Lecture 7 Dynamic Programming

Print to PDF

Contents

Learning objectives

- Identify cases of repeated or redundant computation and address the issues with caching or memoization.
- Describe the basic idea of dynamic programming.
- Identify problems when dynamic programming is applicable.
- Implement simple dynamic programming algorithms.

```
import functools
import numpy as np
import joblib
import sklearn.metrics
from IPython.display import HTML, display
```

1. Caching & memoization

Consider the Fibonacci sequence is defined as follows:

- $F_0 = 0$
- $F_1 = 1.$
- For $n \geq 2$, the sequence is generated by the relation: $F_n = F_{n-1} + F_{n-2}$

This means each number in the sequence is the sum of the two preceding numbers.

We can compute the Fibonacci sequence iteratively and recursively:

```
def fib(n):
    f = np.zeros(n+1, dtype=int)
    f[1] = 1

    for i in range(2,n+1):
        f[i] = f[i-1] + f[i-2]

    return f[-1]

fib(7)
```

```
13
```

```
def fib_recursive(n):
    if n == 0 or n == 1:
        return n

    return fib_recursive(n-1) + fib_recursive(n-2)

fib_recursive(7)
```

```
13
```

The recursive function looks much cleaner. However, computation using the recurrence relation can lead to exponential time complexity due to repeated calculations.

```
%timeit -n1 -r1 fib(35)
```

```
35.3 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%timeit -n1 -r1 fib_recursive(35)
```

```
1e+03 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

The problem is that there is a significant amount of repeated computation in our calculations. For example, `fib(33)` is called by `fib(34)` and `fib(35)`, and it gets much worse deeper down the tree. In fact, `fib(1)` may be called millions of times.

This problem is not unique to Fibonacci numbers. Similar cases can be found in other examples:

- In Lab 3, the function `largest_distance` had repeated computations of distances between vertices.
- In Lab 4, recursive seam carving: this approach also suffers from the same inefficiency due to redundant calculations.

Solutions to redundant computation

To solve the problem of redundant computation, we can use two key concepts: **caching** and **memoization**.

Caching is a general term that refers to the practice of storing data for later use. On the other hand, memoization is a specific form of caching that involves storing the results of function calls so that when the same inputs are encountered again, the previously computed result can be reused. This avoids the need to recompute values that have already been calculated. In Python's context, the term "caching" is frequently used interchangeably with "memoization", which can be confusing.

Python has built-in decorators like `functools.lru_cache`, which implement memoization but use the term "cache." Note that we have introduced Python decorators in DSCI 511.

In the code below, we use the `@functools.lru_cache(maxsize=None)` decorator to implement caching in the Fibonacci function.

```python
@functools.lru_cache(maxsize=None)
def fib_recursive_cache(n):
    if n == 0 or n == 1:
        return n

    return fib_recursive_cache(n-1) + fib_recursive_cache(n-2)
```

```python
%timeit -n1 -r1 fib_recursive_cache(35)
```

```
8.13 μs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

This decorator allows the function to “remember” the results of previous function calls. If `fib_recursive_cache(30)` has already been computed, the result (832040) is saved and used if `fib_recursive_cache(30`) is ever called again. This means that each Fibonacci number is computed only once, significantly reducing the number of recursive calls and speeding up the process.

While this approach speeds up computation by avoiding redundant calculations, it comes at the cost of increased memory usage, since we need to store results for all function calls. This trade-off—using more memory to achieve faster running time—is a common and useful concept in optimization.

Another approach to caching involves storing the cache outside of memory.

`functools.lru_cache` stores the cached results in memory. We can use the `joblib` library to store the cache in a file. This file-backed caching mechanism is referred to as `Memory` in `joblib`, which can be a bit confusing, as the name might suggest it is still stored in memory. Despite the terminology, this method is useful when you want to cache results but avoid using too much RAM, especially for larger datasets or longer-running computations.

```python
memory = joblib.Memory("/tmp", verbose=0)
```

```python
@memory.cache
def fib_recursive_cache2(n):
    if n == 0 or n == 1:
        return n

    return fib_recursive_cache2(n-1) + fib_recursive_cache2(n-2)
```

