Dynamic and Greedy Programming

Practice Problems

Practice Problem 1

- Longest palindromic subsequence
  - You are given a string such as 'marquetteuniversity'
  - You have to find the largest substring that is a palindrome
    - (reads backwards the same as reads forward)
    - E.g. can we do better than marquetteuniversity
      - Yes, there are 'r's we can use
    - marquetteuniversity: 'ruetteur'

Practice Problem 1

- The simplest approach is to:
  - Generate all substrings
    - Check whether they are palindromes
    - Select the palindrome of longest length
- Question 1: What is the complexity of this simple algorithm

Practice Problem 1

- Question 2: If we have a string, how can we reduce it to the same problem involving strings of lesser length
  - How about 'ACCTATGAGCA'?
    - We look at 'ACCTATGAGCA'?
  - How about 'ACCTATGAGAC'?
    - We look at 'ACCTATGAGAC' and 'ACCTATGAGAC'?
- Question 3: How can we make this into an efficient algorithm
  - Subproblem: A quick way to calculate the length of the palindromic substring

Solution 1

- There are 2^n substrings of a string of length n
- Any solution that generates that many strings (or more than a fixed proportion of them) has exponential run-time

Solution 1

- Let $l(s)$ be the length of the longest palindromic substring of a string s.
    - If $s[0] \neq s[-1]$ then the two end letters cannot be both part of a maximum palindrome and we get
        - $l(s) = \max(l(s[1:]), l(s[:-1]))$
            - Recall that in Python $s[1:]$ is the slice obtained by removing the first letter of s and $s[:-1]$ the slice obtained by removing the last letter of s
        - Because the longest palindrome needs to be in one of these two substrings

Solution 1

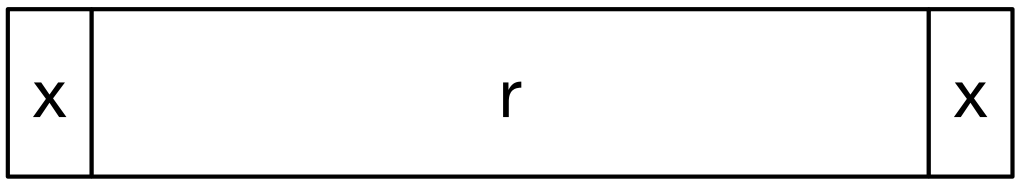
- But what about a case like
  - 'ACCTATGAGCA'
  - Can we say $l('ACCTATGAGCA') = l('CCTATGAGC') + 2$
  - 'ACCTATGAGCA'
  - This cannot be simply asserted
    - It could be the one of ACCTATGAGCA and ACCTATGAGCA could contain a larger palindrome

Solution 1

- If we cannot exclude the possibility, then the recursion formula would be
  - $l(s) = \max\{l(s[1:], l(s:-1), l(s[1,-1]) + 2)$ if $s[0] == s[-1]$

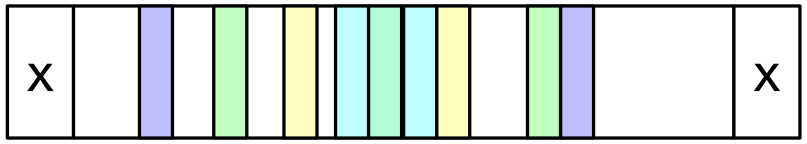
Solution 1

- Happily, this is not necessary
  - Assume that $s[0] == s[-1]$
  - Write x for that letter and $s = xrx$ with a substring r



Solution 1

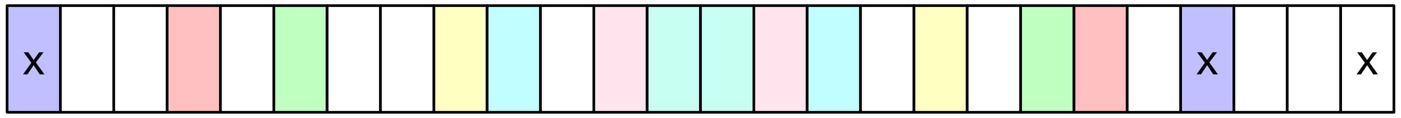
- Assume that the best palindrome is p
  - Where could it be:
    - If it is in the middle



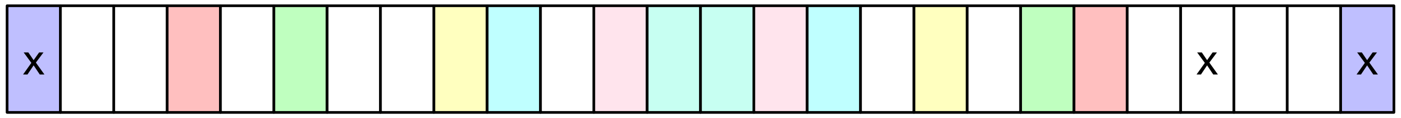
    - We can get a better one by including x

Solution 1

- Assume that the best palindrome is p
  - Where could it be:
    - It must therefore include on of the x
    - We can assume (without loss of generality) that it is the left x



    - But then we can just get the x from the rightmost x



    - A different best palindromic substring, but of equal length

Solution 1

- This gives us our recursion for the length of the largest palindromic substring

```
def lps(astring):
    if len(astring) == 1:
        return 1
    if len(astring) == 0:
        return 0
    if astring[0] == astring[-1]:
        return lps(astring[1:-1])+2
    else:
        return max(lps(astring[1:]), lps(astring[:-1]))
```

Solution 1

- Should we memoize this?
  - For longer strings, yes.
- Run time:
  - In the worst case, we look at two strings of size $n - 1$, so we are looking at 2^n strings.

Solution 1

- Finding the best palindrome
    - Return both the length and the best palindrome so far

```
def lps(astring):
    #print(astring)
    if len(astring) == 1:
        return 1, astring
    if len(astring) == 0:
        return 0, ""
    if astring[0] == astring[-1]:
        length, substring = lps(astring[1:-1])
        return length+2, astring[0]+substring+astring[-1]
    else:
        length1, substring1 = lps(astring[1:])
        length2, substring2 = lps(astring[:-1])
        if length1 < length2:
            return length2, substring2
        else:
            return length1, substring1
```