

Tutorial #9

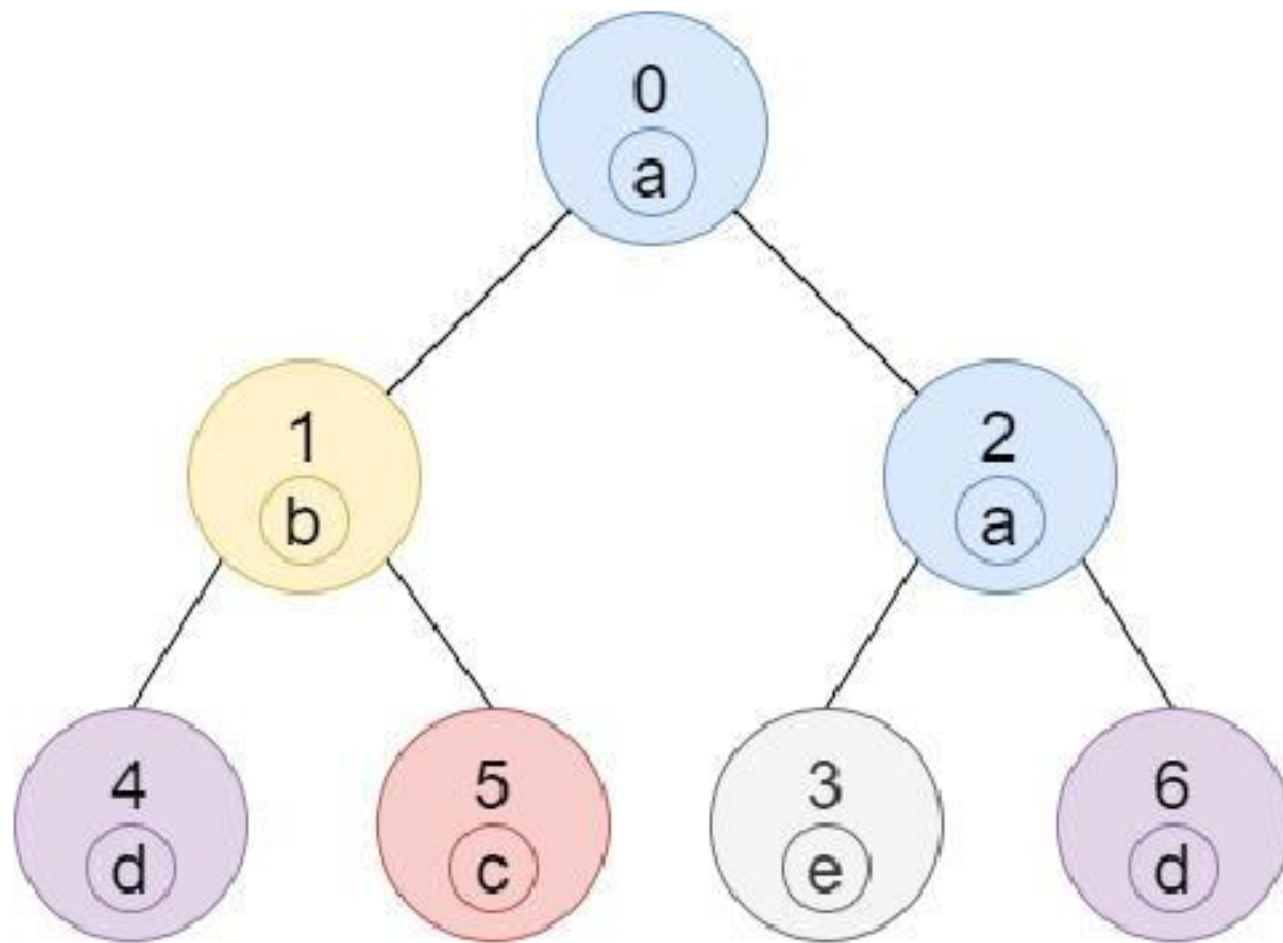
13.01.2023

You are given a tree (i.e. a connected, undirected graph that has no cycles) consisting of n nodes numbered from 0 to $n - 1$ and exactly $n - 1$ edges. The **root** of the tree is the node 0 , and each node of the tree has a **label** which is a lower-case character given in the string `labels` (i.e. The node with the number i has the label `labels[i]`).