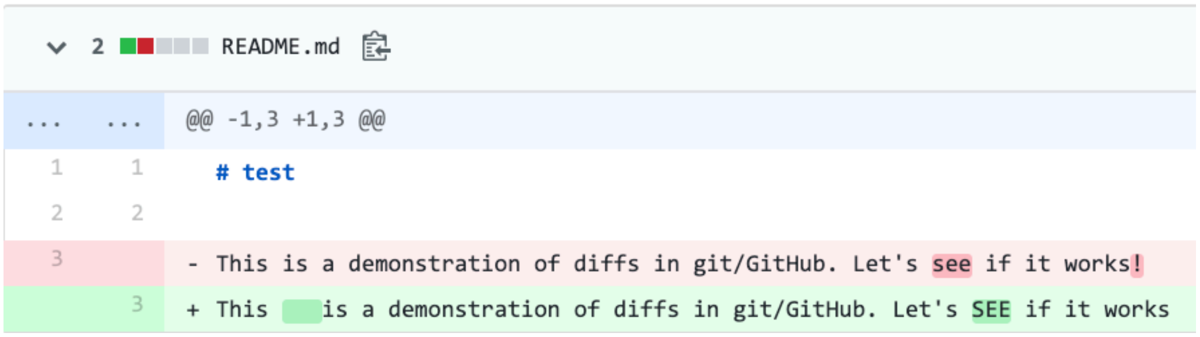
```python
%timeit -n1 -r1 fib_recursive_cache2(35)
```

```
4.68 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

2. Dynamic programming

Attribution: some inspiration for this part of the lecture taken from Sedgewick and Wayne's COS 126 Global Sequence Alignment assignment.

When you look at a commit in GitHub, you just see the "diff" or difference between the two versions.

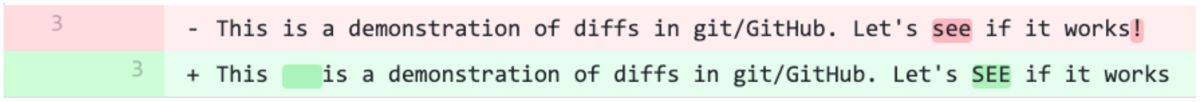


```
2 README.md
@@ -1,3 +1,3 @@
1 1   # test
2 2
3   - This is a demonstration of diffs in git/GitHub. Let's see if it works!
  3 + This   is a demonstration of diffs in git/GitHub. Let's SEE if it works
```

In the highlighted areas, GitHub recognizes changes such as added whitespace, the modification of "see" to "SEE," and the removal of an exclamation mark. But how does GitHub compute this diff? The goal is to highlight the characters that are different between two versions of the file.

This problem of finding the difference between two versions is actually an optimization problem. Technically, you could always describe the diff as "delete the entire old file and add the entire new file," but that wouldn't be a useful feature. Instead, what GitHub (or git) is doing is implicitly aligning the two sequences—finding a way to map one version of the file to the other. The objective is to find the alignment that requires the **minimal number of changes**, meaning we want to minimize the number of characters highlighted in the diff.

For instance, if the sequence x has 4 highlighted characters (in red) and sequence y has 6 highlighted characters (in green), the total number of highlighted characters is 10. The fewer characters we highlight, the better the diff, and now we've framed this as an optimization problem.



```
3   - This is a demonstration of diffs in git/GitHub. Let's see if it works!
  3 + This   is a demonstration of diffs in git/GitHub. Let's SEE if it works
```

We're really aligning the two strings

When we compare two strings and encounter a mismatch, how do we know if we should highlight the mismatch in red (for the original string)? We face a binary decision every time we hit a mismatch.

This decision isn't something we can make one character at a time because each choice affects the overall alignment. The problem is more complex because highlighting a single mismatch can shift how subsequent characters align. As a result, we need to consider the entire string rather than making local decisions. This means we're aiming for an alignment that is optimal across all possible alignments of the two strings.

To find this optimal alignment, one approach is brute force: we could recursively check all possible alignments of the strings and select the one that minimizes the number of changes. We can implement the recursive function below.

(Optional note: This problem can be reformulated as the longest common subsequence problem, or it can be viewed as a form of edit distance.)

```
def num_diffs_recursive(x, y):
    """
    Find the number of characters in the diff between x and y.

