1. Introdução

1.1. **Sobre o Trabalho.**

1.2. **Notação.**

1.3. Matrizes. Explicar o que são matrizes online e offline.

1.4. Implementações. Explicar os padrôes que estou usando pra implementar os programas. Por exemplo: Funções como argumentos, 0-index (em contraste com o 1-index do pseudo-código) e wrappers.

2. Matrizes Monge e monotonicidade

Nesta seção serão apresentados e explorados os conceitos de monotonicidade, convexidade e matrizes Monge, além disso, alguns resultados referentes a estes conceitos serão demonstrados. As definições e os resultados desta seção são fundamentais para o desenvolvimento do restante do trabalho.

Definição 2.1 (Vetor monótono)**.** *Seja $a \in \mathbb{Q}^n$ um vetor, a é dito monótono quando vale uma das propriedades abaixo.*

- *Se para todo $i, j \in [n]$, $i < j \Rightarrow a_i \leq a_j$, a é dito monótono crescente (ou só crescente).*
- *Se para todo $i, j \in [n]$, $i < j \Rightarrow a_i \geq a_j$, a é dito monótono decrescente (ou só decrescente).*

Sabemos que a monotonicidade de vetores pode ser aproveitada para agilizar alguns algoritmos importantes, por exemplo, a busca binária pode ser interpretada como uma otimização da busca linear para vetores monótonos.

Definição 2.2 (Função convexa)**.** *Seja $g : \mathbb{Q} \to \mathbb{Q}$ uma função,*

- *se para todo par de pontos $x, y \in \mathbb{Q}$ e $\lambda \in \mathbb{Q}$ que respeita $0 \leq \lambda \leq 1$, vale $g(\lambda x + (1-\lambda)y) \leq \lambda g(x) + (1-\lambda)g(y)$, g é dita convexa e*
- *se para todo par de pontos $x, y \in \mathbb{Q}$ e $\lambda \in \mathbb{Q}$ que respeita $0 \leq \lambda \leq 1$, vale $g(\lambda x + (1-\lambda)y) \geq \lambda g(x) + (1-\lambda)g(y)$, g é dita côncava.*

Proposição 2.3. *A função $g(x) = x^2$ é convexa.*

Demonstração. Sejam $x, y, \lambda \in \mathbb{Q}$ onde vale $0 \leq \lambda \leq 1$. Queremos provar $(\lambda x + (1-\lambda)y)^2 \leq \lambda x^2 + (1-\lambda)y^2$, isso equivale a

$$\lambda^2 x^2 + (1-\lambda)^2 y^2 + 2\lambda(1-\lambda)xy \leq \lambda x^2 + (1-\lambda)y^2, \text{ ou seja}$$

$$(\lambda^2 - \lambda)(x^2) + ((1-\lambda)^2 - (1-\lambda))y^2 + 2(\lambda - \lambda^2)xy \leq 0, \text{ que é}$$

$$(\lambda^2 - \lambda)(x^2 + y^2 - 2xy) = (\lambda^2 - \lambda)(x+y)^2 \leq 0.$$

□

É interessante definir convexidade também em termos de vetores.

Definição 2.4 (Vetor convexo)**.** *Seja $a \in \mathbb{Q}^n$ um vetor,*

- *se para todo $i, j, k \in [n]$, $i < j < k \Rightarrow a_j \leq \frac{(j-k)a_i + (i-j)a_k}{i-k}$, a é dito convexo e*
- *se para todo $i, j, k \in [n]$, $i < j < k \Rightarrow a_j \geq \frac{(j-k)a_i + (i-j)a_k}{i-k}$, a é dito côncavo.*

Assim como a monotonicidade, a convexidade também é usualmente explorada para agilizar algoritmos, por exemplo, se um vetor é convexo podemos definir o valor mínimo do vetor com uma busca ternária ao invés de percorrer todo o vetor.

Definição 2.5. *Seja $A \in \mathbb{Q}^{n \times m}$, definimos quatro vetores a seguir.*

- *O vetor de índices de máximos das linhas de A guarda na posição i o número* $\max\{j \in [m] \mid A[i][j] \geq A[i][j'] \text{ para todo } j' \in [m]\}$.
- *O vetor de índices de mínimos das linhas de A guarda na posição i o número* $\min\{j \in [m] \mid A[i][j] \leq A[i][j'] \text{ para todo } j' \in [m]\}$.
- *O vetor de índices de máximos das colunas de A guarda na posição j o número* $\max\{i \in [n] \mid A[i][j] \geq A[i'][j] \text{ para todo } i' \in [n]\}$.

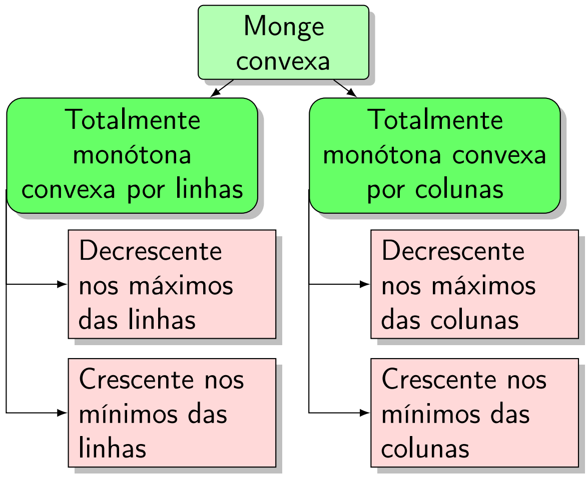

FIGURA 2.6. Comportamento dos vetores de índices ótimos em relação à convexidade.

- *O vetor de índices de mínimos das colunas de A guarda na posição j o número* $\min\{i \in [n] \mid A[i][j] \leq A[i'][j]$ *para todo* $i' \in [n]\}$.

Note que o máximo de uma linha (ou coluna) foi definido como o maior índice que atinge o máximo e o de mínimo foi definido como o menor índice que atinge o mínimo. Esta escolha foi feita para simplificar o Lema 2.9, porém, os algoritmos e resultados discutidos neste trabalho funcionam (com pequenas adaptações) para diversas definições distintas destes vetores.

