Chapter 6

Dynamic programming II - The Recursion Strikes Back

By Sariel Har-Peled, September 14, 2023[①] Version: 0.4

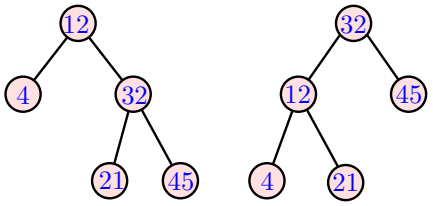
> "No, mademoiselle, I don't capture elephants. I content myself with living among them. I like them. I like looking at them, listening to them, watching them on the horizon. To tell you the truth, I'd give anything to become an elephant myself. That'll convince you that I've nothing against the Germans in particular: they're just men to me, and that's enough."
>
> – – The roots of heaven, Romain Gary.

6.1. Optimal search trees

Given a binary search tree $\mathcal{T}$, the time to search for an element x, that is stored in $\mathcal{T}$, is $O(1 + \text{depth}(\mathcal{T}, x))$, where $\text{depth}(\mathcal{T}, x)$ denotes the depth of x in $\mathcal{T}$ (i.e., this is the length of the path connecting the node containing x with the root of $\mathcal{T}$).

Problem 6.1.1. Given a set of n (sorted) keys $A[1 \ldots n]$, build the best binary search tree for the elements of A.

Note, that we store the values in the internal node of the binary trees. The figure on the right shows two possible search trees for the same set of numbers. Clearly, if we are accessing the number 12 all the time, the tree on the left would be better to use than the tree on the right.



Two possible search trees for the set $A = [4, 12, 21, 32, 45]$.

Usually, we just build a balanced binary tree, and this is good enough. But assume that we have additional information about what is the frequency in which we access the element $A[i]$, for $i = 1, \ldots, n$. Namely, we know that $A[i]$ is going be accessed $\mathsf{f}[i]$ times, for $i = 1, \ldots, n$.

In this case, we know that the total search time for a tree $\mathcal{T}$ is $S(\mathcal{T}) = \sum_{i=1}^{n} (\text{depth}(\mathcal{T}, i) + 1)\mathsf{f}[i]$, where $\text{depth}(\mathcal{T}, i)$ is the depth of the node in $\mathcal{T}$ storing the value $A[i]$. Assume that $A[r]$ is the value stored in the root of the tree $\mathcal{T}$. Clearly, all the values smaller than $A[r]$ are in the subtree $\text{left}_{\mathcal{T}}$, and all values larger than r are in $\text{right}_{\mathcal{T}}$. Thus, the total search time, in this case, is

$$S(\mathcal{T}) = \sum_{i=1}^{r-1} (\text{depth}(\text{left}_{\mathcal{T}}, i) + 1)\mathsf{f}[i] + \overbrace{\sum_{i=1}^{n} \mathsf{f}[i]}^{\text{price of access to root}} + \sum_{i=r+1}^{n} (\text{depth}(\text{right}_{\mathcal{T}}, i) + 1)\mathsf{f}[i].$$

Observe, that if $\mathcal{T}$ is the optimal search tree for the access frequencies $\mathsf{f}[1], \ldots, \mathsf{f}[n]$, then the subtree left_T must be *optimal* for the elements accessing it (i.e., $A[1 \ldots r-1]$ where r is the root).

Thus, the price of $\mathcal{T}$ is

$$S(\mathcal{T}) \quad = \quad S(\mathrm{left}_T) + S(\mathrm{right}_T) + \sum_{i=1}^{n} \mathsf{f}[i],$$

where $S(Q)$ is the price of searching in Q for the frequency of elements stored in Q.

This recursive formula naturally gives rise to a recursive algorithm, which is depicted on the right. The naive implementation requires $O(n^2)$ time (ignoring the recursive call). But in fact, by a more careful implementation, together with the tree $\mathcal{T}$, we can also return the price of searching on this tree with the given frequencies. Thus, this modified algorithm. Thus, the running time for this function takes $O(n)$ time (ignoring recursive calls). The running time of the resulting algorithm is

```
CompBestTreeI (A[i ... j], f[i ... j])
  for r = i ... j do
      T_left  ← CompBestTreeI(A[i ... r − 1], f[i ... r − 1])
      T_right ← CompBestTreeI(A[r + 1 ... j], f[r + 1 ... j])
      T_r ← Tree(T_left, A[r], T_right)
      P_r ← S(T_r)

  return cheapest tree out of T_i, ..., T_j.

CompBestTree (A[1 ... n], f[1 ... n])
      return CompBestTreeI(A[1 ... n], f[1 ... n])
```

$$\alpha(n) = O(n) + \sum_{i=0}^{n-1} (\alpha(i) + \alpha(n-i-1)),$$

and the solution of this recurrence is $O(n3^n)$.

