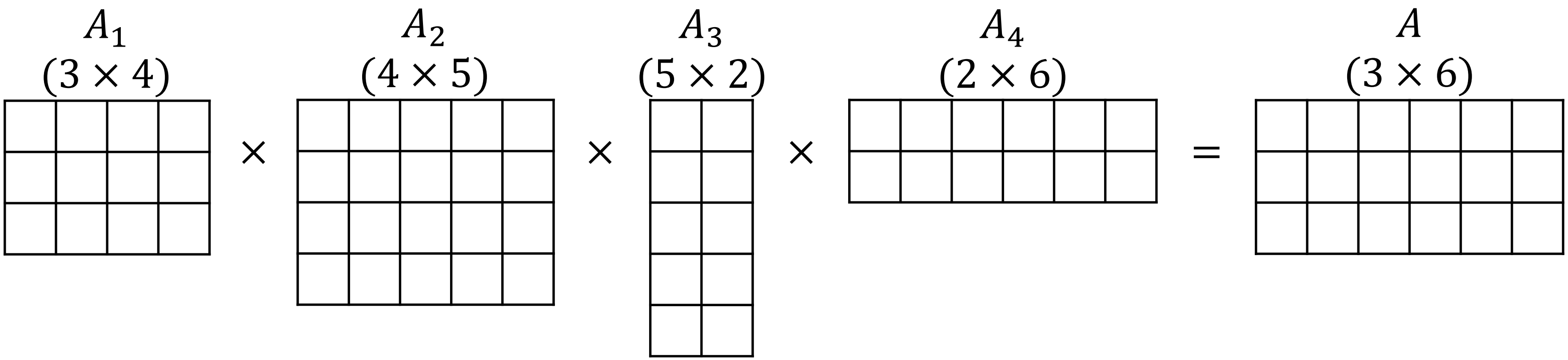
provided $q = q'$

```
MATRIX-MULTIPLY ( p, q, A, q', r, B )
1.  if q ≠ q' then
2.     error "incompatible dimensions"
3.  else
4.     C ← new p × r matrix
5.     for i ← 1 to p do
6.        for j ← 1 to r do
7.           C[i, j] ← 0
8.           for k ← 1 to q do
9.              C[i, j] ← C[i, j] + A[i, k] × B[k, j]
10.    return C
```

Time needed to multiply the $p \times q$ matrix A and the $q \times r$ matrix B is dominated by the total number pqr of scalar multiplications performed in line 7.

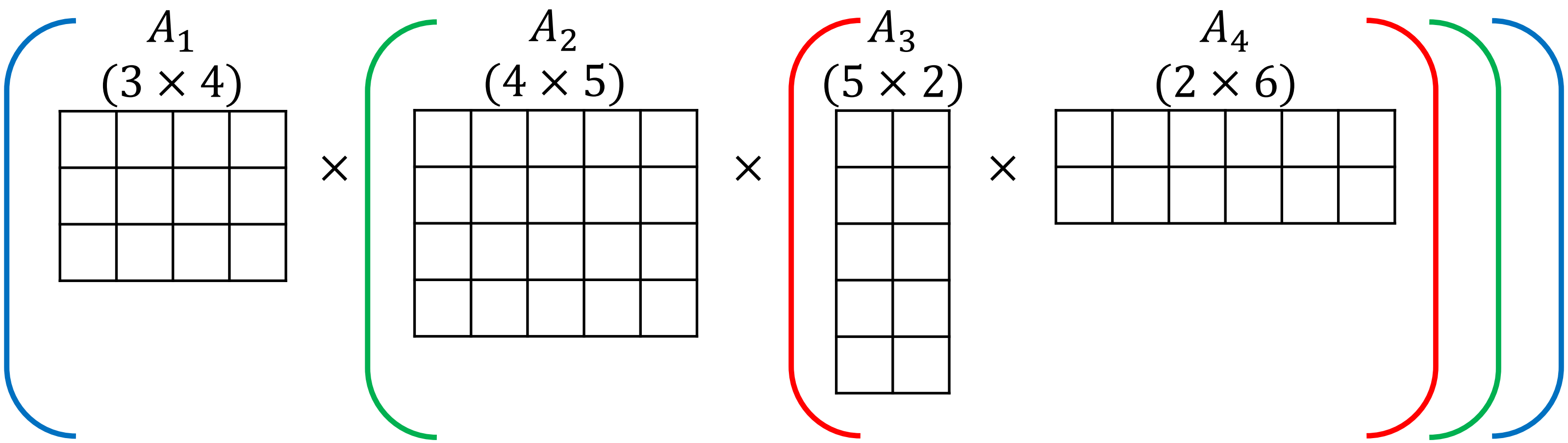
Hence, running time of the algorithm is $\Theta(pqr)$.

Matrix-Chain Multiplication



We can multiply the four matrices on the left hand side in five distinct orders.

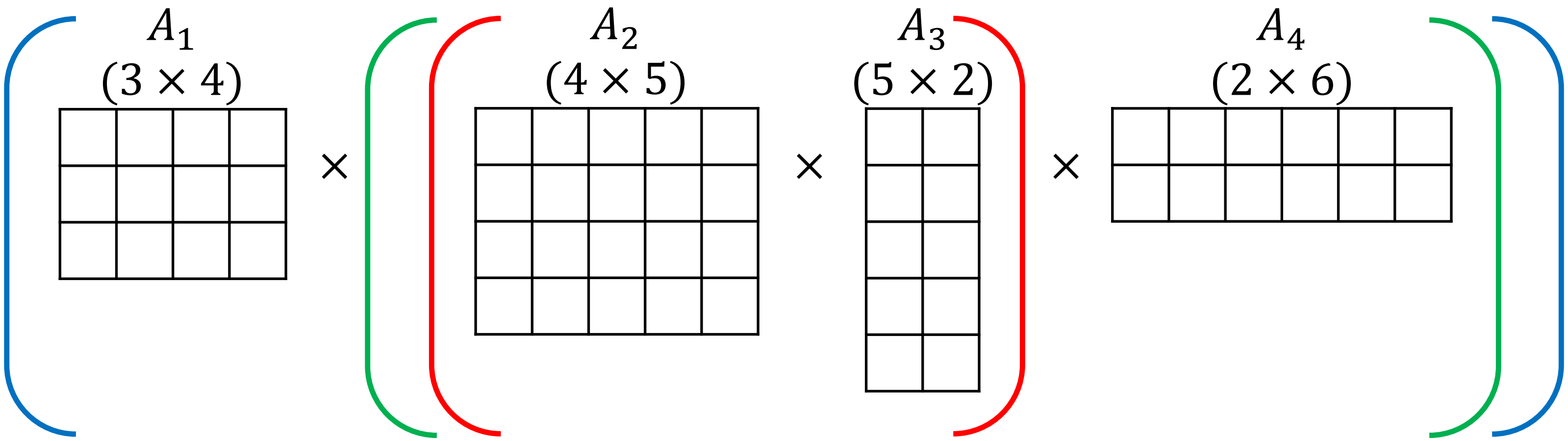
Matrix-Chain Multiplication



number of scalar multiplications

$$= 5 \times 2 \times 6 + 4 \times 5 \times 6 + 3 \times 4 \times 6$$
$$= 252$$

Matrix-Chain Multiplication

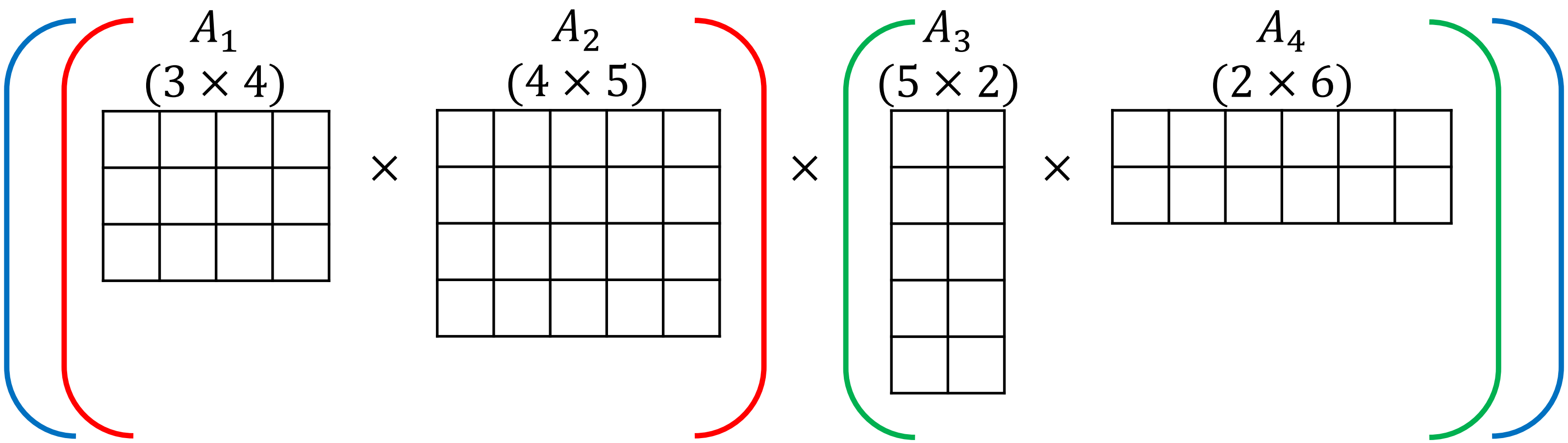


number of scalar multiplications

$$= 4 \times 5 \times 2 + 4 \times 2 \times 6 + 3 \times 4 \times 6$$

$$= 160$$

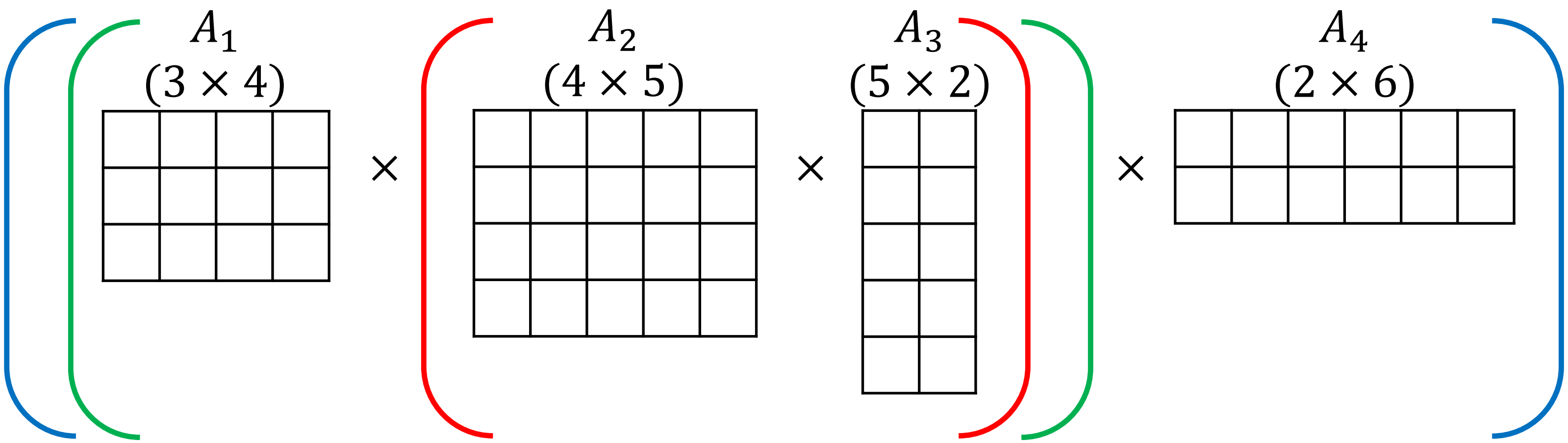
Matrix-Chain Multiplication



number of scalar multiplications

$$= 3 \times 4 \times 5 + 5 \times 2 \times 6 + 3 \times 5 \times 6$$
$$= 210$$

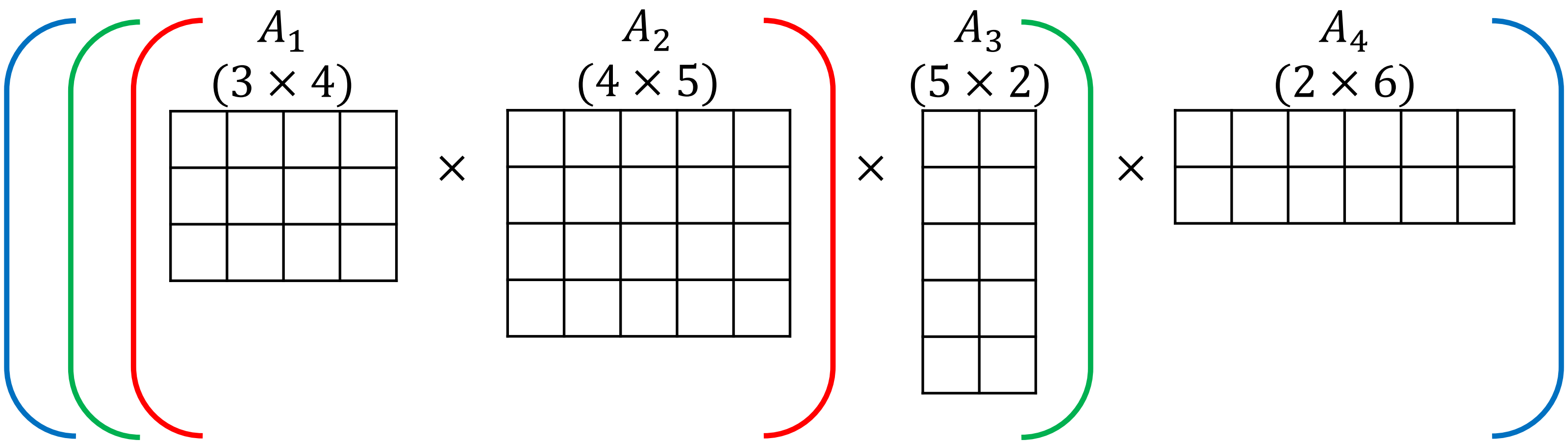
Matrix-Chain Multiplication



number of scalar multiplications

$$= 4 \times 5 \times 2 + 3 \times 4 \times 2 + 3 \times 2 \times 6$$
$$= 100$$

Matrix-Chain Multiplication



number of scalar multiplications

$$= 3 \times 4 \times 5 + 3 \times 5 \times 2 + 3 \times 2 \times 6$$
$$= 126$$

Matrix-Chain Multiplication

The matrix-chain multiplication problem:

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices,
where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$,
fully parenthesize the product $A_1 A_2 \ldots A_n$
in a way that minimizes the number of scalar multiplications.

Matrix-Chain Multiplication

Let $P(n)$ = number of parenthesizations of a sequence of n matrices.

Then

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

Very easy to show that $P(n) = \Omega(2^n)$.

Hence, exhaustively checking all possible parenthesizations of the given chain of matrices does not give an efficient algorithm.

Matrix-Chain Mult: Standard Recursive Algorithm

Let $A_{i\dots j} = A_i A_{i+1} \dots A_{j-1} A_j$ for $1 \le i \le j \le n$.

Let $m(i,j) =$ the minimum number of scalar multiplications needed to compute the matrix $A_{i\dots j}$.

Then $m(1,n) =$ the minimum number of scalar multiplications needed to compute $A_{1\dots n}$ (i.e., solve the entire problem).

$$m(i,j) = \begin{cases} 0, & if\ i = j, \\ \min\limits_{i \le k < j} \left\{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \right\}, & if\ i < j. \end{cases}$$

Matrix-Chain Mult: Standard Recursive Algorithm

Recursive-Matrix-Chain (p, i, j)

if $i = j$ *then*

  return 0

$q \leftarrow \infty$

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

  $q \leftarrow \min \left(\begin{array}{c} q, \\ \text{Recursive-Matrix-Chain}(p, i, k) \\ + \text{Recursive-Matrix-Chain}(p, k+1, j) \\ + p_{i-1} p_k p_j \end{array} \right)$

return q

Matrix-Chain Mult: Standard Recursive Algorithm

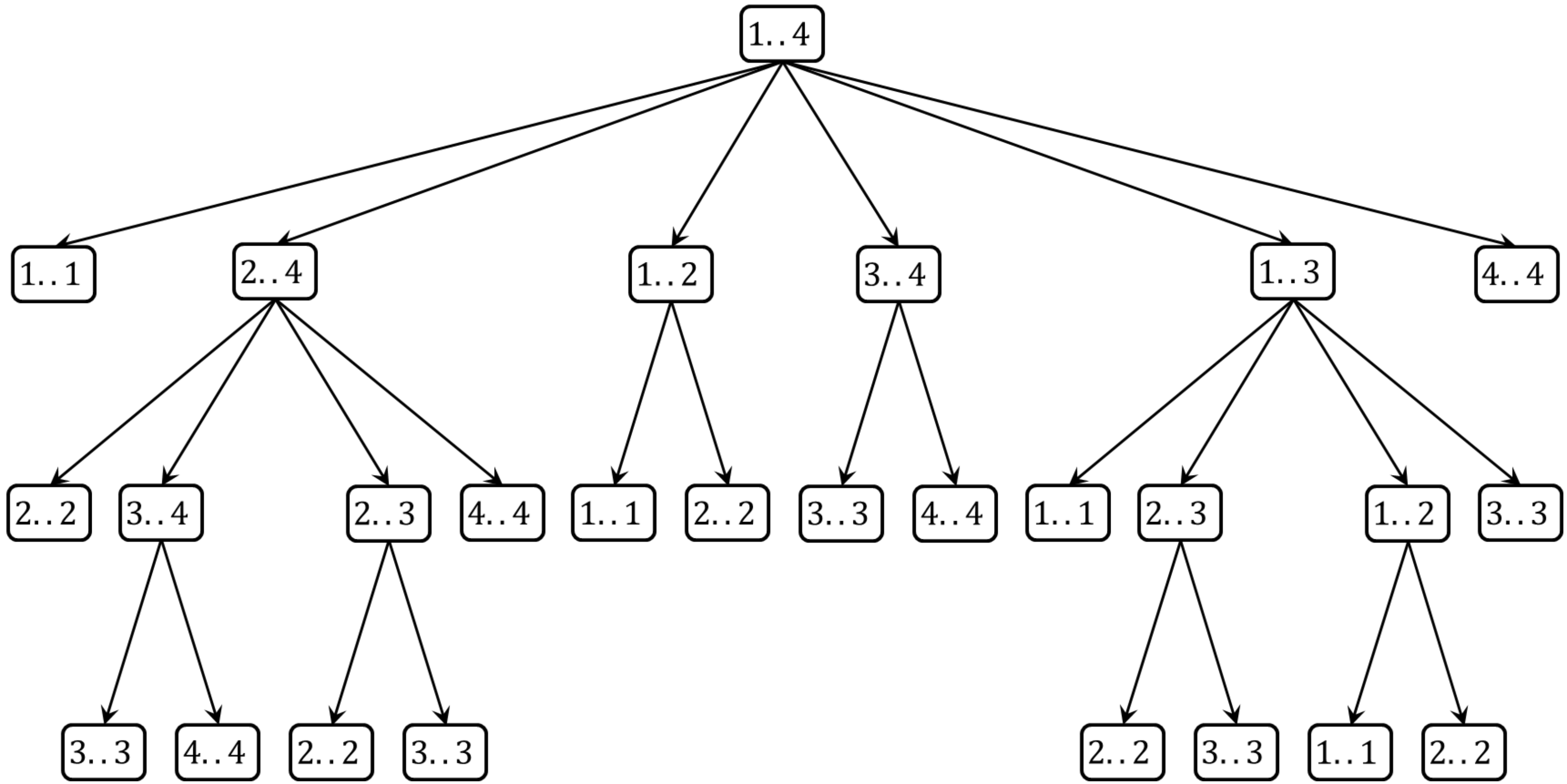
Let $T(n)$ be the running time of the algorithm on an input of size n.

Then

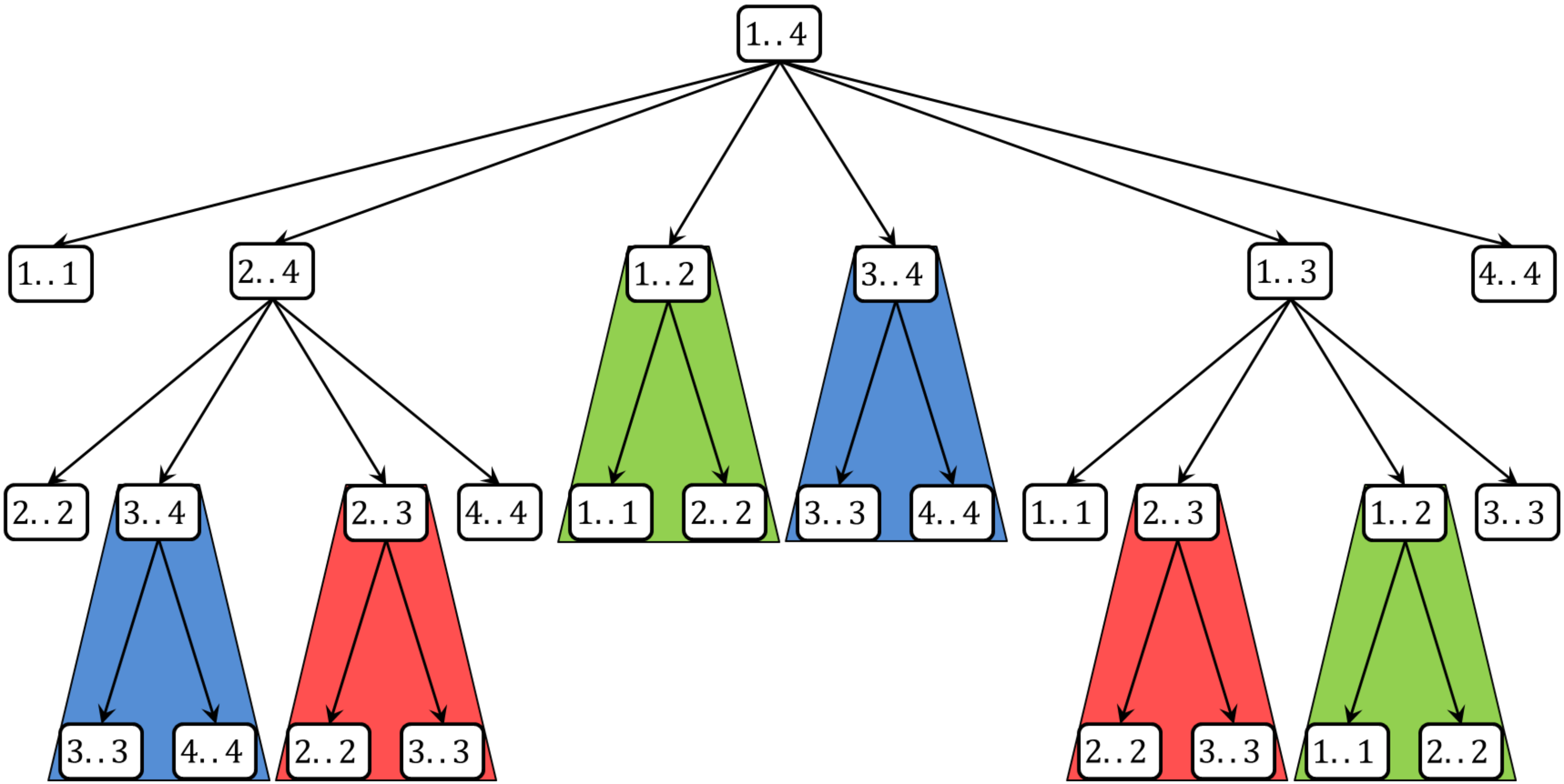
$$T(n) \geq \begin{cases} 1, & if\ n = 1, \\ 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1\right), & if\ n > 1. \end{cases}$$

Solving: $T(n) \geq 2^{n-1} \Rightarrow T(n) = \Omega(2^n)$.

