Longest Increasing Subsequence

Longest Increasing Subsequence

Input: an array A of n numbers.

Output: a longest subsequence of A in which the elements increase strictly.

--- Example ---

Input: array A

12	4	13	9	9	10	2	15

Output: an LIS

12	4	13	9	9	10	2	15

12	4	13	9	9	10	2	15

Reduce LIS to LCS - $O(n^2)$ Time

$LIS_1(A)$ {

Let S be sorted A with the removal of duplicate elements;

return LCS(S, A);

}

Why does this algorithm correctly output an LIS?

DP + Binary Search - O(n log n) Time

12	4	13	9	U

Before U:

IS of length 0: {}
IS of length 1: {12} or {4} or {13} or {9}
IS of length 2: {12, 13} or {4, 9} or {4, 13}

Key Idea:

If {4, 13} is a part of an LIS, one can replace {4, 13} with {4, 9}. There is no need to memoize all IS's.

DP + Binary Search - O(n log n) Time

12	4	13	9	U

Before U:

IS of length 0: {}
IS of length 1: {12} or {4} or {13} or {9}
IS of length 2: {12, 13} or {4, 9} or {4, 13}

Key Idea:

If {4, 13} is a part of an LIS, one can replace {4, 13} with {4, 9}. There is no need to memoize all IS's. We only need to memoize the one whose last element is smallest.

DP + Binary Search - O(n log n) Time

12	4	13	9	U = 3

Before U:

IS of length 0: {}
IS of length 1: {4}
IS of length 2: {4, 9}

Key Idea:

To update the structure, one can perform a *binary search* (Why?) to see where U = 3 can help.

DP + Binary Search - O(n log n) Time

12	4	13	9	U = 3

Before U:

IS of length 0: {}
IS of length 1: {3}
IS of length 2: {4, 9}

Key Idea:

To update the structure, one can perform a *binary search* (Why?) to see where U = 3 can help. Note that at most 1 update is needed. (Why?)

DP + Binary Search - O(n log n) Time

12	4	13	9	U = 3

Before U:

IS of length 0: {}
IS of length 1: {3}
IS of length 2: {4, 9}

Running Time:

O(n) iterations and each iteration needs O(log n) time. The total running time is O(n log n).

Exercise

Find a longest common subsequence of two strings where one string has no duplicate character.

Goal: solve it in O(n log n) time.

Hint: LIS.

Matrix Multiplication

Matrix Multiplication

Input: two n by n matrices A and B.

Output: the product C = AB

One may assume that n is a power of 2. It this is not true, then one can enlarge A and B by appending some zeros.

A First Attempt

Represent $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

where A_{ij} and B_{ij} are n/2 by n/2 submatrices.

Let $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$ where $C_{ij} = \sum_{1 \le k \le 2} A_{ik} B_{kj}$.

$$T(n) = \begin{cases} 8T(n/2) + O(n^2) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}$$

By Master theorem, $T(n) = O(n^3)$.

Strassen's Algorithm

Represent $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

where A_{ij} and B_{ij} are n/2 by n/2 submatrices.

Let $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$ where $\cancel{C_{ij} = \sum_{1 \leq k \leq 2} A_{ik} B_{kj}.}$

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Strassen's Algorithm

Represent $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

where A_{ij} and B_{ij} are n/2 by n/2 submatrices.

Let $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$ where C_{ij} is a linear combination of

$\{M_1, M_2, ..., M_7\}$.

$$T(n) = \begin{cases} 7T(n/2) + O(n^2) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}$$

By Master theorem, $T(n) = O(n^{\log_2 7})$.

Exercise

Read the article "How Can We Speedup Matrix Multiplication?" by Victor Pan. It gives a good entry point to understand other fast matrix multiplication algorithms.

- Optional

The current fastest algorithm runs in

$O(n^{2.37xx})$ time.

Matrix-Chain Multiplication

Matrix-Chain Multiplication

Input: n matrices $A_1, A_2, \ldots, A_n$ where A_i is an r_i by c_i matrix and $r_i = c_{i-1}$ for every i in [2, n].

