# Dynamic Programming

Computing Lab

http://www.isical.ac.in/~dfslab

# General structure

## Key idea

1. Break given problem into subproblems, and subproblems into even smaller subproblems.
2. Solve a subproblem using the answers to smaller subproblems.
3. Solve the given problem using the answers to the subproblems.

## Challenge

Identifying the problem $\longrightarrow$ subproblems $\longrightarrow$ smaller subproblems structure.

**Related ideas:** (more about DP vs. these ideas later)

- Divide and conquer
- Recursion

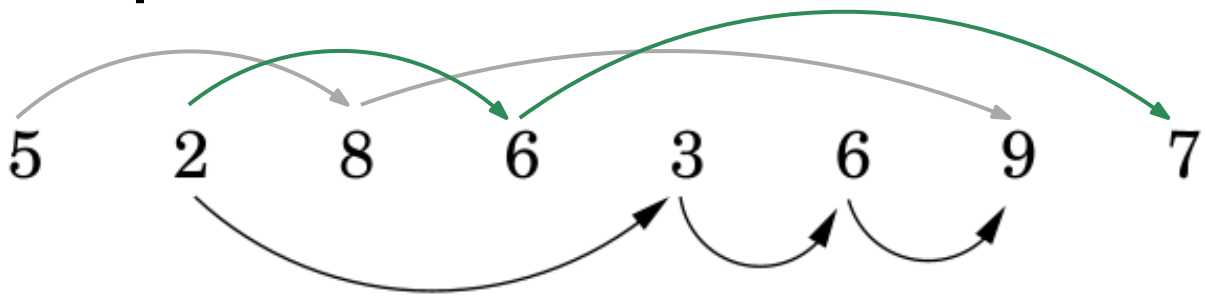
# Outline

# Definition

**Input:** a sequence of numbers $a_1, \ldots, a_n$.

**Increasing subsequence:** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.

**Example:**




Increasing subsequences: $5, 8, 9$ (grey) $2, 6, 7$ (green)
Longest increasing subsequence: $2, 3, 6, 9$ (black) (also $2, 3, 6, 7$)

# Approach

**Case I:** $n = 1 \rightarrow$ trivial

**Case II:** $n = 2 \rightarrow$ need to check whether $a_2 > a_1$

**Case III:** $n = 3$

- **Case III A:** $a_2 < a_1 \rightarrow$ candidates: $\langle a_1 \rangle \quad \langle a_2 \rangle$
- **Case III B:** $a_2 = a_1 \rightarrow$ uninteresting; ignored
- **Case III A:** $a_2 > a_1 \rightarrow$ candidates: $\langle a_1 \rangle \quad \langle a_2 \rangle \quad \langle a_1, a_2 \rangle$

**Question:** Can we forget *any* of the candidates so far?

# Approach

**Case I:** $n = 1 \rightarrow$ trivial

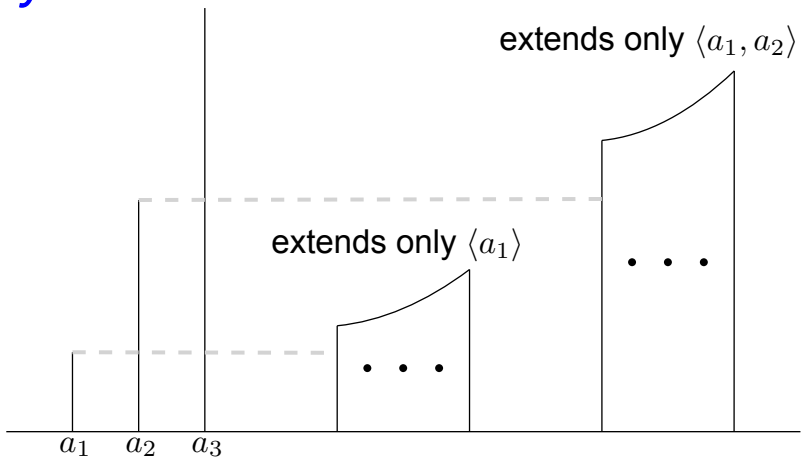
**Case II:** $n = 2 \rightarrow$ need to check whether $a_2 > a_1$