The `edges` array is given on the form `edges[i] = [ai, bi]`, which means there is an edge between nodes a_i and b_i in the tree.

Return an array of size n where `ans[i]` is the number of nodes in the subtree of the i^{th} node which have the same label as node i .

A subtree of a tree T is the tree consisting of a node in T and all of its descendant nodes.



Input: $n = 7$, `edges = (...)`, `labels = "abaedcd"`

Output: `[2,1,1,1,1,1,1]`

①

$Q = \{ \text{all leaves of } T \}$

②

$\forall \text{ leaf } v \in Q:$

$c[v][l(v)] += 1$

merge $c[v]$ into $c[p(v)]$

③

$T = T \setminus Q$

④

if $T = \emptyset:$

stop

$\rightarrow c[v]$ integer
array for
 a, b, c, \dots

$\rightarrow p(v)$ parent of
 v

$\rightarrow l(v)$ label of
 v

Assume you own a company that buys metal poles, cuts them into various lengths and sells them. For simplicity assume you cut each pole in the same way. More specifically, you are given the length n of a pole and for each length $1 \leq i \leq n$ the profit p_i that you can make if you sell a piece of length i . You need to find out how to cut the pole so as to maximize the achieved profit, i.e., the sum of the profits of the individual pieces that you cut.

For example, if the pole has length 4 and $p_1 = 4$, $p_2 = 1$, and $p_3 = 2$, then the solution with the highest profit is to cut the pole into 4 pieces of length 1 each, with a total profit of 16.

You should develop a dynamic programming algorithm for this problem. For $1 \leq j \leq n$ let q_j be the maximum profit that can be achieved by selling a piece of length j . Note that the following recursive relationship holds: $q_j = \max_{1 \leq i \leq j} (p_i + q_{j-i})$ for $j > 0$ and $q_0 = 0$.

1. Give a dynamic programming algorithm for the above algorithm in pseudocode using the above recursive relationship. The algorithm should output the *value* (i.e., the achieved profit) of the optimal solution, not the optimal solution.
2. Give an algorithm that outputs the *optimal solution* (e.g., which lengths to cut), not just its value. For this you can modify the above dynamic programming algorithm so that it outputs additional information and then use this output as input to a second algorithm that outputs the *optimal solution*, not just its value. Give pseudocode for all your algorithms.
3. Analyze the *running time* and the *space usage* of your algorithm(s) from 2. as a function of the input size.
4. Is your algorithm from 1. a polynomial-time algorithm? Give an explanation for your answer.

$$q_j = \max_{1 \leq i \leq j} (p_i + q_{j-i}) \text{ for } j > 0 \text{ and } q_0 = 0.$$

```
n = 4
p = [0, 4, 1, 2]
q = [0 for _ in range(n+1)]
cut = [0 for _ in range(n+1)]

for j in range(n+1):
    maxVal = 0
    for i in range(j+1):
        # just bcs we start at 0 index
        if (i > n-1):
            continue
        if (maxVal < p[i] + q[j-i]):
            maxVal = p[i] + q[j-i]
            cut[j] = i
    q[j] = maxVal

print(q[-1])
```

space
and
time?

space and
time?

```
n = 4
p = [0, 4, 1, 2]
q = [0 for _ in range(n+1)]
cut = [0 for _ in range(n+1)]

for j in range(n+1):
    maxVal = 0
    for i in range(j+1):
        # just bcs we start at 0 index
        if (i > n-1):
            continue
        if (maxVal < p[i] + q[j-i]):
            maxVal = p[i] + q[j-i]
            cut[j] = i
    q[j] = maxVal

while (n > 0):
    print(cut[n])
    n -= cut[n]
```


3.

space usage:

p, q cut, sol are all arrays of maximum length j (sol could be shorter)

$\rightarrow O(n)$

time:

for the first algorithm we only have to look at the number of evaluations of the for-loops (if-statement takes constant time)

$\rightarrow O(n^2)$

second algorithm: while-loops is executed at most n -times (each time n is reduced at least by one!)