Dada uma matriz, encontrar estes vetores é um problema central para este trabalho. Neste momento é interessante classificar algumas matrizes de acordo com propriedades que vão nos ajudar a calcular os vetores de mínimos e máximos de maneira especialmente eficiente.

A Figura 2.6 resume as relações de implicação da classificação que será realizada. Os conceitos ilustrados nela serão apresentados a seguir.

Definição 2.7 (Matriz monótona)**.** *Seja $A \in \mathbb{Q}^{n \times m}$ uma matriz. Se A tiver o vetor de índices de mínimos das linhas monótono, A é dita monótona nos mínimos das linhas.*

Valem também as definições análogas para máximos ou colunas e pode-se especificar monotonicidade crescente ou decrescente.

Definição 2.8 (Matriz totalmente monótona)**.** *Seja $A \in \mathbb{Q}^{n \times m}$ uma matriz.*

- *Se $A[i'][j] \leq A[i'][j']$ implica $A[i][j] \leq A[i][j']$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, A é monótona convexa nas linhas.*
- *Se $A[i][j'] \leq A[i'][j']$ implica $A[i][j] \leq A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, A é monótona convexa nas colunas.*
- *Se $A[i'][j] > A[i'][j']$ implica $A[i][j] > A[i][j']$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, A é monótona côncava nas linhas.*
- *Se $A[i][j'] > A[i'][j']$ implica $A[i][j] > A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, A é monótona côncava nas colunas.*

O motivo do uso dos termos "convexa" e "côncava" em relação a matrizes durante o texto são justificados pelo Teorema 2.15. Note que se uma matriz é totalmente monótona, todas as suas submatrizes são totalmente monótonas no mesmo sentido.

Lema 2.9. *Se $A \in \mathbb{Q}^{n\times m}$ é uma matriz totalmente monótona convexa nas linhas, toda submatriz de A é monótona crescente nos mínimos das linhas e monótona decrescente nos máximos das linhas.*

Se A é totalmente monótona côncava nas linhas, toda submatriz de A é monótona decrescente nos mínimos das linhas e monótona crescente nos máximos das linhas.

As afirmações valem identicamente em termos de colunas.

Demonstração. Considere uma matriz A totalmente monótona convexa nas linhas. Sejam i e i' índices de linhas de A onde $i < i'$. Chamamos de j o índice de máximo da linha i e de j' o índice de máximo da linha i'. Queremos provar que os máximos são decrescentes, portanto, vamos supor por absurdo que $j < j'$. Com isso, teremos $A[i][j'] < A[i][j]$ e $A[i'][j] \leq A[i'][j']$. Porém, já que A é monótona convexa nas linhas, a segunda desigualdade implica em $A[i][j] \leq A[i][j']$, que contradiz a primeira. Portanto, os índices de máximos são decrescentes.

Agora, considere novamente dois índices i e i' quaisquer de linhas de A onde $i < i'$. Denotamos por j o índice de mínimo da linha i' e por j' o índice de mínimo da linha i (note e a inversão no uso de $'$). Vamos supor por absurdo que $j < j'$ e teremos $A[i'][j] \leq A[i'][j']$ e $A[i][j'] < A[i][j]$. E, novamente, usando o fato de que A é monótona convexa nas linhas, obtivemos uma contradição.

Finalmente, se A' é uma submatriz de A, então A' é totalmente monótona convexa nas linhas, portanto monótona crescente nos mínimos das linhas e monótona decrescente nos máximos das linhas.

As demonstrações no caso côncavo e nos casos relacionados a colunas são análogas. □

Definição 2.10 (Monge Convexidade)**.** *Seja $A \in \mathbb{Q}^{n\times m}$.*

(1) *Se vale $A[i][j]+A[i'][j'] \leq A[i][j']+A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$ então A é dita Monge convexa.*
(2) *Se vale $A[i][j]+A[i'][j'] \geq A[i][j']+A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$ então A é dita Monge côncava.*

A desigualdade que define as matrizes Monge é conhecida pelos nomes "Propriedade de Monge" (em inglês, "Monge Property") [6] ou "Desigualdade Quadrangular" (em inglês, "Quadrangle Inequality") [11, 4].

Lema 2.11. *Se A é Monge convexa, A é totalmente monótona convexa tanto nas linhas quanto nas colunas.*

Se A é Monge côncava, A é totalmente monótona côncava tanto nas linhas quanto nas colunas.

Demonstração. Seja A uma matriz Monge convexa. Suponha que vale, para certos $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, $A[i'][j] \leq A[i'][j']$, então, somamos esta desigualdade à definição de Monge convexa e obtemos $A[i][j] \leq A[i][j']$, ou seja, A é totalmente monótona convexa nas linhas.

Por outro lado, se vale, para certos $i, i' \in [n]$ e $j, j' \in [m]$ com $i < i'$ e $j < j'$, $A[i][j'] \leq A[i'][j']$, somamos esta desigualdade à definição de Monge convexa e obtemos $A[i][j] \leq A[i'][j]$, assim, A é totalmente monótona convexa nas colunas.

A prova para o caso côncavo é análoga. □

Teorema 2.12. *Seja $A \in \mathbb{Q}^{n\times m}$.*

Vale $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$ para todo $i \in [n-1]$ e $j \in [m-1]$ se e somente se A é Monge convexa.

Vale $A[i][j] + A[i+1][j+1] \geq A[i][j+1] + A[i+1][j]$ para todo $i \in [n-1]$ e $j \in [m-1]$ se e somente se A é Monge côncava.

Demonstração. Seja $A \in \mathbb{Q}^{n\times m}$. Se A é Monge convexa, vale $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$ para todo $i \in [n-1]$ e $j \in [m-1]$. Vamos mostrar o outro lado desta implicação.

Sejam $i \in [n-1]$ e $j \in [m-1]$ quaisquer, vamos provar que $A[i][j] + A[i+a][j+1] \leq A[i][j+1] + A[i+a][j]$ para todo $0 < a \leq n-i$ com indução em a. A base, onde $a = 1$ vale pela hipótese. Quando $1 < a \leq n-i$ assumimos que a tese vale com $a-1$ e temos $A[i][j] + A[i+a-1][j+1] \leq A[i][j+1] + A[i+a-1][j]$ e, já que $i+a-1 \in [n-1]$, vale $A[i+a-1][j] + A[i+a][j+1] \leq A[i+a-1][j+1] + A[i+a][j]$ e, somando as duas inequações, obtemos $A[i][j] + A[i+a][j+1] \leq A[i][j+1] + A[i+a][j]$. Isso conclui a prova proposta neste parágrafo.