Matrix-Chain Mult: Standard Recursive Algorithm



Matrix-Chain Mult: Standard Recursive Algorithm



Matrix-Chain Mult: Recursion with Memoization

MEMOIZED-MATRIX-CHAIN (p)

$n \leftarrow p.length - 1$
$m[1..n, 1..n] \leftarrow$ new table
for $i \leftarrow 1$ *to* n *do*
    for $j \leftarrow i$ *to* n *do*
        $m[i, j] \leftarrow \infty$
return LOOKUP-CHAIN ($m, p, 1, n$)

LOOKUP-CHAIN (m, p, i, j)

if $m[i, j] < \infty$ *then*
    return $m[i, j]$
if $i = j$ *then*
    $m[i, j] \leftarrow 0$
for $k \leftarrow i$ *to* $j - 1$ *do*
    $q \leftarrow$ LOOKUP-CHAIN (m, p, i, k)
        $+$ LOOKUP-CHAIN (m p, $k + 1$, j)
        $+ p_{i-1} p_k p_j$
    if $q < m[i, j]$ *then*
        $m[i, j] \leftarrow q$
return $m[i, j]$

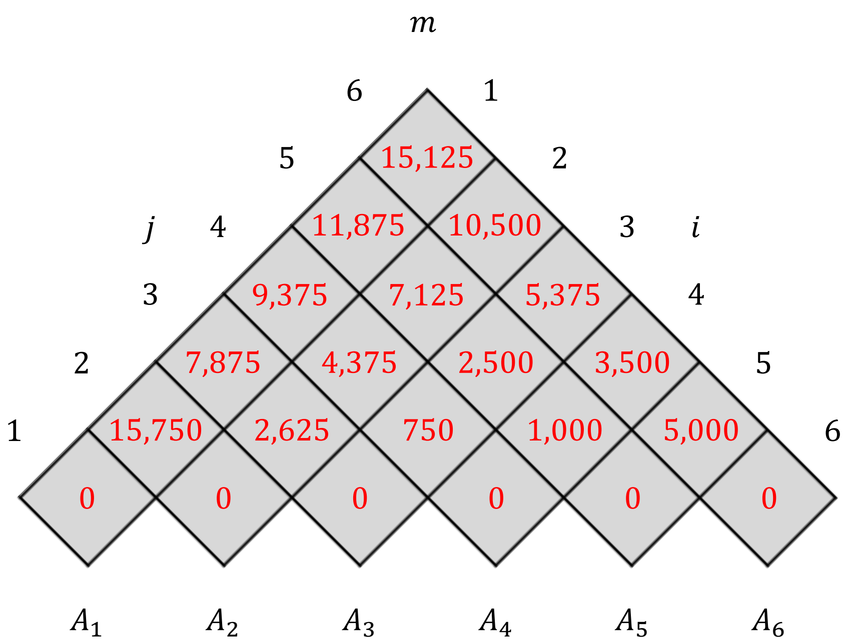
Matrix-Chain Mult: Bottom-up DP

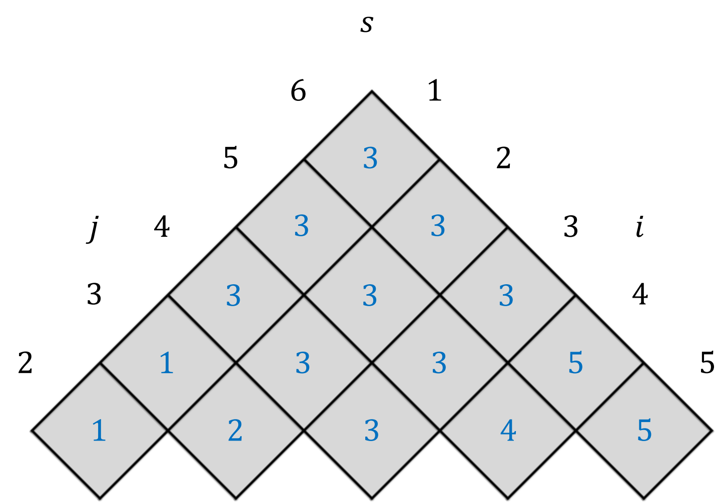
MATRIX-CHAIN-ORDER (p)

$n \leftarrow p.length - 1$
$m[1..n, 1..n] \leftarrow$ new table, $s[1..n-1, 2..n] \leftarrow$ new table
for $i \leftarrow 1$ *to* n *do*
    $m[i,i] \leftarrow 0$
for $l \leftarrow 2$ *to* n *do*   *// l is the chain length*
    for $i \leftarrow 1$ *to* $n - l + 1$ *do*
        $j \leftarrow i + l - 1$
        $m[i,j] \leftarrow \infty$
        for $k \leftarrow i$ *to* $j - 1$ *do*
            $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
            if $q < m[i,j]$ *then*
                $m[i,j] \leftarrow q$
                $s[i,j] \leftarrow k$
return m *and* s

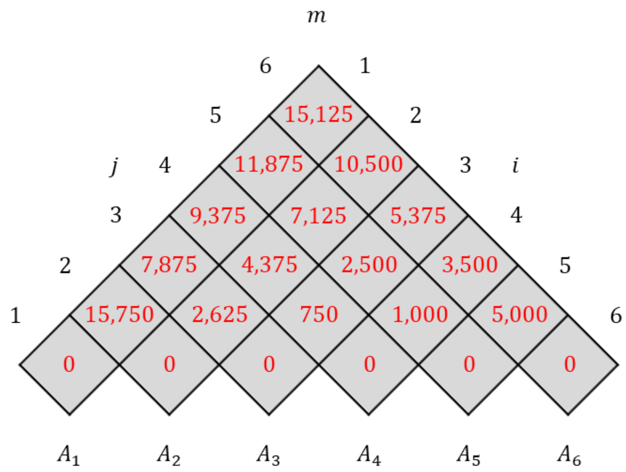
Matrix-Chain Mult: Bottom-up DP

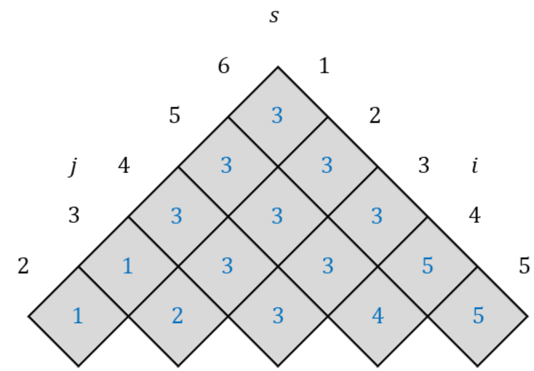
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





Matrix-Chain Mult: Extracting the Solution

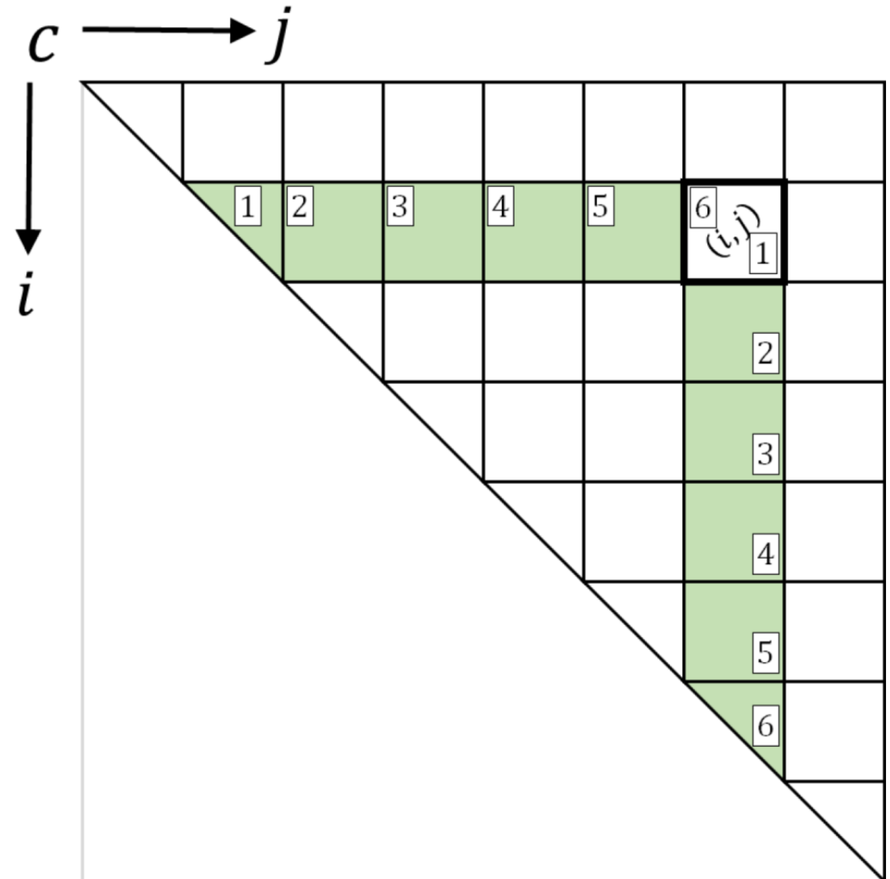




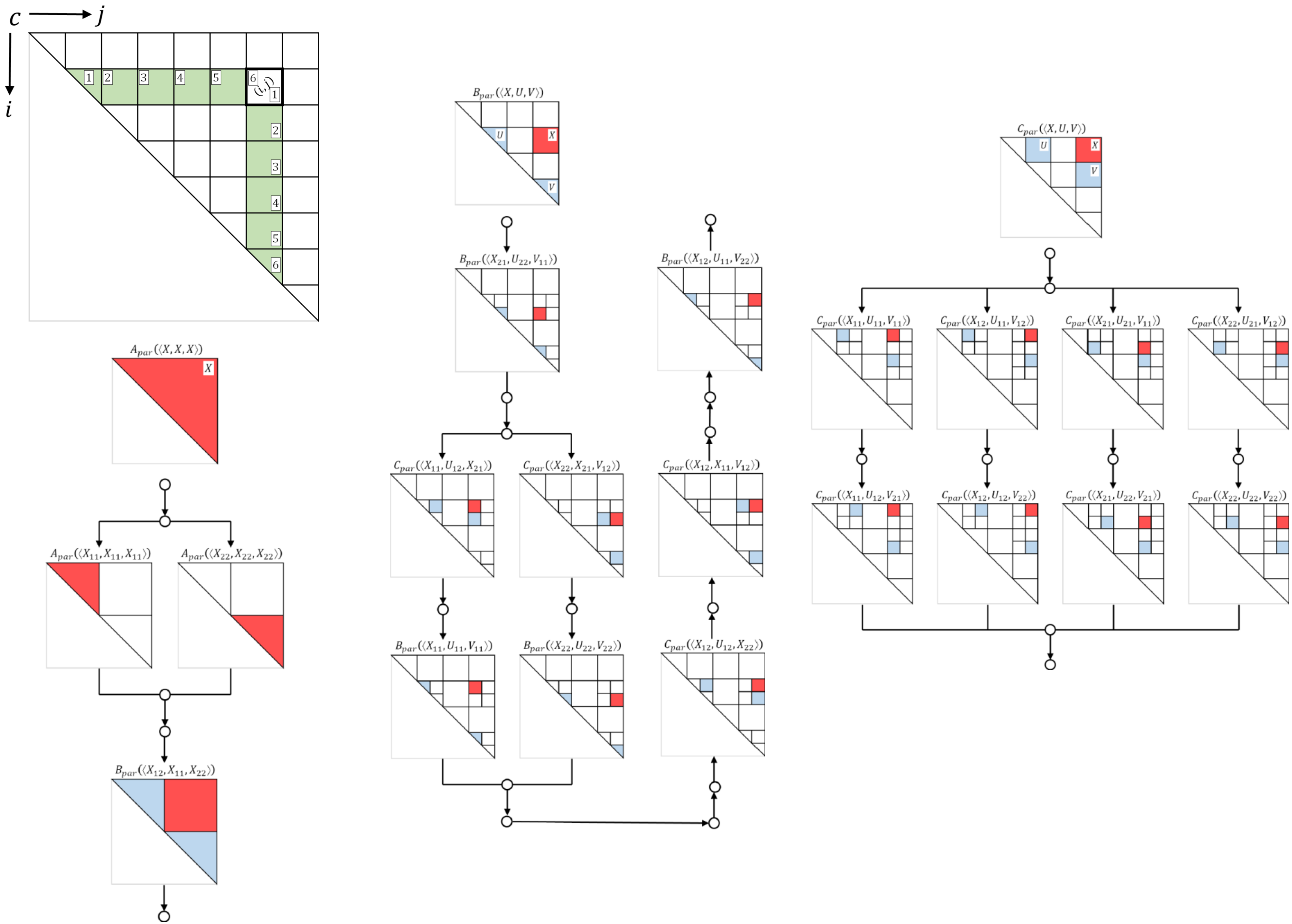
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

```
PRINT-OPTIMAL-PARENS ( s, i, j )

if i = j then
    print "A_i"
else print "("
        PRINT-OPTIMAL-PARENS ( s, i, s[i,j] )
        PRINT-OPTIMAL-PARENS ( s, s[i,j] + 1, j )
        print ")"
```

Matrix-Chain Mult: Recursive Divide-&-Conquer



Matrix-Chain Mult: Recursive Divide-&-Conquer



Matrix-Chain Mult: Recursive Divide-&-Conquer

$\mathcal{A}_{par}(\langle X, X, X \rangle)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop\text{-}par}(\langle X, X, X \rangle)$
2. **else**
3. **par:** $\mathcal{A}_{par}(\langle X_{11}, X_{11}, X_{11} \rangle)$, $\mathcal{A}_{par}(\langle X_{22}, X_{22}, X_{22} \rangle)$
4. $\mathcal{B}_{par}(\langle X_{12}, X_{11}, X_{22} \rangle)$