We can, of course, improve the running time using memoization. There are only $O(n^2)$ different recursive calls, and as such, the running time of CompBestTreeMemoize is $O(n^2) \cdot O(n) = O(n^3)$.

Theorem 6.1.2. *One can compute the optimal binary search tree in $O(n^3)$ time using $O(n^2)$ space.*

Proof: We use the recursive function

$$\gamma(i,j) = \begin{cases} 0 & i > j \\ \mathsf{f}[i] & i = j \\ F[j] - F[i-1] + \min_{k=i}^{j} (\gamma(i,k-1) + \gamma(k+1,j)) & i < j, \end{cases}$$

where $F[t] = F[t-1] + \mathsf{f}[t]$ is the prefix sums of the frequencies. Using memoization, or standard dynamic programming, computing the value $\gamma(1,n)$ requires $O(n^2)$ space (i.e., number of distinct recursive calls), and $O(n^3)$ time. ∎

6.1.0.1. An improved algorithm

A further improvement raises from the fact that the root location is "monotone".

Let $R[i,j]$ denote the location of the root for the subproblem involving the subarray $A[i \ldots j]$.

Lemma 6.1.3. *For all $i < j$, we have $R[i, j-1] \le R[i,j] \le R[i, j+1]$.*

Proof: We only sketch the argument, see the paper by Knuth for full details [Knu71].

So consider an optimal binary search tree for $A[1 \,..\, n]$ with $\mathsf{f}[1 \,..\, n]$. It can be extended directly to an optimal search trees for $n+1$ values, where $A[n+1]$ is larger than all previous values, if the frequency $\mathsf{f}[n+1]$

is zero. Indeed, we just search for $A[n+1]$ in the tree, and replace the leaf we arrive to by a node storing the value $A[n+1]$. Clearly, this does not change the price of tree. Thus, the new tree is still optimal. Note that this did not change the location of the root of the tree (so monotonicity holds).

Now, the idea is to start increasing the frequency of $\mathsf{f}[n+1]$, and argue that the root of the optimal search tree either stays in the same place or move to the right. This is proven by induction. Assume for contradiction this is false, and the root moved to the left. But then, consider the path π' in the new tree $\mathcal{T}'$ to $A[n+1]$, for the frequency values of $f'[1 \,..\, n+1]$. Similarly, let $f[1 \,..\, n+1]$ frequencies before the root jumped to the left, and let $\mathcal{T}$ be their optimal search tree, and let π be the path to $A[n+1]$ in this tree.

Let $\pi = v_1 v_2 \ldots, v_t$ and $\pi' = u_1 u_2, \ldots, u_s$ be the two paths, where $v_i, u_i \in [\![n+1]\!]$ denote the index of the value stored in this nodes. As such, $v_t = n+1$, $u_s = n+1$, and $s < t$ (as otherwise, the new tree would not be cheaper as f is the same as f', except $f'[n+1] > f[n]$. By assumption $u_1 < v_1$ (i.e., the root jumped to the left). Observe that u_2 the optimal root for $A[u_1 + 1 \,..\, n+1]$ can not be larger than v_2 (i.e., the optimal root for $A[v_1 + 1 \,..\, n+1]$. That is $u_2 \le v_2$. If $u_2 = v_2$, then the two subtrees on the right of u_2 (and v_2) corresponds to the same range of values of A, and as such are identical. But this implies that $A[n+1]$ has the same depth in the two subtrees. A contradiction. We conclude that $u_2 < v_2$. We can now applies argument inductively, implying that $u_i < v_i$, for all i. This implies that $u_s < v_s$, which is impossible, as $u_s = n+1$. ∎

The dynamic program. This limits the search space, and we can be more efficient in the search. So, let $c(i, j)$ be the cost for the optimal search tree for $A[i \,..\, j]$. Similarly, let $R(i, j)$ denote the location of the root of tree realizing $c(i, j)$. By the above, we have

$$\ell(i, j-1) \le \ell(i, j) \le \ell(i-1, j).$$

We thus get the following recurrence

$$c(i,j) = \begin{cases} 0 & i > j \\ \mathsf{f}[i] & i = j \\ \min_{r=R(i,j-1)}^{R(i+1,j)} \Big[c(i, r-1) + \mathsf{f}[r] + c(r+1, j) \Big] & i < j \end{cases}$$

When converting this into dynamic program, the idea is to fill the associated table by their diagonals – first filling the main diagonal (i.e., $i = j$), then the next diagonal, and so on. This filling order is illustrated in Figure 6.1.

i,j	1	2	3	4	5	6	7	8	9
1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
3			**0**	**1**	**2**	**3**	**4**	**5**	**6**
4				**0**	**1**	**2**	**3**	**4**	**5**
5					**0**	**1**	**2**	**3**	**4**
6						**0**	**1**	**2**	**3**
7							**0**	**1**	**2**
8								**0**	**1**
9									**0**

Figure 6.1: The order of filling the table. The diagonals with lower values are filled first.

