Paradigmas de Resolução de Problemas

Programação Dinâmica: *Max Range Sum*

Prof. Edson Alves - UnB/FGA

2020

Sumário

Definição

Definição

Max Range Sum

Seja $a = \{a_1, a_2, \ldots, a_N\}$ uma sequência de N elementos. O *max range sum* de a é o intervalo $[i, j]$, com $i \leq j$, tal que a soma

$$\sum_{k=i}^{j} a_k = a_i + a_{i+1} + \ldots + a_j$$

é máxima.

Características do *max range sum*

- Observe que o *max range sum* de uma sequência não é, necessariamente, único
- Por exemplo, para a sequência

$$a = \{1, 1, 1, -5, 3, -2, 0\}$$

  os intervalos $[1, 3]$ e $[5, 5]$ tem, ambos, soma máxima (igual a 3)
- Variante comuns deste prolema é escolher um dentre estes intervalos, ou retornar somente o valor da soma máxima, ignorando o intervalo que a gerou
- Há $N(N-1)/2$ intervalos de índices $[i, j]$, com $i \leq j$, da sequência a
- Uma solução de busca completa que computa a soma de cada intervalo em $O(N)$ solucionaria o problema *max range sum* com complexidade $O(N^3)$

Implementação em $O(N^3)$ do MRS

```cpp
long long MRS(const vector<int>& as)
{
    auto N = as.size();
    long long ans = -oo;

    for (size_t i = 0; i < N; ++i)
    {
        for (size_t j = i; j < N; ++j)
        {
            long long sum = 0;

            for (size_t k = i; k <= j; ++k)
                sum += as[k];

            ans = max(ans, sum);
        }
    }

    return ans;
}
```

Soma dos prefixos

- Uma forma de reduzir a complexidade da solução é computar as somas de cada intervalo em $O(1)$
- Assim a complexidade seria reduzida de $O(N^3)$ para $O(N^2)$
- Esta soma em $O(1)$ pode ser feita precomputando a a soma de todos os prefixos de a
- Um prefixo p_j de a é uma subsequência que tem início no primeiro elemento a_1 de a e termina no elemento a_j
- Seja $ps(i)$ a soma dos elementos do prefixo p_i de a, isto é,

$$ps(i) = \sum_{k=1}^{i} = a_1 + a_2 + \ldots + a_i$$

Soma dos prefixos

- Como p_0 é o prefixo vazio, então $ps(0) = 0$
- Para $1 \leq i \leq N$, vale que

$$ps(i) = a_1 + a_2 + \ldots + a_i = (a_1 + a_2 + \ldots + a_{i-1}) + a_i = ps(i-1) + a_i$$

- Assim, é possível computar $ps(i)$ para todos os i possíveis em $O(N)$
- A soma dos elementos cujos índices pertencem ao intervalo $[i, j]$ então é dada por

$$\begin{aligned}
\sum_{k=i}^{j} a_k &= a_i + a_{i+1} + \ldots + a_j \\
&= (a_1 + \ldots + a_{i-1}) + (a_i + \ldots + a_j) - (a_1 + \ldots + a_{i-1}) \\
&= (a_1 + a_2 + \ldots + a_j) - (a_1 + \ldots + a_{i-1}) \\
&= ps(j) - ps(i-1)
\end{aligned}$$

Implementação do MRS em $O(N^2)$

```cpp
long long MRS(const vector<int>& as)
{
    auto N = as.size() - 1;
    vector<long long> ps(N + 1, 0);

    for (size_t i = 1; i <= N; ++i)
        ps[i] = ps[i - 1] + as[i];

    long long ans = -oo;

    for (size_t i = 1; i <= N; ++i)
        for (size_t j = i; j <= N; ++j)
            ans = max(ans, ps[j] - ps[i - 1]);

    return ans;
}
```