Agora, sejam $i \in [n-1]$ e $j \in [m-1]$ quaisquer, vamos provar que $A[i][j] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j+b]$ para todo $0 < a \leq n-i$ e $0 < b \leq m-j$ por indução b. A base desta indução, onde $b = 1$, foi provada no parágrafo anterior. Se $1 < b \leq m-j$ assumimos que a tese vale para $b-1$, escrevemos $A[i][j] + A[i+a][j+b-1] \leq A[i][j+b-1] + A[i+a][j]$, pela prova do parágrafo anterior, vale $A[i][j+b-1] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j+b-1]$ e, mais uma vez, somando as duas equações provamos a desigualdade $A[i][j] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j]$. Com isso provamos que A é Monge convexa.

A prova para o caso côncavo segue análogamente. □

As matrizes Monge são usadas para resolver uma série de problemas que serão explorados aqui. A condição de Monge é a mais forte apresentada neste trabalho e alguns dos algoritmos apresentados não dependem dela, apenas da monotonicidade ou total monotonicidade, ainda assim, ela leva a resultados úteis que nos permitem provar a pertinência dos algoritmos a problemas, mesmo que o algoritmo usado não se utilize da condição diretamente.

Como consequência desta utilidade, iremos discutir um problema que será resolvido com um algoritmo apresentado somente na Seção 4, o algoritmo SMAWK. Ele não será explicado neste momento, utilizamos ele como caixa preta. Desta forma, poderemos introduzir estes resultados que são importantes em vários momentos deste texto e na aplicação prática dos conhecimentos discutidos aqui de forma suave e motivada.

Problema 2.13. *Definimos a função de custo c de cada vetor v como* $c(v) = \left(\sum_{i=1}^{|v|} v_i\right)^2$.

Dados dois inteiros k e n e um vetor $v \in \mathbb{Q}^n_+$, queremos particionar o vetor v em k subvetores de forma a minimizar a soma dos custos das partes. Formalmente, queremos escolher um particionamento $P_1, P_2, \ldots, P_k$ de v em subvetores que minimize $\sum_{i=1}^{k} c(v_{P_i})$.

Definimos a matriz A para todo $i, j \in [0\,..\,n]$ onde $A[i][j] = c(v[i+1\,..\,j])$ para todo $i \leq j$ e $A[i][j] = +\infty$ caso contrário. A matriz não precisa ser explícitamente calculada, pré-calculamos em $\mathcal{O}(n)$ o vetor a tal que $a_i = \sum_{k=1}^{i} v_k$ para todo $i \in [0\,..\,n]$. Com o vetor a conseguimos calcular cada entrada da matriz A em $\mathcal{O}(1)$ quando necessário.

Podemos resolver o Problema 2.13 com programação dinâmica. Um subproblema de parâmetros i e ℓ é da forma: Melhor particionamento do vetor $v[i+1\,..\,n]$ em ℓ partes. Definimos a matriz $E \in \mathbb{Q}^{[k]\times[0\,..\,n]}$ de respostas desses subproblemas, assim, se ℓ e i definem um subproblema o seu valor ótimo é guardado em $E[\ell][i]$. Todas as outras entradas da matriz E têm valor $+\infty$. Escrevemos E como uma recorrência, para todo $\ell \in [k]$ e $i \in [0\,..\,n]$,

$$E[\ell][i] = \begin{cases} A[i][n] & \text{, se } \ell = 1, \\ \min\limits_{j=i}^{n} A[i][j] + E[\ell-1][j] & \text{, se } \ell \leq k \text{ e} \\ +\infty & \text{, caso contrário.} \end{cases}$$

É fácil resolver a recorrência definida acima em tempo $\mathcal{O}(kn^2)$. Vamos simplificar a definição de E. Fixados $i \in [0\,..\,n]$ e $\ell \in [1\,..\,k]$, se $j < i$ então $A[i][j] = +\infty$ e podemos escrever $E[\ell][i] = \min\limits_{j=0}^{n} A[i][j] + E[\ell-1][j]$. Definimos a matriz B_ℓ onde, para todo $j \in [0\,..\,n]$, a entrada $B_\ell[i][j] = A[i][j] + E[\ell-1][j]$. Além disso, definimos a matriz B_1 onde, para todo $i, j \in [0\,..\,n]$ vale $B_1[i][j] = A[i][j] + A[i][n]$. Note que $\min\limits_{j=0}^{n} B_1[i][j] = A[i][i] + A[i][n] = A[i][n] = E[1][i]$. Desta forma, para todo $\ell \in [k]$ e $i \in [0\,..\,n]$ vale $E[\ell][i] = \min\limits_{j=0}^{n} B_\ell[i][j]$.

Com esta formulação, reduzimos o problema original a encontrar os mínimos das linhas de B_ℓ para todo $l \in [k]$. O algoritmo SMAWK encontra mínimos de linhas de matrizes $n+1 \times n+1$ totalmente monótonas por linhas em tempo $\mathcal{O}(n)$. Vamos mostrar que as matrizes B_ℓ são totalmente monótonas convexas por linhas para podermos aplicar o SMAWK.

Lema 2.14. *Sejam $A, B \in \mathbb{Q}^{n \times m}$ matrizes e $c \in \mathbb{Q}^m$ um vetor tais que para todo $i \in [n]$ e $j \in [m]$, $B[i][j] = A[i][j] + c[j]$. Se A é Monge convexa, B é Monge convexa.*

O mesmo resultado vale se $c \in \mathbb{Q}^n$ e $B[i][j] = A[i][j] + c[i]$. Os resultados análogos valem nos casos de concavidade.