To this end, let $C[n \times n]$ be a two dimensional array. Similarly, let $R[n \times n]$ be an associate table with the locations of the root. Let $D(\ell) = \{(i, j) \mid j = i + \ell\}$ be the ℓth diagonal of T. The basic idea is that all the entries of $D(i+1)$ can be computed from the entries of $D(i)$. The code for doing this is depicted in Figure 6.2.

```
FillDiagonal(k):
  for i = 1,...,n − k do
    j = i + k
    α = R[i, j − 1]
    β = R[i + 1, j]
    τ = +∞, ρ = −1
    for r = α,...,β do
      if k = 0 then
          ρ = i, τ = f[i]
      else
          c = C[i, r − 1] + f[i] + C[r + 1, j]
          if c < τ then
            ρ = r, τ = c
    R[i, j] = ρ
    C[i, j] = τ
```

```
OptBST(f[1 .. n]):
  C = empty(n × n), T = empty(n × n)
  for k = 0,...,n do
    FillDiagonal(k)
  return C[1, n]
```

Figure 6.2: Optimal binary search trees. Filling the kth diagonal, an computing the optimal value.

Lemma 6.1.4. *Assume that C and R were filled correctly for the first $k-1$ diagonals. Then computing the values for the kth diagonal can be done in $O(n)$ time.*

Proof: Clearly, the running time of filing the entries of this diagonal is proportional to

$$\sum_{i=1}^{n-k} (R[i+1, i+k] - R[i, i+k-1] + 1) = n - k + R[n-k+1, n] - R[1, k] \leq 2n.$$

Here, we are using that terms cancel out, that the terms are monotonically increasing. ■

Filling each diagonal takes $O(n)$ time, and there are $n+1$ diagonals. We conclude that the overall running time is $O(n^2)$.

Theorem 6.1.5. *Given frequencies $\mathsf{f}[1 \,..\, n]$, one can compute the optimal binary search tree in $O(n^2)$ time using $O(n^2)$ space.*

6.2. Optimal Triangulations

Given a convex polygon P in the plane, we would like to find the triangulation of P of minimum total length. Namely, the total length of the diagonals of the triangulation of P, plus the (length of the) perimeter of P are minimized. See Figure 6.3.

Definition 6.2.1. A set $S \subseteq \mathbb{R}^d$ is ***convex*** if for any to $x, y \in S$, the segment xy is contained in S.

A ***convex polygon*** is a closed cycle of segments, with no vertex pointing inward. Formally, it is a simple closed polygonal curve which encloses a convex set.

A ***diagonal*** is a line segment connecting two vertices of a polygon which are not adjacent. A *triangulation* is a partition of a convex polygon into (interior) disjoint triangles using diagonals.

Observation 6.2.2. *Any triangulation of a convex polygon with n vertices is made out of exactly $n-2$ triangles.*

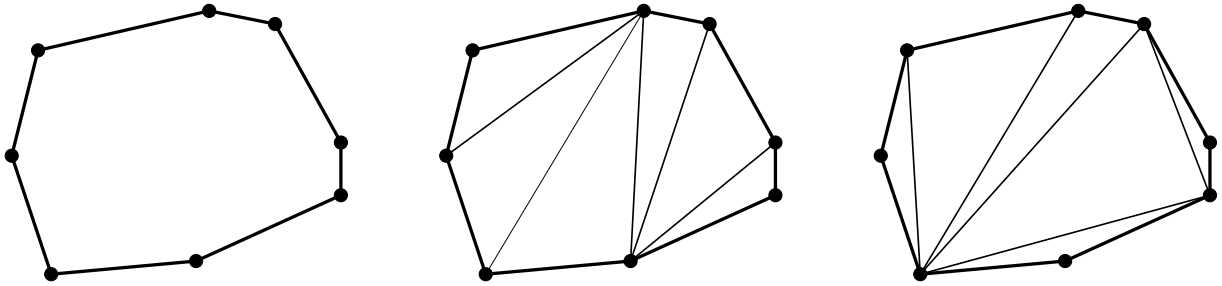
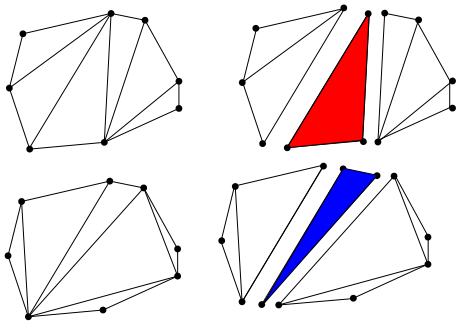
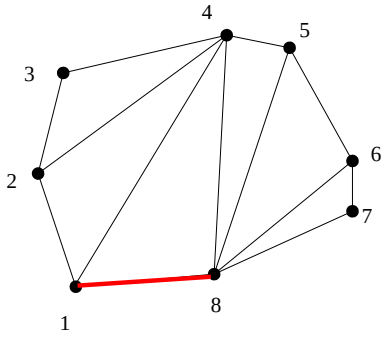


