CS3230 Midterm Solution Sketches

Ryan Chew

September 18, 2019

Q1 Palindrome

Q1(i) Palindromic Substrings

Here, we want to detect which substrings of $A[1 \dots n]$ are palindrome.

We create a 2D array $B[i][j]$, where $B[i][j]$ is true if and only if the substring $A[i \dots j]$ is a palindrome.

Our base cases are the empty substrings ($i > j$), and single letters ($i = j$).

$$B[i][j] = \begin{cases} \texttt{TRUE} & (i \geq j) \\ ? & (i < j) \end{cases}$$

The cases $i < j$ correspond to substrings of length at least 2. These are palindromes, if and only if both the start and end letters are the same, and the remaining letters $A[i+1 \dots j-1]$ form a palindrome.

$$B[i][j] = \begin{cases} \texttt{TRUE} & (i \geq j) \\ \texttt{TRUE} & (i < j \wedge B[i+1][j-1]) \\ \texttt{FALSE} & (\text{otherwise}) \end{cases}$$

We need to justify that we do not have any 'cyclic dependencies' in our recursive equation. Intuitively, when we examine the length l substring $A[i \dots i + (l-1)]$, we look back only at length $l-2$ substrings. By starting with length $0, 1$ strings (corresponding to our base cases), and evaluating states in order of increasing length, we only examine previously computed state(s).

Algorithm 1 Return a 2D boolean array B, representing which A substrings are palindromes.

```
function FindPalindromes(A[1...n])
    B ← int[n][n]
    for i ← [1, n] do                          ▷ We skip filling in the negative length cells
        B[i][i − 1] ← 1
        B[i][i] ← 1
    end for
    for l ← [1, n] do
        for i ← [1, n] do
            if A[i] = A[i + (l − 1)] then
                B[i][i + (l − 1)] ← B[i + 1][i + (l − 2)]
            end if
        end for
    end for
    return B
end function
```

Now, we have a 2-dimensional DP recurrence. We have $i, j \in [1, n]$, and so have n^2 possible states. Approximately half of them have $i \geq j$, and so are base cases computable in $O(1)$ time.

The remaining cases $i < j$, examine one previous state $B[i+1][j-1]$, and evaluate in $O(1)$ time. All states are computed in constant time, and we have n^2 states. Hence our total time complexity is $O(n^2)$.

Q1(ii) Palindrome Partition

We want to break down a word, into a sequence of (non-empty) palindromes.

First, we should show a solution always exists. We can always break a length n string into n characters. As a single character is by itself a palindrome, every word can be trivially constructed from at most n palindromes.

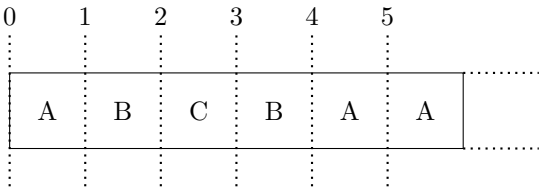
Also, for any non-empty (length ≥ 1) word, it is impossible to use less than 1 palindrome. Concatenating a negative number of palindromes does not make sense, while joining 0 palindromes only gives you an empty string.

Now, we consider prefixes $A[1 \dots k]$. If the prefix is empty $k = 0$, then we need 0 palindromes.

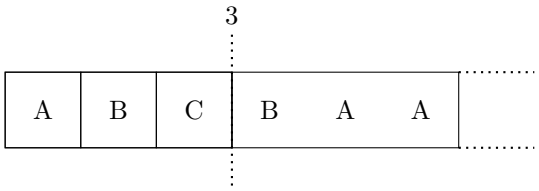
$$S[0] = 0$$

For any length $k \geq 1$ prefix of A, the minimum number of palindromes $S[k]$ is within $[1, k]$. In other words, each word has at least 1 non-empty palindrome.