Demonstração. Sejam A, B e b definidos como no enunciado do teorema. Suponha que A é Monge convexa. Vale, para quaisquer $1 \leq i < i' \leq n$ e $1 \leq j < j' \leq m$, $A[i][j] + A[i'][j'] \leq A[i'][j] + A[i][j']$, logo, vale $A[i][j] + b[j] + A[i'][j'] + b[j'] \leq A[i'][j] + b[j'] + A[i][j'] + b[j]$ que é $B[i][j] + B[i'][j'] \leq B[i'][j] + B[i][j']$. A prova para o caso onde $c \in \mathbb{Q}^n$ e $B[i][j] = A[i][j] + c[i]$ é análoga, bem como as provas para os casos côncavos. □

Suponha que A é Monge convexa. Todas as matrizes B_ℓ se encaixam perfeitamente nas hipóteses do Lema 2.14 e, por isso, são Monge convexas, portanto, totalmente monótonas convexas por linhas. Basta provar que A é Monge convexa.

Teorema 2.15. *Sejam $A \in \mathbb{Q}^{n \times n}$ uma matriz, $w \in \mathbb{Q}^n_+$ um vetor e $g : \mathbb{Q} \to \mathbb{Q}$ uma fução tais que para todo $i, j \in [n]$ vale $A[i][j] = g\left(\sum\limits_{k=1}^{j} w_k - \sum\limits_{k=1}^{i} w_k\right)$. Se g é convexa, A é Monge convexa. Similarmente, se g é côncava, A é Monge côncava.*

Antes de demonstrar este teorema, vamos provar que A é Monge convexa utilizando o resultado. Considere a função g tal que $g(x) = x^2$ se $x \geq 0$ e $g(x) = +\infty$ caso contrário. Vale, para todo $i, j \in [n]$, $A[i][j] = g\left(\sum\limits_{k=1}^{j} v_k - \sum\limits_{k=1}^{i} v_k\right)$. Vamos provar que g é convexa. Sejam $x \leq y \in \mathbb{Q}$ e $\lambda \in [0, 1]$, escrevemos $z = \lambda x + (1-\lambda)y$. Se $0 \leq x$, pela Proposição 2.3, sabemos $g(z) \leq \lambda g(x) + (1-\lambda)g(y)$. Se $x < 0$, vale $\lambda g(x) + (1-\lambda)g(y) = +\infty \geq g(z)$, independente do valor de $g(z)$. Assim, g é convexa e aplicamos o teorema para concluir que A é Monge convexa.

Com isso já que o nosso problema se reduziu a encontrar, para todos os $\ell \in [k]$ os mínimos das linhas da matriz B_ℓ e estas são Monge convexas, elas também são totalmente monótonas convexas por linhas e podemos encontrar seus máximos em $\mathcal{O}(n)$, resolvendo o problema todo em $\mathcal{O}(kn)$.

Nos falta provar o Teorema 2.15.

Demonstração. Sejam A e g quaisquer que respeitem as condições do enunciado. Sejam ainda $i, i', j, j' \in [n]$ onde $i < i'$ e $j < j'$. Escrevemos $a = \sum_{k=1}^{i'} w_k - \sum_{k=1}^{i} w_k$, $b = \sum_{k=1}^{j'} w_k - \sum_{k=1}^{j} w_k$ e $z = \sum_{k=1}^{j} w_k - \sum_{k=1}^{i'} w_k$. Desta forma, temos $g(z) = A[i'][j]$, $g(z+a+b) = A[i][j']$, $g(z+a) = A[i][j]$ e $g(z+b) = A[i'][j']$, portanto, $g(z+a) + g(z+b) \leq g(z) + g(z+a+b)$ se e somente se $A[i][j] + A[i'][j'] \leq A[i][j'] + A[i'][j]$ (A é Monge convexa).

Consideramos o caso onde $0 < a \leq b$. Temos $0 < a \leq b < a+b$, ou seja, $z < z+a \leq z+b < z+a+b$. Definimos $\lambda = \frac{a}{a+b}$. Já que $z + a = \lambda z + (1-\lambda)(z+a+b)$ e $z + b = (1-\lambda)z + \lambda(z+a+b)$, por convexidade de g, obtemos $g(z+a) \leq \lambda g(z) + (1-\lambda)g(z+a+b)$ e $g(z+b) \leq \lambda g(z+a+b) + (1-\lambda)g(z)$. Somando, obtemos $g(z+a) + g(z+b) \leq g(z) + g(z+a+b)$.

Se considerarmos o caso onde $0 < b \leq a$, seguimos o mesmo raciocínio e obtemos, novamente, $g(z+a) + g(z+b) \leq g(z) + g(z+a+b)$. Falta considerar o caso onde $0 = a = b$, neste caso, $g(z) = g(z+a) = g(z+b) = g(z+a+b)$ e vale $(z+a) + g(z+b) \leq g(z) + g(z+a+b)$. Portanto, A é Monge convexa. $\square$

3. Divisão e Conquista

Nesta seção será apresentada uma técnica que chamamos de Divisão e Conquista. A ideia é citada por Aggarwal [3] e é um tópico recorrente em competições de programação, sendo conhecida como "Divide and Conquer Optimization" [1, 2] e geralmente aplicada a problemas de programação dinâmica.

Além disso, as hipóteses deste algoritmo são mais fracas do que as do algoritmo SMAWK, apresentado na Seção 4, portanto, todo problema para o qual aquela solução pode ser aplicada, esta também pode. Ao final desta seção, apresentamos exemplos de aplicações em programação dinâmica.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$, listamos os casos de uso deste algoritmo:

- Se A é monótona nos mínimos das linhas podemos encontrar os índices de mínimos das linhas em tempo $\mathcal{O}((n+m)\lg(n))$,
- se A é monótona nos máximos das linhas podemos encontrar os índices de máximos das linhas em tempo $\mathcal{O}((n+m)\lg(n))$,
- se A é monótona nos mínimos das colunas podemos encontrar os índices de mínimos das colunas em tempo $\mathcal{O}((n+m)\lg(m))$ e
- se A é monótona nos máximos das colunas podemos encontrar os índices de máximos das colunas em tempo $\mathcal{O}((n+m)\lg(m))$.

Apresentaremos o caso em que A é crescente nos mínimos das linhas. É fácil manipular o algorimto para trabalhar com os outros casos.