Output: the product $C = A_1 A_2 \ldots A_n$

--- Observation ---

1. Matrix multiplication is **associative**, so all possible parenthesizations yield the same product. For example, $((A_1A_2)A_3) = (A_1(A_2A_3))$.

2. **However**, the way to parenthesize the matrix-chain multiplication changes the performance dramatically.

Matrix-Chain Multiplication

```
Matrix-Multiply(A[p][q], B[q][r]){

    for i = 1 to p {
        for j = 1 to r {
            C[i][j] = 0;
            for k = 1 to q{
                C[i][j] = C[i][j] + A[i][k]B[k][j];
            }
        }
    }
}
```

Multipling two matrices of dimension p by q and dimension q by r needs O(pqr) scalar operations.

Matrix-Chain Multiplication

Let A_1 be a 10 by 100 matrix.
Let A_2 be a 100 by 5 matrix.
Let A_3 be a 5 by 50 matrix.

Calculating $((A_1A_2)A_3)$ needs 10*100*5 + 10*5*50 = 7500 scalar operations.

Calculating $(A_1(A_2A_3))$ needs 100*5*50 + 10*100*50 = 75000 scalar operations.

One is faster than the other by 10 times.

Divide and Conquer

McMul(a, b){ // return the least number of scalar operations to multiply the matrix chain A_a, A_{a+1}, … A_b.

```
  if(a == b) return 0;

  int lno_op = ∞;

  for k = a to b-1  { // Guess (A_a … A_k)(A_k+1 … A_b) is optimal
     if (McMul(a, k) + McMul(k+1, b) + r_a c_k c_b < lno_op)
          lno_op = McMul(a, k) + McMul(k+1, b) + r_a c_k c_b;
  }
  return lno_op;
}
```

The initial call is McMul(1, n).
The total number of subproblems invoked by this recursive algorithm is exponential in n, but there are only $O(n^2)$ different subproblems.

Exercise

Prove the number of subproblems invoked by the divide and conquer procedure in the previous page is exponential in n.

(Hint. Guess the number is $2^{\Omega(n)}$.)

Dynamic Programming

McMul(a, b, sol[][]){ // return the least number of scalar operations to multiply the matrix chain $A_a, A_{a+1}, \dots A_b$.

```
  if(sol[a][b] < ∞) return sol[a][b];
  if(a == b) return 0;

  int lno_op = ∞;

  for k = a to b-1  { // Guess (A_a … A_k)(A_k+1 … A_b) is optimal
     if (McMul(a, k, sol) + McMul(k+1, b, sol) + r_a c_k c_b < lno_op)
         lno_op = McMul(a, k, sol) + McMul(k+1, b, sol) + r_a c_k c_b;
  }
  return sol[a][b] = lno_op;
}
```

The initial call is McMul(1, n, sol[][] = {∞}).

Each of the $O(n^2)$ subproblems needs $O(n)$ operations. The total runtime is $O(n^3)$.

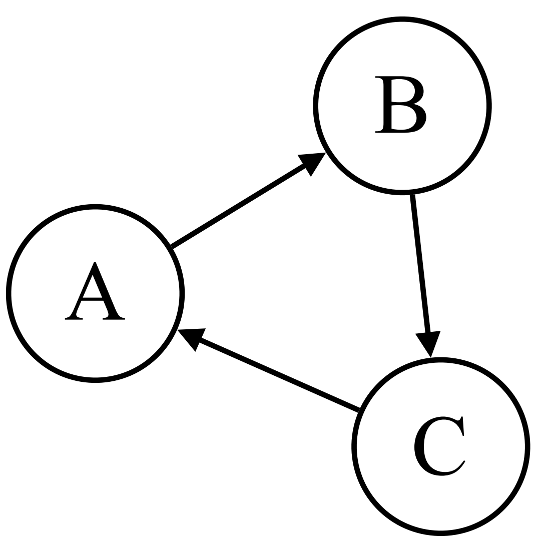
DP on Trees

Property of Trees

If graph T is a tree, then T has **no cycle**.

