Design and Analysis of Algorithms

Part 4

Dynamic Programming

Giulio Chiribella

Hilary Term 2019

Dynamic programming

Dynamic programming is a very general and very powerful ***optimisation technique***.
The term "dynamic programming" was coined by Bellman in the 1950s.
At that time, "programming" meant "planning, optimising".
"Dynamic programming" meant *"optimisation in a sequence of steps"*.

The paradigm of dynamic programming:

- Define a sequence of subproblems, with the following properties:
  1. the subproblems are ordered from the smallest to the largest
  2. the largest problem is the problem we want to solve
  3. the optimal solution of a subproblem can be constructed from the optimal solutions of smaller subproblems
     (this property is called ***Optimal Substructure***)
- Solve the subproblems from the smallest to the largest.
  When a subproblem is solved, store its solution in an array,
  so that it can be used to solve the next subproblems.

The change-making problem [DPV, Exercise 6.17]

Change-Making Problem

Input: Positive integers $1 = x_1 < x_2 < \cdots < x_n$ and v

Task: Given an unlimited supply of coins of denominations $x_1, \ldots, x_n$, find the minimum number of coins needed to sum up to v.

Key question of dynamic programming: *What are the subproblems?*

For $0 \leq u \leq v$, compute the minimum number of coins needed to make value u, denoted as $C[u]$

For $u = v$, $C[u]$ is exactly the solution of the problem.

Optimal substructure: for $u \geq 1$, one has

$$C[u] = 1 + \min\{\, \textsc{Coins}[u - x_i] : 1 \leq i \leq n \ \wedge \ u \geq x_i \,\}\,.$$

$C[u]$ can be computed from the values of $C[u']$ with $u' < u$.

Pseudocode for the Change-Making Problem

CHANGE-MAKING$(x_1, \ldots, x_n; v)$

Input: Positive integers $1 = x_1 < x_2 < \cdots < x_n$ and v
Output: Minimum number of coins needed to sum up to v

```
1  C[0] = 0
2  for u = 1 to v
3        C[u] = 1 + min{ C[u − x_i] : 1 ≤ i ≤ n ∧ u ≥ x_i }
4  return C[v]
```

Running time analysis

The array COINS$[1 \,..\, v]$ has length v, and each entry takes $O(n)$ time to compute. Hence running time is $O(nv)$.

Knapsack problem [DPV 6.4]

A burglar finds much more loot than he had expected:

- His knapsack holds a total weight of at most W kg ($W \in \mathbb{N}$).
- There are n items to pick from, of weight $w_1, \cdots, w_n \in \mathbb{N}$, and value $v_1, \cdots, v_n \in \mathbb{N}$ respectively.

Problem: find the most valuable combination of items he can fit into his knapsack, without exceeding its capacity.

Example Take $W = 10$ and

Item	Weight (kg)	Value (£)
1	6	30
2	3	14
3	4	16
4	2	9

Two versions of the problem:

1. *Only one of each item available*: items 1 and 3 (total: £46)
2. *Unlimited quantities of each*: 1 of item 1 and 2 of item 4 (total: £48)

Knapsack with repetition

(*Unlimited quantities of each item*)

Key question of dynamic programming: *What are the subproblems?*

Define

$$K[w] \;=\; \text{maximum value achievable with a knapsack of capacity } w$$

Optimal substructure: if the optimal solution to $K[w]$ includes item i, then removing this item leaves an optimal solution to $K[w - w_i]$.

Since we don't know which item i is in the optimal solution,
we must try all possibilities:

$$K[w] \;=\; \max\{\, K[w - w_i] + v_i \;: i \in \{1\,..\,n\}, w_i \leq w \,\}$$

Knapsack with repetition, cont'd

Hence we get the following simple and elegant algorithm:

1 $K[0] = 0$
2 **for** $w = 1$ **to** W
3 $\quad K[w] = \max\{\, K[w - w_i] + v_i \;: i \in \{1, \ldots, n\}, w_i \leq w \,\}.$
4 **return** $K[W]$.

Remark: line 3 calls for an algorithm for computing the maximum of n elements. It implicitly contains a **for** loop, with counter i going from 1 to n.