Algoritmo de Kadane

- De forma surpreendente, é possível resolver solucionar o *max range sum* em $O(N)$
- Em geral, o algoritmo de Kadane é apresentado de tal forma que leva a crer que o mesmo seja um algoritmo guloso
- Porém ele é, de fato, um algoritmo de programação dinâmica
- Seja $s(k)$ a maior soma dentre todos os intervalos da forma $[i, k]$, com $1 \leq i \leq k$
- O caso base ocorre quando $k = 1$: neste caso, $s(1) = a_1$
- Para $k > 1$ há duas transições possíveis:
    1. estender o intervalo anterior, adicionando a_k
    2. desprezar o intervalo anterior, e retornar a_k

Algoritmo de Kadane

- Destas duas possibilidades, o algoritmo seleciona a melhor das duas
- A segunda transição só é a melhor quando $s(k-1) < 0$, e esta característica é que gera a percepção de estratégia gulosa do algoritmo
- Em notação matemática,

$$s(k) = \max\{\ s(k-1) + a_k, a_k\ \}$$

- Uma vez computado os valores de $s(k)$ para $k \in [1, N]$, a solução do problema é dada por

$$MRS(a) = \max\{\ s(1), s(2), \ldots, s(N)\ \}$$

- Como há $O(N)$ estados possíveis e as transições são feitas em $O(1)$, a complexidade da solução é $O(N)$

Implementação do algoritmo de Kadane

```
int kadane(const vector<int>& as)
{
    vector<int> s(as.size());
    s[0] = as[0];

    for (size_t i = 1; i < as.size(); ++i)
        s[i] = max(as[i], s[i - 1] + as[i]);

    return *max_element(s.begin(), s.end());
}

```

Implementação “gulosa” do algoritmo de Kadane

```cpp
int kadane(const vector<int>& as)
{
    int ans = as[0], sum = ans;

    for (size_t i = 1; i < as.size(); ++i)
    {
        if (sum < 0)
            sum = 0;

        sum += as[i];
        ans = max(ans, sum);
    }

    return ans;
}
```

Max 2D Range Sum

Definição

Max 2D Range Sum

Seja A uma matriz $n \times m$. O *max 2D range sum* da matriz A é a submatriz $B_{r\times s}$ de A cuja soma

$$\sum_{i=1}^{r}\sum_{j=1}^{s} b_{ij}$$

é máxima.

Características do *Max 2D Range Sum*

- Assim como no caso unidimensional, a solução não é, necessariamente, única
- Uma submatriz B de A pode ser determinada pelas coordenadas (a, b) de seu canto superior esquerdo e pelas coordenadas (c, d) do seu canto inferior direito
- No pior caso, soma todos os elementos de B tem complexidade $O(nm)$
- O total de submatrizes de A é $O(n^2m^2)$
- Assim, uma algoritmo *naive* para o problema tem complexidade $O(n^3m^3)$

Implementação naive do Max 2D Range Sum

```
int MSR(int N, int M, const vector<vector<int>>& A)
{
    int ans = -oo;

    for (int a = 0; a < N; ++a)
        for (int b = 0; b < M; ++b)
            for (int c = a; c < N; ++c)
                for (int d = b; d < M; ++d)
                {
                    int sum = 0;

                    for (int i = a; i <= c; ++i)
                        for (int j = b; j <= d; ++j)
                            sum += A[i][j];

                    ans = max(ans, sum);
                }

    return ans;
}
```

Max 2D Range Sum com soma de prefixos

- A ideia da soma de prefixos pode ser estendida para o caso bidimensional, permitindo somar os elementos de uma submatriz B de A em $O(1)$
- Seja $p(i, j)$ a soma dos elementos da submatriz B cujo canto superior esquerdo é $(1, 1)$ e o canto inferior direito é (i, j)
- Os casos base acontecem quando $i = 1$ ou $j = 1$, os quais se reduzem à soma de prefixos unidimensional:

$$
\begin{aligned}
p(1,1) &= a_{11} \\
p(1,j) &= p(1,j-1) + a_{1j}, \quad \text{se } j > 1 \\
p(i,1) &= p(i-1,1) + a_{i1}, \quad \text{se } i > 1
\end{aligned}
$$

