Dynamic Programming 3

In this lecture we explore a dynamic programming solution to finding the longest palindrome substring of a given string. To understand this problem, let's start off with the definition of a palindrome, and a palindromic substring:

- A **Palindrome** is a sequence of characters which reads the same forwards and backwards, such as *racecar*
- A **palindromic substring** is a substring of a string (the characters are adjacent), such that the substring is a palindrome. For example, in the word *bananas*, the substring *anana* is a palindromic substring.

1 The longest palindromic substring

The **longest palindromic substring** problem is the problem of finding the maximum length of all palindromic substrings of a sequence. This substring may not not be unique. For example, in the sequence "ecacbeb" there are two substrings of length three that are palindromes: "cac" and "beb". There are many different versions of this problem that you may come across: some where we try to find *all* maximum-length palindromes, and some where we try to find just one. In this lecture we look at the simpler problem of finding the *length* of the longest palindromic substring.

1.1 The brute-force approach

Given a string s of length n, suppose we simply check *all* possible substrings to see if they are palindromes. This represents a brute-force method for solving the above problem. In order to check all possible substrings, we would need a method to both *enumerate* them all, and a method to *check* each substring.

1. How many possible substrings are contained within a string of length n?

   This is a counting problem of determining the number of substrings of a sequence. We can count these possibilities in a systematic way:

   - Any substring that starts on the first character can finish on any character from a_1 to a_n. Therefore the number of substrings starting at a_1 is n.
   - Any substring that begins on the second character can finish on any character from a_2 to a_n, for a total of $n-1$ possibilities.
   - Continuing in this way, the number of substrings starting on any character a_i is exactly $n-i+1$

   The total number of substrings is thus:

   $$n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$$

   So there are $O(n^2)$ possible substrings of s.

2. How quickly can we check if a substring is a palindrome?

   Any substring of s between index i and j can be verified as a palindrome by looping from the front and back of the substring towards the middle of the substring. At each iteration we check if the characters at either end match up:

This method loops through the substring, and therefore takes time $O(k)$ where $k = j - i$. Here is a quick look at the pseudocode for verifying the substring from $s[i]$ to $s[j]$:

```
for k = i to ⌊(i + j)/2⌋
    if s[k] ≠ s[j + i − k]
        return: not palindrome
return: is palindrome
```

The brute-force algorithm:

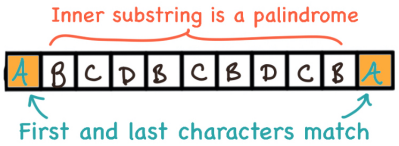
Since there are $O(n^2)$ possible substrings, and it takes time $O(k)$ to verify each substring where $k \leq n$, the entire brute-force approach runs in time $O(n^3)$. With a few summation tricks, we can show this is in fact $\Theta(n^3)$. In the next section we look at a more efficient dynamic programming solution.

2 Dynamic Programming Solution

The key to finding a dynamic programming solution lies in our ability to recursively define the value of an optimal solution. Simply said, we would like to find how the palindrome problem on a large substring relates to the same problem on a smaller substring.

Suppose our string is stored in the array $s[1 \ldots n]$, and consider a particular substring $s[i, \ldots, j]$. By comparing the first and last characters of the substring we have the following two cases:

Case 1: Suppose the first and last characters of the substring are the **same**: $s[i] = s[j]$. Then the string $s[i \ldots j]$ is a palindrome *if and only if* the inner substring is a palindrome:



Therefore if $s[i] = s[j]$ **and** if the inner substring $s[i + 1 \ldots j - 1]$ is a palindrome, then $s[i, \ldots, j]$ is a palindrome.

Case 2: Suppose the first and last characters of the string are **not the same**. Then certainly $s[i \ldots j]$ cannot be a palindrome.

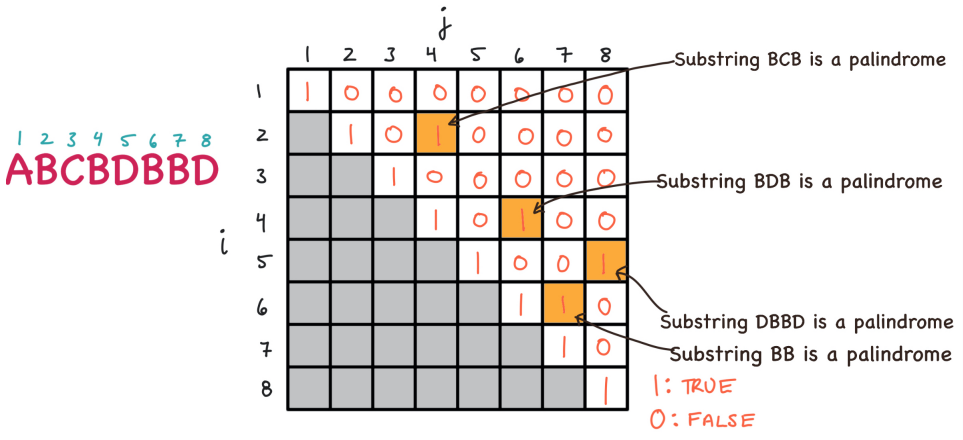
First and last characters do not match.
The string cannot be a palindrome.