Running time

- The algorithm fills a one-dimensional table of length $W + 1$.
- Each entry of the table takes $O(n)$ to compute.
- Hence total running time is $O(nW)$.

Knapsack without repetition

Suppose that *only one of each item is available*.

Now, the earlier idea does not work:
knowing the value of $K[w - w_i]$ does not help,
because we don't know whether or not item i has already been used.

To refine our approach, we add a second parameter, $0 \leq j \leq n$.
Subproblems: Compute

$$K[w, j] \;=\; \left\{ \begin{array}{l} \text{maximum value achieveable with a knapsack of} \\ \text{capacity } w\text{, choosing from items } 1, \cdots, j \end{array} \right.$$

Our sequence of subproblems has *two indices*, w and j.
The last element of the sequence is $K[W, n]$, the answer to the problem.

Optimal substructure: If we remove item j we should have an optimal solution for $j - 1$. We only need to find out whether item j is useful or not:

$$K[w, j] \;=\; \max\{\, K[w - w_j, j - 1] + v_j, K[w, j - 1] \,\}$$

Knapsack without repetition, cont'd

Thus we have

```
 1  for j = 0 to n
 2        K[0, j] = 0
 3  for w = 0 to W
 4        K[w, 0] = 0
 5  for j = 1 to n
 6        for w = 1 to W
 7              if w_j > w
 8                    K[w, j] = K[w, j − 1]
 9              else K[w, j] = max{ K[w − w_j, j − 1] + v_j, K[w, j − 1] }
10  return K[W, n].
```

The algorithm fills out a two-dimensional table,
with $W + 1$ rows and $n + 1$ columns.

Each table entry takes constant time, so the running time is $O(nW)$.

Longest increasing subsequences [DPV 6.2]

Let $S = (a_1, \cdots, a_n)$ be a sequence of numbers.
A *subsequence* of S is a sequence of the form $T = (a_{i_1}, a_{i_2}, \cdots, a_{i_k})$
where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.
An *increasing subsequence* is one in which

$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

The Longest Increasing Subsequence (LIS) Problem

Input: A sequence of numbers $S = (a_1, \cdots, a_n)$.

Task: Find a longest *increasing subsequence* of S

Example. The longest increasing subsequences of

$$(5,\ 2,\ 8,\ 6,\ 3,\ 6,\ 1,\ 9,\ 7)$$

are $(2, 3, 6, 9)$ and $(2, 3, 6, 7)$.

Dynamic programming approach

Subproblems: For $1 \leq j \leq n$, find a longest subsequence among the increasing subsequences ending at j .

Optimal substructure: suppose that a longest increasing subsequence contains a_i, namely $T = (a_{i_1}, \ldots, a_i, \ldots, a_j)$.
Then, $(a_{i_1}, \ldots, a_i)$ must be a longest subsequence among the increasing subsequences ending at i.
Here, i could be any index such that $i < j$ and $a_i < a_j$.

Computing the length: define

$$L[j] \;=\; \text{length of longest increasing subsequence ending at } j.$$

To compute $L[j]$, we need the values of $L(i)$ for $i < j$:

$$L[j] \;=\; 1 + \max\{\, L(i) : 1 \leq i < j,\; a_i < a_j \,\}.$$

Dynamic programming approach (cont'd)

Constructing a longest increasing subsequence: how to recover the longest subsequence itself?

- While computing $L[j]$, write down the position $P[j]$ of the *penultimate* entry of the longest increasing subsequence ending at j.
- The optimal subsequence can be reconstructed from these backpointers.

Example: $S = (5,\ 2,\ 8,\ 6,\ 3,\ 6,\ 1,\ 9,\ 7)$

$j = 1$	$T_1 = (5)$	$P[1] = \text{NIL}$
$j = 2$	$T_2 = (2)$	$P[2] = \text{NIL}$
$j = 3$	$T_3 = (\mathbf{5}, 8)$	$P[3] = 1$
$j = 4$	$T_4 = (\mathbf{5}, 6)$	$P[4] = 1$
$j = 5$	$T_5 = (\mathbf{2}, 3)$	$P[5] = 2$
$j = 6$	$T_6 = (2, \mathbf{3}, 6)$	$P[6] = 5$
$j = 7$	$T_7 = (1)$	$P[7] = \text{NIL}$
$j = 8$	$T_8 = (2, 3, \mathbf{6}, 9)$	$P[8] = 6$
$j = 9$	$T_9 = (2, 3, \mathbf{6}, 7)$	$P[9] = 6$