$\rightarrow O(n)$

so both algorithms together take $O(n^2)$

4.

Yes, since the input needs to consist of the n potentially different p_i values, the input size is at least n , while the running time is $O(n^2)$.

Problem 2 (8 points)

If an array $A[1 \dots n]$ of numbers is given, an increasing subsequence is a set $I = \{i_1, i_2, \dots, i_k\} \subseteq \{1, \dots, n\}$ of indices such that $i_1 < i_2 < \dots < i_k$ and $A[i_1] < A[i_2] < \dots < A[i_k]$. The length of I is its cardinality k , and the end value of I is $A[i_k]$. Let $L(A)$ denote the length of the longest increasing subsequence of A . We want to develop an algorithm for finding $L(A)$.

Let $P_i(j)$ be the smallest end value of an increasing subsequence of length j in the array $A[1 \dots i]$ and ∞ if there is no increasing subsequence of length j in $A[1 \dots i]$. Clearly, $L(A) = \max \{j : P_n(j) < \infty\}$. The following table gives an example for $n = 6$.

i	$A[i]$	$P_i(1)$	$P_i(2)$	$P_i(3)$	$P_i(4)$	$P_i(5)$
1	2	2	∞	∞	∞	∞
2	5	2	5	∞	∞	∞
3	3	2	3	∞	∞	∞
4	1	1	3	∞	∞	∞
5	4	1	3	4	∞	∞
6	6	1	3	4	6	∞

1. Fill in the following table:

i	$A[i]$	$P_i(1)$	$P_i(2)$	$P_i(3)$	$P_i(4)$	$P_i(5)$	$P_i(6)$
1	1		∞	∞	∞	∞	∞
2	6			∞	∞	∞	∞
3	3				∞	∞	∞
4	5					∞	∞
5	8						∞
6	4						

2. Show that for $2 \leq i \leq n$, $2 \leq j \leq n$, $P_i(j) = A[i]$ if $P_{i-1}(j-1) < A[i] < P_{i-1}(j)$ and otherwise $P_i(j) = P_{i-1}(j)$.
3. Outline a dynamic programming algorithm for computing $L(A)$. (Do not forget to specify in which order table entries are computed.)
4. Give an algorithm for finding $L(A)$ in time $O(n \log L(A))$.

1. Fill in the following table:

i	$A[i]$	$P_i(1)$	$P_i(2)$	$P_i(3)$	$P_i(4)$	$P_i(5)$	$P_i(6)$
1	1	1	∞	∞	∞	∞	∞
2	6	1	6	∞	∞	∞	∞
3	3	1	3	∞	∞	∞	∞
4	5	1	3	5	∞	∞	∞
5	8	1	3	5	8	∞	∞
6	4	1	3	4	8	∞	∞

Show that for $2 \leq i \leq n$, $2 \leq j \leq n$, $P_i(j) = A[i]$ if $P_{i-1}(j-1) < A[i] < P_{i-1}(j)$ and otherwise $P_i(j) = P_{i-1}(j)$.

1.2 To prove the claim, we will first express it as a logic proposition.

Let:

$$X := P_{i-1}(j-1) < A[i] \wedge A[i] < P_{i-1}(j)$$

$$Y := P_i(j) = A[i]$$

$$Z := P_i(j) = P_{i-1}(j)$$

The expression we should prove is $(X \rightarrow Y) \wedge (\neg X \rightarrow Z)$

Proof of $(X \rightarrow Y)$

Show table,
easier to understand

- Base Step ($i = 2, j \geq 2$):

$$P_1(j-1) < A[2] \wedge A[2] < P_1(j) \rightarrow P_2(j) = A[2] \xrightarrow{P_1(j)=\infty}$$

$$P_1(j-1) < A[2] \rightarrow P_2(j) = A[2]$$

For $j = 2$ the expression holds because, if the premise is true, $A[2]$ can be added to the end of the increasing subsequence of length 1 in $A[1]$, which contains only $A[1]$. For $j > 2$, the expression holds because the premise is always false (the premise is false because there can be no subsequence of length $j-1$ in $A[1]$, which makes $P_1(j-1) = \infty$).