The above relationship indicates that we should maintain a table that stores which substring are palindromes so that we can look them up and determine if the larger substrings are also palindromes. Since each substring has both a starting and ending index, we need a double array, $L[i, j]$ to store the possible palindrome. The entries in the array are defined as follows:

The table: $L[1 \ldots n, 1 \ldots n]$:

- Given a string of characters stored in the array: $s[1 \ldots n]$
- $L[i, j]$ is **true** for $i \leq j$ if the substring $s[i \ldots j]$ is a palindrome.
- Since all substrings of length 1 are automatically palindromes, then $L[i, i] =$ **true** for all $1 \leq i \leq n$

Below is an example of the table for the string $ABCBDBBD$. Note that the lower triangular part of the table is not used, since it is impossible to have substring that is indexed backwards.



We next turn to the problem of how to fill up this table. From the above table description, we can see that the values $L[i,i]$ are initialized to true. These represent all the substrings of length one. Subrings of length 2 can also be easily initialized. If any two adjacent characters are the same, then $L[i,i+1] = \text{true}$. The remaining table values can be filled in using the description of Case 1 above.

Recursively defining $L[i,j]$

Given a string contained in the array $s[1 \ldots n]$, we recursively define $L[i,j]$ as follows:

- **Substrings of length 1:** For all $i = 1 \ldots n$

$$L[i,i] = \text{true}$$

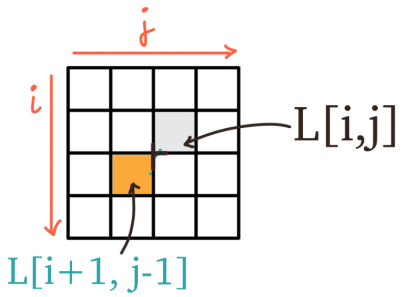
- **Substrings of length 2:** For all $i = 1 \ldots n-1$, if $s[i] = s[i+1]$

$$L[i,i+1] = \text{true}$$

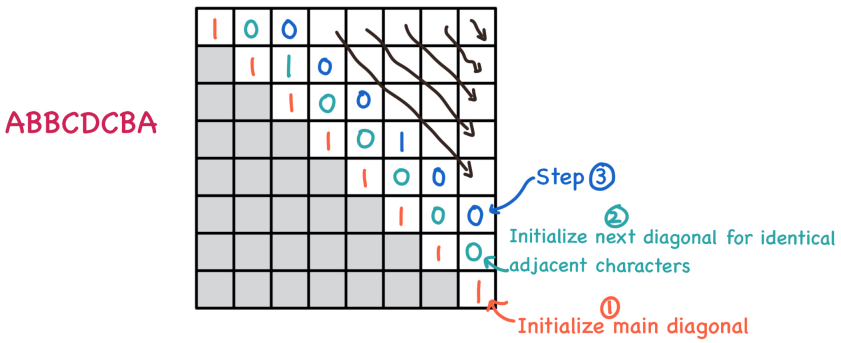
- **All other substrings:** For $1 < j - i$ if $s[i] = s[j]$ and if $L[i+1, j-1] = \text{true}$, then