Pseudocode for Longest Increasing Subsequence Problem

LONGEST-INCREASING-SUBSEQUENCE(A)

Input: An integer array A.
Output: An array B containing a longest increasing subsequence of A.

```
1   L[1] = 1;   P[1] = NIL
2   k = 1                          // longest incr. subseq. found so far ends at k
3   for j = 2 to n                 // determines where longest incr. subseq. ends
4       L[j] = 1;   P[j] = NIL
5       for i = 1 to j − 1         // finds longest incr. subseq. ending at j
6           if A[i] < A[j] ∧ L[i] ≥ L[j]
7               L[j] = 1 + L[i];   P[j] = i
8       if L[j] > L[k]
9           k = j
10  Create new array B of length L[k]
11  for j = L[k] downto 1          // writes the longest incr. subseq. into B
12      B[j] = A[k];   k = P[k]
```

Loop invariants

Loop invariant of the for loop at lines 3-9:

(I1) For every $t \leq j-1$, $L[t]$ is the length of a longest increasing subsequence ending in t.

(I2) $P[t]$ is the position of the penultimate entry in such subsequence.

(I3) k is the position of the last entry in a longest increasing subsequence contained in the subarray $A[1\,..\,j-1]$.

Loop invariant of the for loop at lines 5-7:

(I1) $L[j]$ is the length of a longest increasing subsequence of the form $(a_{i_1}, \ldots, a_{i_m}, a_j)$ with $i_m < i$.

(I2) $P[j]$ is the position of the penultimate entry in such subsequence.

Loop invariant of the for loop at lines 11-12:

(I1) $B[j+1\,..\,L[k]]$ is an increasing subsequence of A.

(I2) $A[k] < B[j+1]$.

(I3) $B[j+1] = A[p]$ for some $p > k$.

Exercise: prove initialisation, termination, and maintenance.

Analysis

Runtime

LONGEST-INCREASING-SUBSEQUENCE(A) runs in $O(n^2)$ due to the nested **for** loops in lines 3–9.

Example

Running the algorithm on $A = [5,\ 2,\ 8,\ 6,\ 3,\ 6,\ 1,\ 9,\ 7]$ produces the following values:

i	1	2	3	4	5	6	7	8	9
A	5	2	8	6	3	6	1	9	7
L	1	1	2	2	2	3	1	4	4
P	NIL	NIL	1	1	2	5	NIL	6	6
B	2	3	6	9					

Longest simple paths: where things go wrong [CLRS 15.3]

A path in a graph is said to be ***simple*** if it **does not visit the same vertex twice.**

Longest Simple Path Problem

Input: A weighted directed graph $G = (V, E)$ and two vertices $a, c \in V$.

Task: Find $l[a, c]$:= length of the *longest simple path* from a to c.

Suppose we know that the longest simple path from a to c passes through b. Then, we may be tempted to consider $l[a, b]$ and $l[b, c]$ as *subproblems*.

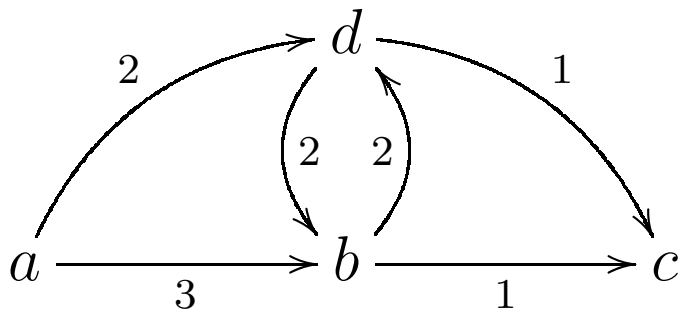
Unfortunately, in general we have

$$l[a, c] \neq l[a, b] + l[b, c].$$

This is because the longest simple path from a to b may have a vertex in common with the longest simple path from b to c. Hence, we cannot use these two paths to construct the longest *simple* path from a to c.