3.1. **Técnica.** Dada uma matriz $A \in \mathbb{Q}^{n \times m}$ monótona crescente nos mínimos das linhas, queremos encontrar o vetor de índices de mínimos das linhas de A. Isto é, para todo $i \in [n]$, queremos encontrar

$$R[i] = \min\{j \mid A[i][j] \leq A[i][j'] \text{ para todo } j' \in [n]\}.$$

Se, para alguma linha i, encontrarmos o valor $R[i]$, sabemos que para todo $i' < i$, $R[i'] \leq R[i]$ e, para todo $i' > i$, $R[i'] \geq R[i]$, isto é, sabemos que os mínimos de menor índice das outras linhas se encontram nas submatrizes $A[1\,..\,i-1][1\,..\,R[i]]$ e $A[i+1\,..\,n][R[i]\,..\,m]$. Seguindo o paradigma de divisão e conquista, vamos resolver o mesmo problema para estas submatrizes e, consequentemente, resolver o problema original.

Algoritmo 3.1 Mínimos das linhas com divisão e conquista

1: **função** FindRowMin_DC(A, r_s, r_t, c_s, c_t)
2: $\quad \ell \leftarrow \lceil (r_s + r_t)/2 \rceil$
3: $\quad R[\ell]$ índice de mínimo da linha ℓ.
4: $\quad$ **se** $i > r_s$ **então**
5: $\quad\quad R[r_s\,..\,\ell-1] \leftarrow$ FindRowMin_DC($A, r_s, \ell-1, c_s, R[\ell]$)
6: $\quad$ **se** $i < r_t$ **então**
7: $\quad\quad R[\ell+1\,..\,r_t] \leftarrow$ FindRowMin_DC($A, \ell+1, r_t, R[\ell], c_t$)
8: $\quad$ **devolve** R

Note que na linha 3 o mínimo só precisa ser buscado entre os índices c_s e c_t, inclusive, pois estamos resolvendo o problema para a submatriz $A[r_s\,..\,r_t][c_s\,..\,c_t]$.

3.2. **Análise.** Será feita uma análise do tempo de execução do algoritmo acima no pior caso assumindo que as atribuições feitas nas linhas 5 e 7 custam tempo constante, futuramente, na Subseção 3.3, iremos apresentar uma implementação em C++ que está de acordo com a análise realizada.

Se A é uma matriz e r_s, r_t, c_s e c_t são índices tais que $r_t - r_s = n > 0$ e $c_t - c_s = m > 0$, o tempo gasto por FINDROWMIN_DC(A, r_s, r_t, c_s, c_t) pode ser expresso pela seguinte recorrência:

$$T(n,m) = \begin{cases} m & \text{, se } n = 1, \\ m + \max\limits_{j\in[m]} T(1,j) & \text{, se } n = 2, \\ m + \max\limits_{j\in[m]} \left\{ \begin{array}{c} T(\lceil n/2\rceil - 1, m - j + 1) \\ +T(\lfloor n/2\rfloor, j) \end{array} \right\} & \text{, caso contrário.} \end{cases}$$

Proposição 3.2. *Para todo $n, m \geq 1$, $T(n,m) \leq (m+n)\lg(2n)$ e, portanto, a técnica da divisão e conquista consegue encontrar o mínimo de todas as linhas em tempo $\mathcal{O}((m+n)\lg(n))$*

Demonstração. Vamos usar indução em n para provar a tese. Se $n = 1$ e $m \geq 1$, $T(1,m) = m \leq (m+1)\lg(2)$. Se $n = 2$ e $m \geq 1$, existe $j \in [m]$ tal que $T(2,m) = m + r \leq 2m \leq (m+2)\lg(4)$. Agora, se $n \geq 3$ e $m \geq 1$, existe um $j \in [m]$ tal que

$$T(n,m) = m + T(\lceil n/2\rceil - 1, j) + T(\lfloor n/2\rfloor, m - j + 1).$$

Assumimos para $1 \leq n' < n$ e $m' \geq 1$ que $T(n', m') \leq (m' + n')\lg(2n')$. Com isso, já que $1 \leq \lceil n/2\rceil - 1 < n$, $1 \leq \lfloor n/2\rfloor < n$, $j \geq 1$ e $m - j + 1 \geq 1$, temos, com a equação acima e o fato de que $\lceil n/2\rceil - 1 \leq \lfloor n/2\rfloor \leq n/2$,

$$\begin{aligned} T(n,m) &\leq m + (j + \lceil n/2\rceil - 1 + m - j + 1 + \lfloor n/2\rfloor)\lg(n) \\ &= m + (m+n)\lg(n) < (m+n)(\lg(n)+1) = (m+n)\lg(2n). \end{aligned}$$

□

3.3. **Implementação.** Para implementar o Algoritmo 3.1 com a complexidade desejada, devemos tomar cuidado com as atribuições feitas nas linhas 5 e 7. A forma como elas foram apresentadas sugere que os vetores R recebidos pelas funções sejam recebidos e copiados para o vetor R. Ao invés de fazer isso, passaremos o endereço do vetor R recursivamente e garantir que cada chamada só complete o subvetor $R[r_c \mathinner{..} r_t]$, referente a seu subproblema. Além disso, como explicado na Subseção 1.4, a matriz A será passada como uma função e não como uma matriz.

A implementação em C++ do algoritmo apresentado, levando em conta as considerações acima, pode ser encontrada em `implementacao/FindRowMax_DC.cpp`.

3.4. **Aplicação em programação dinâmica.** Utilizaremos a técnica apresentada aqui para resolver uma adaptação do problema "Internet Trouble" da Final Brasileira da Maratona de Programação de 2016. A prova em questão pode ser encontrada no link `http://maratona.ime.usp.br/hist/2016/resultados/contest.pdf`.

Problema 3.3. *Definimos a função de custo c de cada vetor v como $c(v) = \sum_{i=1}^{|v|} \min(i-1, m-i)v_i$.*

Sejam n e k inteiros onde $1 \leq k \leq n$ e seja $h \in \mathbb{N}^n$ um vetor, queremos encontrar uma partição P de h em até k subvetores $h_{P_1}, h_{P_2}, \ldots, h_{P_k}$ não vazios de forma que $\sum_{i=1}^{k} c(h_{P_i})$.