**Case III:** $n = 3$

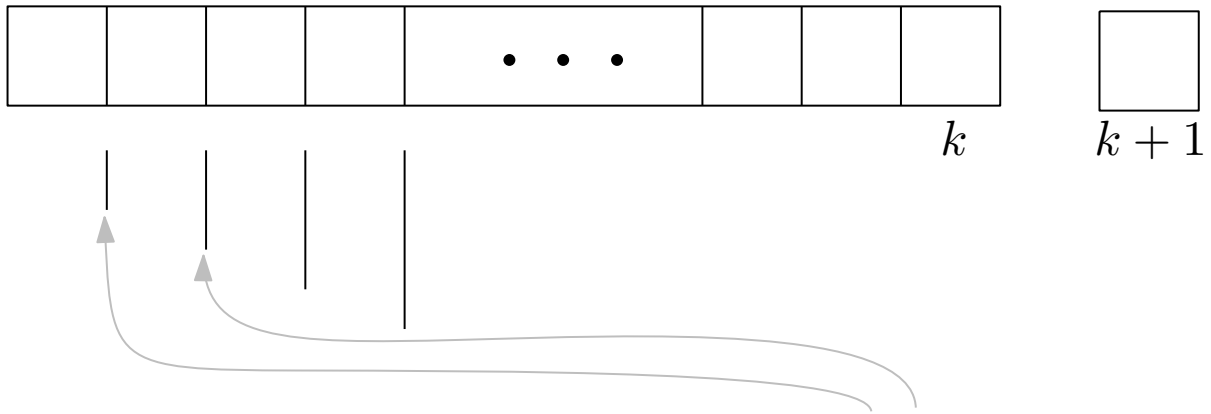
- **Case III A:** $a_2 < a_1 \rightarrow$ candidates: $\langle a_1 \rangle \quad \langle a_2 \rangle$
- **Case III B:** $a_2 = a_1 \rightarrow$ uninteresting; ignored
- **Case III A:** $a_2 > a_1 \rightarrow$ candidates: $\langle a_1 \rangle \quad \langle a_2 \rangle \quad \langle a_1, a_2 \rangle$

**Question:** Can we forget *any* of the candidates so far?

**Answer:** **NO!** Consider



# Problem $\rightarrow$ subproblem $\rightarrow$ smaller subproblem structure

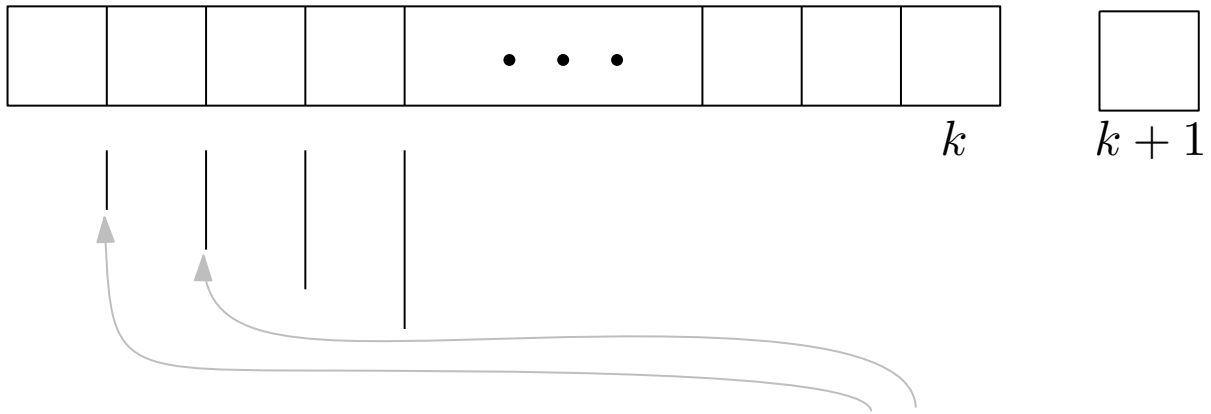



Store the LIS **up to and including** $i = 0, 1, 2, \ldots$ (*how?*)