Failure of optimal substructure

Example:



We have $l[a, c] = 6,\ l[a, b] = 4$ and $l[b, c] = 3$

The Longest Simple Path Problem does not have ***optimal substructure***: the solution to the problem is *not* a composite of solutions to the subproblems.
We also say that the solution is not ***compositional***.

Lesson of the example: DP is *not* applicable to every optimization problem. For DP to be applicable, the problem must have *optimal substructure*.

Dynamic Programming vs Divide-and-Conquer

- DP is an *optimization* technique and is applicable only to problems with *optimal substructure*.
  D&C is not normally used to solve optimization problems.
- Both DP and D&C split the problem into parts, find solutions to the parts, and combine them into a solution of the larger problem.
  - In D&C, the subproblems are *significantly smaller* than the original problem (e.g. half of the size, as in MERGE-SORT) and "do not overlap" (i.e. they do not share sub-subproblems).
  - In DP, the subproblems are not significantly smaller and are overlapping.
- In D&C, the dependency of the subproblems can be represented by a tree. In DP, it can be represented by a directed path from the smallest to the largest problem (or, more accurately, by a *directed acyclic graph*, as we will see later in the course).

The edit distance [DPV 6.3]

How does a spell checker determine words that are close by?

Idea: find how many ***edits*** are needed to transform one word into another.

An ***edit*** is an insertion, or deletion, or character substitution.

Example: SNOWY and SUNNY

$$\text{SNOWY} \xrightarrow{\text{ins}} \text{SUNOWY} \xrightarrow{\text{sub}} \text{SUNNWY} \xrightarrow{\text{del}} \text{SUNNY}$$

The ***edit distance*** between two words is the minimum number of edits needed to transform one word into the other.

Alignments

An ***alignment*** of two words is an arrangement of their letters on two lines, with the letters of one word on one line and the letters of the other word on the other line, possibly including blank characters □.

Examples:

□	S	N	O	W	□	Y	5 mismatched columns
S	U	N	□	□	N	Y	

S	□	N	O	W	Y	3 mismatched columns
S	U	N	N	□	Y	

Observation: every alignment identifies a sequence of edits that transforms one word into the other.
The number of edits is equal to the number of mismatched columns.
Hence,
edit distance = minimum number of mismatched columns, with the minimisation running over all possible alignments.

The Edit Distance Problem

Edit Distance Problem

Input: Two strings (i.e. character arrays) $x[1\,..\,m]$ and $y[1\,..\,n]$

Task: Compute the edit distance between them.

What is a good subproblem?

- For $1 \leq i \leq m$ and $1 \leq j \leq n$,
  find the edit distance between the prefixes $x[1\,..\,i]$ and $y[1\,..\,j]$,
  denoted as $E[i, j]$.
- Our task is to compute $E[m, n]$.

Edit Distance Problem

$E[i, j]$:= edit distance between the prefixes $x[1\,..\,i]$ and $y[1\,..\,j]$.

Express $E[i, j]$ in terms of appropriate subproblems

The respective rightmost columns in an alignment of $x[1\,..\,i]$ and $y[1\,..\,j]$ must be one of three cases:

Case 1		*Case 2*		*Case 3*
$x[i]$	or	□	or	$x[i]$
□		$y[j]$		$y[j]$

- *Case 1.* Cost = 1, and it remains to align $x[1\,..\,i-1]$ with $y[1\,..\,j]$.
- *Case 2.* Cost = 1, and it remains to align $x[1\,..\,i]$ with $y[1\,..\,j-1]$.
- *Case 3.* Cost = 1 if $(x[i] \neq y[j])$ and 0 otherwise, and it remains to align $x[1\,..\,i-1]$ with $y[1\,..\,j-1]$.

Writing $\delta(i, j) := 1$ if $(x[i-1] \neq y[j-1])$ and 0 otherwise, we have:

$$E[i, j] = \min\{\, E[i-1, j] + 1,\ E[i, j-1] + 1,\ E[i-1, j-1] + \delta(i, j)\,\}$$

Computing $E[i, j]$

The answers to all subproblems $E[i, j]$, with $0 \leq i \leq m$ and $0 \leq j \leq n$, form a two-dimensional array.