Podemos dar ao problema acima a interpretação do problema "Internet Trouble" citado. Temos várias cidades dispostas em uma linha e queremos escolher $k+1$ destas cidades para instalar torres de distribuição de energia de forma a minimizar o custo de alimentar todas as cidades com energia. Nesta adaptação, as cidades 1 e n são escolhas obrigatórias. Se na cidade de índice i existem h_i habitantes, o custo de tranferir energia de uma torre a d cidades de distância para esta cidade é dado por dh_i. Note que, se há uma torre na própria cidade, o custo é considerado 0.

Definimos, para todo $1 \leq i \leq j \leq n$ o custo $A[i][j]$ de escolher o subvetor $v[i \,..\, j]$ como uma das partições. Os valores com índices de linhas maiores do que índices de colunas não fazem sentido na nossa modelagem, portanto, queremos torná-los inválidos, já o problema é de minimização definimos seus valores como $+\infty$. Assim, para todo $1 \leq i, j \leq n$,

$$A[i][j] = \begin{cases} c(v[i \,..\, j]) & \text{, se } i \leq j \text{ e} \\ +\infty & \text{, c.c.} \end{cases}$$

Perceba que, se criarmos em tempo $\mathcal{O}(n)$ dois vetores $p, q \in \mathbb{N}^{[0\,..\,n]}$ onde, para todo $i \in [0 \,..\, n]$, $p_i = \sum\limits_{k=1}^{i} v_k$ e $q_i = \sum\limits_{k=1}^{i} kv_k$, podemos calcular cada entrada de A em $\mathcal{O}(1)$ quando necessário.

Queremos resolver o problema com programação dinâmica. Definimos para todo $\ell \in [k]$ e $i \in [n]$ com $\ell \leq n-i+1$ o problema de encontrar o melhor particionamento do subvetor $v[i \,..\, n]$ em ℓ partes. Definimos como $E[\ell][i]$ o valor ótimo atingido neste subproblema. Se $\ell = 1$, escrevemos $E[\ell][i] = A[i][n]$ e se $1 < \ell \leq k$ escrevemos $E[\ell][i] = \min\limits_{j=i}^{n-\ell+2} A[i][j] + E[\ell-1][j]$. Esta recorrência define um programa dinâmico que pode ser resolvido trivialmente em tempo $\mathcal{O}(kn^2)$.

Com $\ell > 1$, definindo $B_\ell[i][j] = A[i][j] + E[\ell-1][j]$ podemos reescrever $E[\ell][i] = \min\limits_{j=i}^{n-\ell+2} B[i][j]$ e, se definirmos $B_\ell[i][j] = +\infty$ para todo i que desrespeite $i \leq j$ ou $\ell \leq n-i+1$, teremos $E[\ell][i] = \min\limits_{j=1}^{n} B_\ell[i][j]$. Esta formulação reduz o problema de programação dinâmica a encontrar, para todo $1 < \ell \leq k$ fixo, os mínimos das linhas da matriz B_ℓ. Com isso, basta provar que B_ℓ é monótona crescente nos mínimos das linhas e aplicar a técnica da divisão e conquista para resolver o problema em tempo $\mathcal{O}(kn\log(n))$.

Vamos, primeiramente, provar que A é Monge convexa. Queremos mostrar que vale, para todo $i, j \in [n]$, a desigualdade $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$ e usar o Teorema 2.12 para concluir que A é Monge convexa. Se $j \leq i$, $A[i+1][j] = +\infty$, logo, já que $A[i][j+1] \geq 0$, a desigualdade vale. Consideramos que $i < j$. Escrevemos $a = \lfloor \frac{i+j}{2} \rfloor$ e $b = \lceil \frac{i+j}{2} \rceil$, note que $b = \lfloor \frac{i+j+1}{2} \rfloor$.

Temos $A[i][j] = \sum\limits_{k=i}^{j} \min(k-i, j-k)h_k = \sum\limits_{k=i}^{a}(k-i)h_k + \sum\limits_{k=a+1}^{j}(j-k)h_k$ e $A[i][j+1] = \sum\limits_{k=i}^{j+1} \min(k-i, j-k+1)h_k = \sum\limits_{k=i}^{b}(k-i)h_k + \sum\limits_{k=b+1}^{j}(j-k+1)h_k$. Se $i+j$ é par, $a = b$ e vale $A[i][j+1] = \sum\limits_{k=i}^{a}(k-i)h_k + \sum\limits_{k=a+1}^{j}(j-k)h_k = A[i][j] + \sum\limits_{k=a+1}^{j} h_k$. Se $i+j+1$ é ímpar, $b-1 = a$ e $b = \frac{i+j+1}{2}$, logo $b - i = j - b + 1$ e teremos $(k-b)h_b = (j-b+1)h_b$, o que leva a $A[i][j+1] = \sum_{k=i}^{a}(k-i)h_k + \sum_{k=a+1}^{j}(j-k+1)h_k = A[i][j] + \sum\limits_{k=a+1}^{j} h_k$.

O parágrafo acima nos mostrou que $A[i][j+1] = A[i][j] + \sum_{k=a+1}^{j} h_k$. Com um raciocínio parecido, conseguiremos concluir $A[i+1][j+1] = A[i+1][j] + \sum_{k=b+1}^{j} h_k$. Assim, $A[i][j+1] - A[i+1][j+1] = A[i][j] + \sum_{k=a+1}^{j} h_k - A[i+1][j] - \sum_{k=b+1}^{j} h_k$. Sabemos que $a \leq b$, portanto, obtivemos $A[i][j+1] - A[i+1][j+1] \geq A[i][j] - A[i+1][j]$, isto é $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$. Provamos que A é Monge convexa.

Sabemos que A é Monge convexa, pelo Teorema 2.14 todas as matrizes B_ℓ são Monge convexas, portanto, monótonas decrescentes nos mínimos das linhas e podemos aplicar a técnica da Divisão e Conquista para encontrar seus mínimos de linhas em tempo $\mathcal{O}(n)$. Já que o problema consiste em encontrar estes mínimos para todas as matrizes B_ℓ com $\ell \in [k]$, conseguimos resolver o problema em tempo $\mathcal{O}(kn \lg(n))$. Vale notar que a Subseção 4.6 ensina a resolver este mesmo problema em tempo $\mathcal{O}(kn)$.

