◂ Circular logic ✴ garethrees.org ✴ Software archaeology and technical debt ▸

The tabular method

Gareth Rees, 2013-06-11

Dynamic programming is an important technique for writing combinatorial algorithms, in which you divide a problem into sub-problems, like in good ol' *divide-and-conquer*, except that the sub-problems may overlap, so you build up a table of these problems and their solutions in order to avoid repeated work. But the name sucks: the technique is no more or less 'dynamic' than many other techniques, and 'programming' is a fossil: as we'll see below, the 'program' referred to is the *output* of the technique, not the implementation of the technique itself. I can recall being confused by the name when studying algorithms as an undergraduate: I knew that 'linear programming' referred to the optimization of linear functions under linear constraints, but what did the 'dynamic' in 'dynamic programming' refer to? The name was a small but genuine barrier to understanding the technique.

So where did the name come from? It was coined by **Richard Bellman** (famous for the **Bellman–Ford algorithm** for finding shortest paths in a graph). In his autobiography *Eye of the Hurricane* he wrote:

> I spent the Fall quarter (of 1950) at **RAND**. My first task was to find a name for multistage decision processes.
>
> An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named **[Charles Erwin] Wilson**. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."[1]

So it's not surprising that the name sucks: it was deliberately chosen to be obfuscatory! Sixty years of confusion is long enough, so I propose the name **tabular method** or **tabular technique** since the key to the technique is the building up of a table of solutions to sub-problems.[2]

✴

Let's have a very simple example: Fibonacci numbers. These are defined by the recurrence relation[3] $F(n) = n$ if $n \leq 1$ or

$$F(n) = F(n-1) + F(n-2)$$

otherwise. If we program this in the naïve way:

```
def F(n):
    """Return the nth Fibonacci number."""
    if n <= 1: return n
    else: return F(n - 1) + F(n - 2)
```

then it runs very slowly for even moderately large n:

```
>>> from timeit import timeit
>>> [timeit(lambda:F(i), number=1) for i in range(20, 40, 5)]
[0.007075786590576172, 0.06089305877685547, 0.63248610496521, 7.062050104141235]
```

Tracing the execution of a small example shows what's going on:

```
def trace(frame, event, arg):
    if event == 'call': print(frame.f_code.co_name, frame.f_locals)
    return trace

>>> import sys
>>> sys.settrace(trace)
>>> F(4)
F {'n': 4}
F {'n': 3}
F {'n': 2}
F {'n': 1}
F {'n': 0}
F {'n': 1}
F {'n': 2}
F {'n': 1}
F {'n': 0}
3
```

Some of the sub-problems are being solved multiple times: $F(2)$ and $F(0)$ are calculated twice, and $F(1)$ three times. But the **tabular method** comes to the rescue: by storing results in a table, no result needs to be calculated more than once. Here's a really simple implementation:

```
def F(n, table={0: 0, 1: 1}):
    """Return the nth Fibonacci number."""
    try:
        return table[n]
```

```
        except KeyError:
            return table.setdefault(n, F(n - 1) + F(n - 2))
```

And this is many orders of magnitude faster:

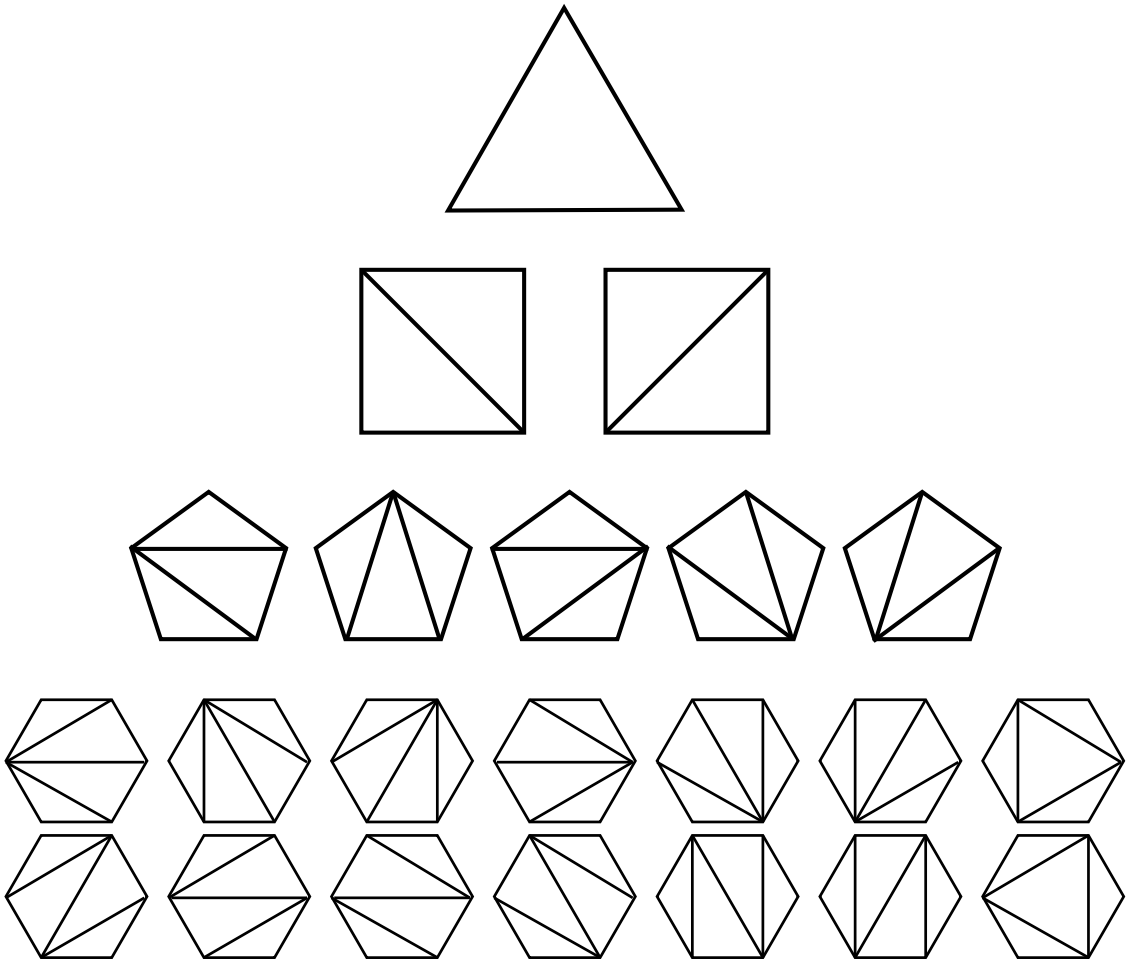
```
>>> timeit(lambda:F(1000), number=1)
0.009588003158569336
```

This implementation is unsatisfactory, however, because the computation of the result is mixed in with the logic for building up and querying the table. It would be clearer if we could separate these parts of the program, and also more general, because the logic for building up and querying the table of results is going to be the same in many cases. In Python it's straightforward to turn this approach into a **decorator** so that you can apply it to any function with a single hashable argument:[4]

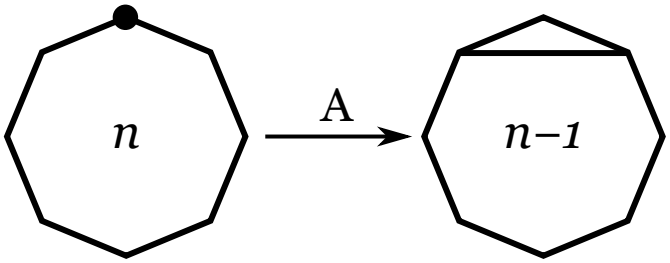
```
from functools import wraps

def memoized(table=None):
    """Return a memoizer for functions with a single (hashable) argument.
    The optional argument table gives the initial state of the table
    mapping arguments to results.

    """
    if table is None:
        table = dict()
    def memoizer(f):
        @wraps(f)
        def wrapper(arg):
            try:
                return table[arg]
            except KeyError:
                return table.setdefault(arg, f(arg))
        return wrapper
    return memoizer

@memoized({0: 0, 1: 1})
def F(n):
    """Return the nth Fibonacci number."""
    return F(n - 1) + F(n - 2)
```