$\mathcal{B}_{par}(\langle X, U, V \rangle)$

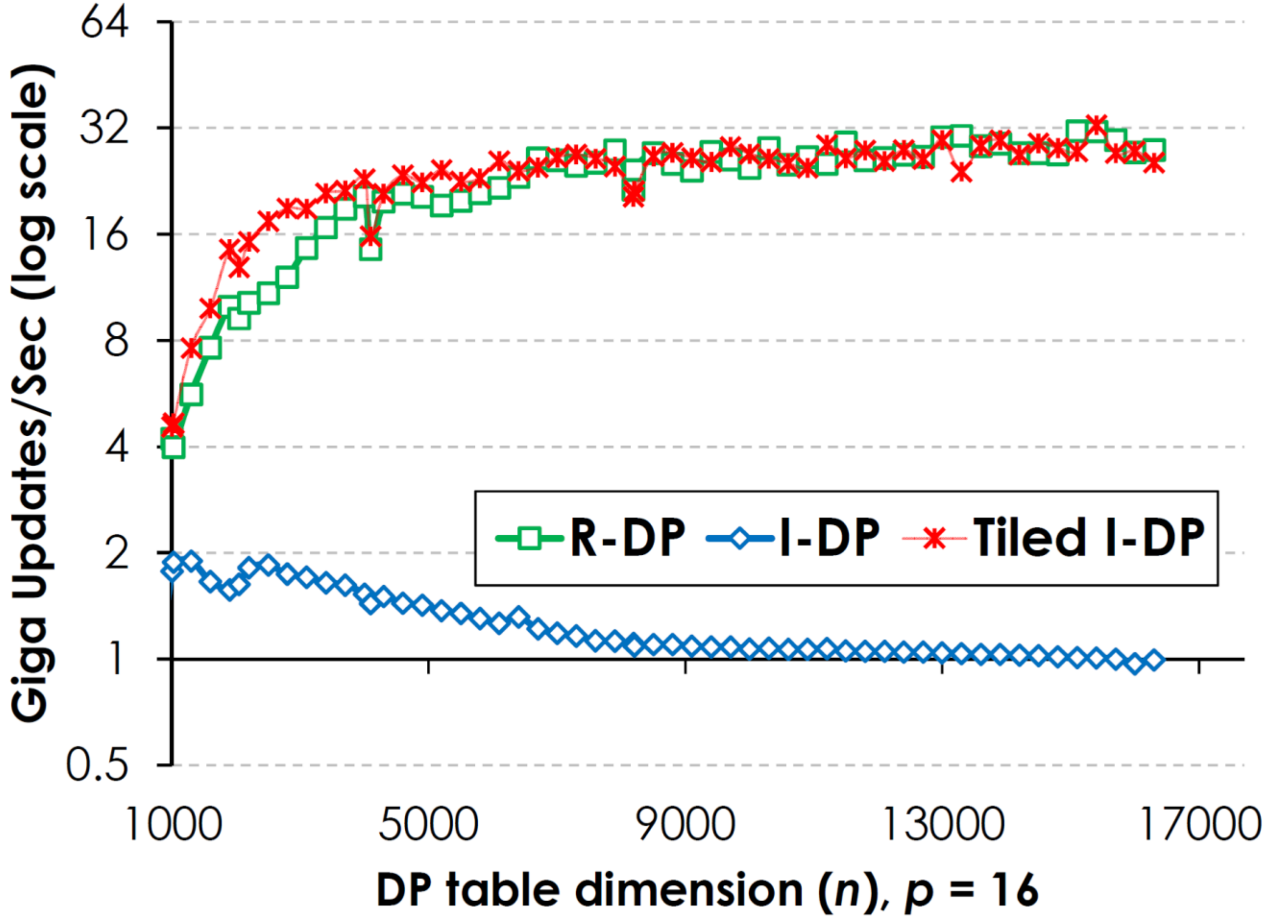
1. **if** X is a small matrix **then** $\mathcal{B}_{loop\text{-}par}(\langle X, U, V \rangle)$
2. **else**
3. $\mathcal{B}_{par}(\langle X_{21}, U_{22}, V_{11} \rangle)$
4. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, X_{21}, V_{12} \rangle)$
5. **par:** $\mathcal{B}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{B}_{par}(\langle X_{22}, X_{22}, V_{22} \rangle)$
6. $\mathcal{C}_{par}(\langle X_{12}, U_{12}, X_{22} \rangle)$
7. $\mathcal{C}_{par}(\langle X_{12}, X_{11}, V_{12} \rangle)$
8. $\mathcal{B}_{par}(\langle X_{12}, U_{11}, V_{22} \rangle)$

$\mathcal{C}_{par}(\langle X, U, V \rangle)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop\text{-}par}(\langle X, U, V \rangle)$
2. **else**
3. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{11}, V_{12} \rangle)$,
   $\mathcal{C}_{par}(\langle X_{21}, U_{21}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{21}, V_{12} \rangle)$
4. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{12}, V_{22} \rangle)$,
   $\mathcal{C}_{par}(\langle X_{21}, U_{22}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$

Matrix-Chain Mult: Empirical Performance

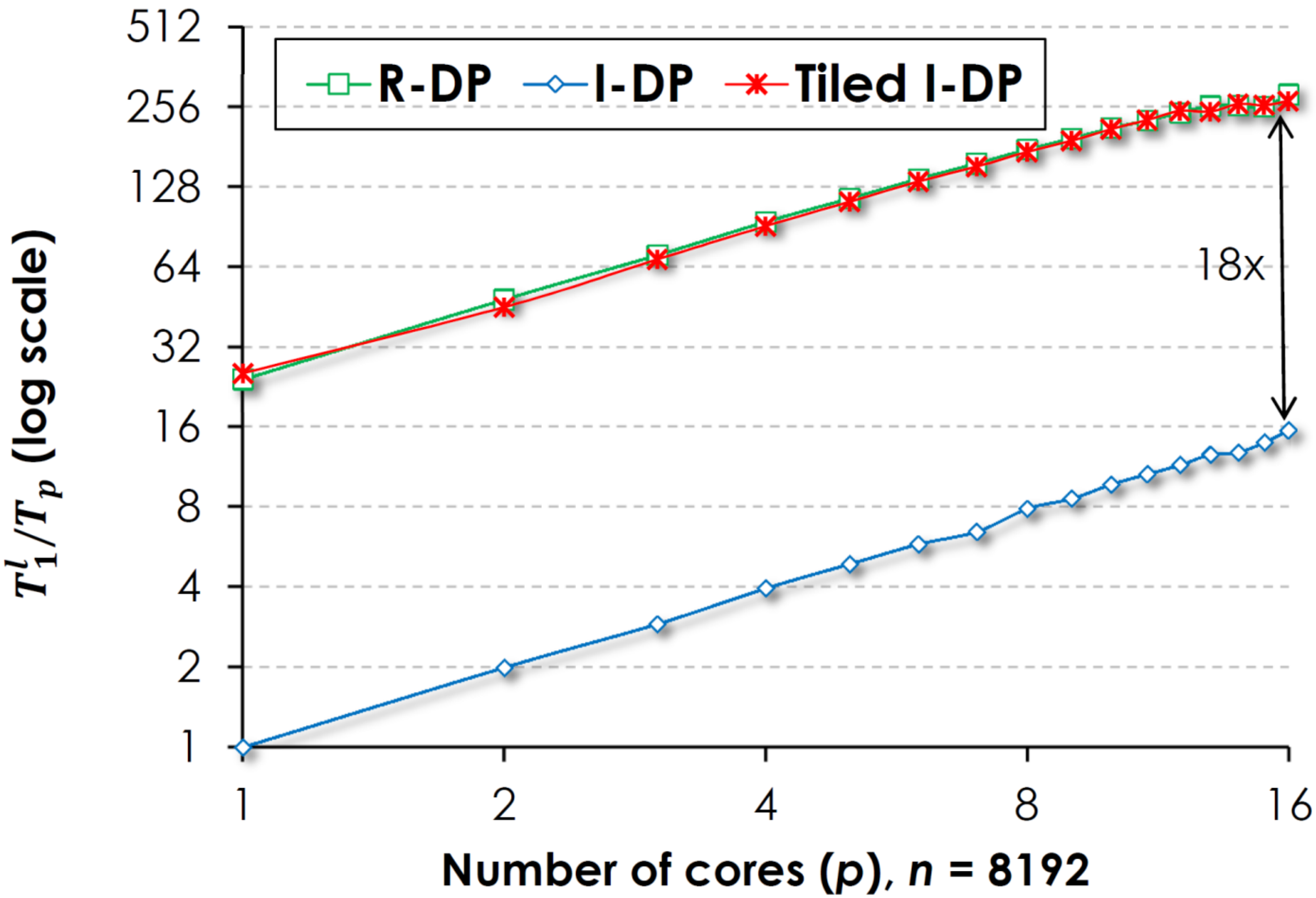
R-DP: recursive divide-&-conquer (BASE_SIZE = 64 × 64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2 × 8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

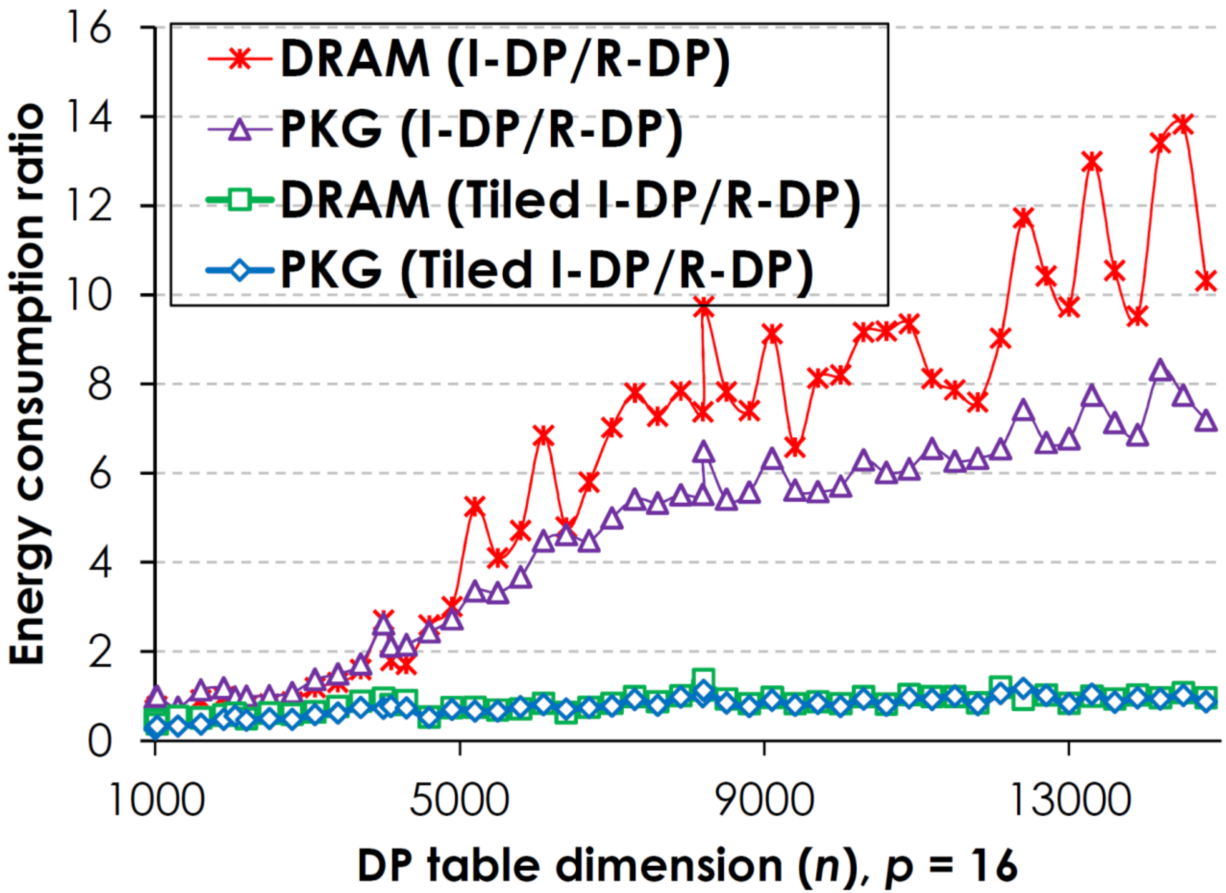
R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