To use D&C or DP, we cannot allow that the dependency graph of the recursive algorithm has a cycle. (Why?)

For example, if a recursive algorithm has three subproblems A, B, C where A calls B, B calls C, and C calls A, then the dependency graph has a cycle. Such an algorithm does not halt because it enters an endless loop.



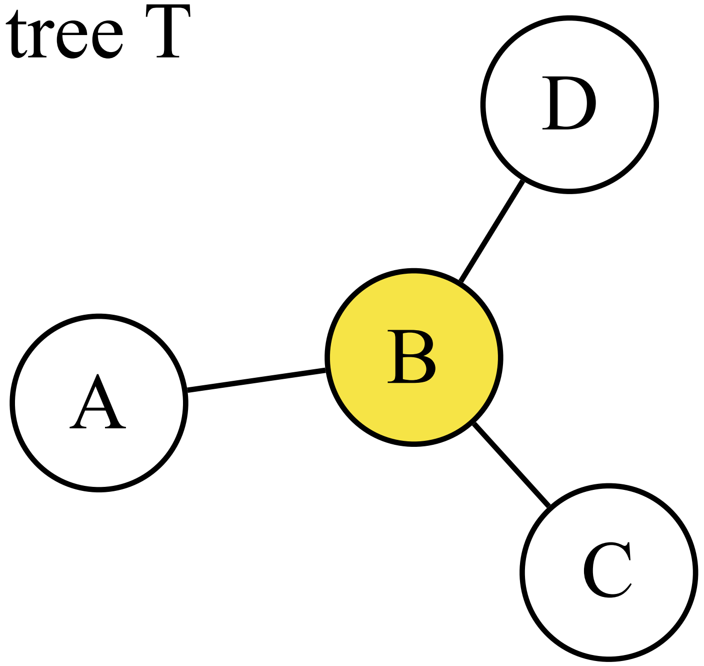
Explain why the recursive algorithms mentioned previously have no cycle.

Maximum Independent Set on Trees

Input: a tree $T = (V, E)$.

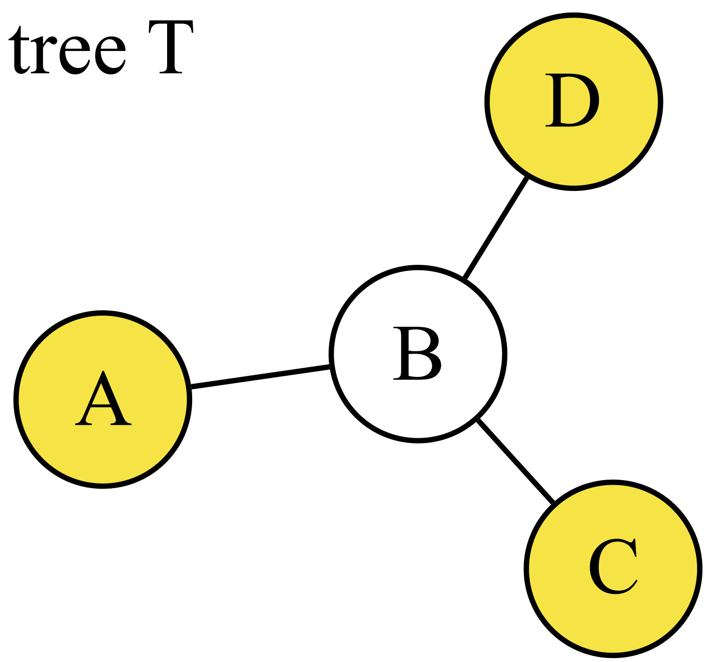
Output: a subset I of V so that $|I|$ is **maximized** and for every pair of nodes $u, v \in I$, the edge $(u, v) \notin E$.