- Induction Step:

$$P_{i-1}(j-1) < A[i] \wedge A[i] < P_{i-1}(j) \rightarrow P_i(j) = A[i]$$

Given the premise, the conclusion follows since $A[i]$ can be added as the new end value of the increasing subsequence of length $j-1$ in $A[1, \dots, i-1]$ with minimum end value.

Proof of $(\neg X \rightarrow Z)$:

- Base Step $(i = 2, j \geq 2)$:

$$P_1(j-1) \geq A[2] \vee A[2] \geq P_1(j) \rightarrow P_2(j) = P_1(j) \xrightarrow{P_1(j)=\infty}$$

$$P_1(j-1) \geq A[2] \rightarrow P_2(j) = P_1(j)$$

If the premise is true, then $A[1, 2]$ has no increasing subsequence of length 2. Hence $P_2(j) = P_1(j) = \infty$.

- Induction Step:

$$P_{i-1}(j-1) \geq A[i] \vee A[i] \geq P_{i-1}(j) \rightarrow P_i(j) = P_{i-1}(j)$$

Given the premise, $A[i]$ either cannot be appended to the increasing subsequence of length $j-1$ with minimum end value in $A[1, \dots, i-1]$ or is not smaller than the end value of the increasing subsequence of length j with minimum end value in $A[1, \dots, i-1]$. Hence, the conclusion follows in either case.

1.3 We outline the algorithm DP_LA.

```

 $X \leftarrow 1$ 
 $P_1(1) \leftarrow A[1]$ 
for  $j \in \{2, 3, \dots, n\}$  do
    |    $P_1(j) \leftarrow \infty$ 
end
for  $i \in \{2, 3, \dots, n\}$  do
    |    $P_i(1) \leftarrow \min(A[i], P_{i-1}(1))$ 
end
for  $j \in \{2, 3, \dots, n\}$  do
    |   for  $i \in \{2, 3, \dots, n\}$  do
    |       |   if  $P_{i-1}(j-1) < A[i] \wedge A[i] \leq P_{i-1}(j)$  then
    |       |       |    $P_i(j) \leftarrow A[i]$ 
    |       |       |    $X \leftarrow j$ 
    |       |       |   else
    |       |       |       |    $P_i(j) \leftarrow P_{i-1}(j)$ 
    |       |   end
    |   end
end
return  $X$ 
```

Algorithm 1: DP_LA()

We start by filling out the first row and the first columns of the table. Then we go through all lines in a top down fashion and, for each line, we fill from left to right. Note that this algorithm has time complexity $O(n^2)$.

1.4 We outline the algorithm Fast_LA.

```
 $X \leftarrow 1$   
 $P[1] \leftarrow A[1]$   
for  $j \in \{2, 3, \dots, n\}$  do  
     $P[j] \leftarrow \infty$   
end  
for  $i \in \{2, 3, \dots, n\}$  do  
    if  $P[x] < A[i]$  then  
         $X \leftarrow x + 1$   
         $j \leftarrow x$   
    else  
         $j \leftarrow \text{BinarySearch}(P[1, \dots, X], A[i])$   
         $P[j] \leftarrow A[i]$   
    end  
return  $X$ 
```

Algorithm 2: Fast_LA()

In the algorithm, we only use $O(n)$ memory. This is possible because the $P_i(j)$ table presented in the question has the following property: Each row i is equal to the previous row except for one cell, whose value is updated to $A[i]$. Hence, it is not necessary to keep the whole table, rather only the last row, which is updated in place. This row is always sorted in ascending order, so a simple binary search is enough to find the right spot to place $A[i]$. Since the binary search always searches only in the current largest increasing subsequence, then it has time complexity $\log L(A)$. This sums up to a total complexity of $O(n * \log L(A))$.

Problem 3 (6 points) The graph coloring problem is to color the vertices of a given graph such that no adjacent vertices have the same color. The problem is to minimize the number of colors needed. Give a branch and bound algorithm to solve the problem to optimality. Analyze the running time of your algorithm!