Max 2D Range Sum com soma de prefixos

- Quando $i > 1$ e $j > 1$, o valor de $p(i, j)$ pode ser obtido utilizando-se o princípio da inclusão/exclusão:

$$p(i, j) = p(i, j - 1) + p(i - 1, j) - p(i - 1, j - 1) + a_{ij}$$

- A matriz abaixo ilustra esta situação:

$$A = \begin{bmatrix} \underline{a_{11}} & \underline{a_{12}} & \dots & \underline{a_{1(j-1)}} & a_{1j} & \dots & a_{1m} \\ \underline{a_{21}} & \underline{a_{22}} & \dots & \underline{a_{2(j-1)}} & a_{2j} & \dots & a_{2m} \\ \underline{a_{(i-1)1}} & \underline{a_{(i-1)2}} & \dots & \underline{a_{(i-1)(j-1)}} & a_{(i-1)j} & \dots & a_{(i-1)m} \\ \underline{a_{i1}} & \underline{a_{i2}} & \dots & \underline{a_{i(j-1)}} & a_{ij} & \dots & a_{im} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \dots & \dots & \dots & a_{nm} \end{bmatrix}$$

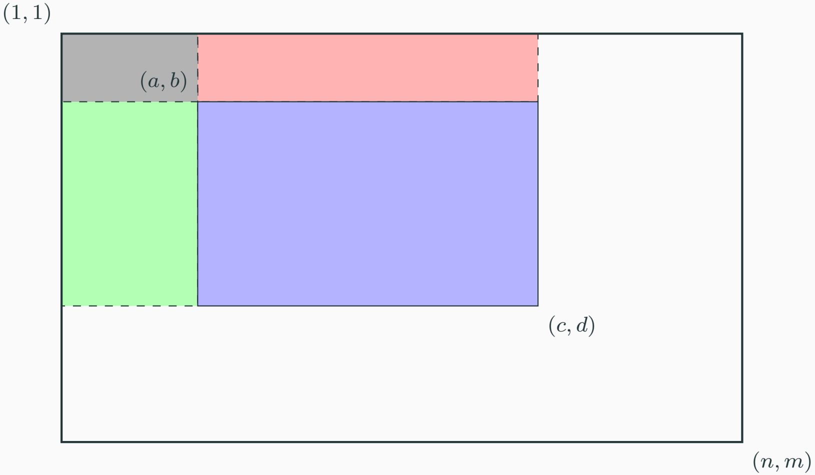
Max 2D Range Sum com soma de prefixos

- Assuma que $p(0, j) = p(i, 0) = 0$
- É possível computar a soma dos elementos de uma matriz B cujo canto superior esquerdo é o ponto (a, b) e o ponto inferior direito é (c, d) a partir dos valores de p:

$$S(a, b, c, d) = p(c, d) - p(a - 1, d) - p(c, b - 1) + p(a - 1, b - 1)$$

- Esta expressão novamente utiliza o princípio de inclusão/exclusão
- Ela permite computar as somas S em $O(1)$, e os valores de p são computados em $O(nm)$
- Assim a complexidade do algoritmo se reduz para $O(n^2m^2)$

Visualização da soma das submatrizes a partir dos valores de p



Max 2D Range Sum PD

```
int MSR(int N, int M, const vector<vector<int>>& A)
{
    vector<vector<int>> p(N + 1, vector<int>(M + 1, 0));

    for (int i = 1; i <= N; ++i)
        for (int j = 1; j <= M; ++j)
            p[i][j] = p[i][j-1] + p[i-1][j] - p[i-1][j-1] + A[i][j];

    int ans = -oo, sum;

    for (int a = 0; a < N; ++a)
        for (int b = 0; b < M; ++b)
            for (int c = a; c < N; ++c)
                for (int d = b; d < M; ++d)
                {
                    sum = p[c][d] - p[a-1][d] - p[c][b-1] + p[a-1][b-1];
                    ans = max(ans, sum);
                }

    return ans;
}
```