tree T



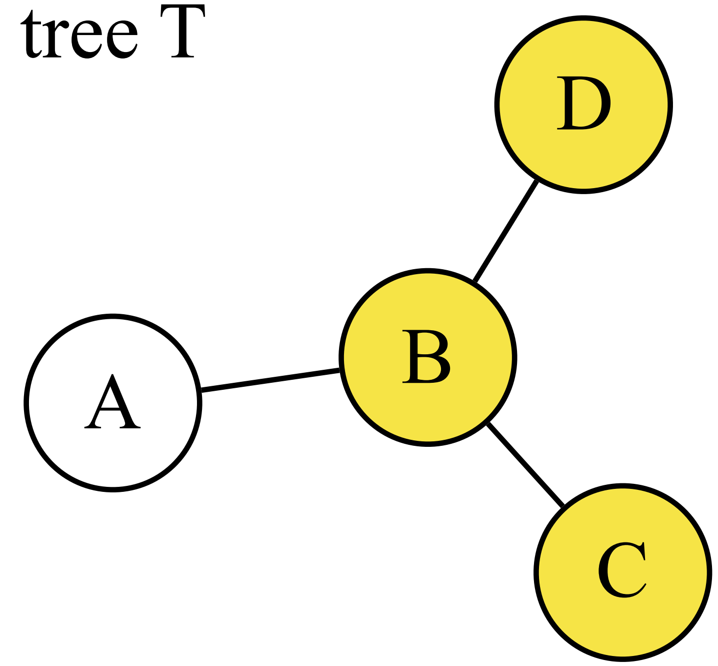
{B} is an independent set.

tree T



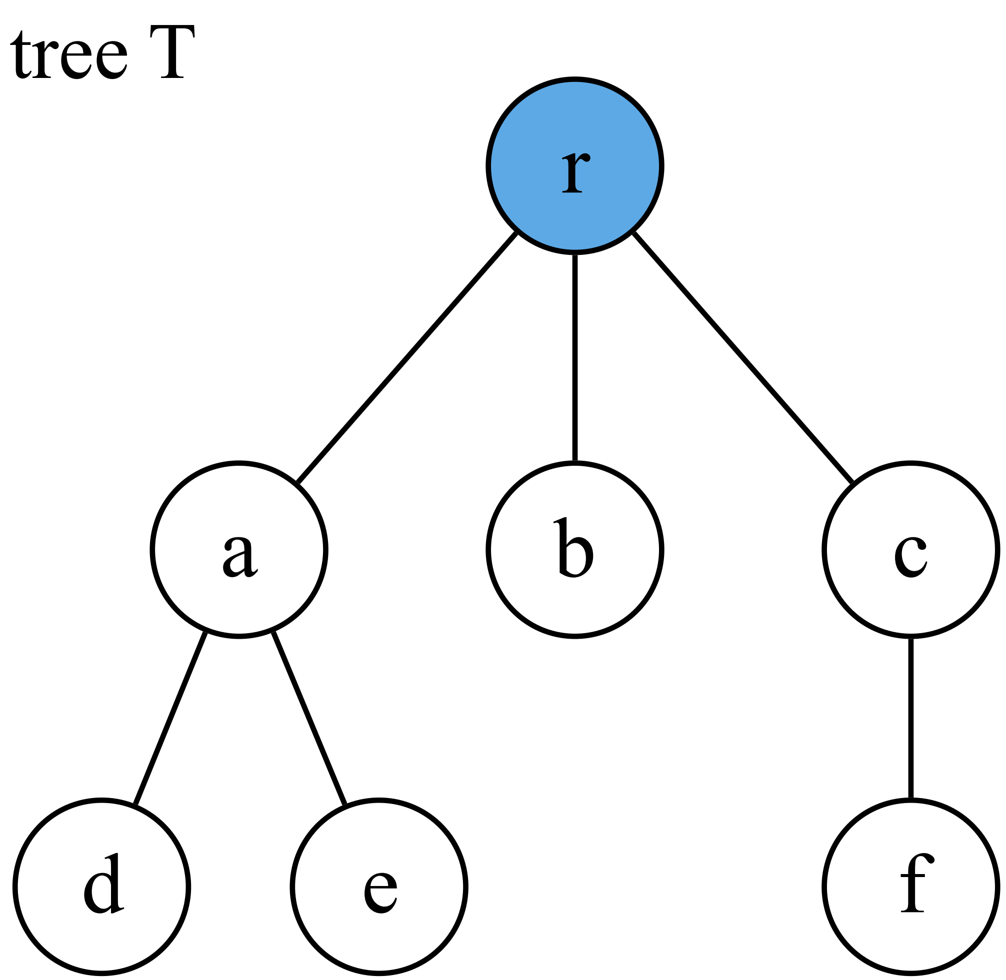
{A, C, D} is a larger independent set.