Note that an upper bound for the graph coloring in arbitrary graphs is n . We start our Branch-and-Bound algorithm with this bound in order to bound enumeration branches throughout the algorithm.

We outline the core of our Branch and Bound in the algorithm BB_coloring.

We list below the global variables we use alongside with their initializations:

$$\begin{aligned} num_colors &\leftarrow 0 \\ colored[1, \dots, n] &\leftarrow [Null, \dots, Null] \\ best_sol[1, \dots, n] &\leftarrow [1, \dots, n] \\ best_mark &\leftarrow n \\ colors_used[1, \dots, n] &\leftarrow [0, \dots, 0] \end{aligned}$$

To obtain the chromatic number of a graph $G = (V, E)$, we run the following call to our Branch-and-Bound algorithm:

if $remaining = \emptyset$ **then**

if $num_colors < best_mark$ **then**

$best_sol \leftarrow copy(colored)$

$best_mark \leftarrow num_colors$;

else

 Select $u \in remaining$

foreach $i \in \{1, \dots, n\} \setminus \{colored[v], \forall v \in N(u)\}$ **do**

if $color_used[i] = 0$ **then**

if $num_colors \geq best_mark - 1$ **then**

 Continue ;

$num_colors \leftarrow num_colors + 1$

$colors_used[i] \leftarrow colors_used[i] + 1$

$colored[u] \leftarrow i$

 BB_coloring($remaining \setminus \{u\}$)

$colors_used[i] \leftarrow colors_used[i] - 1$

if $color_used[i] = 0$ **then**

$num_colors \leftarrow num_colors - 1$

end

$colored[u] \leftarrow Null$

return

Base case
// leaf of enumeration tree

Bound \rightarrow don't use more colors than previous
// bound enumeration

Branch

Algorithm 3: BB_coloring($remaining$)


We start with $remaining = V$


We now analyze the complexity of our Branch and Bound algorithm. Each time we call the algorithm, it makes at most as many recursive calls as the current value ($num_cols - 1$). Since the maximum depth of a recursion stack is n , the overall complexity is $O(n^n)$. However notice that it should be much faster in practice because of the bounding operation.

Problem 4 (6 points) Given a graph, the longest path problem is to find the longest path (i.e. sequences of edges that touches each vertex at most once). Give an exhaustive search algorithm to solve the problem to optimality. Analyze the running time of your algorithm!

In the algorithm BB_LP, we outline the core of our Exhaustive Search.

```
if  $remaining = \emptyset$  then
    |   if  $curr\_mark > best\_mark$  then
    |       |    $best\_path \leftarrow copy(curr\_path)$ 
    |       |    $best\_mark \leftarrow curr\_mark$  ;
else
    |   foreach  $v \in remaining \cap N(u)$  do
    |       |    $curr\_mark \leftarrow curr\_mark + weight(u, v)$ 
    |       |    $curr\_path \leftarrow curr\_path \cup \{(u, v)\}$ 
    |       |   BB_LPP( $remaining \setminus \{v\}, v$ )
    |       |    $curr\_mark \leftarrow curr\_mark - weight(u, v)$ 
    |       |    $curr\_path \leftarrow curr\_path \setminus \{(u, v)\}$ 
    |   end
return
```

 Base Case
// leaf of enumeration tree

 Branch

Algorithm 4: BB_LPP($remaining, u$)

We list below the global variables we use alongside with their initializations:

$$\begin{aligned} \textit{curr_mark} &\leftarrow 0 \\ \textit{curr_path} &\leftarrow \emptyset \\ \textit{best_path} &\leftarrow \emptyset \\ \textit{best_mark} &\leftarrow 0 \end{aligned}$$

For the main call to the algorithm, let s be an artificial node sharing a 0-weighted edge with each node from V . We explicit the main call below:

$$BB_LPP(V, s)$$

We now analyze the complexity of our Exhaustive Search algorithm. Each time we call the algorithm, it makes at most as many recursive calls as $|N(u)|$. Since the maximum depth of a recursion stack is n , the overall complexity is $O(\Delta^n)$, in which Δ is the highest degree in the graph.