Figure 6.3: A polygon and two possible triangulations of the polygon.

Our purpose is to find the triangulation of P that has the minimum total length. Namely, the total length of diagonals used in the triangulation is minimized. We would like to compute the optimal triangulation using divide and conquer. As the figure on the right demonstrate, there is always a triangle in the triangulation, that breaks the polygon into two polygons. Thus, we can try and guess such a triangle in the optimal triangulation, and recurse on the two polygons such created. The only difficulty, is to do this in such a way that the recursive subproblems can be described in succinct way.





To this end, we assume that the polygon is specified as list of vertices $1 \ldots n$ in a clockwise ordering. Namely, the input is a list of the vertices of the polygon, for every vertex, the two coordinates are specified. The key observation, is that in any triangulation of P, there exist a triangle that uses the edge between vertex 1 and n (red edge in figure on the left).

In particular, removing the triangle using the edge $1 - n$ leaves us with two polygons which their vertices are *consecutive* along the original polygon.

Let $M[i, j]$ denote the price of triangulating a polygon starting at vertex i and ending at vertex j, where every diagonal used contributes its length twice to this quantity, and the perimeter edges contribute their length exactly once. We have the following "natural" recurrence:

$$M[i,j] = \begin{cases} 0 & j \leq i \\ 0 & j = i+1 \\ \min_{i<k<j}(\Delta(i,j,k) + M[i,k] + M[k,j]) & Otherwise \end{cases}.$$

Where $Dist(i,j) = \sqrt{(x[i]-x[j])^2 + (y[i]-y[j])^2}$ and $\Delta(i,j,k) = Dist(i,j) + Dist(j,k) + Dist(i,k)$, where the ith point has coordinates $(x[i], y[i])$, for $i = 1, \ldots, n$. Note, that the quantity we are interested in is $M[1,n]$, since it the triangulation of P with minimum total weight.

Using dynamic programming (or just memoization), we get an algorithm that computes optimal triangulation in $O(n^3)$ time using $O(n^2)$ space.

6.3. Matrix Multiplication

We are given two matrix: (i) A is a matrix with dimensions $p \times q$ (i.e., p rows and q columns) and (ii) B is a matrix of size $q \times r$. The product matrix AB, with dimensions $p \times r$, can be computed in $O(pqr)$ time using the

standard algorithm.

A	1000×2
B	2×1000
C	1000×2

Things becomes considerably more interesting when we have to multiply a chain for matrices. Consider for example the three matrices A, B and C with dimensions as listed on the left. Computing the matrix $ABC = A(BC)$ requires $2 \cdot 1000 \cdot 2 + 1000 \cdot 2 \cdot 2 = 8,000$ operations. On the other hand, computing the same matrix using $(AB)C$ requires $1000 \cdot 2 \cdot 1000 + 1000 \cdot 1000 \cdot 2 = 4,000,000$. Note, that matrix multiplication is associative, and as such $(AB)C = A(BC)$.

Thus, given a chain of matrices that we need to multiply, the exact ordering in which we do the multiplication matters as far to multiply the order is important as far as efficiency.

Problem 6.3.1. The input is n matrices $M_1, \ldots, M_n$ such that M_i is of size $D[i-1] \times D[i]$ (i.e., M_i has $D[i-1]$ rows and $D[i]$ columns), where $D[0 \ldots n]$ is array specifying the sizes. Find the ordering of multiplications to compute $M_1 \cdot M_2 \cdots M_{n-1} \cdot M_n$ most efficiently.