Computing $E[i, j]$ for $0 \leq i \leq m$, $0 \leq j \leq n$

- *Initialization*. Set $E[0, 0] := 0$.
  For $1 \leq i \leq m$, set $E[i, 0] := i$.
  For $1 \leq j \leq n$, set $E[0, j] := j$.
- For $1 \leq i \leq n$ and $1 \leq j \leq n$, we can fill in the array $E[i, j]$ row by row, from top to bottom, moving from left to right across each row, using the relation

$$E[i, j] = \min\{\, E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \delta(i, j) \,\}$$

Running time: $O(mn)$

Exercise (easy) Present the dynamic-programming algorithm to compute $E[m, n]$ in pseudo-code.

Example revisited

Find the edit distance between "SNOWY" and "SUNNY".

Using recurrence

$$E[i,j] = \min\{\, E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\delta(i,j)\,\}$$

we have:

	$E[i,j]$		S	N	O	W	Y
		0	1	2	3	4	5
	0	0	1	2	3	4	5
S	1	1	0	1	2	3	4
U	2	2	1	1	2	3	4
N	3	3	2	1	2	3	4
N	4	4	3	2	2	3	4
Y	5	5	4	3	3	3	3

The answer is $E[5,5] = 3$.

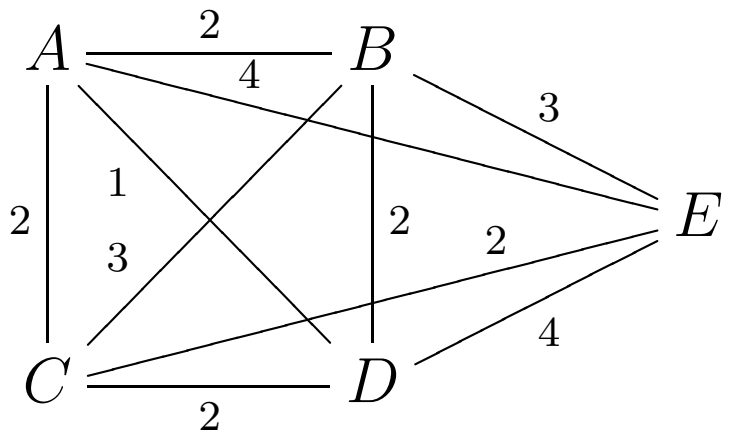
Travelling salesman problem [DPV 6.6]

- Starting from his hometown, a salesman will conduct a journey in which each target city is visited exactly once before he returns home.
- Given the pairwise distance between cities, what is the best order in which to visit them, so as to minimize overall distance travelled?

Model the problem as a complete (undirected) graph with vertex-set $\{1, \cdots, n\}$ *and edge lengths given as a matrix* $D = (d_{ij})$.

Task: Find a tour that starts and ends at 1, includes all other vertices exactly once, and has minimum total length.

Example



What is the optimal tour from A?

TSP is hard to solve [DPV, page 235]

Theorem 1. *The Travelling Salesman Problem is NP-hard.*

Message: TSP is solvable in polynomial time iff $\boldsymbol{P} = \boldsymbol{NP}$, which is unlikely.

An (important) digression A Millennium Prize Problem: *Is* $\boldsymbol{P} = \boldsymbol{NP}$*?*
`www.claymath.org/millennium-problems/p-vs-np-problem`
`www.cs.toronto.edu/~sacook/homepage` *"The P versus NP Problem"*

Brute force method for TSP:

Evaluate every possible route, and return the best one.

Cost: Since there are $(n-1)!$ possible routes and computing the length of each route costs $\Omega(n)$ time, the running time of this strategy is $\Omega(n!)$.

Dynamic programming yields a much faster solution, though not a polynomial one.

Dynamic programming approach

Subproblems: For every subset $S \subseteq \{ 1, \cdots , n \}$ containing 1, and for every element $j \in S , j \neq 1$, find the shortest path that starts from 1, ends in j, and passes only once through all the other nodes in S.
Define $C[S, j]$ to be the length of such path.