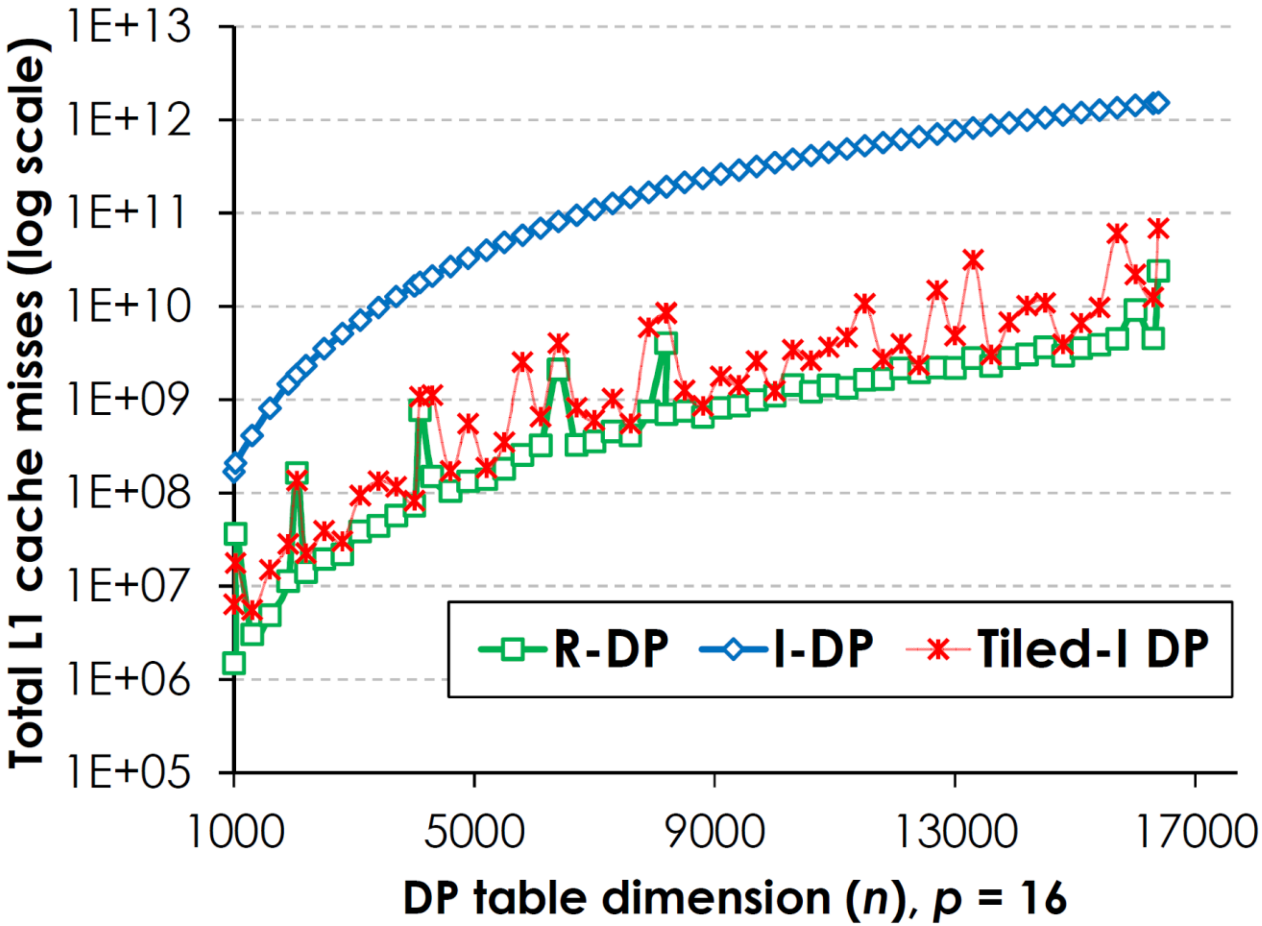
R-DP: recursive divide-&-conquer (BASE_SIZE = 64 × 64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

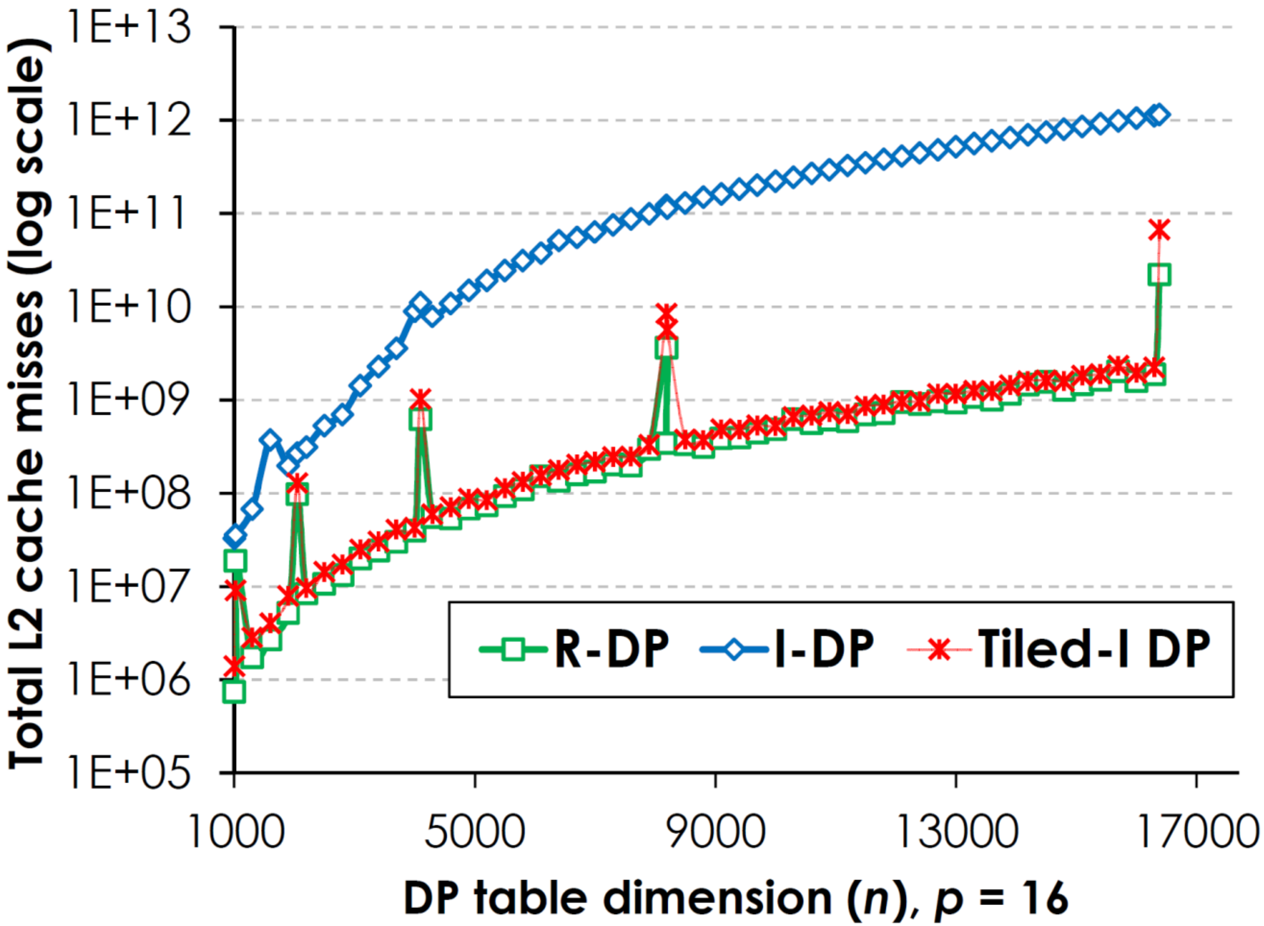
R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

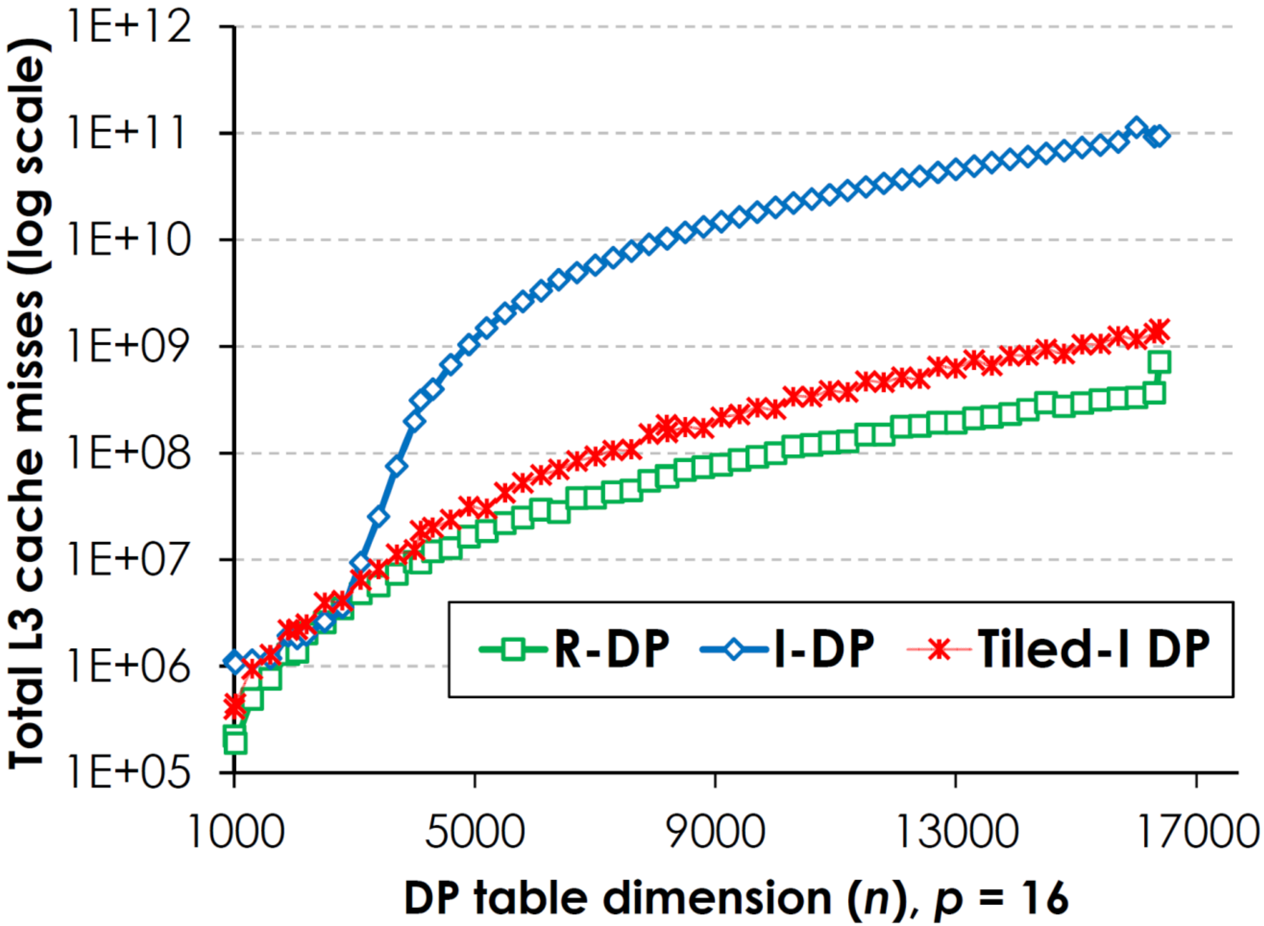
R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

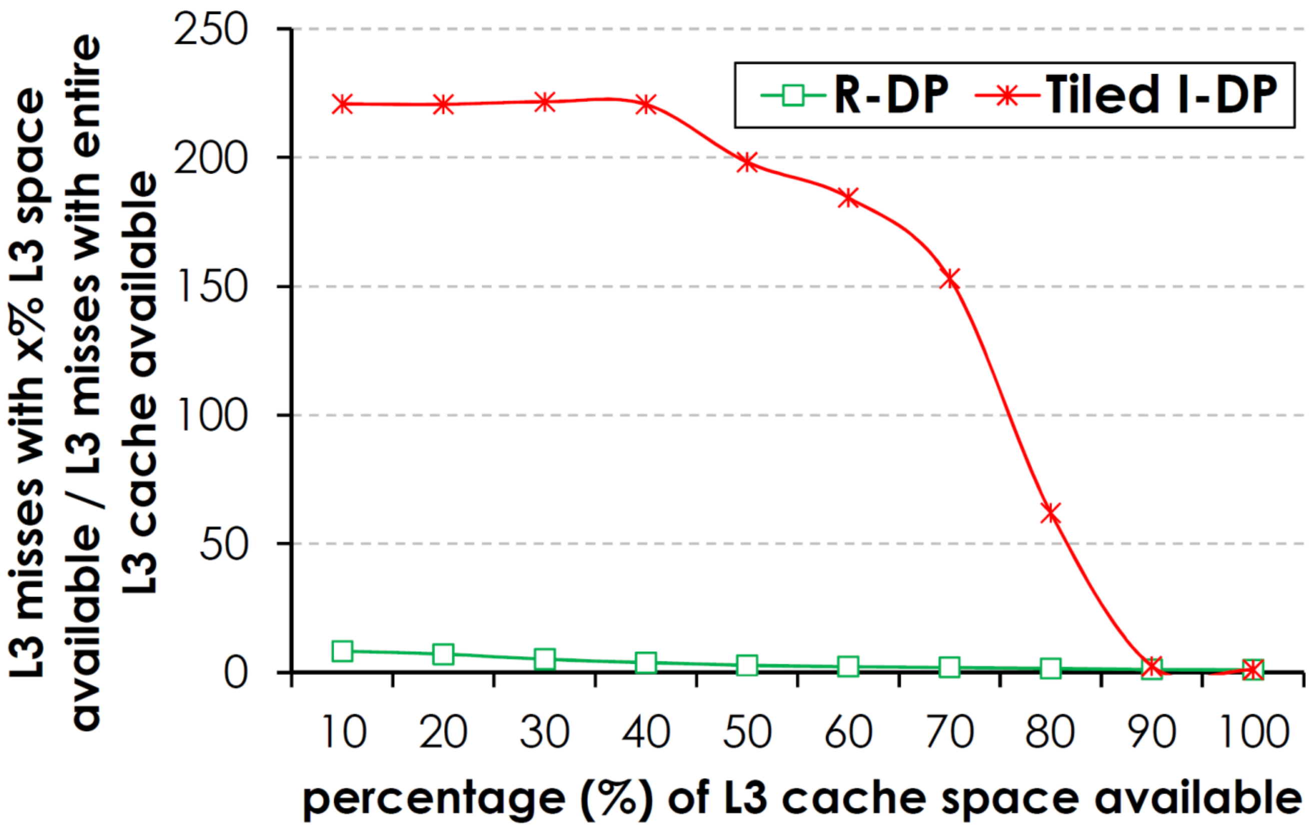
R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