$$L[i,j] = \text{true}$$

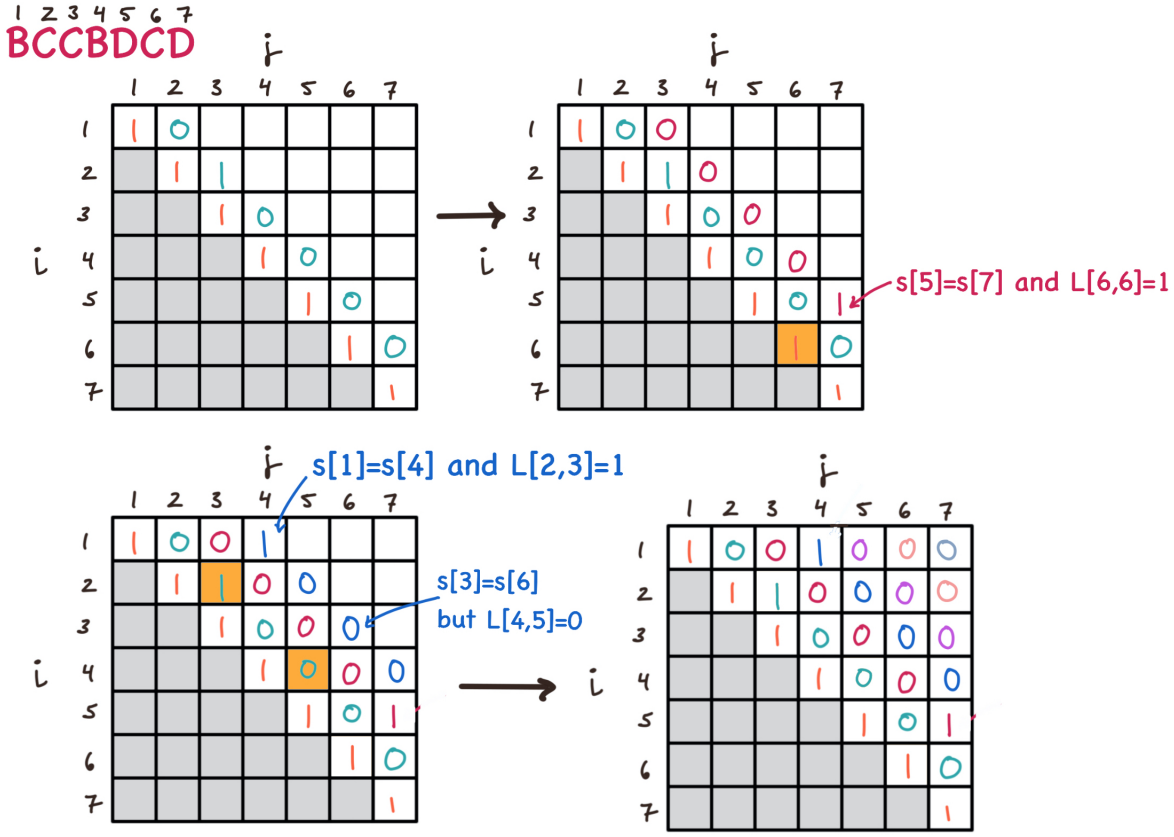
Let's look at how this recursive relationship can be used to fill the table. Notice that in order to determine $L[i,j]$, we require a look-up of $L[i+1, j-1]$. Visually, this is like referencing the element to our lower left diagonal :



If we were to fill in the table row by row, then the cell entries could not be filled because the values that they referenced would not yet be filled. A similar problem arises if we fill in the table column by column. Instead, notice that the table is initialized across the diagonal, and therefore we could use these two initial diagonals to complete the next diagonal entry:



Taking the example $BCCBDCD$, we demonstrate the table-filling process below:



When the table is filled, it contains all information regarding which substrings are palindromes and which are not. But the actual value of maximum length of all palindromes is not stored in the table. Therefore, the only way we can keep track of this information is to update a **max-length** variable each time a new palindrome is found by the dynamic program. The final algorithm for finding the longest palindromic substring in string s is described below:

LPSubstring(s)

Step 1: Initialize the main diagonal of the table. Loop from $i = 1$ to $i = n$ and set

$$L[i, i] = \text{true}$$

Initialize the max-length variable to 1

Step 2: Initialize the second diagonal of the table. Loop through from $i = 1$ to $i = n - 1$

$$\begin{aligned} &\text{If } s[i] = s[i+1] \\ &\qquad L[i, i+1] = \text{true} \\ &\qquad \text{max-length} = 2 \end{aligned}$$

Step 3: Fill in the rest of the upper part of the table, one diagonal at a time.
For each substring from index i to index j of length k, we verify if the first and last characters are the same, and if $L[i+1, j-1] = \text{true}$. The pseudocode is:

for $k = 3$ to $k = n$
  for $i = 1$ to $i = n - k + 1$
    j is the right endpoint: $j = i + k - 1$
    If $s[i] = s[j]$ and if $L[i+1, j-1]$ is true
      $L[i, j] = \text{true}$
      max-length $= k$

When the algorithm is complete, the length of the longest palindromic substring is stored in the variable max-length. In the practice problems, we look at how to update the above steps so that we could also return the specific substring that corresponds to the maximum-length palindromic substring.