- The subproblems form a sequence ordered by the size of S and by the value of j (e.g. in lexicographic order).
- When $|S| = n$, we have the shortest path from 1 to j passing through all the other nodes.
- After all the subproblems with $|S| = n$ are solved,
  the solution of the original TPS problem is obtained by
  1. adding the node 1 in the end of the shortest path from 1 to j, so that it becomes a cycle
  2. finding the shortest cycle.
     Explicitly, the length of the shortest cycle is

$$L_{\min} = \min\{ C[\{ 1, \cdots , n \}, j] + d_{j1} : 1 < j \leq n \} .$$

Dynamic programming approach (cont'd)

Optimal Substructure: for $1 < j \leq n$,

$$C[S, j] \;=\; \min\{\, C[S \setminus \{\, j \,\}, i] + d_{ij} : i \in S \setminus \{1, j\} \,\}$$

(base case: $C[\{\, 1 \,\}, 1] = 0$)

Running time:

- There are 2^n subsets of $\{\, 1, \ldots, n \,\}$
- For each subset, there are at most $O(n)$ values for j.
  Hence, the array $C[S, j]$ contains $O(n\, 2^n)$ entries.
- Computing one entry of the array $C[S, j]$ takes at most time $O(n)$

In total, the running time is $O(n^2\, 2^n)$.
Much better than the $\Omega(n!)$ time of the brute force solution!

cf. Stirling's formula: $n! \;=\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \Theta\left(\frac{1}{n}\right)\right]$

Addendum: back to the Longest Increasing Subsequence

We have seen a dynamic programming algorithm that finds a longest increasing subsequence (LIS) of an array $A[1, \ldots, n]$ in time $O(n^2)$.

Can we do better?

Yes! In the following we will see an ingenious algorithm that achieves running time $O(n \log r)$, where r is the length of the LIS.

The idea

For a fixed $j \in \{1, \ldots, n\}$, let

- r be the ***length of the LIS in*** $A[1, \ldots, j]$
- $K[1, \ldots, r]$ be the array where $K[i]$ is the ***position of the smallest last element of an increasing subsequence of length*** i ***in*** $A[1, \ldots, j]$
- $P[j]$ be the index of the penultimate element in an increasing subsequence ***ending at*** j and ***having maximal length***

Example:
$A[1] = 4\,, A[2] = 8\,, A[3] = 2\,, A[4] = 1\,, A[5] = 9$
For $j = 5$ we have $r = 3\,$, $P[5] = 2$, and $K[1] = 4\,, K[2] = 2\,, K[3] = 5$.

Fact: for $j = n$, the length r, the value $K[r]$, and the array $P[1, \ldots, n]$ are ***enough to reconstruct the LIS***: $K[r]$ is the position of the last element of the LIS, $P[K[r]]$ is the position of the second to last, $P[P[K[r]]]$ is the position of the third to last, and so on.

Computing r, $K[1,\ldots,r]$ and $P[j]$

Observe that, for every j, one has $A[K[1]] < A[K[2]] < \cdots < A[K[r]]$.

Algorithm for computing r, $K[1,\ldots,r]$ and $P[1,\ldots,n]$

- Start from $j = 1$. Set $r = 1$, $K[1] = 1$, $P[1] = \text{NIL}$, and $K[0] = \text{NIL}$
- **for** $j = 2$ **to** n , do the following:
  1. **if** $A[j] > A[K[r]]$,
     update r to $r + 1$,
     set $K[r] = j$ and $P[j] = K[r-1]$
  2. **else**
     use binary search to find the smallest index $i \in \{1,\ldots,r\}$ such that $A[j] \leq A[K[i]]$.
     Update $K[i]$ to j.
     Set $P[j] = K[i-1]$.

Analysis: The **for** loop runs for $O(n)$ times, and each execution takes at most time $O(\log r)$, the time of binary search. Total time: $O(n \log r)$.

Constructing the Longest Increasing Subsequence

Given the length r, the value $K[r]$, and the array $P[1, \ldots, n]$, one can construct the LIS with the following algorithm:

PRINT-LIS(A, P, r, K)

Input: Integer arrays A and P, integer r, and integer array K.
Output: An array B containing a longest increasing subsequence of A.

```
1  k = K[r]
2  for j = r downto 1
3        B[j] = A[k];   k = P[k]
```

Note that printing the LIS takes only $O(r)$ time.
In total, the time required to find the LIS is $O(n \log r)$.