**Question:** What is the LIS that ends at $k + 1$?

**Sub-question:** Which increasing sequences end at $k + 1$?

# Problem → subproblem → smaller subproblem structure




Store the LIS **up to and including** $i = 0, 1, 2, \ldots$ (*how?*)

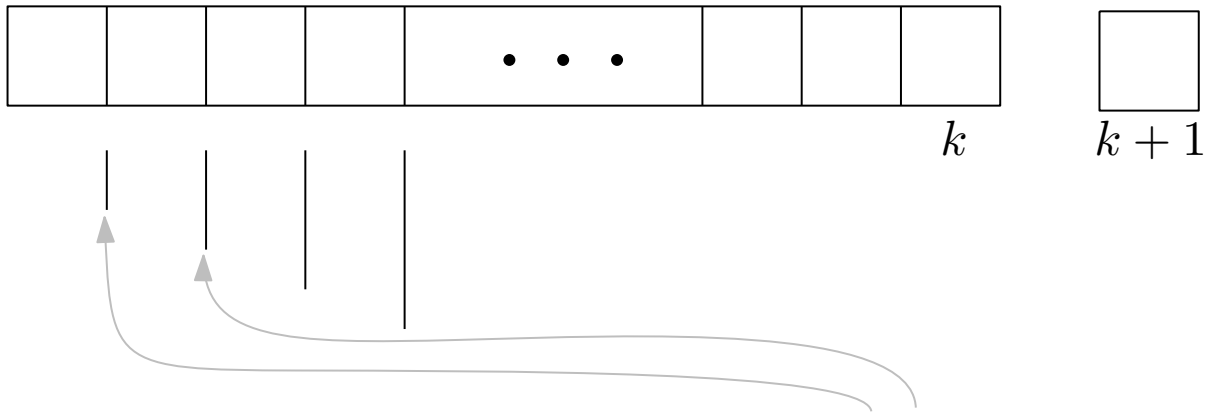
**Question:** What is the LIS that ends at $k + 1$?

**Sub-question:** Which increasing sequences end at $k + 1$?

**Algorithm:**

```
for (k = 1; k < ac-1; k++)
    for (j = 0; j < k; j++)
        if (A[k] > A[j] &&
            length[k] < length[j] + 1) { /* try <= instead of < */
            length[k] = length[j] + 1;
            predecessor[k] = j;
        }
```

# Problem $\rightarrow$ subproblem $\rightarrow$ smaller subproblem structure




Store the LIS **up to and including** $i = 0, 1, 2, \ldots$ (*how?*)

**Question:** What is the LIS that ends at $k + 1$?

**Sub-question:** Which increasing sequences end at $k + 1$?

**Algorithm:**

```
for (k = 1; k < ac-1; k++)
    for (j = 0; j < k; j++)
        if (A[k] > A[j] &&
            length[k] < length[j] + 1) { /* try <= instead of < */
            length[k] = length[j] + 1;
            predecessor[k] = j;
        }
```

How to make this more efficient?