tree T



{B, C, D} is **not** an independent set.

Maximum Independent Set on Trees



Pick a node as the root.

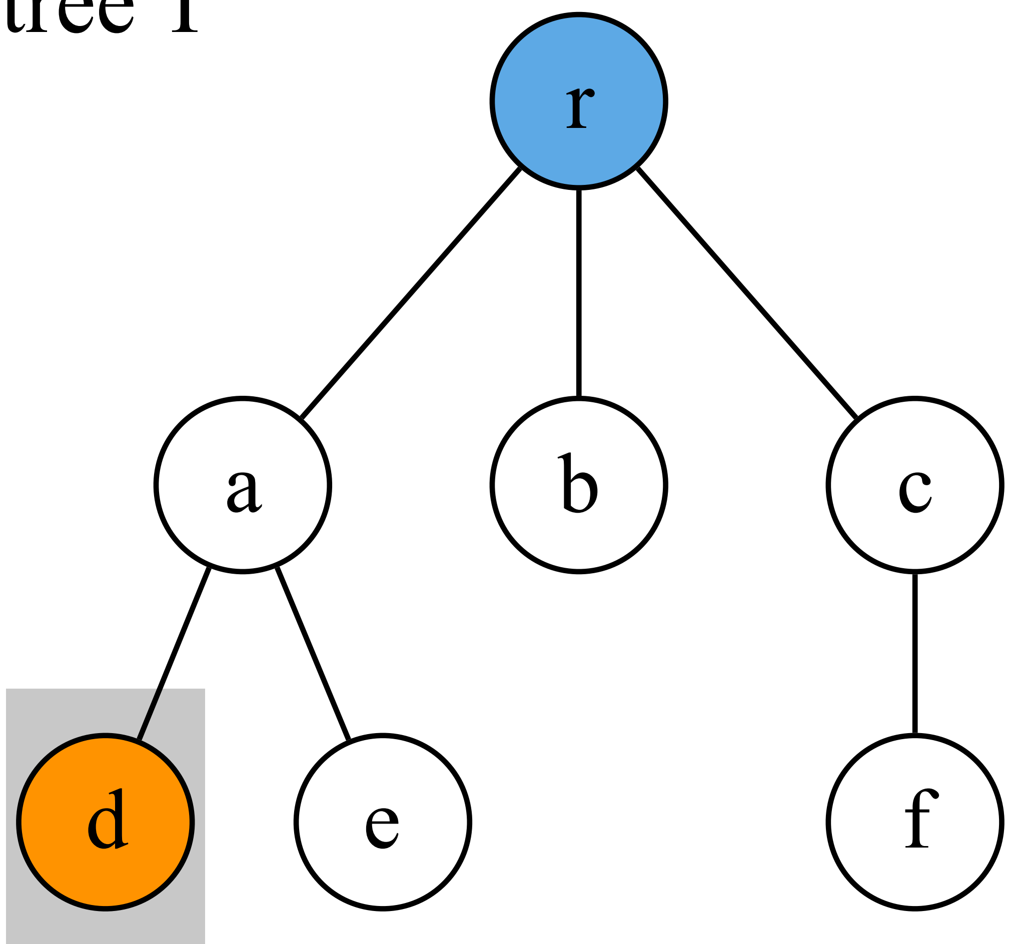
Let O[x] be the largest size of such a node set S that S is an independenet set of the subtree rooted at x, and $x \in S$.