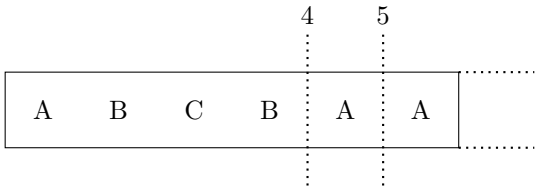
To shrink the problem, we break the prefix into a smaller prefix, and the final palindrome. This cut can be done at several locations.



Not all cuts are valid. If we try cutting at position 3:



Our last 'palindrome' is `BAA`, but its reversal is `AAB`. Clearly, not all cuts lead to valid palindromes.
Now, we want the best (minimal) previous cut, and add the final palindrome to produce our whole string.



One of these cuts must be the last cut of the minimal palindrome sequence. Hence, we should take the minimum of all of these.

$$S[0] = 0$$

$$S[n] = \min_{\substack{cut \in [0,n-1] \\ A[cut+1 \ldots n] \text{ is a palindrome}}} \overbrace{S[cut]}^{\text{Number of palindromes in prefix } A[1 \ldots cut]} + \overbrace{1}^{\text{Final palindrome } A[cut+1 \ldots n]}$$

$$= \min_{\substack{cut \in [0,n-1] \\ B[cut+1][n]}} S[cut] + 1$$

This is defined for every word, as the last cut $n-1$ is valid (as the single last letter forms a palindrome).
In the end, $S[n]$ will contain the minimum number of palindromes partitioning the word.

Algorithm 2 Given the computed B array, find minimum number of palindromes to combine to A.

```
function PalindromePartition(B[1...n][1...n])
    S ← int[0...n]                                  ▷ Note the 0-based indexing
    S[0] ← 0
    for prefix ← [1, n] do
        S[prefix] ← prefix
        for cut ← [0, prefix − 1] do
            if B[cut][prefix] then
                S[prefix] ← min(S[prefix], S[cut] + 1)
            end if
        end for
    end for
    return S[n]
end function
```

How long did we take to compute this? At each state $S[i], i > 0$, we examine i previous states ($S[0..i-1]$).

Each state transition takes $O(1)$ time to compute. Hence, the overall time complexity is:

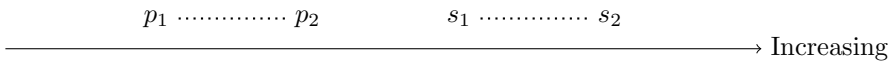
$$
\begin{aligned}
T(n) &\in \sum_{i=1}^{n} O(1) \cdot i \\
T(n) &\leq \sum_{i=1}^{n} ki = k \sum_{i=1}^{n} i && (\exists k > 0) \\
&= k \cdot \frac{n^2 + n}{2} \in O(n^2)
\end{aligned}
$$

Q2 Ski and Skier

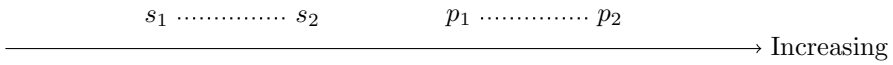
We want to show the $n = 2$ cases, i.e. for 2 skiers and hills. Let the skiers be $s_1 \leq s_2$, and hill heights be $p_1 \leq p_2$. We want to show that pairing $(s_1, p_1), (s_2, p_2)$ is no worse than pairing $(s_1, p_2), (s_2, p_1)$.

One key observation is that we can swap the list of skiers S and list of heights P, and our final answer still remains the same. After all, we are only concerned about pairing and minimizing the total difference.

Take the example, where we have $s_1 > p_1$.

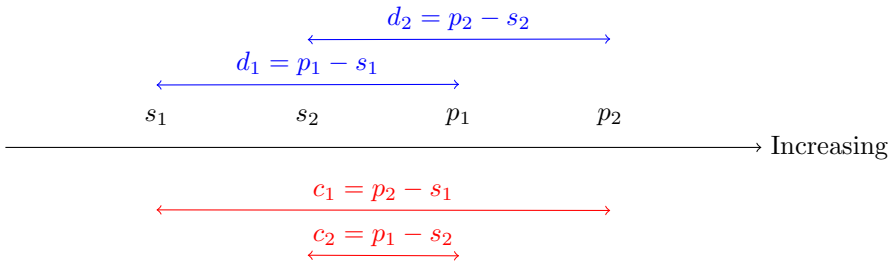


We can reverse the roles of the lists.