# Some implementation details

```
k--;
max_length = length[k];
end_of_lis = k--;
while (k >= 0) {
    if (length[k] > max_length) { /* try >= instead of > */
        max_length = length[k];
        end_of_lis = k;
    }
    k--;
}
```

# Some more implementation details

```
/* Print the actual LIS: sorry this is confusing */
l = length[end_of_lis];
printf("Longest increasing subsequence (%d elements): ", l);
for (i = l-1, j = end_of_lis;
     j != -1 && i >= 0;
     i--, j = predecessor[j])
    /* Being lazy, sorry: reusing the length array to store actual
lis */
    length[i] = j;
assert(i == -1 && j == -1);
for (i = 0; i < l; i++)
    printf("A[%d] = %d,   ", length[i], A[length[i]]);
```

# Outline

# Definition

**Input:** two strings, $s = s_1 s_2 s_3 \ldots s_m$ and $t = t_1 t_2 t_3 \ldots t_n$.

(Indexing starts from 1 to simplify discussion; code will use indexing from 0 as usual.)

**Edit distance:** minimum no. of insertions, deletions, substitutions requimygreen to convert $s$ to $t$

$\equiv$ minimum cost of *aligning* $s$ and $t$ using insertions, deletions, substitutions.

**Examples:**

| a | s | t | r | o | n | o | m | e | r | _ | _ | _ | _ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | S | S | I | I | I | I |
| a | s | t | r | o | n | o | m | i | c | a | l | l | y |

Edit distance = 6

| a | s | t | r | o | n | o | m | e | r |
|---|---|---|---|---|---|---|---|---|---|
| | | | D | | | S | S | S | D |
| a | s | t | _ | o | n | i | s | h | _ |

Edit distance = 5

# Problem $\rightarrow$ subproblem $\rightarrow$ smaller subproblem structure

**Motivation:**
For LIS, we looked at "prefixes" of the given array $\Rightarrow$ for edit distance, we consider prefixes of $s$ and $t$.

**Approach:**
Let $E(i, j)$ be the edit distance between $s_1 s_2 s_3 \ldots s_i$ and $t_1 t_2 t_3 \ldots t_j$ (we want $E(m, n)$).

**Question:** How to express $E(i, j)$ in terms of $E()$ for smaller strings?

**Key idea:** Consider the last position in the alignment of $s_{1\ldots i}$ and $t_{1\ldots j}$.

# Algorithm

| Case I | Case II | Case III |
|---|---|---|
| $s[i]$ | _ | $s[i]$ |
| _ | $t[j]$ | $t[j]$ |
| D | I | S / NOP |

$$E(i,j) = \min\{E(i-1,j)+1,\ \ E(i,j-1)+1,\ \ E(i-1,j-1)+(s[i] \stackrel{?}{=} t[j])\}$$

# Implementation

## Initialisation

```
edit_info[0][0].distance = 0;
edit_info[0][0].operation = NOP;
for (i = 1; i < m+1; i++) {
    edit_info[i][0].distance = i;
    edit_info[i][0].operation = DELETE;
}
for (j = 1; j < n+1; j++) {
    edit_info[0][j].distance = j;
    edit_info[0][j].operation = INSERT;
}
```

# Implementation

## Updates

```
for (i = 1; i < m+1; i++) {
    for (j = 1; j < n+1; j++) {
        e = &edit_info[i][j];
        e->distance = edit_info[i-1][j].distance + 1;
        e->operation = DELETE;

        if (edit_info[i][j-1].distance + 1 < e->distance) {
            e->distance = edit_info[i][j-1].distance + 1;
            e->operation = INSERT;
        }

        same = (s[i-1] == t[j-1]) ? 1 : 0;
        d = edit_info[i-1][j-1].distance + (1-same);
        if (d < e->distance) {
            e->distance = d;
            e->operation = 1-same; // see note before #define NOP,
SUBST, etc.
        }
```

## Computing the edits

```
if (NULL == (ops = Malloc(m+n, uint))) // I think this is actually
unnecessarily conservative
    ERR_MESG("levenstein: out of memory\n");
for (i = m, j = n, k = 0; i>=1 || j>=1; k++) {
    ops[k] = edit_info[i][j].operation;
    switch (ops[k]) {
    case INSERT:
        j--; break;
    case DELETE:
        i--; break;
    case NOP:
    case SUBST:
        i--, j--; break;
    default:
        fprintf(stderr, "Unknown operation %u\n", ops[k]);
        exit(1);
    }
}
```