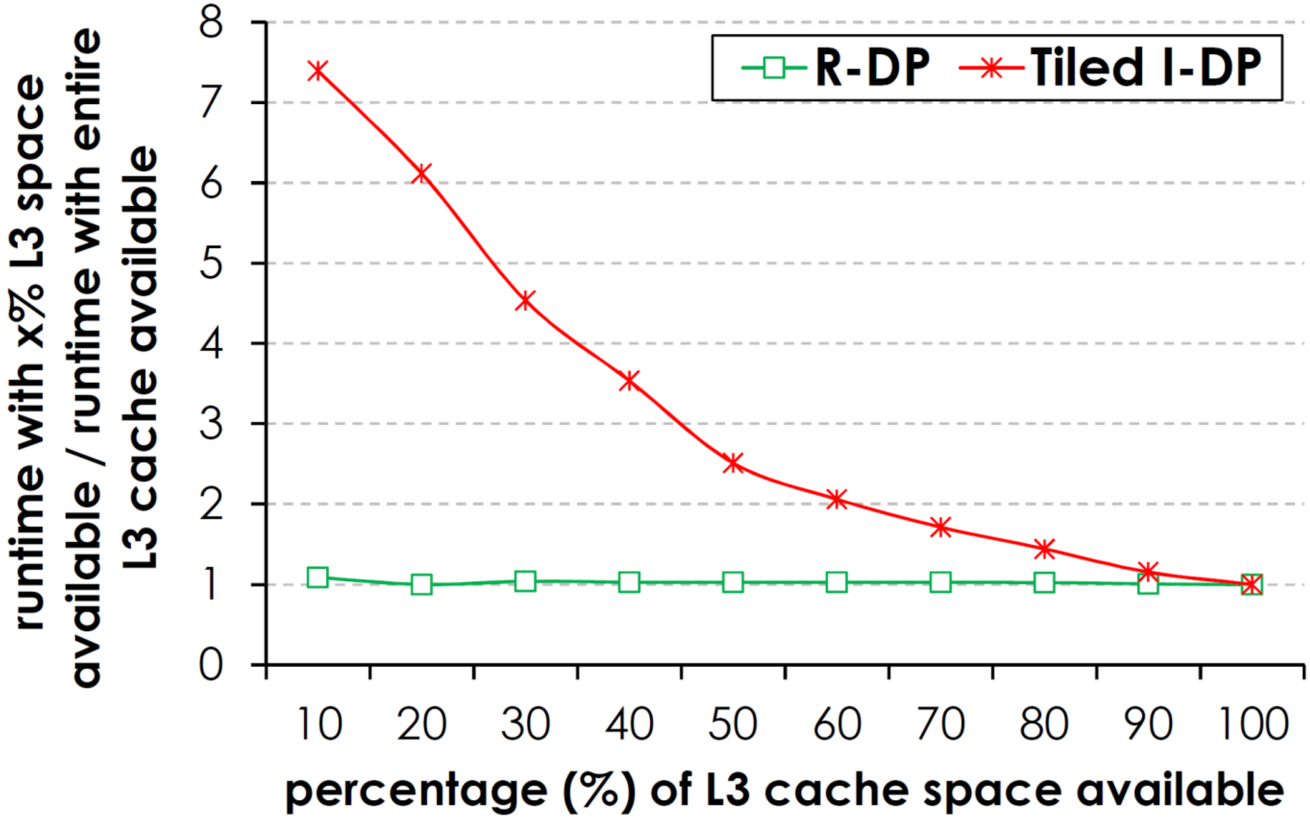
R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2×8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64 × 64),
I-DP: iterative DP, **Tiled I-DP:** tiled iterative DP



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Dynamic Programming vs. Divide-and-Conquer

- Dynamic programming, like the divide-and-conquer method, solves problems by combining solutions to subproblems
- Divide-and-conquer algorithms
  - partition the problem into disjoint subproblems,
  - solve the subproblems recursively, and
  - then combine their solutions to solve the original problem
- In contrast, dynamic programming applies when the subproblems overlap — that is, when subproblems share subsubproblems
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Elements of Dynamic Programming

An optimization problem must have the following two ingredients for dynamic programming to apply.

1) Optimal substructure
   - an optimal solution to the problem contains within it optimal solutions to subproblems

2) Overlapping subproblems
   - subproblems share subsubproblems and/or subsubsubproblems and/or subsubsubsubproblems, and so on

Dynamic Programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1) Characterize the structure of an optimal solution.
2) Recursively define the value of an optimal solution.
3) Compute the value of an optimal solution, typically in a bottom-up fashion.
4) Construct an optimal solution from computed information.

If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.

If we perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Longest Common Subsequence (LCS)

A *subsequence* of a sequence X is obtained by deleting zero or more symbols from X.

Example:

$X =$ *abcba*

$Z =$ *bca* $\leftarrow$ obtained by deleting the 1^{st} '*a*' and the 2^{nd} '*b*' from X

A *Longest Common Subsequence* (*LCS*) of two sequence X and Y is a sequence Z that is a subsequence of both X and Y, and is the longest among all such subsequences.

Given X and Y, the *LCS problem* asks for such a Z.

LCS: Optimal Substructure

Given two sequences: $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$

Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y.

For $0 \le i \le m$, let $X_i = \langle x_1, x_2, \dots, x_i \rangle$. We define Y_i and Z_i similarly.

Then

(1) If $x_m = y_n$,

then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

(2) If $x_m \ne y_n$,

then $z_k \ne x_m$ implies that Z is an LCS of X_{m-1} and Y.

(3) If $x_m \ne y_n$,

then $z_k \ne y_n$ implies that Z is an LCS of X and Y_{n-1}.

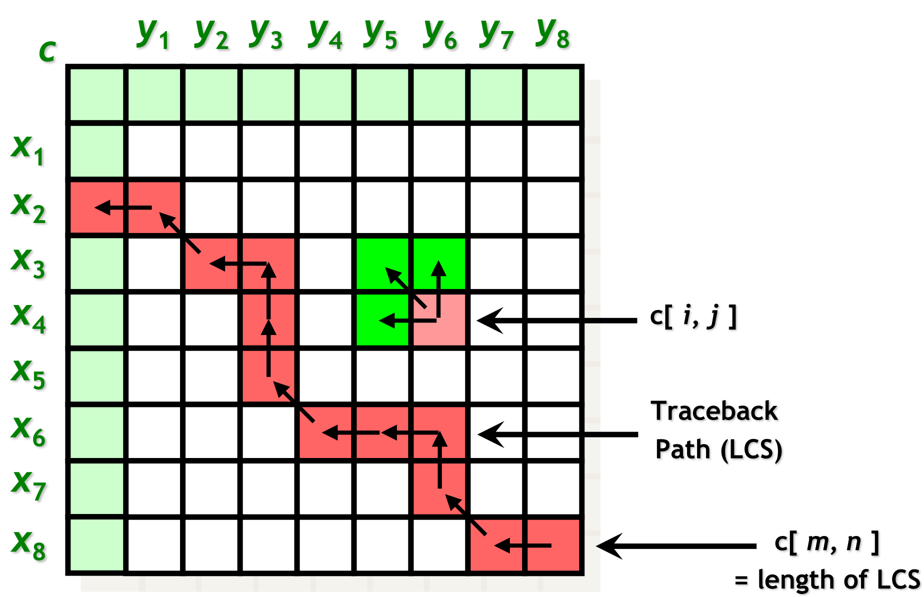
LCS: Recurrence

Given two sequences: $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$

For $0 \le i \le m$ and $0 \le j \le n$,

let $c[i,j]$ be the length of an LCS of X_i and Y_j. Then

$$c[i,j] = \begin{cases} 0, & if\ i = 0 \lor j = 0, \\ c[i-1, j-1] + 1, & if\ i, j > 0 \land x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\}, & otherwise. \end{cases}$$



LCS: Bottom-up DP

```
LCS-LENGTH ( X, Y )
m ← X.length
n ← Y.length
b[1 … m, 1 … n] ← new table, c[0 … m, 0 … n] ← new table
for i ← 1 to m
    c[i, 0] ← 0
for j ← 0 to n
    c[0, j] ← 0
for i ← 1 to m
    for j ← 1 to n
        if x_i = y_j
            c[i, j] ← c[i − 1, j − 1] + 1
            b[i, j] ← "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] ← c[i − 1, j]
            b[i, j] ← " ↑"
        else c[i, j] ← c[i, j − 1]
            b[i, j] ← "←"
```

Running time $= \Theta(mn)$

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0						
6	A	0						
7	B	0						

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0						
7	B	0						

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0						

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A (circled)
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A (circled)	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4 (circled)
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	*B*	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	*C*	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	*B*	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	*D*	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	*A*	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	*B*	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		B	D	Ⓒ	A	Ⓑ	Ⓐ
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	Ⓒ	0	↑ 1	↑ 1	↖ ②	← 2	↑ 2	↑ 2
4	Ⓑ	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ ③	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	Ⓐ	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ ④
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		Ⓑ	D	Ⓒ	A	Ⓑ	Ⓐ
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	Ⓑ	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	Ⓒ	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	Ⓑ	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	Ⓐ	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Bottom-up DP

i	j	0	1	2	3	4	5	6
	y_j		(*B*)	*D*	(*C*)	*A*	(*B*)	(*A*)
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	(*B*)	0	↖ (1)	← 1	← 1	↑ 1	↖ 2	← 2
3	(*C*)	0	↑ 1	↑ 1	↖ (2)	← 2	↑ 2	↑ 2
4	(*B*)	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ (3)	← 3
5	*D*	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	(*A*)	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ (4)
7	*B*	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS: Constructing an LCS