4. SMAWK

Nesta seção discutiremos o algoritmo SMAWK. Ele é conhecido pela sua aplicação no problema de encontrar o vértice mais distante de cada vértice num polígono convexo em tempo linear [3]. Ao final desta seção serão citadas esta e outras aplicações deste algoritmo.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$, listamos os casos de uso deste algoritmo:

- Se A é totalmente monótona convexa ou côncava nas linhas podemos encontrar os índices de mínimos e máximos das linhas em tempo $\mathcal{O}(n+m)$ e
- se A é totalmente monótona convexa ou côncava nas colunas podemos encontrar os índices de mínimos e máximos das colunas em tempo $\mathcal{O}(n+m)$.

Apresentaremos o caso onde A é totalmente monótona convexa nas linhas e estamos interessados nos índices de mínimos. É fácil manipular o algoritmo para trabalhar com os outros casos.

4.1. Técnica Primordial.

Para facilitar a compreensão do algoritmo SMAWK, iremos apresentar uma técnica parecida com a Divisão e Conquista apresentada na Seção 3 e mostrar uma otimização desta técnica que leva ao algoritmo SMAWK.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$ totalmente monótona convexa por linhas, queremos encontrar o índice de mínimo de cada uma das linhas de A. Se para uma dada linha i onde $i > 0$ e $i < n$ conhecermos os índices ℓ e r de mínimos das linhas $i-1$ e $i+1$, respectivamente, já que A tem os índices de mínimos das linhas crescente (por ser totalmente monótona) basta buscar o índice de mínimo da linha i no intervalo entre ℓ e r (inclusive). Além disso, se i é a primeira linha da matriz podemos considerar $\ell = 1$ ou se i é a última linha da matriz podemos considerar $r = n$ sem perder a validade do fato de que basta buscar entre ℓ e r.

Após realizar as observações acima note que, já que A é totalmente monótona, remover qualquer linha de A mantém a total monotonicidade e não altera o índice de mínimo de outra linha. Com esta observação, concluímos que podemos remover todas as linhas pares da matriz, resolver o problema recursivamente para a matriz resultante e utilizar este resultado para calcular os índices de interesse para as linhas pares da matriz. Vamos provar que encontrar estes índices de mínimos custa tempo $\mathcal{O}(m)$.

Definimos a sequência t de forma que a i-ésima linha ímpar de A busca seu máximo entre as colunas t_{i-1} e t_i, inclusive. Sabemos que $0 = t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_{\lfloor n/2 \rfloor} \leq n$. Podemos escrever o tempo gasto por todas as buscas de mínimo como $\sum_{i=1}^{\lfloor n/2 \rfloor} t_i - t_{i-1} + 1 = \mathcal{O}(n+m)$, ou seja, o trabalho feito para encontrar os mínimos das colunas ímpares dados os mínimos das colunas pares custa tempo $\mathcal{O}(n+m)$.

Agora, com uma análise similar à realizada para a técnica da Divisão e Conquista é fácil concluir que uma implementação desta técnica que consiga remover as linhas pares da matriz (e adicionar elas de volta) em tempo $\mathcal{O}(1)$ resolve o problema em tempo $\mathcal{O}((n+m)\lg(n))$, assim como a técnica da divisão e conquista.

4.2. Reduce.

Chamamos de ótimas as células de uma matriz que são mínimo de alguma linha e as colunas que contém o índice de mínimo de pelo menos uma linha. Note que uma matriz contém no máximo n colunas ótimas, pois cada linha faz com que exatamente uma célula seja ótima.

Queremos agilizar a técnica apresentada acima. Para isso, vamos adicionar a nova hipótese de que a matriz A é quadrada, ou seja, $n = m$. Lembre que a cada passo, removemos as $\lfloor n/2 \rfloor$ linhas pares da matriz gerando uma nova matriz A', resolvemos o problema recursivamente para A' e usamos a solução de A' para resolver para as linhas restantes de A. Quando removemos linhas da nossa A, ela deixa de ser quadrada e passa a ser uma matriz com mais colunas do que linhas, isto é, $m \geq n$. Queremos remover colunas não ótimas da matriz A com mais colunas do que linhas fazendo com que A se torne quadrada.

Vamos desenvolver o algoritmo Reduce a partir de um índice de linha k e de algumas invariantes:

(1) Vale $1 \leq k \leq n$,
(2) apenas colunas não ótimas foram removidas da matriz e
(3) toda célula em uma coluna de índice menor ou igual a k que possua índice de linha menor do que índice de coluna é não ótima. A Figura 4.1 representa, em azul, a célula de índice k, k e, em preto, as células que, segundo esta invariante, são não ótimas.

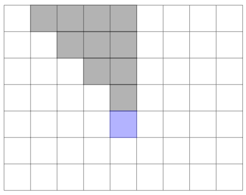

Figura 4.1. Invariante 3 do Reduce.

Vamos comparar $A[k][k]$ com $A[k][k+1]$ e considerar dois casos. Em cada um dos casos, concluiremos que algumas células da matriz A são não ótimas. A Figura 4.2 mostra, hachuradas em vermelho, as células que são descobertas não ótimas quando $A[k][k] > A[k][k+1]$ e, com linhas verticais verdes, as células que são descobertas não ótimas quando $A[k][k] \leq A[k][k+1]$. Vamos provar estas implicações.

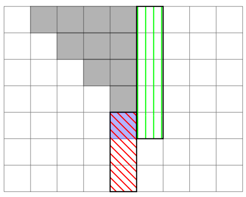

Figura 4.2. Casos do Reduce.

Se $A[k][k] > A[k][k+1]$, as entradas com índice de linha maior ou igual a k na coluna k são não ótimas. Primeiramente, a célula (k, k) é não ótima como consequência direta da desigualdade. Agora, suponha que existe alguma linha $i > k$ tal que $A[i][k] \leq A[i][k+1]$. Pela total monotonicidade convexa por linhas de A, isso implica em $A[k][k] \leq A[k][k+1]$, um absurdo.

Se $A[k][k] \leq A[k][k+1]$, as células da coluna $k+1$ com índices de linha menores ou iguals a k são não ótimas. A célula $(k, k+1)$ é não ótima pela desigualdade apresentada. Suponha que existe alguma linha $i < k$ tal que $A[i][k] > A[i][k+1]$. Pela contrapositiva da total monotonicidade, temos $A[k][k] > A[k][k+1]$, um absurdo.

