Strings e Programação Dinâmica

Maior Subsequência Palíndroma

Prof. Edson Alves - UnB/FGA

Sumário

Maior subsequência palíndroma

Definição

- Uma variante da maior sequência comum é o problema de se encontrar a maior subsequência de uma string S que forma um palíndromo (*Longest Palindrome Subsequence* – LPS)

Definição

- Uma variante da maior sequência comum é o problema de se encontrar a maior subsequência de uma string S que forma um palíndromo (*Longest Palindrome Subsequence* – LPS)
- Uma maneira de se enunciar este problema é a seguinte: qual é o maior palíndromo que pode ser formado removendo m $(0 \leq m \leq n)$ caracteres, de quaisquer posições, de uma string S de tamanho n?

Definição

- Uma variante da maior sequência comum é o problema de se encontrar a maior subsequência de uma string S que forma um palíndromo (*Longest Palindrome Subsequence* – LPS)
- Uma maneira de se enunciar este problema é a seguinte: qual é o maior palíndromo que pode ser formado removendo m $(0 \leq m \leq n)$ caracteres, de quaisquer posições, de uma string S de tamanho n?
- Este problema sempre tem solução, pois uma string com apenas um caractere é um palíndromo (o mesmo vale para strings vazias)

Definição

- Uma variante da maior sequência comum é o problema de se encontrar a maior subsequência de uma string S que forma um palíndromo (*Longest Palindrome Subsequence* – LPS)
- Uma maneira de se enunciar este problema é a seguinte: qual é o maior palíndromo que pode ser formado removendo m ($0 \leq m \leq n$) caracteres, de quaisquer posições, de uma string S de tamanho n?
- Este problema sempre tem solução, pois uma string com apenas um caractere é um palíndromo (o mesmo vale para strings vazias)
- O tamanho do maior palíndromo, ou o palíndromo em si, pode ser determinado por meio de programação dinâmica

Formulação em programação dinâmica da LPS

- Os casos bases ocorrem strings vazias ou com um único caractere

Formulação em programação dinâmica da LPS

- Os casos bases ocorrem strings vazias ou com um único caractere
- Se $LPS[i, j]$ é o tamanho da maior subsequência palíndroma da substring $S[i..j]$, então

$$LPS[i, i] = 1, \qquad LPS[i, j] = 0, \quad \text{se } i > j$$

Formulação em programação dinâmica da LPS

- Os casos bases ocorrem strings vazias ou com um único caractere
- Se $LPS[i, j]$ é o tamanho da maior subsequência palíndroma da substring $S[i..j]$, então

$$LPS[i, i] = 1, \qquad LPS[i, j] = 0, \quad \text{se } i > j$$

- São três transições possíveis: a primeira é remover o caractere mais à esquerda de $S[i..j]$:

$$LPS[i, j] = LPS[i + 1, j]$$

Formulação em programação dinâmica da LPS

- Os casos bases ocorrem strings vazias ou com um único caractere
- Se $LPS[i, j]$ é o tamanho da maior subsequência palíndroma da substring $S[i..j]$, então

$$LPS[i, i] = 1, \qquad LPS[i, j] = 0, \quad \text{se } i > j$$

- São três transições possíveis: a primeira é remover o caractere mais à esquerda de $S[i..j]$:

$$LPS[i, j] = LPS[i + 1, j]$$

- A segunda transição remove o caractere mais à direita $S[i..j]$:

$$LPS[i, j] = LPS[i, j - 1]$$

Formulação em programação dinâmica da LPS

- Os casos bases ocorrem strings vazias ou com um único caractere
- Se $LPS[i,j]$ é o tamanho da maior subsequência palíndroma da substring $S[i..j]$, então

$$LPS[i,i] = 1, \qquad LPS[i,j] = 0, \quad \text{se } i > j$$

- São três transições possíveis: a primeira é remover o caractere mais à esquerda de $S[i..j]$:

$$LPS[i,j] = LPS[i+1,j]$$

- A segunda transição remove o caractere mais à direita $S[i..j]$:

$$LPS[i,j] = LPS[i,j-1]$$

- No último caso, casos os caracteres que estão nos extremos da strings sejam iguais, eles podem ser parte do palíndromo:

$$LPS[i,j] = LPS[i+1,j-1] + 2, \quad \text{se } S[i] = S[j]$$

Implementação *top-down* da LPS

```cpp
int dp(const string& s, int i, int j)
{
    if (i > j)
        return 0;

    if (i == j)
        return 1;

    if (st[i][j] != -1)
        return st[i][j];

    st[i][j] = max(dp(s, i + 1, j), dp(s, i, j - 1));

    if (s[i] == s[j])
        st[i][j] = max(st[i][j], dp(s, i + 1, j - 1) + 2);

    return st[i][j];
}
```

Implementação *top-down* da LPS

```
int lps(const string& s)
{
    memset(st, -1, sizeof st);

    return dp(s, 0, s.size() - 1);
}

int main()
{
    string s;
    cin >> s;

    cout << lps(s) << '\n';

    return 0;
}
```

Identificação da maior subsequência palíndroma

- O algoritmo proposto para a LPS tem complexidade $O(n^2)$ tanto para a execução quanto para a memória

Identificação da maior subsequência palíndroma

- O algoritmo proposto para a LPS tem complexidade $O(n^2)$ tanto para a execução quanto para a memória
- Para recuperar a string que corresponde à LPS, é preciso manter o registro das operações utilizadas em cada transição:
  - `'B'`: os caracteres dos extremos são mantidos
  - `'K'`: o único caractere da string é mantido
  - `'L'`: remover o primeiro caractere
  - `'R'`: remover o último caractere

Identificação da maior subsequência palíndroma

- O algoritmo proposto para a LPS tem complexidade $O(n^2)$ tanto para a execução quanto para a memória
- Para recuperar a string que corresponde à LPS, é preciso manter o registro das operações utilizadas em cada transição:
  - `'B'`: os caracteres dos extremos são mantidos
  - `'K'`: o único caractere da string é mantido
  - `'L'`: remover o primeiro caractere
  - `'R'`: remover o último caractere
- Usando um valor sentinela (zero) para estados que não forem atingidos, é possível remontar o palíndromo em $O(n^2)$

Implementação da identificação da LPS

```cpp
int dp(const string& s, int i, int j)
{
    if (i > j)
        return 0;

    if (i == j) {
        ps[i][j] = 'K';
        return 1;
    }

    if (st[i][j] != -1)
        return st[i][j];

    st[i][j] = max(dp(s, i + 1, j), dp(s, i, j - 1));
    ps[i][j] = dp(s, i + 1, j) > dp(s, i, j - 1) ? 'L' : 'R';

    if (s[i] == s[j]) {
        st[i][j] = max(st[i][j], dp(s, i + 1, j - 1) + 2);
        ps[i][j] = st[i][j] > dp(s, i + 1, j - 1) + 2 ? ps[i][j] : 'B';
    }
```

Implementação da identificação da LPS

```
    return st[i][j];
}

string lps(const string& s)
{
    memset(st, -1, sizeof st);
    memset(ps, 0, sizeof ps);

    int n = s.size();

    dp(s, 0, n - 1);

    int i = 0, j = n - 1;
    string L = "", R = "";

    while (i <= j)
    {
        auto p = ps[i][j];
```

Implementação da identificação da LPS

```
        switch (p) {
        case 'L':
            ++i;
            break;

        case 'R':
            --j;
            break;

        case 'K':
            L += s[i];
            ++i;
            break;

        default:
            L += s[i]; R = s[i] + R;
            ++i; --j;
            break;
        }
    }
```

Implementação da identificação da LPS

```
    return L + R;
}

int main()
{
    string s;
    cin >> s;

    cout << lps(s) << '\n';

    return 0;
}
```

Referências

1. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.