Let Z[x] be the largest size of such a node set S that S is an independenet set of the subtree rooted at x, and $x \notin S$.

The desired answer is $\max\{O[r], Z[r]\}$.

Maximum Independent Set on Trees

tree T



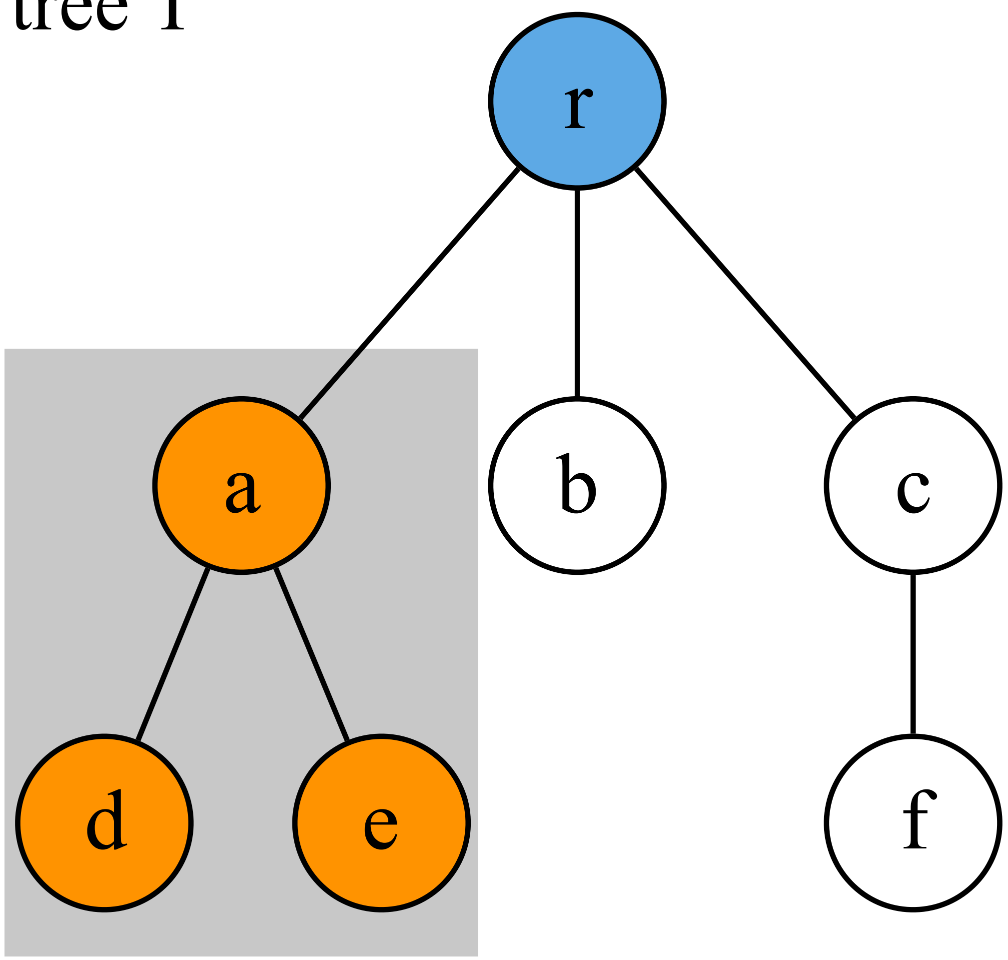
(O[d], Z[d]) = (1, 0).

(O[e], Z[e]) = (1, 0).

(O[f], Z[f]) = (1, 0).

Maximum Independent Set on Trees

tree T



(O[d], Z[d]) = (1, 0).

(O[e], Z[e]) = (1, 0).

(O[f], Z[f]) = (1, 0).