    Parameters
    ----------
    x : str
        The first string
    y : str
        The second string

    Returns
    -------
    int
        The number of highlights

    Examples
    --------
    >>> num_diffs_rec("This is a!", "this  is a")
    4
    >>> num_diffs_rec("xxHello", "Hellox")
    3
    """
    if len(x) == 0:
        return len(y) # Highlight the rest of y
    if len(y) == 0:
        return len(x) # Highlight the rest of x

    if x[0] == y[0]:  # A match
        return num_diffs_recursive(x[1:], y[1:])
    else:
        return 1 + min( num_diffs_recursive(x[1:], y), num_diffs_recursive(x, y
num_diffs_recursive("This is a!", "this  is a")
```

```
4
```

```
num_diffs_recursive("xxHello", "Hellox")
```

```
3
```

Note: the code that actually returns the highlighted characters is slightly more complicated – we'll defer that until later.

How does this recursive function work?

Back to the example:

```
This is a demonstration of diffs in git/GitHub. Let's see if it works!
This    is a demonstration of diffs in git/GitHub. Let's SEE if it works
```

The first mismatch between the two strings is at the character `i` in the first string and ` ` in the second string. Our goal is to find the alignment that results in the minimum number of highlights. To achieve this, we need to compare two options:

- Highlight `i` and add 1 to the score for highlighting the `i`.
- Highlight ` ` and add 1 to the score for highlighting the ` `.

We evaluate both possibilities and choose the one with the lower score. This process is done recursively for the rest of the string, considering every mismatch and updating the score accordingly.

The base case for this recursive approach occurs when we reach the end of one of the strings. In that case, we must highlight the remaining characters in the other string, as no alignment is possible anymore.

How slow is this code?

- As usual, the brute force solution is unusably slow.

```
%timeit -n1 -r1 num_diffs_recursive("This'll be slow", "Yeah, right...")
```

```
10.2 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

This implementation is pretty much useless!

This is the same issue as we saw earlier with the Fibonacci numbers: repeated computations for subproblems. Each recursive call is recomputing solutions for overlapping subproblems, which leads to inefficiency. If you were to draw out the recursion tree, you'd see just how many subproblems are being recomputed multiple times—this is why it's slow!

One solution could be to use the memoization trick we discussed earlier, which would store results and prevent repeated work. However, there are some challenges with this approach:

- We don't have a clear idea of how much memory the memoization will take — will it be too much for the system?
- We also don't know how long the memoized code will take to run.
- While memoization can work, it can feel like a heavy-handed approach and we cannot clearly see what the code is doing.

For these reasons, a better solution would be to directly write efficient code for this problem, avoiding the need for memoization altogether.

Dynamic programming

Dynamic programming is an algorithm for solving certain optimization problems. While it only applies to specific types of problems, it is extremely fast when it works. This concept is somewhat similar to linear programming (we will discuss it in the next lecture), but dynamic programming tackles a different class of problems. It's also closely related to memoization. See here for a discussion of memoization vs. dynamic programming.

Dynamic programming has a wide range of applications, including:

- DNA sequence alignment
- Text hyphenation
- Running certain machine learning models (e.g. hidden Markov models)
- Finding the differences between two files (like we're discussing now!)
- Image resizing (this week's lab!)
- and many more…

Historically, the word "programming" was often used to describe optimization problems.

We can implement the function using dynamic programming:

```python
def num_diffs(x, y, return_table=False):
    """
    Compute the number of highlighted characters in the
    diff between x and y using dynamic programming.

    Parameters
    ----------
    x : str
        The first string
    y : str
        The second string

    Returns
    -------
    numpy.ndarray
        The dynamic programming table.
        The last element is the result.

    Examples
    --------
    >>> num_diffs("This is a!", "this  is a")[-1,-1]
    4
    >>> num_diffs("xxHello", "Hellox")[-1,-1]
    3
    """
    M = len(x)
    N = len(y)

    opt = np.zeros((M+1, N+1), dtype=int)
    opt[:,0] = np.arange(M+1)
    opt[0,:] = np.arange(N+1)

    for i in range(1,M+1):
        for j in range(1,N+1):
            if x[i-1] == y[j-1]:
                opt[i,j] = opt[i-1, j-1]
            else:
                opt[i,j] = 1 + min( opt[i-1,j], opt[i,j-1] )

    return opt if return_table else opt[-1,-1]
```

```python
num_diffs("This is a!", "this  is a")
```

```
4
```

```python
num_diffs("xxHello", "Hellox")
```

```
3
```

How does this function work?

The recursive implementation works in a “top-down” manner. It starts by solving the big problem — comparing the entire two strings — and then breaks it down into smaller subproblems.

In contrast, dynamic programming works in a “bottom-up” manner. Instead of starting with the full problem, it begins with the smallest subproblems and gradually builds up to the bigger ones. This avoids recomputing the same subproblems multiple times. Essentially, dynamic programming is the recursive solution combined with memoization, but implemented deliberately and efficiently.