✴

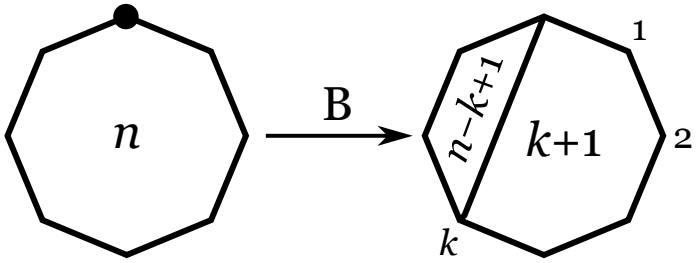
Now for a more interesting example:[5] how many ways are there to triangulate an oriented convex polygon with n sides?[6]



A difficulty that often arises in combinatorial problems like this is the avoidance of double-counting. To show the kind of trap that lurks in these problems, consider the following superficially plausible approach. Pick a distinguished vertex of the polygon (marked with a dot in the diagrams below) and consider what happens to it in the triangulation. Case A: the vertex gets removed as an *ear*, leaving the remainder with $n-1$ sides:



Case B: an edge of the triangulation meets the vertex. Numbering the other vertices $0, 1, 2, \ldots, k, \ldots, n-1$ clockwise from the distinguished vertex, an edge to vertex k divides the polygon into two smaller polygons with $k+1$ and $n-k+1$ sides:



So we have $T(n) = 1$ if $n \leq 3$ or

$$T(n) = T(n-1) + \sum_{2 \le k \le n-2} T(k+1)T(n-k+1)$$

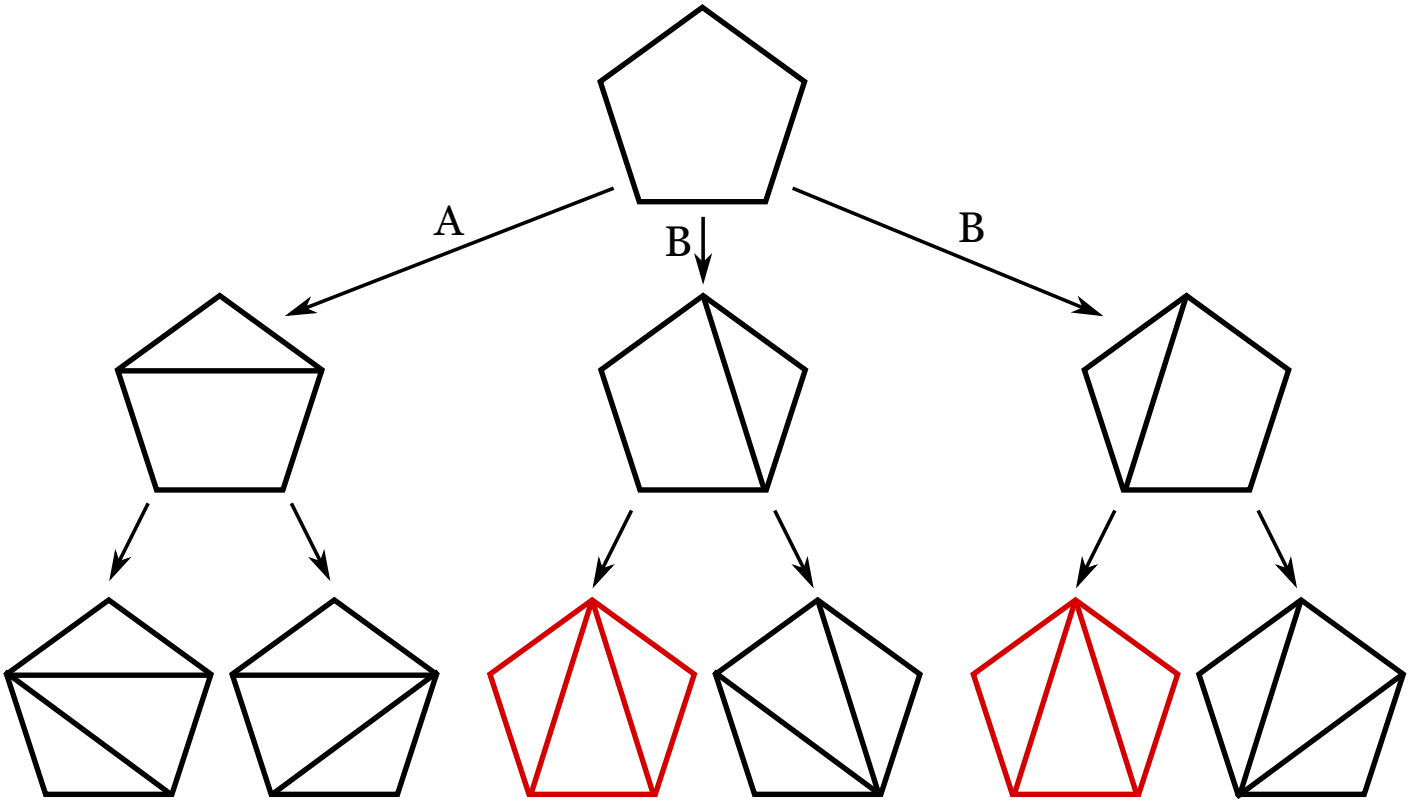
otherwise. In code, that's:

```
@memoized({2: 1, 3: 1})
def T(n):
    """Return number of ways to triangulate a convex polygon with n sides."""
    return T(n - 1) + sum(T(k + 1) * T(n - k + 1) for k in range(2, n - 1))
```

This is correct for $n = 3$ and $n = 4$, but goes wrong thereafter. As shown in **the diagram above**, there are 5 ways to triangulate a pentagon and 14 ways to triangulate a hexagon, but the program above outputs 6 and 22:

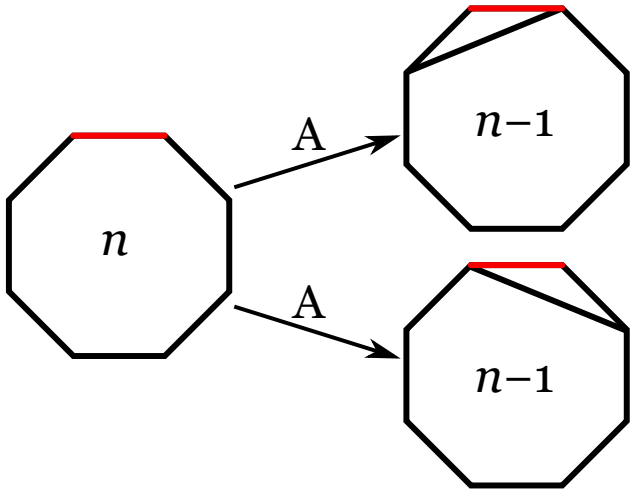
```
>>> [T(i) for i in (3, 4, 5, 6)]
[1, 2, 6, 22]
```

The reason for this error is that there is double-counting in case B. Any triangulation in which two edges meet the distinguished vertex will be counted twice. For example, the triangulations of the pentagon are counted like this:

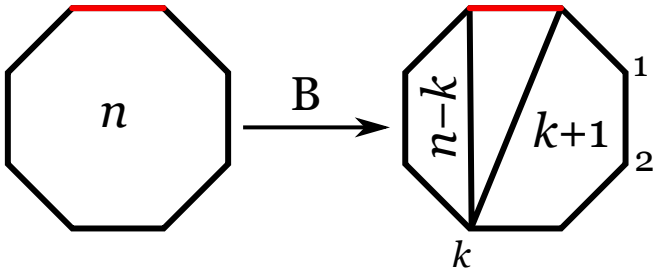


There are a couple of ways to repair the recurrence. In case B we could make sure to examine only the *first* edge to meet the distinguished vertex (the one with the smallest k). Or we could change the recurrence so that it looks at a distinguished *side* of the polygon instead, the idea being that no side of the polygon can participate in more than one triangle of the triangulation, which prevents double-counting.

Let's try the second of these approaches. Pick a distinguished side of the polygon (shown in red in the figure below). Exactly one triangle in the triangulation contains this side. Case A: the triangle is an ear (which can happen in two ways):



Case B: the triangle splits the polygon into two parts with $k+1$ and $n-k$ sides:



We can combine these two cases into one by adopting the convention that in case A the polygon is split into two parts with $n-1$ and 2 sides. This allows the recurrence to be expressed as $T(n) = 1$ if $n \leq 3$ or

$$T(n) = \sum_{1 \leq k \leq n-2} T(k+1)T(n-k)$$

otherwise. (In fact we only need the base case for $n = 2$.) In code that's:

```
@memoized({2: 1})
def T(n):
    """Return number of ways to triangulate a convex polygon with n sides."""
    return sum(T(k + 1) * T(n - k) for k in range(1, n - 1))
```

which gives the right answer for $n = 5$ and beyond:

```
>>> [T(n) for n in range(3, 13)]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
```

1. ↩ Quoted from Stuart Dreyfus, "**Richard Bellman on the birth of dynamic programming**". Note that this account can't be the whole story, because (as Russell and Norvig point out in *Artificial Intelligence: A Modern Approach*) Bellman's paper "**On the theory of dynamic programming**" (*Proc. Natl. Acad. Sci. U.S.A.* 38:716–319) was published in 1952, but Wilson did not become Secretary of Defense until January 1953.

2. ↩ If you have a better name, please suggest it in the comments! But note that terms like 'divide' or 'decomposition' are not ideal in this context as they have the nuance of *disjoint*. The important thing about the tabular method is that it works well when the sub-problems overlap.

3. ↩ There are other recurrence relations for the Fibonacci numbers that lead to more efficient ways to compute them, but I wanted a really simple example. If you actually wanted to compute moderately large Fibonacci numbers, then you could use the recurrence

$$F(2n) = (2F(n-1) + F(n))F(n)$$
$$F(2n+1) = F(n)^2 + F(n+1)^2$$

which leads to the following code:

```
@memoized({0: 0, 1: 1, 2: 1, 3: 2})
def F(n):
    """Return the nth Fibonacci number."""
    if n % 2 == 0:
        a = F(n // 2 - 1)
        b = F(n // 2)
        return (2 * a + b) * b
    else:
        a = F(n // 2)
        b = F(n // 2 + 1)
        return a * a + b * b

>>> timeit(lambda:F(1000), number=1)
0.00013494491577148438
```

4. ↩ See the **Python decorator library** for a **more general memoization decorator**.

5. ↩ For an even more complicated example, see "**Square triangulations**".

6. ↩ It's well known that this question is answered by the **Catalan numbers** (sequence **A000108**): the number of ways to triangulate an oriented convex polygon with $n+2$ sides is

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

So the example is not truly realistic, but it's hard to come up with an example that's simple enough not to get bogged down in detail (as would happen if I picked something like **Levenshtein distance**).

Comments

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Comment as a Guest, or login: facebook

Name

Email

Website (optional)

Displayed next to your comments.

Not displayed publicly.

If you have a website, link to it here.

Subscribe to None

Submit Comment