Max 2D Range Sum com algoritmo de Kadane

- Assim como no caso unidimensional, o algoritmo de Kadane pode ser utilizado para reduzir a complexidade do *max 2D range sum*
- A ideia é a seguinte: para cada par de colunas (i, j), com $1 \leq i \leq j \leq m$, deve-se aplicar o algoritmo de Kadane na sequência $r = \{r_1, r_2, \ldots, r_N\}$, onde

$$r_k = \sum_{t=i}^{j} a_{kt}$$

- Deste modo, o algoritmo de Kadane identifica, dentre todas as submatrizes B de A que iniciam na coluna i e terminam na coluna j, a que possui maior soma
- Como o algoritmo de Kadane roda em $O(n)$ e há $O(m^2)$ pares de colunas, esta solução para o *max 2D range sum* tem complexidade $O(nm^2)$

Implementação do max 2D range sum com kadane em $O(nm^2)$

```cpp
int MSR(int N, int M, const vector<vector<int>>& A)
{
    vector<vector<int>> p(N + 1, vector<int>(M + 1, 0));
    int ans = -oo;

    for (int i = 1; i <= M; ++i)
    {
        vector<int> r(N + 1, 0);

        for (int j = i; j <= M; ++j)
        {
            for (int k = 1; k <= N; ++k)
                r[k] += A[k][j];

            ans = max(ans, kadane(N, r));
        }
    }

    return ans;
}
```

Max Square Sub-Matrix

Definição

Max Square Sub-Matrix with all 1s

Seja A uma matrix $n \times m$ tal que $a_{ij} \in [0, 1]$, para todo $1 \leq i \leq n, 1 \leq j \leq m$. O problema consiste em identificar a submatriz quadrada B de A, de maior área possível, tal que todos os elementos de B são iguais a 1.

Características do problema

- Este é um problema semelhante, mas não idêntico, ao *max 2D range sum*
- Aqui, a solução B não é, necessariamente, a submatriz de maior soma
- Novamente, uma solução de força bruta checaria $O(m^2n^2)$ submatrizes em $O(mn)$ cada, tendo complexidade $O(n^3m^3)$
- Porém, é possível resolver tal problema com complexidade $O(mn)$ por meio de um algoritmo de programação dinâmica

Solução de programação dinâmica

- Seja $S(i, j)$ a dimensão k da submatriz $B_{k \times k}$ de maior área, cujos elementos são todos iguais a 1, que tem o elemento a_{ij} esta localizado no canto inferior direito de B
- Naturalmente, $S(i, j) = 0$ se $a_{ij} = 0$
- Se $a_{ij} = 1$, então

$$S(i, j) = \min\{\ S[i][j-1], S[i-1][j], S[i-1][j-1]\ \} + 1$$

- Esta transição tenta construir o maior quadrado possível com o canto inferior esquerdo em a_{ij} a partir dos quadrados ótimos para seus vizinhos à oeste, norte e noroeste
- Os casos base ocorrem quando $i = 1$ ou $j = 1$: nestes casos, $S(i, 1) = a_{i1}$ e $S(1, j) = a_{1j}$
- A solução do problema corresponde ao valor máximo de $S(i, j)$
- A complexidade do algoritmo é $O(mn)$: são $O(mn)$ estados, com transições em $O(1)$

Implementação *bottom-up* do *max square sub-matrix*

```
int MSS(int N, int M)
{
    for (int i = 0; i < N; ++i)
        S[i][0] = A[i][0];

    for (int j = 0; j < M; ++j)
        S[0][j] = A[0][j];

    int ans = 0;

    for (int i = 1; i < N; ++i)
        for (int j = 1; j < M; ++j)
        {
            S[i][j] = A[i][j] == 0 ? 0 :
                min({ S[i-1][j], S[i][j-1], S[i-1][j-1] }) + 1;

            ans = max(ans, S[i][j]);
        }

    return ans;
}
```

Referências

1. **CORMEN**, Thomas H.; **LEISERSON**, Charles E.; **RIVEST**, Ronald; **STEIN**, Clifford. *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
2. GeeksForGeeks. Maximum size square sub-matrix with all 1s, acesso em 24/09/2020.
3. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
4. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.