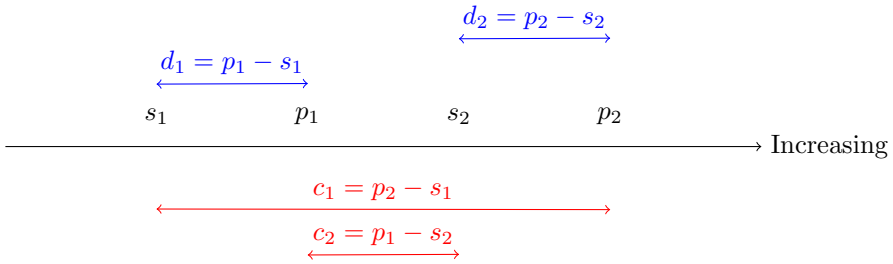
This way, we have a $s_1 \leq s_2$ and $p_1 \leq p_2$, with $s_1 \leq p_1$. Our least value is always s_1.
Hence, we reduce it to 3 cases:

- $s_1 \leq s_2 \leq p_1 \leq p_2$



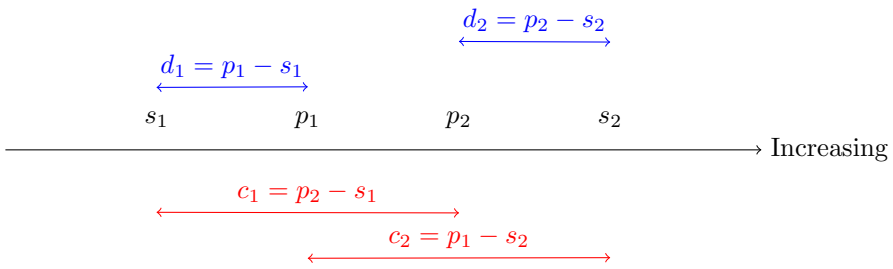
Here, $d_1 + d_2 = c_1 + c_2$, and so using the direct pairs d instead of cross pairs c is no worse.

- $s_1 \leq p_1 < s_2 \leq p_2$



In this case, $d_1 + d_2 = c_1 + c_2 - 2(s_2 - p_1) \leq c_1 + c_2$. Again, using direct pairing is at least no worse.

- $s_1 \leq p_1 \leq p_2 < s_2$



In this case, $d_1 + d_2 = c_1 + c_2 - 2(p_2 - p_1) \leq c_1 + c_2$. Again, using direct pairing is at least no worse.

In conclusion, for $n = 2$, pairing (s_1, p_1) and (s_2, p_2) results in a solution at least as good as the one with inversion.

By extension, for general n, removing inversions results in a solution at least as good. If we started with an optimal solution with some inversions, removing those inversions results in a solution equal in cost. We shall prove this.

First, assume we have $s_i < s_j$ and $p_k < p_l$. In the current optimal solution, we have paired $(s_i, p_l), (s_j, p_k)$. Ignoring all the other pairing, we can treat this identically with the $n = 2$ case. We swap the pairing to $(s_i, p_k), (s_j, p_l)$. By our previous proof, our total cost either stayed the same or decreased. We cannot have decreased the cost, as by definition, the optimal solution has least cost.

As an length n array has at most $\binom{n}{2}$ inversions, we do a finite number of swaps, and end with a solution with no inversions. The whole time, the cost has stayed constant.

The final inversion-free solution directly corresponds to pairing the sorted lists directly. This is exactly our greedy solution, and so our algorithm is optimal.

This is a form of exchange argument, where we start with some optimal solution. Then, we **exchange (adjust)** that solution, closer towards our greedy solution, without increasing the cost. Eventually, we end up with our greedy solution, which has optimal cost.