We defined a 2D array which we called `opt`. The entry `opt[i,j]` will store the result of `num_diffs(x[:i], y[:j])`, which is the solution for the (i, j) subproblem. That is, each cell `opt[i][j]` in this table represents the minimum edit distance (or number of character differences) between the first i characters of string x and the first j characters of string y.

Initialization:

The first row and column of the opt table are filled to represent the costs of transforming one string into an empty string:

- `opt[i][0]` indicates the cost of deleting i characters from x to match an empty string.
- `opt[0][j]` indicates the cost of inserting j characters into x to match y.

Filling the Table:

- If `x[i]` equals `y[j]`, then `opt[i,j]=opt[i-1,j-1]`. This is because there is no additional highlight needed when the characters match, so we carry forward the previous result.
- If they are not equal, then there are two possibilities: highlight in x or highlight in y. We choose the better option (i.e., the one with the fewer number of highlights). We already have the results for these subproblems, so we can make this decision efficiently.

By following this process, we build up the solution iteratively, rather than repeating computations as we would in a top-down recursive approach.

Let's see what `opt` looks like with a concrete example:

```
x, y = "xxHello", "Hellox"

opt = num_diffs(x, y, return_table=True)
opt
```

```
array([[0, 1, 2, 3, 4, 5, 6],
       [1, 2, 3, 4, 5, 6, 5],
       [2, 3, 4, 5, 6, 7, 6],
       [3, 2, 3, 4, 5, 6, 7],
       [4, 3, 2, 3, 4, 5, 6],
       [5, 4, 3, 2, 3, 4, 5],
       [6, 5, 4, 3, 2, 3, 4],
       [7, 6, 5, 4, 3, 2, 3]])
```

Let's recreate this table according to its definition:

```
same = 0*opt
for i in range(len(x)+1):
    for j in range(len(y)+1):
        same[i,j] = num_diffs_recursive(x[:i], y[:j])
same
```

```
array([[0, 1, 2, 3, 4, 5, 6],
       [1, 2, 3, 4, 5, 6, 5],
       [2, 3, 4, 5, 6, 7, 6],
       [3, 2, 3, 4, 5, 6, 7],
       [4, 3, 2, 3, 4, 5, 6],
       [5, 4, 3, 2, 3, 4, 5],
       [6, 5, 4, 3, 2, 3, 4],
       [7, 6, 5, 4, 3, 2, 3]])
```

This illustrates how silly the recursive solution is!! Each element of this table `same` was computed from scratch. But actually we can get each element in $O(1)$ time with the previous elements. That's how the dynamic programming code works.

Computational cost

Now that we understand how dynamic programming works for this problem, we can analyze the computational cost.

- Memory usage: Since we are using a 2D array `opt` with dimensions M x N, where M is the length of string x and N is the length of string y, the memory usage is $O(MN)$.
- Runtime: Similarly, the time complexity is also $O(MN)$, as we need to fill in every entry of the opt array by solving each subproblem exactly once.

Thus, both the memory and runtime scale with the product of the lengths of the two strings, making the algorithm efficient given the problem constraints.

3. Dynamic programming: backtracking

In dynamic programming, once we have computed the solution using the opt table, the next step is to recover the highlights. This process is called **backtracking**.

Backtracking involves tracing back through the `opt` table to figure out the exact sequence of decisions (highlights) that led to the optimal solution.

```
opt
```

```
array([[0, 1, 2, 3, 4, 5, 6],
       [1, 2, 3, 4, 5, 6, 5],
       [2, 3, 4, 5, 6, 7, 6],
       [3, 2, 3, 4, 5, 6, 7],
       [4, 3, 2, 3, 4, 5, 6],
       [5, 4, 3, 2, 3, 4, 5],
       [6, 5, 4, 3, 2, 3, 4],
       [7, 6, 5, 4, 3, 2, 3]])
```