Again, let us define a recurrence for this problem, where $M[i, j]$ is the amount of work involved in computing the product of the matrices $M_i \cdots M_j$. We have

$$M[i,j] = \begin{cases} 0 & j = i \\ D[i-1] \cdot D[i] \cdot D[i+1] & j = i+1 \\ \min_{i \le k < j} (M[i,k] + M[k+1,j] + D[i-1] \cdot D[k] \cdot D[j]) & j > i+1. \end{cases}$$

Again, using memoization (or dynamic programming), one can compute $M[1, n]$, in $O(n^3)$ time, using $O(n^2)$ space.

6.4. Longest increasing subsequence

Given an array of numbers $A[1 \ldots n]$ we are interested in finding the ***longest increasing subsequence*** (***LIS***). For example, if $A = [6, 3, 2, 5, 1, 12]$ the longest increasing subsequence is $2, 5, 12$. To this end, let $M[i]$ denote longest increasing subsequence having $A[i]$ as the last element in the subsequence. The recurrence on the maximum possible length, is

$$M[n] = \begin{cases} 1 & n = 1 \\ 1 + \max\limits_{1 \le k < n, A[k] < A[n]} M[k] & \text{otherwise.} \end{cases}$$

The length of the longest increasing subsequence is $\max_{i=1}^{n} M[i]$. Again, using dynamic programming, we get an algorithm with running time $O(n^2)$ for this problem. It is also not hard to modify the algorithm so that it outputs this sequence (you should figure out the details of this modification).

6.5. Speeding up dynamic programming using data-structure magic

6.5.1. The data-structure: A mildly modified BST

Key and value. A standard approach in storing items in a data-structure is to associate with each data element being stored a key which is used in identifying the item. Thus, an item stored in a data-structure is a pair $e = (k, v)$, where k is the ***key*** used in ordering the items, and v is the ***value***. Neither keys nor values might be

unique. In terms of low-level implementation, in many cases, the key and value might be the same, and might be pointers to somewhere else storing the real information in the key/item. Sometime it would be useful to think about each data-item having additional data stored with each item. Technically, this can be stored together with the value field. To simplify the presentation, we refer to such additional information as ***associated*** data with the item.

What we need. We need a data-structure that supports the following operations. It supports inserting points $p_i = (k_i, v_i)$ in $O(\log n)$ time. Given a query $\hbar$, it reports the point (k, v) currently in the data-structure with maximum v among all the points having $k < \hbar$. The query also takes $O(\log n)$ time (if n items are currenty stored).

Modifying a binary search trees. To implement this data-structure, we are going to use any binary search tree that supports insertion in logarithmic time (say red-black tree, AVL trees, or treaps). We store the items in sorted increasing order by their x-coordinate. These data-structure are binary search trees that have $O(\log n)$ depth at any point in time. Assume for the time being that are just using a binary search tree. For every node v of this tree, let P_v be the subset of points stored in this subtree. The algorithm would maintain for each node α in the tree the quantity

$$p_\alpha = \arg \max_{(k,v) \in P_\alpha} v.$$

For a binary search tree $\mathcal{T}$ of depth d, one can easily maintain this quantity, by computing it for a node by taking the maximum candidate from both children, together with the point stored in the node. Specifically, if the children of α are β and γ, then

$$p_u = \arg \max_{(k,v) \in \{p_\alpha, p_\beta, d_\alpha\}} v,$$

where d_α denotes the original data point stored at α. Thus, if the maximum depth of the tree is d, then insertion can be done in $O(d)$ time, where one needs to recompute this quantity along the insertion path, which can be done in $O(d)$ time.

Given a query value $\hbar$, and a BST $\mathcal{T}$, one can search for the path of $\hbar$ in the BST tree $\mathcal{T}$. All the values stored in $\mathcal{T}$ that are smaller than $\hbar$ are either on the search path π itself, or are stored in subtrees that are left children of nodes on π. This results in a list of size $\leq$ 2depth($\mathcal{T}$) nodes, that the items stored in them. or in their subtrees form all the items with key values smaller than $\hbar$. Thus, scanning the nodes on π, and using P_α for the left children, and direct comparisons on the items stored on the path itself, one can answer a query in $O(\text{depth}(\mathcal{T}))$ time.

Balanced BST. If one uses one of the standard BST that guarantees depth $O(\log n)$, then things get slightly more involved. The key observation is that one needs to recompute the quantity P_α for all the nodes α modified by the insertion (or deletion). However, these data-structures use only rotations to guarantee that the depth remains $O(\log n)$, so one can recompute the values of p_α for every node α that is involved in a rotation in constant time. This readily implies the following.