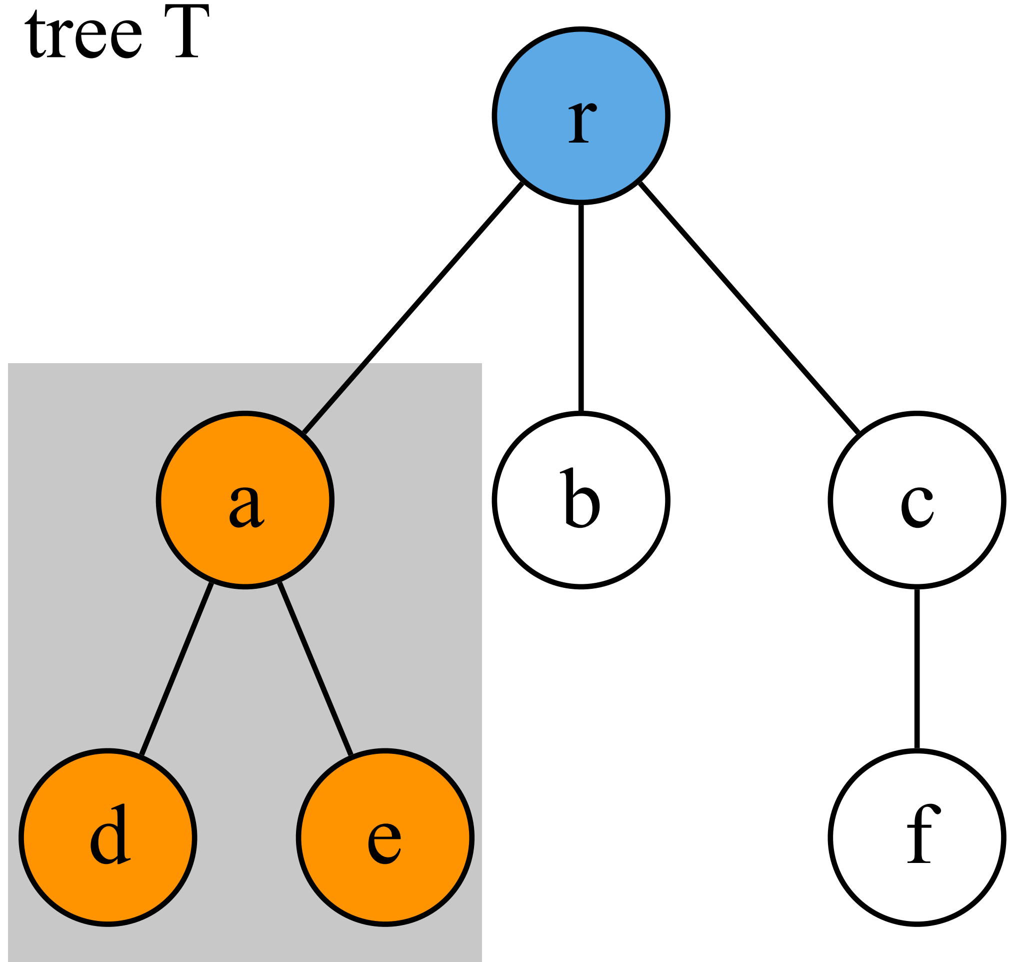
Focus on the subtree rooted at node a.

O[a] = Z[d] + Z[e] + 1 = 1.

Z[a] = max{Z[d], O[d]} + max{Z[e], O[e]} + 0 = 2.

Maximum Independent Set on Trees

tree T



(O[d], Z[d]) = (1, 0).

(O[e], Z[e]) = (1, 0).

(O[f], Z[f]) = (1, 0).

Focus on the subtree rooted at node a.