Q3 Increment or Divide?

We are given a starting value of 1, and 2 available operations:

- Increment the current number ($+1$)
- Double the current number ($\times 2$)

What is the shortest way to construct any positive integer n?

This seems to have a very close relation to n's binary representation. Let's try out same base cases, with a pretty naive algorithm. If the target number is even, our final operation is $\underset{\times 2}{\Rightarrow}$. Else, if it is odd, our final operation is $\underset{+1}{\Rightarrow}$.

Target	Binary	Sequence	Length
1	1	1	0
2	10	$1 \underset{\times 2}{\Rightarrow} 10$	1
3	11	$1 \underset{\times 2}{\Rightarrow} 10 \underset{+1}{\Rightarrow} 11$	2
4	100	$1 \underset{\times 2}{\Rightarrow} 10 \underset{\times 2}{\Rightarrow} 100$	2
5	101	$1 \underset{\times 2}{\Rightarrow} 10 \underset{\times 2}{\Rightarrow} 100 \underset{+1}{\Rightarrow} 101$	3
6	110	$1 \underset{\times 2}{\Rightarrow} 10 \underset{+1}{\Rightarrow} 11 \underset{\times 2}{\Rightarrow} 110$	3

If we try around some more, we find that we can do no better.

Here, we can see that $\underset{\times 2}{\Rightarrow}$ grows the number by introducing a 0 at end, while $\underset{+1}{\Rightarrow}$ is used to turn the 0 to 1. Furthermore, we never actually cause a carry when we $\underset{+1}{\Rightarrow}$.

How many operations does this naive algorithm take? We started with 1, but we need to grow to the same length of n in binary

$$l(n) = \lfloor \log_2 n \rfloor + 1$$

Similarly, we introduce a 1 with $\underset{+1}{\Rightarrow}$. At first, we only have a single 1 bit, but n may have multiple 1-bits.

$$b(n) = \text{Number of 1-bits in binary representation of } n$$

Hence, we can show our algorithm takes this many operations:

$$\begin{aligned} T(n) &= (\text{Number of bits to grow by}) + (\text{Number of one bits to insert}) \\ &= (l(n) - 1) + (b(n) - 1) \end{aligned}$$

Surprisingly, this is optimal! We shall prove that we can do no better. Let $B(n)$ be the best number of operations we need to construct n.

Now, for our inductive hypothesis. Let P_n be the statement that our algorithm is optimal for n.

$$P_n \equiv B(n) = T(n)$$

We have already proved that our algorithm is optimal for cases $n \in \{1, 2, 3, 4\}$.

Now, for $n > 4$, we assume $\{P_1, P_2, P_3, \ldots, P_{n-1}\}$. In other words, our algorithm is optimal, when producing anything from 1 to $n-1$. When we represent n in binary, we have 2 cases (note that xyz may be empty):

- $n = (\ldots xyz1)_2$

  Here, n is an odd number, as its binary representation ends in 1. This means the previous operation must have been an increment. As the last digit is 1, decrementing n to get $n-1$ does not borrow at all, and so the length remains the same, but the number of 1-bits decreases by 1.

  $$\begin{aligned} B(n) &= 1 + B(n-1) \\ &= 1 + T(n-1) && \text{(Inductive hypothesis)} \\ &= 1 + (l(n-1) - 1) + (b(n-1) - 1) \\ &= 1 + (l(n) - 1) + ((b(n) - 1) - 1) \\ &= (l(n) - 1) + (b(n) - 1) \\ &= T(n) \end{aligned}$$

  Hence our algorithm is optimal for this case.

- $n = (\ldots xyz1\underbrace{0\ldots0}_{t})_2$

  In this case, n is an even number. For this case, we could have built n either by doubling ($\frac{n}{2} \underset{\times 2}{\Rightarrow} n$) or incrementing ($n-1 \underset{+1}{\Rightarrow} n$). We'll examine both possibilities.