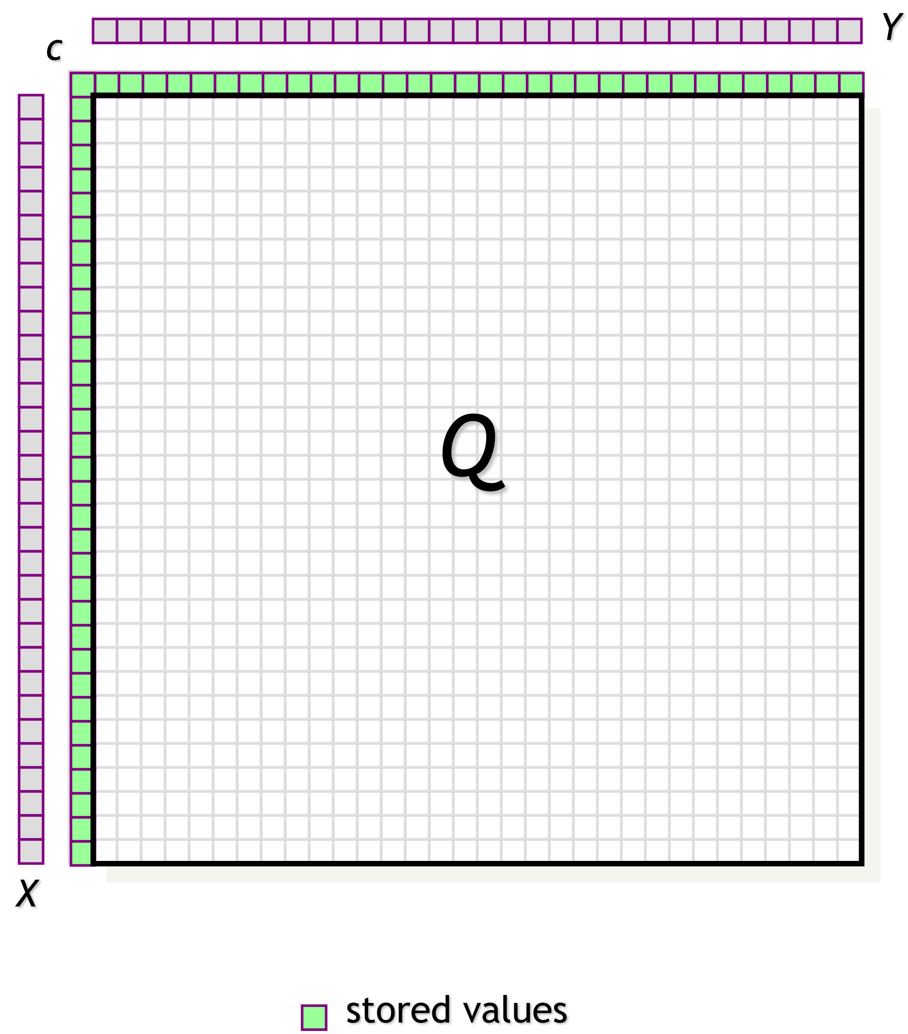
```
PRINT-LCS ( b, X, i, j )

    if i = 0 or j = 0
        return
    if b[i, j] = "↖"
            PRINT-LCS ( b, X, i − 1, j − 1 )
            print x_i
    elseif b[i, j] = "↑"
            PRINT-LCS ( b, X, i − 1, j )
    else PRINT-LCS ( b, X, i, j − 1 )
```

Running time $= O(m + n)$

LCS: Linear Space with Recursive Divide-&-Conquer

$Q \equiv c[1 \ldots n,\ 1 \ldots n]$

$n = 2^q$



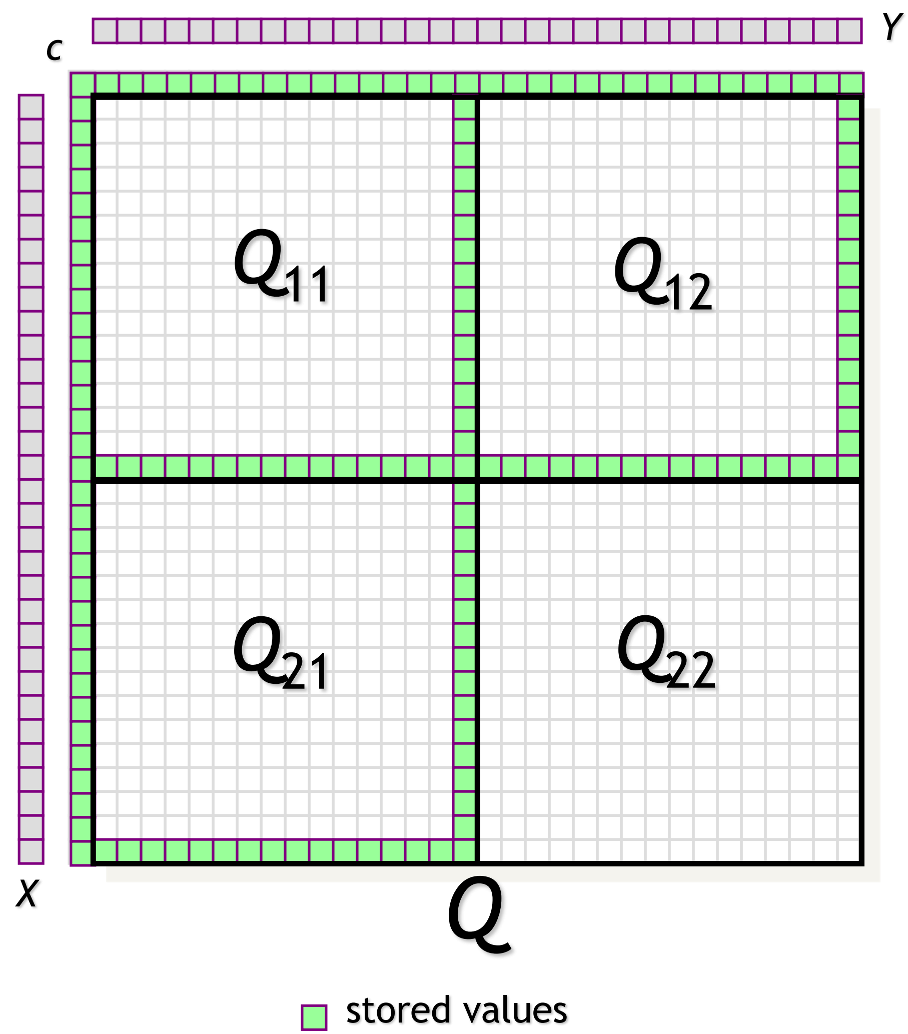
LCS: Linear Space with Recursive Divide-&-Conquer

$Q \equiv c[1 \ldots n, 1 \ldots n]$

$n = 2^q$

1. **Decompose Q:**
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**
   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)



LCS: Linear Space with Recursive Divide-&-Conquer

$Q \equiv c[1 \dots n, 1 \dots n]$

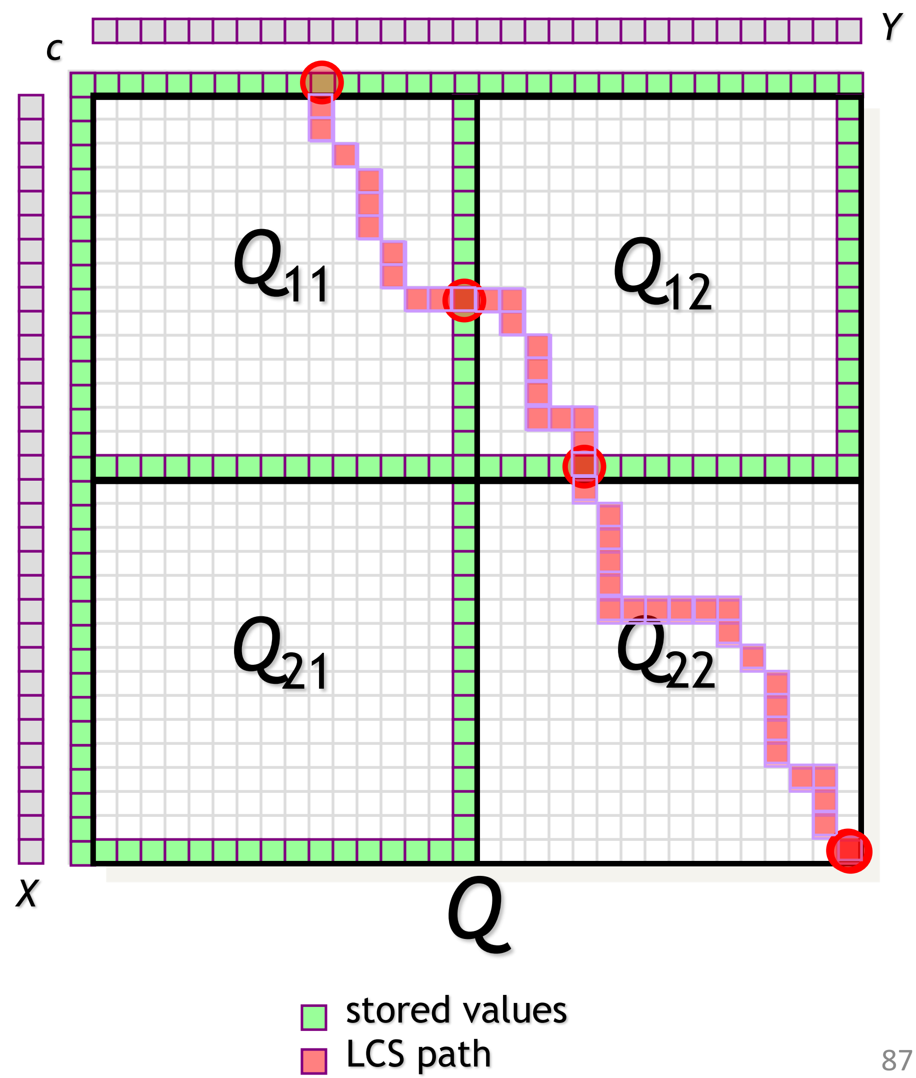
$n = 2^q$

1. **Decompose Q:**
   Split Q into four quadrants.

2. **Forward Pass (Generate Boundaries):**
   Generate the right and the bottom boundaries of the quadrants recursively. (of at most 3 quadrants)

3. **Backward Pass (Extract LCS-Path Fragments):**
   Extract LCS-Path fragments from the quadrants recursively. (from at most 3 quadrants)

4. **Compose LCS-Path:**
   Combine the LCS-Path fragments.



Optimal Binary Search Trees (OPBST)

Given (1) a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct key in sorted order (so that $k_1 < k_2 < \cdots < k_n$),

(2) for $i \in [1, n]$, probability p_i that a search will be for k_i,

(3) for $i \in [1, n-1]$, probability q_i that a search will be for a key (say, d_i) between k_i and k_{i+1},

(4) probability q_0 that a search will be for a key (say, d_0) smaller than k_1, and

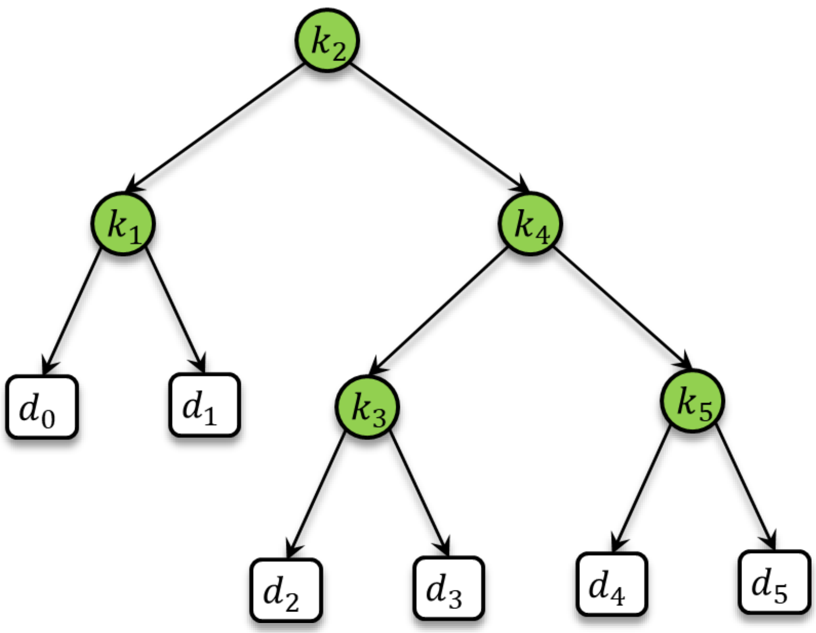
(5) probability q_n that a search will be for a key (say, d_n) larger than k_n.

