# Chapter 6

# Dynamic programming II - The Recursion Strikes Back

By Sariel Har-Peled, December 30, 2014[①] Version: 0.4

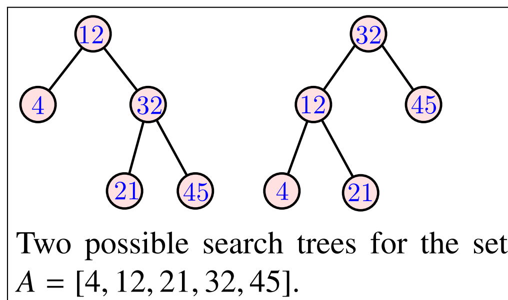
> "No, mademoiselle, I don't capture elephants. I content myself with living among them. I like them. I like looking at them, listening to them, watching them on the horizon. To tell you the truth, I'd give anything to become an elephant myself. That'll convince you that I've nothing against the Germans 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 $\mathsf{T}$, the time to search for an element $x$, that is stored in $\mathsf{T}$, is $O(1 + \mathrm{depth}(\mathsf{T}, x))$, where $\mathrm{depth}(\mathsf{T}, x)$ denotes the depth of $x$ in $\mathsf{T}$ (i.e., this is the length of the path connecting $x$ with the root of $\mathsf{T}$).

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

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

[①]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

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

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

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

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 $\mathsf{T}$, 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.*

A further improvement raises from the fact that the root location is "monotone". Formally, if $R[i, j]$ denotes the location of the element stored in the root for the elements $A[i \ldots j]$ then it holds that $R[i, j-1] \le R[i, j] \le R[i, j+1]$. This limits the search space, and we can be more efficient in the search. This leads to $O(n^2)$ algorithm. Details are in Jeff Erickson class notes.

## 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 6.1.

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.*

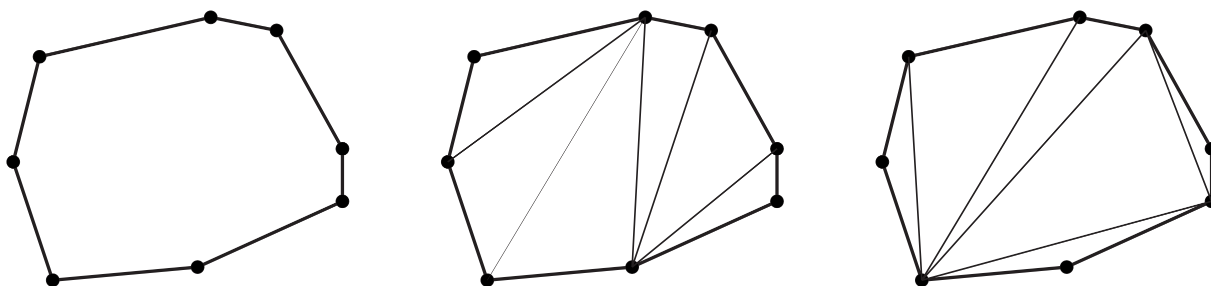
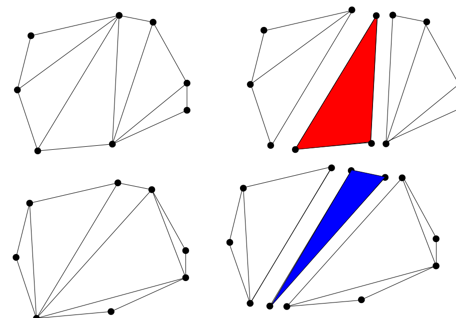
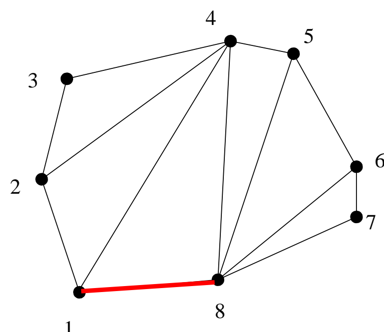


Figure 6.1: A polygon and two possible triangulations of the polygon.



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 $1 \dots n$ 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:

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

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 $i$th point has coordinates $(x[i], y[i])$, for $i = 1, \dots, n$. 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.

| $A$ | $1000 \times 2$ |
|---|---|
| $B$ | $2 \times 1000$ |
| $C$ | $1000 \times 2$ |

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 Ascending Subsequence

Given an array of numbers $A[1 \ldots n]$ we are interested in finding the ***longest ascending subsequence***. For example, if $A = [6, 3, 2, 5, 1, 12]$ the longest ascending 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). A better $O(n \log n)$ solution is possible using some data-structure magic.

## 6.5. Pattern Matching

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.

Tidbit

*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.

```
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 $O(mn^2)$ 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.