Case II in slide 12

Case I in slide 12

Case III in slide 12

# Time and space complexity related issues

- Provided implementation: $O(mn)$ space, $O(mn)$ time
- Easy extension: compute edit distance only in $O(m)$ space, $O(mn)$ time
- Will not cover: edit distance **and** alignment in $O(m)$ space, $O(mn)$ time

# Outline

# Problem statement

**Problem:** Multiply $A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$

**Example:** $A_{10 \times 20} \times B_{20 \times 100} \times C_{100 \times 30}$

**Options:**

1. $\left(A_{10 \times 20} \times B_{20 \times 100}\right) \times C_{100 \times 30}$
2. $A_{10 \times 20} \times \left(B_{20 \times 100} \times C_{100 \times 30}\right)$

# Problem statement

**Problem:** Multiply $A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$

**Example:** $A_{10 \times 20} \times B_{20 \times 100} \times C_{100 \times 30}$

**Options:**

1. $\left(A_{10 \times 20} \times B_{20 \times 100}\right) \times C_{100 \times 30}$
2. $A_{10 \times 20} \times \left(B_{20 \times 100} \times C_{100 \times 30}\right)$

No. of arithmetic operations
$\approx 10 \times 20 \times 100 + 10 \times 100 \times 30$
$= 50,000$

No. of arithmetic operations
$\approx 20 \times 100 \times 30 + 10 \times 20 \times 30$
$= 66,000$

# Problem statement

**Problem:** Multiply $A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$

**Example:** $A_{10 \times 20} \times B_{20 \times 100} \times C_{100 \times 30}$

**Options:**

1. $(A_{10 \times 20} \times B_{20 \times 100}) \times C_{100 \times 30}$
2. $A_{10 \times 20} \times (B_{20 \times 100} \times C_{100 \times 30})$

No. of arithmetic operations
$\approx 10 \times 20 \times 100 + 10 \times 100 \times 30$
$= 50,000$

No. of arithmetic operations
$\approx 20 \times 100 \times 30 + 10 \times 20 \times 30$
$= 66,000$

# Top-down approach

$$A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$$

**Top down approach:**

1. Break-up for the final / top-level multiplication

$$\left( A^{(1)} \times \cdots \times A^{(k)} \right) \times \left( A^{(k+1)} \times \cdots \times A^{(m)} \right)$$

$$k = 1, \ \ldots, \ m - 1$$

2. Recursively solve left sub-chain and right sub-chain

Disadvantage: the same sub-problem is solved many times.

# Bottom-up approach

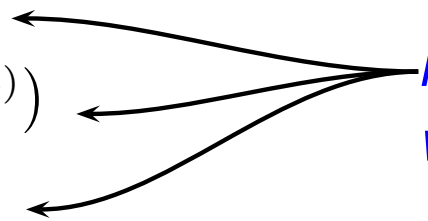
$$A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$$

1. Only one way to compute $A^{(i)} \times A^{(i+1)}$
2. Two ways to compute $A^{(i)} \times A^{(i+1)} \times A^{(i+2)}$
3. Three ways to compute $A^{(i)} \times A^{(i+1)} \times A^{(i+2)} \times A^{(i+3)}$:
   (a) $A^{(i)} \times \left(A^{(i+1)} \times A^{(i+2)} \times A^{(i+3)}\right)$
   (b) $\left(A^{(i)} \times A^{(i+1)}\right) \times \left(A^{(i+2)} \times A^{(i+3)}\right)$
   (c) $\left(A^{(i)} \times A^{(i+1)} \times A^{(i+2)}\right) \times A^{(i+3)}$

# Bottom-up approach

$$A^{(1)}_{n_1 \times n_2} \times A^{(2)}_{n_2 \times n_3} \times \cdots \times A^{(m)}_{n_m \times n_{m+1}}$$