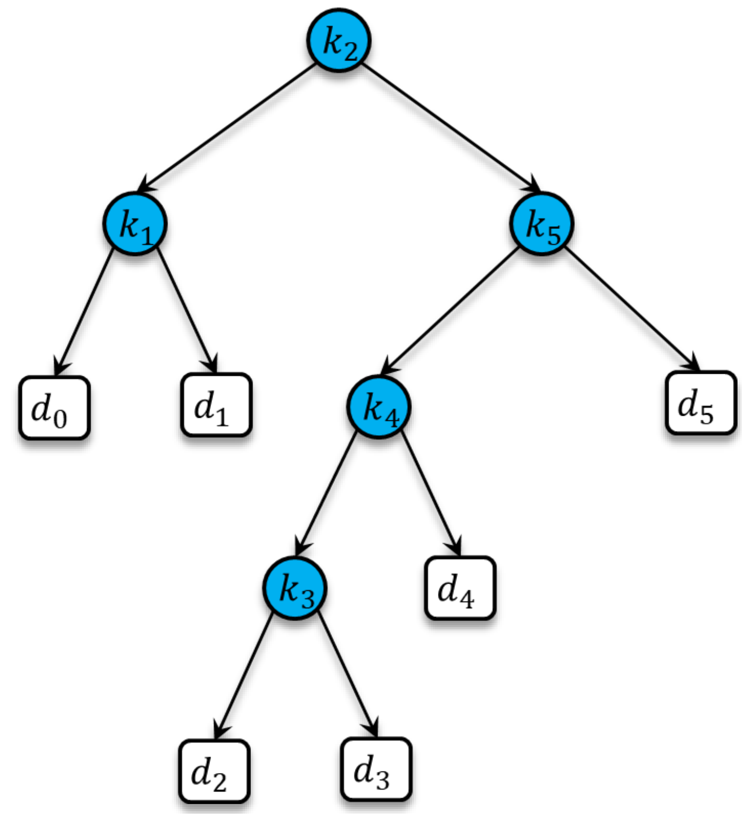
So, $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$

Construct a binary search tree T from keys in K such that the following expected search cost in T is minimized:

$$\sum_{i=1}^{n} (depth(k_i) + 1).\, p_i + \sum_{i=0}^{n} (depth(d_i) + 1).\, q_i$$

Optimal Binary Search Trees (OPBST)

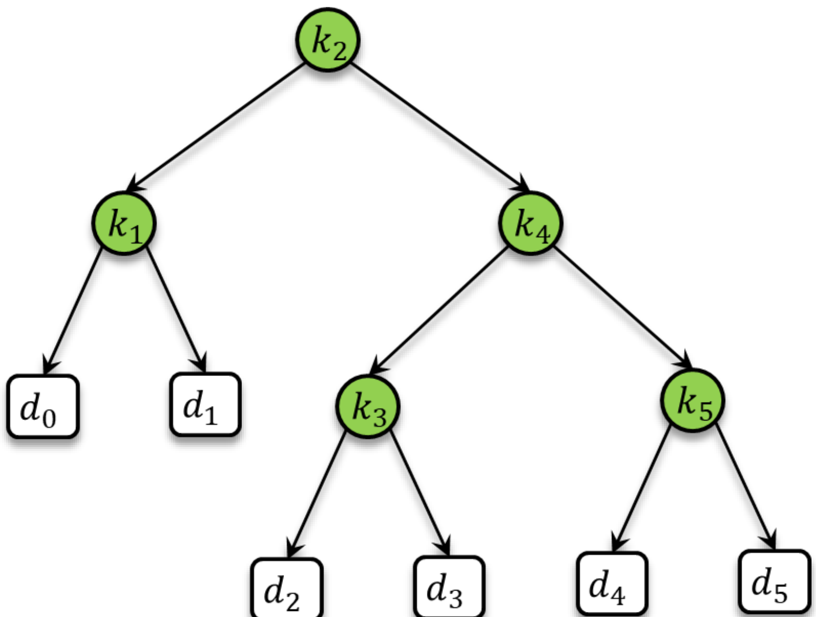




k_i	k_1	k_2	k_3	k_4	k_5
p_i	0.15	0.10	0.05	0.10	0.20

d_i	d_0	d_1	d_2	d_3	d_4	d_5
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal Binary Search Trees (OPBST)



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Binary Search Trees (OPBST)

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

OPBST: Recurrence

Let $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ for $1 \le i \le j \le n$.

Let $e(i,j) =$ expected cost of searching an optimal binary search tree containing the keys $k_i, \dots, k_j$.

Then $e(1,n) =$ expected cost of searching an optimal binary search tree containing $k_1, \dots, k_n$ (i.e., containing all keys).

If k_r is the root of an optimal subtree containing $k_i, \dots, k_j$, then

$$\begin{aligned} e(i,j) &= p_r + \{e(i,r-1) + w(i,r-1)\} \\ &\quad + \{e(r+1\,,j) + w(r+1,j)\} \\ &= e(i,r-1) + e(r+1,j) + w(i,j) \end{aligned}$$

Hence,

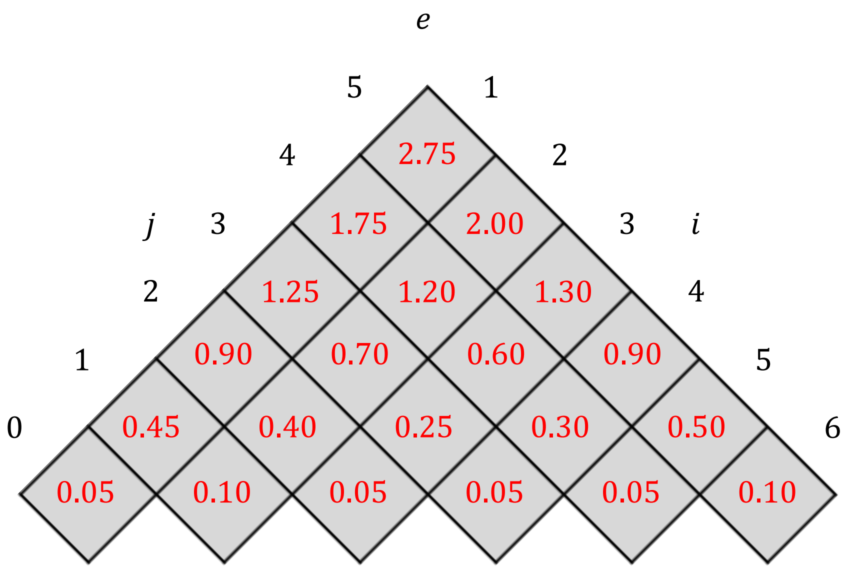
$$e(i,j) = \begin{cases} q_{i-1}, & if\ j = i-1, \\ \min\limits_{i \le r \le j} \{e(i,r-1) + e(r+1,j) + w(i,j)\}, & if\ i < j. \end{cases}$$

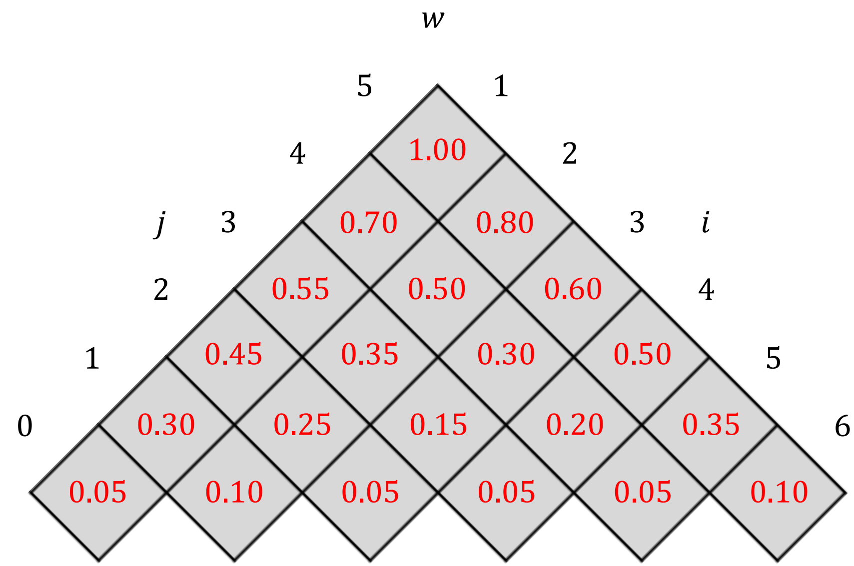
OPBST: Bottom-up DP (Cubic Time)

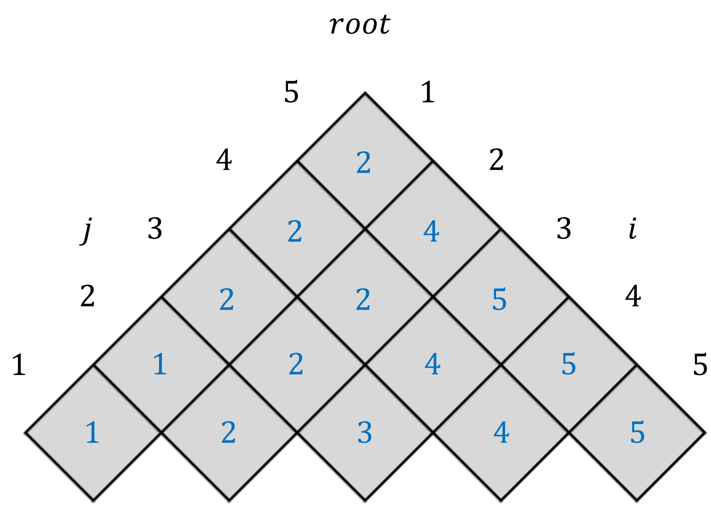
```
OPTIMAL-BST ( p, q, n )
    e[1..n+1, 0..n] ← new table,
    w[1..n+1, 0..n] ← new table,
    root[1..n, 1..n] ← new table
    for i ← 1 to n+1 do
        e[i, i−1] ← q_{i−1}
        w[i, i−1] ← q_{i−1}
    for l ← 1 to n do
        for i ← 1 to n−l+1 do
            j ← i+l−1
            e[i, i] ← ∞
            w[i, j] ← w[i, j−1] + p_j + q_j
            for r ← i to j do
                t ← e[i, r−1] + e[r+1, j] + w[i, j]
                if t < e[i, j] then
                    e[i, j] ← t
                    root[i, j] ← r
    return e and root
```

Running time $= \Theta(n^3)$

OPBST: Bottom-up DP (Cubic Time)







OPBST: Bottom-up DP (Quadratic Time)

```
OPTIMAL-BST ( p, q, n )
1.  e[1..n+1, 0..n] ← new table,
    w[1..n+1, 0..n] ← new table,
    root[1..n, 1..n] ← new table
2.  for i ← 1 to n + 1 do
3.      e[i, i − 1] ← q_{i−1}
4.      w[i, i − 1] ← q_{i−1}
5.  for l ← 1 to n do
6.      for i ← 1 to n − l + 1 do
7.          j ← i + l − 1
8.          e[i, j] ← ∞
9.          w[i, j] ← w[i, j − 1] + p_j + q_j
10.         for r ← root[i, j − 1] to root[i + 1, j] do
11.             t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
12.             if t < e[i, j] then
13.                 e[i, j] ← t
14.                 root[i, j] ← r
15. return e and root
```

Running time $= \Theta(n^2)$

Longest Increasing Subsequence (LIS)

An *Increasing Subsequence* L of a given sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of numbers is obtained by deleting zero or more numbers from A such that every number $x \in L$ is larger than the number immediately preceding x in L.

A *Longest Increasing Subsequence* (*LIS*) of A has the maximum length among all increasing subsequences of A.

Longest Increasing Subsequence (LIS)

Let's augment the given sequence $A = \langle a_1, a_2, \dots, a_n \rangle$ to include a sentinel value $a_0 = -\infty$. Thus $\langle a_0, a_1, a_2, \dots, a_n \rangle$ is our augmented sequence.

Let $LIS(i)$ be the length of the longest increasing subsequence of $\langle a_i, a_{i+1}, \dots, a_n \rangle$ that starts at a_i.

Then

$$LIS(i) = 1 + \max_{i<j\leq n} \left\{ LIS(j) \mid a_j > a_i \right\}$$

Running time $= \Theta\left(n^2\right)$.

Subset Sum

Given an array $A[1..n]$ of n positive integers and a target integer T, determine if any subset of the numbers in A sum up to T.

Subset Sum

Given an array $A[1..n]$ of n positive integers and a target integer T, determine if any subset of the numbers in A sum up to T.

Let $S(i,t)$ be $True$ iff some subset of $A[i..n]$ adds up to t.

Then

$$S(i,t) = \begin{cases} True, & if\ t = 0, \\ False, & if\ t < 0\ or\ i > n, \\ S(i+1,t) \vee S(i+1,t-A[i]), & otherwise. \end{cases}$$

Running time $= \Theta(nT)$.

The resulting DP algorithm is called a *pseudo-polynomial time algorithm* because its running time depends on the numeric value of the input.

The Knapsack Problem

You have a knapsack of integer weight capacity W.

There are n items to pick from with the i^{th} item having weight w_i and value v_i, where $1 \leq i \leq n$. All weight values are integers.

You need to pickup the most valuable combination of items that fit in your knapsack

Unbounded Knapsack:

Pick up as many copies of each item as you want.

0/1 Knapsack:

Pick up at most one copy of each item.

The Knapsack Problem

You have a knapsack of integer weight capacity W.

There are n items to pick from with the i^{th} item having weight w_i and value v_i, where $1 \leq i \leq n$. All weight values are integers.

You need to pickup the most valuable combination of items that fit in your knapsack

Unbounded Knapsack:

Pick up as many copies of each item as you want.

Let $K(w) =$ maximum value achievable with a knapsack of capacity w.

Then $K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\}$

Running time $= \Theta(nW)$.

The Knapsack Problem

You have a knapsack of integer weight capacity W.

There are n items to pick from with the i^{th} item having weight w_i and value v_i, where $1 \le i \le n$. All weight values are integers.

You need to pickup the most valuable combination of items that fit in your knapsack

0/1 Knapsack:

Pick up at most one copy of each item.

Let $K(w, i) =$ maximum value achievable with a knapsack of capacity w and items $1, 2, \ldots, i$.

Then $K(w, i) = \max\{K(w - w_i, i - 1) + v_i, K(w, i - 1)\}$

Running time $= \Theta(nW)$.