Lemma 6.5.1. *Any standard BST data-structure that supports insertion/deletion in $O(\log n)$ time, and uses rotations for balancing, can be modified as described above, to support inserting or deleting items (k, v), in $O(\log n)$ time, such that in addition, given a query value $\hbar$, the data-structure can report, in $O(\log n)$ time, the stored item (k, v) with maximum v among all the points having $k < \hbar$.*

6.5.2. Longest increasing subsequence (LIS) in $O(n \log n)$ time

The basic idea is to interpret the task at hand as computing the longest path in a DAG, and then computing this longest path using the above data-structure to avoid computing the graph explicitly.

To this end, the input is an array $A[1 \ldots n]$. The graph $\mathsf{G} = (\llbracket n \rrbracket, \mathsf{E})$ would have an edge $i \to i'$ if $i' < i$ and $A[i'] < A[i]$. Clearly, the longest increasing subsequence is the longest path in G.

The algorithm builds the data-structure $\mathcal{D}$ of Lemma 6.5.1. If a point $(k : A[i], v : t)$ is stored in $\mathcal{D}$ this implies that the longest increasing subsequence ending in $A[i]$ (and using it) has length t. So assume we had computed and stored these items for the first i entries of A, and let D_i denote this data set (which is currently stored in $\mathcal{D}$). The algorithm now handles $A[i+1]$. Clearly, the longest sequence ending in $A[i+1]$, is realized by appending $A[i+1]$ to the longest sequence ending before it, with a value smaller than $A[i+1]$. Specifically, let

$$Q_i = \{(k, v) \in D_i \mid k < A[i+1]\}.$$

Clearly, we are interested in the item (k, v) in Q_i with maximum v (i.e., longest sequence). That is, using a query to $\mathcal{D}$, the algorithm computes the item

$$q_i = \arg \max_{(k,v) \in Q_i} v.$$

If D_i is empty, then $q_i = (0, 0)$. This takes $O(\log n)$ time. The longest sequence ending at $A[i+1]$ then has length $v(q_i) + 1$. Namely, the algorithm inserts the point $(k : A[i+1], v : v(q_i) + 1)$ to the data-structure (with the associated data of this item being $i + 1$). Clearly, this takes $O(\log n)$ time.

The algorithm now repeats this for $i = 1, \ldots, n$. The longest increasing sequence has length that is the maximum value v for any item inserted into the data-structure $\mathcal{D}$ during this process. The algorithm extracts this item in $O(\log n)$ time using a query on the with key value $+\infty$, and report this value. It is easy to verify that by using the associated data attached to the items, one can recover the subsequence realizing this solution.

We thus get the following (maybe surprising) result.

Theorem 6.5.2. *Given a sequence of n real numbers, $\alpha_1, \ldots, \alpha_n$, one can compute the longest increasing subsequence of the input in $O(n \log n)$ time.*

6.5.3. Longest common subsequence in almost linear time in number of matches

In the ***longest common subsequence*** (***LCS***) problem, one is given two sequences $A[1 \,..\, n]$ and $B[1 \,..\, m]$, and wants to find the longest subsequence of numbers that appear in both sequences. A straightforward modification of the dynamic program for edit-distance yields $O(nm)$ time algorithm. Indeed, this is exactly edit-distance, where the price of aligning a character with a different character costs 2, and insertion and deletion both have unit cost. Clearly, minimizing this edit distance is equivalent to maximizing matching columns, which is exactly the value of the LCS.

Here, we consider the more interesting case where there are only "few" matching pairs. Formally, let

$$K = \{(i, j) \mid A[i] = B[j]\},$$

and let $k = |K|$. The set K can be computed in $O(n \log n + m \log m + k)$ time (how?).

One can reduce the problem of LCS to LIS! Indeed, sort the points of K in increasing value of x coordinate. Among points with the same x coordinate, we sort them with decreasing y orientates. Let $p_1, p_2, \ldots, p_k$ be the points of K in this sorted order, and consider the associated sequence

$$S \equiv y(p_1), y(p_2), \ldots, y(p_k).$$

Lemma 6.5.3. *Let ℓ be the length of the longest (strictly) increasing subsequence (LIS) of S. Then ℓ is also the length of the LCS of the original instance.*

Proof: Let $T \equiv y_1 < y_2 < \ldots < y_\ell$ be the LIS of S, and let $Q \equiv q_1, q_2, \ldots, q_t$ be the corresponding points of K. Observe that all the points of K with the same i value are sorted by decreasing j coordinate. Namely, they form a consecutive decreasing subsequence of S, and T can contain only one point of K for such a subsequence. It follows that the points of Q have all increasing i coordinates, and increasing j coordinates. Namely, Q encodes an common-subsequence of the original input of length ℓ.