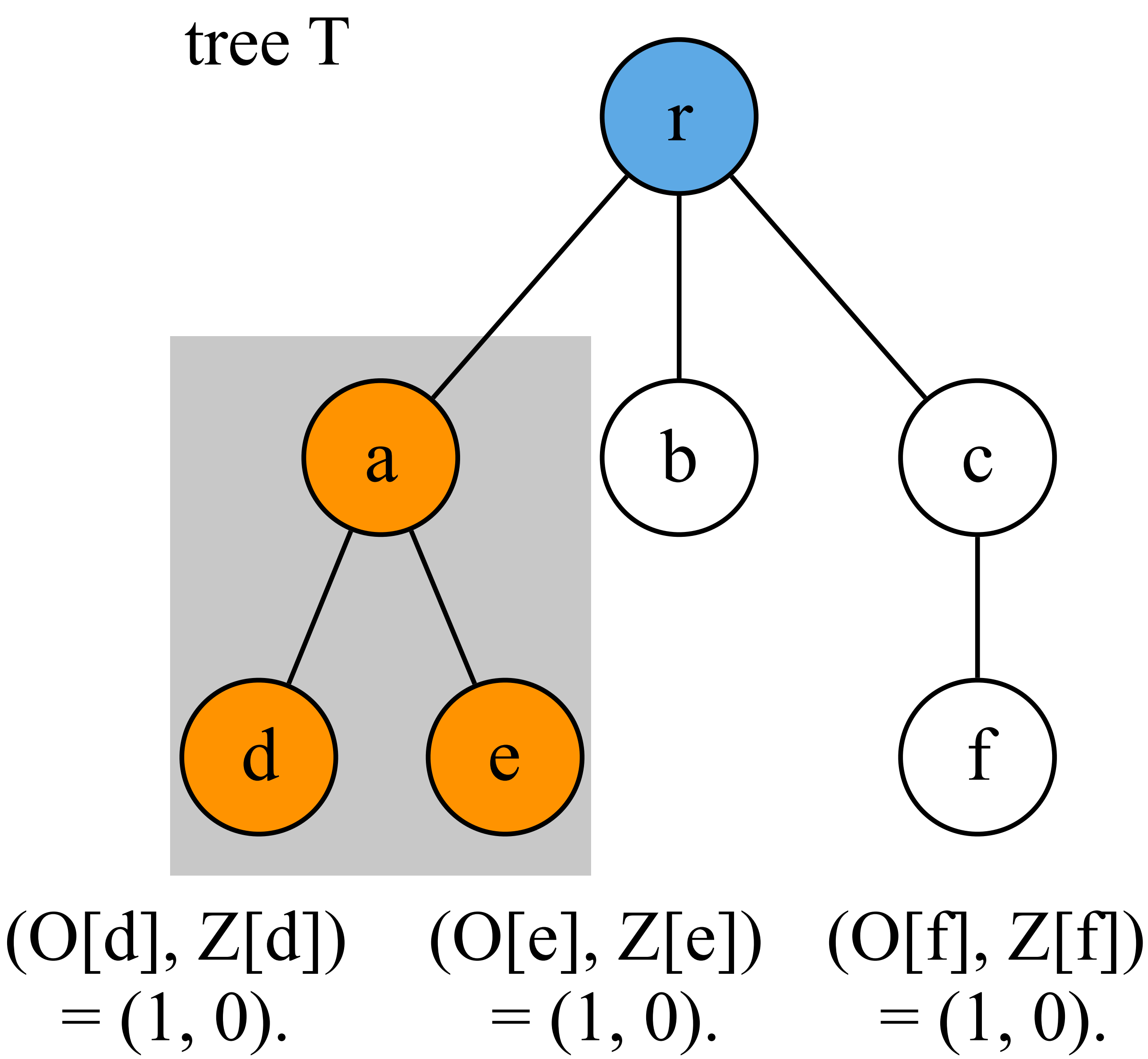
O[a] = Z[d] + Z[e] + 1 = 1.

Z[a] = max{Z[d], O[d]} + max{Z[e], O[e]} + 0 = 2.

If tree T has n nodes, then the MIS of T can be found in O(n) time. Why?

There might be some nodes that have $n^{1/2}$ child nodes.

Maximum Independent Set on Trees



Focus on the subtree rooted at node a.

O[a] = Z[d] + Z[e] + 1 = 1.

Z[a] = max{Z[d], O[d]} + max{Z[e], O[e]} + 0 = 2.

If tree T has n nodes, then the MIS of T can be found in O(n) time. Why?

However, if we amortize the cost to the n-1 edges. Each edge needs O(1) cost.