  - Doubling

    $$\frac{n}{2} \underset{\times 2}{\Rightarrow} n$$
    $$\frac{n}{2} = (\ldots xyz1\underbrace{0\ldots0}_{t-1})_2$$

    Here, $\frac{n}{2}$ is 1 bit shorter, but the number of 1-bits remains the same.

    $$\begin{aligned} b(n-1) &= b(n) \\ l(n-1) &= l(n) - 1 \\ B(n) &= 1 + B\left(\frac{n}{2}\right) \\ &= 1 + T\left(\frac{n}{2}\right) \\ &= 1 + \left(l\left(\frac{n}{2}\right) - 1\right) + \left(b\left(\frac{n}{2}\right) - 1\right) \\ &= 1 + ((l(n) - 1) - 1) + (b(n) - 1) \\ &= (l(n) - 1) + (b(n) - 1) \\ &= T(n) \end{aligned}$$

  - Incrementing

    $$n - 1 \underset{+1}{\Rightarrow} n$$
    $$n - 1 = (\ldots xyz0\underbrace{1\ldots1}_{t})_2$$

    Here, as we have t trailing zeroes, we must borrow from a 1-bit. We lose that 1-bit, and turn the t trailing 0-bits into 1-bits.
    The length may either stay the same (if we borrowed from the middle), or decrease by 1 (if we borrowed from the first and only bit, e.g. $10000_2 \to 1111_2$).

    $$\begin{aligned} b(n-1) &= b(n) + t - 1 \\ l(n) - 1 \leq l(n-1) &\leq l(n) \\ B(n) &= 1 + B(n-1) \\ &= 1 + T(n-1) \\ &= 1 + (l(n-1) - 1) + (b(n-1) - 1) \\ &\leq 1 + (l(n) - 1) + ((b(n) + t - 1) - 1) \\ &= (l(n) - 1) + (b(n) - 1) + t \\ &= T(n) + t \end{aligned}$$

  Hence, incrementing is at least as good as doubling, if and only if the number of trailing zeroes $t \leq 0$. However, by virtue of ending with a 0-bit, we have at least 1 trailing 0-bit. Hence, doubling is always better than incrementing.

By induction, this statement holds for every positive integer n. In conclusion, our algorithm always uses the optimal number of operations, and so is optimal.

Q4 Recursive Squaring

We want to square a k-digit number n. First, we divide its digits into 3 portions a, b, c.

$$\begin{aligned}
m &= \left\lfloor \frac{k}{3} \right\rfloor \\
n &= \underbrace{a \dots a}_{m+k \bmod 3} \mid \underbrace{b \dots b}_{m} \mid \underbrace{c \dots c}_{m} \\
n &= a \cdot 10^{2m} + b \cdot 10^{m} + c \\
n^2 &= (a \cdot 10^{2m} + b \cdot 10^{m} + c)^2 \\
&= a^2 \cdot 10^{4m} + b^2 \cdot 10^{2m} + c^2 + 2ab \cdot 10^{3m} + 2ac \cdot 10^{2m} + 2bc \cdot 10^{m} \\
&= a^2 \cdot 10^{4m} + 2ab \cdot 10^{3m} + (b^2 + 2ac) \cdot 10^{2m} + 2bc \cdot 10^{m} + c^2
\end{aligned}$$

We need $a^2, 2ab, (b^2+2ac), 2bc, c^2$. The question gives us a hint to first compute $a^2, c^2, (a+b+c)^2, (a-b+c)^2$.

$$\begin{aligned}
(a+b+c)^2 &= a^2 + b^2 + c^2 + 2ab + 2ac + 2bc \\
(a-b+c)^2 &= a^2 + b^2 + c^2 - 2ab + 2ac - 2bc
\end{aligned}$$

Here, we can remove a^2, c^2 from the equation, to help us isolate the terms we want.