It is not hard to verify that this argument also works in the other direction. Given an LCS of length ℓ, it can be interpreted as a sequence of points T of length ℓ in the points of Q such that both coordinates are increasing. This in turn leads to an increasing subsequence of length ℓ in S, ■

Theorem 6.5.4. *Computing the LCS of $A[1 \ldots n]$ and $B[1 \ldots m]$ can be done in $O(n \log n + m \log m + k \log k)$ time, where k is the number of matching pairs of values in the two sequences.*

Proof: We yet to describe how to compute K. To this end, for each value $A[i]$ attach the index i as an additional (value) information. Do the same to B. Now, sort copies of both A and B. Now, scanning the sorted arrays, in a merge sort fashion, it is straightforward to extract all the points of K. With some care, this takes $O(n \log n + m \log m + k)$ time.

Now that this preprocessing is done, the above reduction of LCS to LIS takes $O(k \log k)$ time. Computing the LIS takes $O(k \log k)$ time, and translating it back to an LCS for the original instance takes $O(k)$ time. ■

6.6. Pattern Matching

Tidbit 6.6.1. *Magna Carta* or *Magna Charta* - the great charter that King John of England was forced by the English barons to grant at Runnymede, June 15, 1215, traditionally interpreted as guaranteeing certain civil and political liberties. tidbit

Assume you have a string S = "*Magna Carta*" and a pattern P = "?*ag* $*$ *at* $*$ *a* $*$ " where "?" can match a single character, and "*" can match any substring. You would like to decide if the pattern matches the string.

We are interested in solving this problem using dynamic programming. This is not too hard since this is similar to the edit-distance problem that was already covered.

```
IsMatch(S[1...n], P[1...m])
  if m = 0 and n = 0 then return TRUE.
  if m = 0 then return FALSE.
  if n = 0 then
      if P[1...m] is all stars then return TRUE
      else return FALSE
  if (P[m] = '?') then
      return IsMatch(S[1...n-1], P[1...m-1])
  if (P[m] ≠ '*') then
      if P[m] ≠ S[n] then return FALSE
      else return IsMatch(S[1...n-1], P[1...m-1])
  for i = 0 to n do
      if IsMatch(S[1...i], P[1...m-1]) then
          return TRUE
  return FALSE
```

The resulting code is depicted on the left, and as you can see this is pretty tedious. Now, use memoization together with this recursive code, and you get an algorithm with running time $O\left(mn^2\right)$ and space $O(nm)$, where the input string of length n, and m is the length of the pattern.

Being slightly more clever, one can get a faster algorithm with running time $O(nm)$.

BTW, one can do even better. A $O(m + n)$ time is possible but it requires Knuth-Morris-Pratt algorithm, which is a fast string matching algorithm.

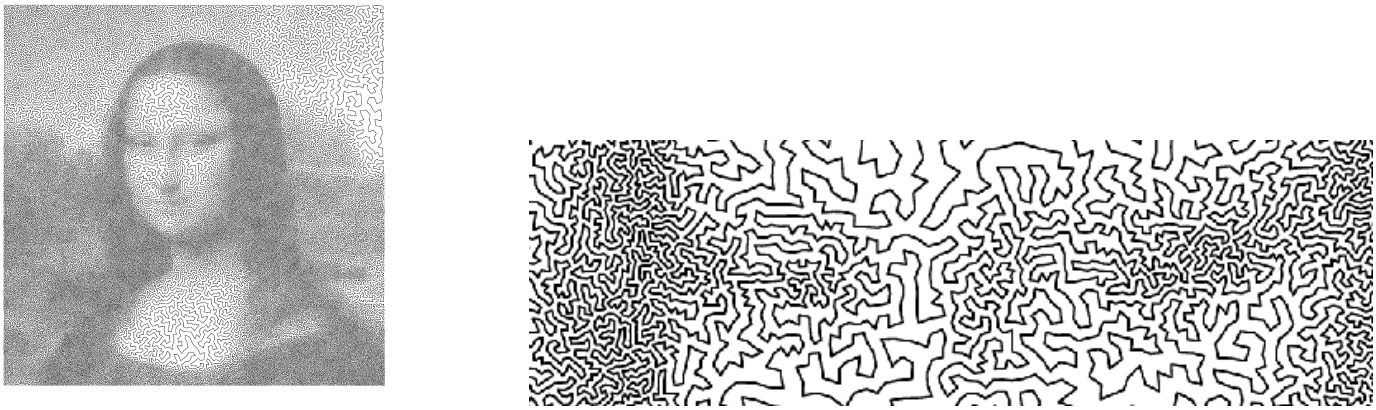


Figure 6.4: A drawing of the Mona Lisa by solving a TSP instance. The figure on the right is the TSP in the eyes region.