Putting everything together

LONGEST-INCREASING-SUBSEQUENCE-BS(A)

Input: An integer array A.
Output: An array B containing a longest increasing subsequence of A.

```
 1  r = 1;   K[1] = 1;   P[1] = NIL;   K[0] = NIL
 2  for j = 2 to n
 3      i_0 = 1;   i_1 = r + 1
 4      while i_0 < i_1
 5          i_m = ⌊(i_0 + i_1)/2⌋
 6          if A[j] ≤ A[K[i_m]]
 7              i_1 = i_m
 8          else i_0 = i_m + 1
 9      if i_0 > r
10          r = r + 1
11      K[i_0] = j;   P[j] = K[i_0 − 1]
12  Create new array B of length r
13  k = K[r]
14  for j = r downto 1
15      B[j] = A[k];   k = P[k]
```

Correctness (optional material)

Invariant of the for loop at lines 2-11:
(I1) r is the length of the LIS in $A[1\,..\,j-1]$.
(I2) For $i \in \{1, .., r\}$, $A[K[i]]$ is the smallest last value of any increasing subsequence of length i within $A[1..j-1]$
(I3) The sequence $A[K[1]], \ldots, A[K[r]]$ is strictly increasing.
(I4) $P[j]$ is the penultimate entry in the longest increasing subsequence ending at j.

Initialisation. At initialisation, $j = 2$, $r = 1$, $A[K[1]] = A[1]$, the sequence $A[K[1]]$ is trivially increasing, and $P[1] = \text{NIL}$. Hence, (I1), (I2), (I3), and (I4) hold.

Termination. At termination, $j = n + 1$. Then, (I1) guarantees that r is the length of the LIS in $A[1\,..\,n]$, (I2) guarantees that $k = K[r]$ is the index of the last entry of a LIS, and (I4) guarantees that the LIS can be backtracked from k: in reverse order, the indices of the LIS are $k, P[k], P[P[k]], P[P[P[k]]]$, and so on.

Correctness (cont'd)

Maintenance. Assume that (I1), (I2), (I3), and (I4) hold for a given value $1 \leq j \leq n$. We have to show that, after the iteration of the **for** loop, (I1), (I2), (I3), and (I4) hold for $j + 1$.

We take for granted that the **while** loop has the following property:

- if $i_0 \leq r$, then i_0 is the smallest index $i \in \{1, \ldots, r\}$ such that $A[j] \leq A[K[i]]$
- if $i_0 > r$, then $i_0 = r + 1$ and $A[j]$ is larger than every element in $\{A[K[1]], A[K[2]], \ldots, A[K[r]]\}$.

The first item just states that the **while** loop implements a search for the smallest index $i \in \{1, \ldots, n\}$ such that $A[j] \leq A[K[i]]$. The second item specifies what happens when the search finds no such index.

Correctness (cont'd)

Case 1. If $i_0 \leq r$, then $\big(A[K[1]], \ldots, A[K[i_0 - 1]], A[j]\big)$ is a longest increasing subsequence ending in j. It has length $i_0 \leq r$, and its penultimate entry has index $P[j] = K[i_0 - 1]$. Hence, r remains the length of the LIS (I1), and (I4) holds after the execution of line 12. Moreover, (I2) is still satisfied after setting setting $K[i_0] = j$. Finally, since i_0 is the smallest index i such that $A[j] \leq A[K[i]]$, and since the subsequence $A[K[1]], \ldots, A[K[r]]$ was strictly increasing before setting $K[i_0] = j$, (I3) still holds after setting $K[i_0] = j$.

Case 2. If $i_0 > r$, then $A[j] \geq A[K[i]]$ for every $i \in \{1, \ldots, r\}$. Hence, we have found an increasing subsequence of length $r + 1$, ending in j, and having $K[r]$ as its penultimate entry. Then, setting $r = r + 1$ restores (I1), and setting $P[j] = K[r - 1]$ restores (I4). Moreover, $A[K[r]]$ is the smallest last element of an increasing subsequence of length r within $A[1\,..\,j]$, thus restoring (I2). Finally, the sequence $A[K[1]], \ldots, A[K[r-1]], A[K[r]]$ is strictly increasing, and therefore (I3) holds.