$$\begin{aligned}
(a+b+c)^2 - a^2 - c^2 &= b^2 + 2ab + 2ac + 2bc \\
(a-b+c)^2 - a^2 - c^2 &= b^2 - 2ab + 2ac - 2bc \\
2(b^2 + 2ac) &= (a+b+c)^2 + (a-b+c)^2 - 2a^2 - 2c^2 \\
b^2 + 2ac &= \frac{(a+b+c)^2 + (a-b+c)^2}{2} - a^2 - c^2
\end{aligned}$$

And now, we have sieved out $b^2 + 2ac$, one of the terms we need. We have $2ab, 2bc$ remaining.

$$\begin{aligned}
ab + bc &= \frac{(a+b+c)^2 - a^2 - c^2 - (b^2 + 2ac)}{2} \\
-(ab + bc) &= \frac{(a-b+c)^2 - a^2 - c^2 - (b^2 + 2ac)}{2}
\end{aligned}$$

And now we are stuck, as we can only generate multiples of $ab + bc$. What final term can we multiply, in order to separate them? Obtaining something of the form $n(ab) + m(bc), n \neq m$ would help. However, we also want multiples of $a^2, c^2, (b^2 + 2ac)$ to appear, in order to cancel them out. Let's try generically, with $(a + jb + kc)$.

$$\begin{aligned}
(a+jb+kc)^2 &= a^2 + j^2b^2 + k^2c^2 + 2jab + 2kac + 2jkbc \\
(a+jb+kc)^2 - a^2 - k^2c^2 &= j^2\left(b^2 + \frac{k}{j^2}2ac\right) + 2jab + 2jkbc
\end{aligned}$$

We set $k = j^2$, so as to cancel out the fraction and obtain $b^2 + 2ac$.

$$\begin{aligned}
(a+jb+j^2c)^2 - a^2 - j^4c^2 &= j^2\left(b^2 + 2ac\right) + 2jab + 2j^3bc \\
2jab + 2j^3bc &= (a+jb+j^2c)^2 - a^2 - j^4c^2 - j^2\left(b^2 + 2ac\right)
\end{aligned}$$

To minimize the amount of additions to do, we pick $j = 2$. (It still works for larger j, but we spend more work.)

$$\begin{aligned}
4(ab + 4bc) &= (a+2b+4c)^2 - a^2 - 16c^2 - 4\left(b^2 + 2ac\right) \\
ab + 4bc &= \frac{(a+2b+4c)^2 - a^2}{4} - 4c^2 - \left(b^2 + 2ac\right)
\end{aligned}$$

Now, we can separate out ab and bc.

$$\begin{aligned} bc &= \frac{(ab + 4bc) - (ab + bc)}{3} \\ ab &= (ab + bc) - bc \end{aligned}$$

What is the recurrence relation for our time complexity? For an k-digit number, we squared 5 at-most-$\left(\frac{k}{3} + 2\right)$-digit numbers. (The $+2$ is from a's extra $k \bmod 3$ digits.) Hence, we have a recursive step of $5T(\frac{k}{3} + 2)$.

Then, we performed many (but a constant number!) of additions and subtractions. Each addition/subtraction is linear in bit length, hence our work term is $c \cdot O(k) = O(k)$.

$$T(k) \leq 5T\left(\frac{k}{3} + 2\right) + O(k)$$

To get rid of the ugly $+2$, we use a domain transform. We define a shifted version $S(k)$.

$$\begin{aligned} S(k) &= T(k+3) \\ &\leq 5T\left(\frac{k+3}{3} + 2\right) + O(k) \\ &= 5T\left(\frac{k}{3} + 3\right) + O(k) \\ &= 5S\left(\frac{k}{3}\right) + O(k) \end{aligned}$$

By Master Theorem, we have critical exponent $c_{crit} = \log_3 5$ and work term exponent $c = 1 < c_{crit}$. Hence, the recursive terms dominate, and we obtain:

$$\begin{aligned} S(k) &\in O\left(n^{\log_3 5}\right) \\ T(k) &\in O\left(n^{\log_3 5}\right) \end{aligned}$$