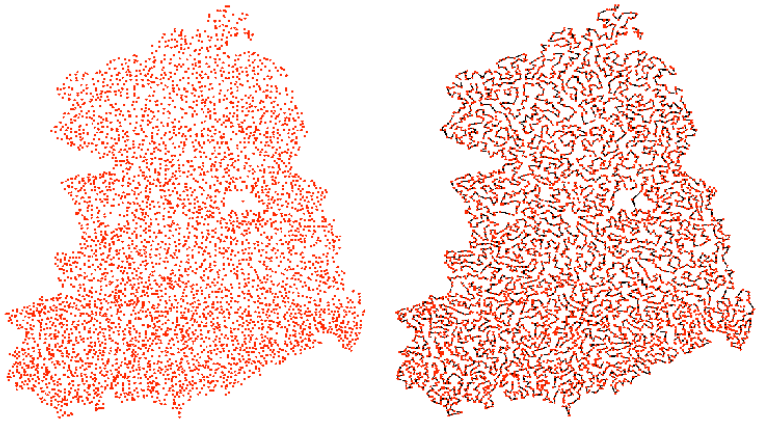


Figure 6.5: A certain country and its optimal TSP tour.

6.7. Slightly faster TSP algorithm via dynamic programming

TSP: Traveling Salesperson Problem

Instance: A graph $G = (V, E)$ with non-negative edge costs/lengths. Cost $c(e)$ for each edge $e \in E$.
Question: Find a tour of minimum cost that visits each node.

No polynomial time algorithm known for TSP– the problem is NP-Hard.

Even an exponential Time algorithm requires some work. Indeed, there are $n!$ potential TSP tours. Clearly, $n! \leq n^n = \exp(n \ln n)$ and $n! \geq (n/2)^{n/2} = \exp((n/2)\ln(n/2))$. Using Stirling's formula, we have $n! \simeq \sqrt{n}(n/e)^n$, which gives us a somewhat tighter estimate $n! = \Theta(2^{cn \log n})$ for some constant $c > 1$.

So, naively, any running time algorithm would have running time (at least) $\Omega(n!)$. Can we do better? Can we get a $\approx 2^{O(n)}$ running time algorithm in this case?

Towards a Recursive Solution.

(A) Order the vertices of V in some arbitrary order: $v_1, v_2, \ldots, v_n$.

(B) opt(S): optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to S. We would like to compute opt(V).

Can we compute opt(S) recursively?

(A) Say $v \in S$. What are the two neighbors of v in optimum tour in S?
(B) If u, w are neighbors of v in an optimum tour of S then removing v gives an optimum *path* from u to w visiting all nodes in $S - \{v\}$.

Path from u to w is not a recursive subproblem! Need to find a more general problem to allow recursion.

We start with a more general problem: TSP Path.

TSP Path

Instance: A graph G = (V, E) with non-negative edge costs/lengths($c(e)$ for edge e) and two nodes s, t.
Question: Find a path from s to t of minimum cost that visits each node exactly once.

We can solve the regular TSP problem using this problem.
We define a recursive problem for the optimum TSP Path problem, as follows:

$$\text{opt}(u, v, S): \quad \text{optimum TSP Path from } u \text{ to } v \text{ in the graph restricted to } S \quad \text{s.t.} \quad u, v \in S.$$

(A) What is the next node in the optimum path from u to v?
(B) Suppose it is w. Then what is opt(u, v, S)?
(C) $\text{opt}(u, v, S) = c(u, w) + \text{opt}(w, v, S - \{u\})$
(D) We do not know w! So try all possibilities for w.

A Recursive Solution.

(A) $\text{opt}(u, v, S) = \min_{w \in S, w \neq u, v}(c(u, w) + \text{opt}(w, v, S - \{u\}))$
(B) What are the subproblems for the original problem opt(s, t, V)? For every subset $S \subseteq$ V, we have the subproblem opt(u, v, S) for $u, v \in S$.

As usual, we need to bound the number subproblems in the recursion:

(A) number of distinct subsets S of V is at most 2^n
(B) number of pairs of nodes in a set S is at most n^2
(C) hence number of subproblems is $O(n^2 2^n)$

Exercise 6.7.1. *Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.*

Lemma 6.7.2. *Given a graph G with n vertices, one can solve TSP in $O(n^3 2^n)$ time.*

The disadvantage of dynamic programming solution is that it uses a lot of memory.

Bibliography

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.