1. Only one way to compute $A^{(i)} \times A^{(i+1)}$
2. Two ways to compute $A^{(i)} \times A^{(i+1)} \times A^{(i+2)}$
3. Three ways to compute $A^{(i)} \times A^{(i+1)} \times A^{(i+2)} \times A^{(i+3)}$:
   (a) $A^{(i)} \times \left(A^{(i+1)} \times A^{(i+2)} \times A^{(i+3)}\right)$
   (b) $\left(A^{(i)} \times A^{(i+1)}\right) \times \left(A^{(i+2)} \times A^{(i+3)}\right)$
   (c) $\left(A^{(i)} \times A^{(i+1)} \times A^{(i+2)}\right) \times A^{(i+3)}$



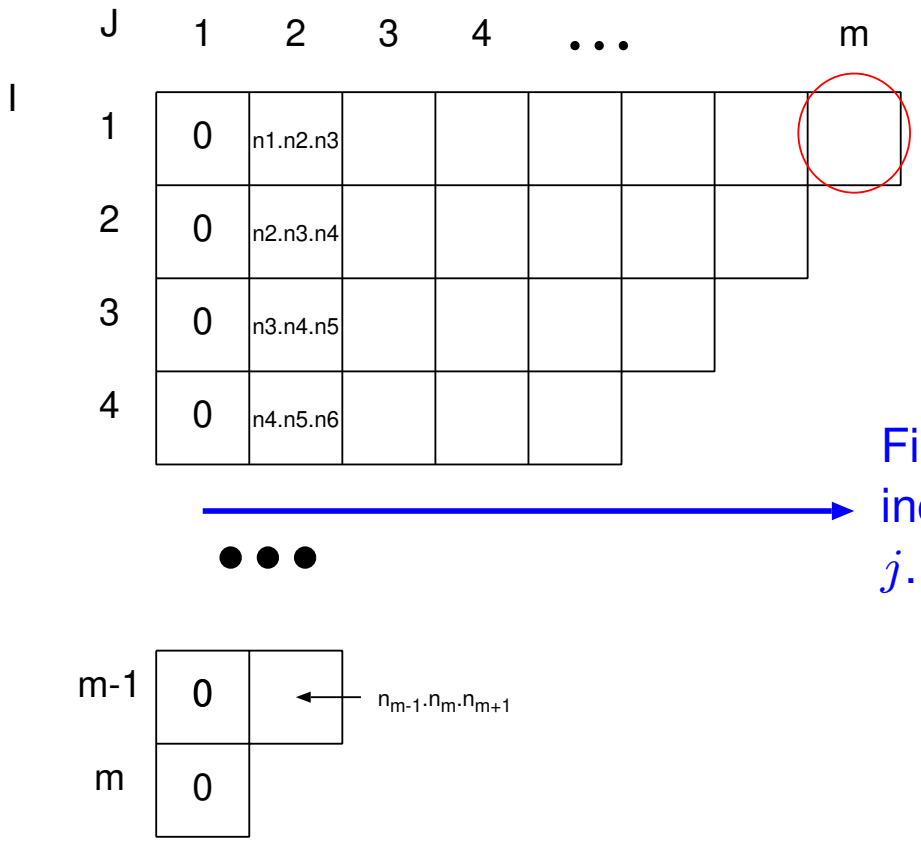
*We already know the best way to do this.*

# Algorithm

$$Cost[i,j] = \begin{cases} 0 & \text{if } j = 1 \\ \min_{1 \le k \le j-1} \left\{ Cost[i,k] + Cost[i+k, j-k] + n_i n_{i+k} n_{i+j} \right\} & \text{if } j > 1 \end{cases}$$

position

length of chain




This is the value we want.

Fill in the table in increasing order of $j$.

Store cost and position in table.