Intuitively, during the backtracking process, we check where we might have come from in the previous step to determine how to reconstruct the highlights. We begin at the bottom-right corner of the `opt` table and **backtrack** to the top-left corner, following the optimal path based on the decisions made during the computation. The function provided below includes some additional HTML visualization code that you can ignore.

```
dark_green  = '<span style="background-color: rgba(0,255,0,0.5)">'
dark_red    = '<span style="background-color: rgba(255,0,0,0.5)">'
light_green = '<span style="background-color: rgba(0,255,0,0.05)">'
light_red   = '<span style="background-color: rgba(255,0,0,0.05)">'

def show_diff(x, y, align=False):
    opt = num_diffs(x, y, return_table=True)

    x_highlight = ''
    y_highlight = ''
    i = len(x)
    j = len(y)
    while i > 0 or j > 0:
        if i > 0 and j > 0 and x[i-1] == y[j-1]:
            x_highlight = x[i-1] + x_highlight
            y_highlight = y[j-1] + y_highlight
            i -= 1
            j -= 1
        elif j > 0 and opt[i, j] == opt[i, j-1] + 1:
            y_highlight = dark_green + y[j-1] + '</span>' + y_highlight
            if align:
                x_highlight = " " + x_highlight
            j -= 1
        else:
            x_highlight = dark_red   + x[i-1] + '</span>' + x_highlight
            if align:
                y_highlight = " " + y_highlight
            i -= 1

    x_highlight = light_red   + x_highlight + "</span>"
    y_highlight = light_green + y_highlight + "</span>"

    display(HTML('<code>' + x_highlight + '</code>' + '<br>' + '<code>' + y_hig
```

```
x, y = "xxHello", "Hellox"
show_diff(x, y)
```

```
xxHello
Hellox
```

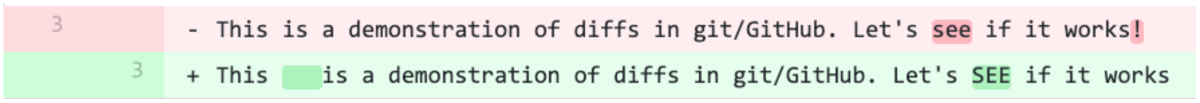
Note: the colours do not render properly on GitHub - you need to run the notebook locally to see the highlighting.

```
before = "This is a demonstration of diffs in git/GitHub. Let's see if it works
after  = "This    is a demonstration of diffs in git/GitHub. Let's SEE if it wo

show_diff(before, after)
```

```
This is a demonstration of diffs in git/GitHub. Let's see if it works!
This is a demonstration of diffs in git/GitHub. Let's SEE if it works
```

Compared to this:



```
3     - This is a demonstration of diffs in git/GitHub. Let's see if it works!
   3  + This  is a demonstration of diffs in git/GitHub. Let's SEE if it works
```

Both highlights match, with a slight difference on highlighting of the spaces. This is because the solution to the optimization problem is not unique - both are optimal. Arbitrary choices made in the code will determine which one is returned by the code.

Why could we use dynamic programaming to find the diffs between documents?

- This comes back to how `opt` is computed. The optimal solution at a given step can be described in terms of optimal solutions of subproblems.

```
x, y = "xxHello", "Hellox"
opt = num_diffs(x, y, return_table=True)
opt
```

```
array([[0, 1, 2, 3, 4, 5, 6],
       [1, 2, 3, 4, 5, 6, 5],
       [2, 3, 4, 5, 6, 7, 6],
       [3, 2, 3, 4, 5, 6, 7],
       [4, 3, 2, 3, 4, 5, 6],
       [5, 4, 3, 2, 3, 4, 5],
       [6, 5, 4, 3, 2, 3, 4],
       [7, 6, 5, 4, 3, 2, 3]])
```

Consider `opt[3,1]`

```
x[:3]
```

```
'xxH'
```

```
y[:1]
```

```
'H'
```

Here, because the last letters match (both `H`), we can just grab the score from the diagonal entry, that is, `opt[3,1] = opt[2,0]`

```
print(opt[3,1], opt[2,0])
```

```
2 2
```

Now consider `opt[5,2]`

```
x[:5]
```

```
'xxHel'
```

```
y[:2]
```

```
'He'
```

We know this is the right alignment:

```
show_diff(x[:5], y[:2])
```

```
xxHel
He
```

But how do we get there?

We need to look at two options: the `l` is highlighted or the `e` is highlighted. Since they don't match, one of them has to be highlighted.

(Note: neither is highlighted in the final result. But for this particular subproblem, one of them has to be! This "line of reasoning" isn't going to end up being used in the final result at all. But we still have to compute it because we don't know in advance what will end up being optimal for the entire problem.)

If we highlight `l` in the first string, then we have the following subproblem:

```
show_diff(x[:4], y[:2])
```

```
xxHe
He
```

with the following cost:

```
opt[4,2]
```

```
2
```

If we highlight `e` in the second string, then we have the following subproblem:

```
show_diff(x[:5], y[:1])
```

```
xxHel
H
```

with the following cost:

```
opt[5,1]
```

```
4
```

Which one should we highlight?

- We should highlight `l` in the first string, which leads to a smaller cost `opt[4,2]+1`

```
opt[5,2]
```

```
3
```