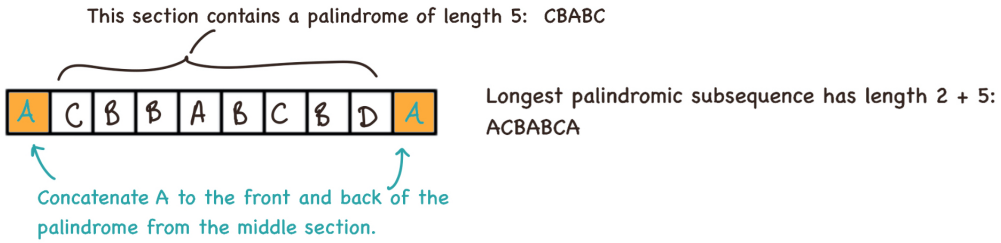
Runtime: This algorithm carries out a constant number of operations per cell of the table. Since there are $\Theta(n^2)$ entries in the table, the algorithm runs in time $\Theta(n^2)$. This is a substantial improvement of the $\Theta(n^3)$ brute-force algorithm.

3 Longest Palindromic Subsequence

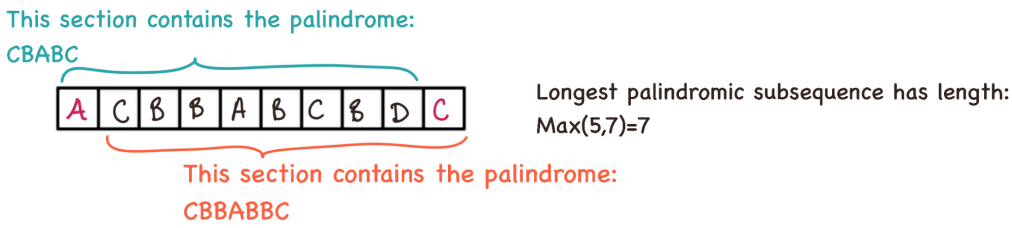
In this section we look at a variation of the above problem. Instead of considering only sub*strings* of the original sequence, we consider *subsequences*. The problem is to determine the length of the **longest possible palindromic subsequence**. For example, the string $s =$ "ABBCDABBC" contains the subsequence "BBABB" which is a palindrome of length five. Note that "BBABB" is not a substring of s. A subsequence of a string may skip characters, as long as the order is maintained, whereas the substring from the previous section could not.

The dynamic programming solution to this problem requires a small update to the the recursive definition seen the previous section. Suppose we consider the substring $s[i, \ldots, j]$ and we would like to determine the length of the longest palindromic subsequence within this substring. By comparing the first and last characters of the substring, we have the following two cases:

Case 1: Suppose $s[i] = s[j]$. These two characters extend the longest palindromic subsequence of $s[i+1 \ldots j-1]$ by two:



Case 2: Suppose $s[i] \neq s[j]$. Then the longest palindromic subsequence cannot include both $s[i]$ and $s[j]$. Therefore, we take the maximum of the two options: the longest palindromic subsequence of the substring $s[i+1 \ldots j]$ and that of $s[i \ldots j-1]$:



The dynamic programming table $P[1 \ldots n, 1 \ldots n]$ is used to store the exact length of the longest palindromic subsequence.

Recursively defining $P[i, j]$

Given a string contained in the array $s[1 \ldots n]$, $P[i, j]$ is set to the length of the longest palindromic subsequence of $s[i \ldots j]$.

- **Substrings of length 1:** For all $i = 1 \ldots n$

$$P[i, i] = 1$$

- **Substrings of length 2:** For all $i = 1 \ldots n - 1$,

  if $s[i] = s[i + 1]$ then:

$$P[i, i + 1] = 2$$

  otherwise $P[i, i + 1] = 1$

- **All other substrings:** For $1 < j - i$

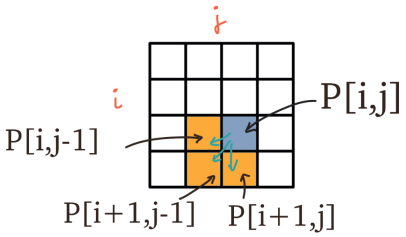
  if $s[i] = s[j]$ then

$$P[i, j] = 2 + P[i + 1, j - 1]$$

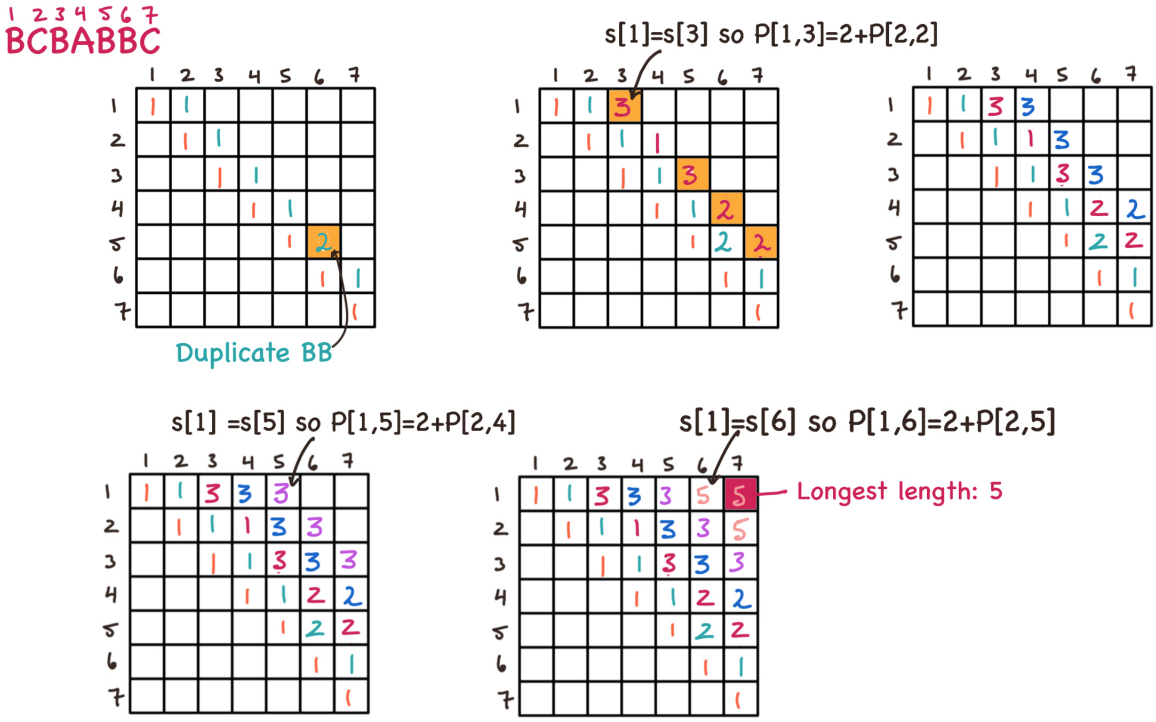
  otherwise

$$P[i, j] = \max\{P[i + 1, j], P[i, j - 1]\}$$

The above recursive definition indicates that cells in the table are referenced in the following way:



As in the previous section, we should fill in the cells of the table starting with the diagonal and moving out towards the top right corner. This order-of-fill guarantees that any look-up we make to the table is to a cell that will have already been filled The example below demonstrates the result of filling out the table for the string: $BCBACCB$:



LPSubsequence(s)

Step 1: Initialize the main diagonal of the table. Loop from $i = 1$ to $i = n$ and set

$$P[i, i] = 1$$

Step 2: Initialize the second diagonal of the table. Loop through from $i = 1$ to $i = n - 1$

$$\begin{aligned} &\text{If } s[i] = s[i+1] \\ &\qquad P[i, i+1] = 2 \\ &\text{else } P[i, i+1] = 1 \end{aligned}$$

Step 3: Fill in the rest of the upper part of the table, one diagonal at a time.
For each substring from index i to index j of length k, we verify if the first and last characters are the same, and set $P[i, j]$ accordingly:

$$\begin{aligned} &\text{for } k = 3 \text{ to } k = n \\ &\quad \text{for } i = 1 \text{ to } i = n - k + 1 \\ &\qquad j \text{ is the right endpoint: } j = i + k - 1 \\ &\qquad \text{If } s[i] = s[j] \\ &\qquad\quad P[i, j] = 2 + P[i+1, j-1] \\ &\qquad \text{else } P[i, j] = \max\{P[i+1, j], P[i, j-1]\} \end{aligned}$$

Final result: The maximum length of a palindromic subsequence of s is found in $P[1, n]$, since this entry represents all possible subsequences of the original string s.

Runtime: This algorithm carries out a constant number of operations per cell of the table. Since there are $\Theta(n^2)$ entries in the table, the algorithm runs in time $\Theta(n^2)$. This is a substantial improvement of the $\Theta(2^n)$ brute-force algorithm.

In the practice problems this week, we also look at how to reconstruct the solution for the above palindrome problem. Notice that this table stores the *length* of the longest palindrome subsequence in entry $P[1, n]$, but we would need an extra step to actually reconstruct what that palindrome subsequence is.