# Outline

# Definition

**Input:** a tree $T(V, E)$

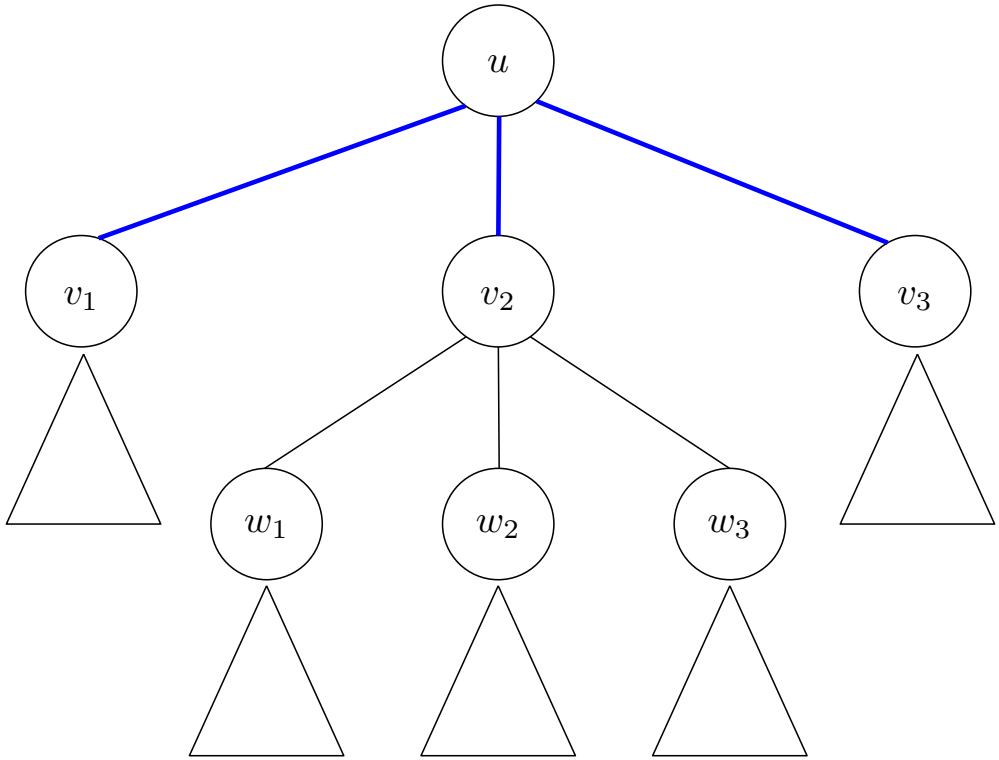
**Independent set:** $S \subset V$ s.t. $\forall u, v \in S,\ (u, v) \notin E$

**Required output:** an independent set that has the largest cardinality among all independent sets of $T$

# Problem $\rightarrow$ subproblem $\rightarrow$ smaller subproblem structure

**Preprocessing:** convert the tree to a *rooted tree* (how?)

**Bottom-up approach:**




$$I(u) = \max\{1 + \sum_{w \in \text{grandchildren}(u)} I(w), \sum_{v \in \text{children}(u)} I(v)\}$$

# Exercises

1. Implement the matrix chain multiplication algorithm.

   **Input format:** Note that a chain containing $N$ matrices can be represented as $N + 1$ positive integers. Take these numbers as command-line arguments.

   **Output format:** Print the smallest number of multiplication operations that are needed to compute the product, as well as the bracketing (how?).

2. *Longest common subsequence:* Given two strings, $s = s_1 s_2 s_3 \ldots s_m$ and $t = t_1 t_2 t_3 \ldots t_n$, find a subsequence $u = u_1 u_2 \ldots u_l$ of maximum length that occurs in both $s$ and $t$.

# References


- Chapter 6 of *Algorithms* by Dasgupta, Papadimitriou, Vazirani. `https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf`
- Chapter 15 of *Introduction to Algorithms* (2nd ed.) by Cormen, Leiserson, Rivest, Stein.