Com estas observações estamos prontos para deduzir um algoritmo que elimina exatamente $m-n$ colunas de A.

Algoritmo 4.3 Algoritmo REDUCE

função REDUCE(A)
  $k \leftarrow 1$
  enquanto A tem mais linhas do que colunas
    se $A[k][k] > A[k][k+1]$ **então**
      Remove a coluna k
      $k \leftarrow \max(1, k-1)$
    senão
      se $k = n$ **então**
        Remove a coluna $k+1$
      senão
        $k \leftarrow k+1$
  devolve A

É fácil ver que as invariantes são válidas neste algoritmo. Olhamos para o primeiro passo, $k = 1$, nenhuma coluna foi removida ainda e não há elementos com índices de linha e coluna menores do que k, logo, as Invariantes 1, 2 e 3 valem. Em todo passo do loop, $A[k][k+1]$ existe, pois $k \leq n$ e $n \leq m$. Consideramos o caso onde $A[k][k] > A[k][k+1]$, a Invariante 1 sempre se mantém trivialmente, já provamos que a coluna k é não ótima neste caso, portanto, a Invariante 2 se mantém mesmo após a remoção da coluna k. Agora, se $k = 1$, vale a 3 por vacuidade e, no caso contrário, já que a k decresce, a Invariante 3 também se mantém.

Em outro caso, valem $A[k][k] \leq A[k][k+1]$ e $k = n$. Foi provado que os elemntos de linhas menores ou iguals a k na coluna coluna $k+1$ são não ótimos, porém, isto representa toda a coluna $k+1$, assim, remover ela mantém a Invariante 2. As outras duas invariantes se mantém trivialmente. Agora, falta considerar o caso onde $A[k][k] \leq A[k][k+1]$ e $k < n$. Neste caso, foi provado, novamente, que as células com índices menores ou iguais a k na coluna $k+1$ são inválidos. Estes são exatemente os elementos com índices de linhas menores do que índices de colunas na coluna $k+1$, o que faz com que a Invariante 3 se mantenha ao incrementarmos o valor de k. As outras duas invariantes se mantém trivialmente neste caso.

Além disso, o algoritmo, a cada passo, incrementa k ou remove uma coluna de A. Sabemos que k nunca passa de n e, já que a matriz tem m colunas, não podemos remover mais do que m colunas. Supondo que a cada remoção de coluna k seja decrementado, chegamos a uma quantidade máxima de $2m+n$ passos. Supondo que as remoções sejam feitas em tempo constante, o tempo de cada passo é constante, portanto, atingimos uma complexidade de $\mathcal{O}(m)$ operações no algoritmo REDUCE, já que $n \leq m$.

4.3. **SMAWK.** Recebemos uma matriz $A \in \mathbb{Q}^{n \times m}$ totalmente monótona convexa por linhas. Primeiramente, vamos transformar a matriz A em uma matriz quadrada. Se A tem mais colunas do que linhas, basta aplicar o algorimto REDUCE em A para fazer com que ela fique quadrada e tenha os mesmos índices de mínimos. Se A tem mais linhas do que colunas, basta adicionar colunas sem que os mínimos ou a total monotonicidade sejam prejudicados, para isso, adicionamos, ao final da matriz, colunas $n - m$ com entradas infinitas.

Agora estamos prontos para descrever e aplicar o algoritmo SMAWK na matriz modificada. Vamos misturar as ideias apresentadas da técnica primordial, apresentada na Subseção 4.1, e do Reduce, apresentado na Subseção 4.2. Em cada passo, removemos as linhas pares da matriz, aplicamos o algoritmo Reduce para manter esta nova matriz quadrada e resolvemos o problema recursivamente para a nova matriz. Com a solução desta instância, descobrimos os resultados para as linhas restantes da matriz original. O Algoritmo 4.4 descreve este processo.

4.4. **Análise.** O tempo gasto pelo algoritmo SMAWK depende apenas da dimensão n da matriz recebida. Escrevemos $T(n)$ a recorrência que define o tempo gasto pelo algoritmo para todo $n \geq 1$. Sabemos que $T(1) = \mathcal{O}(1)$. Com $n > 1$, a retirada de linhas pares será implementada em tempo constante, o algoritmo Reduce é, então, aplicado a uma matriz com $\lfloor n/2 \rfloor$ linhas e n colunas, gastando tempo $\mathcal{O}(n)$ e depois os máximos das lunhas ímpares de A são achados à partir das linhas pares de A na forma descrita na Subseção 4.1, o que custa tempo $\mathcal{O}(n)$. Assim, para todo $n > 1$, $T(n) = \mathcal{O}(n) + T(\lfloor n/2 \rfloor)$, o que nos leva a $T(n) = \mathcal{O}(n)$.

Se a matriz recebida tiver menos colunas do que linhas, a transformação inicial custa tempo $\mathcal{O}(n)$, no outro caso, custa tempo $\mathcal{O}(m)$, onde m é a quantidade de colunas. Assim, podemos escrever a complexidade no caso geral como $\mathcal{O}(n + m)$.

4.5. **Implementação.** Queremos encontrar uma maneira eficiente de remover as linhas pares da matriz, mas não podemos gerar explicitamente uma nova matriz. Queremos representar, a cada passo, todas as linhas que podem ser visitadas. Se k é um inteiro não negativo arbitrário, as linhas da matriz são da forma $1+k$, as linhas visitáveis após a retirada de todas as pares são da forma $1+2k$, as visitáveis depois de duas remoções são da forma $1 + 4k$ e assim por diante, ou seja, depois de t remoções de linhas pares, podemos visitar as linhas da forma $1 + 2^t k$. Assim, basta guardar o inteiro $p = 2^t$ para representar todas as linhas que podem ser visitadas pelo algoritmo em um dado passo. Remover todas as linhas pares é dobrar o valor de p.

Agora, precisamos representar as colunas visitáveis em A. Já que não há uma regra fixa para a remoção de colunas, precisamos de alguma estrutura de dados que nos permita iterar pelos seus valores em ordem e remover um valor eficientemente sempre que visitado. Guardaremos uma lista duplamente ligada com todos os índices de colunas válidos, já que iteramos pelas colunas e, quando removemos uma coluna, ela é sempre vizinha da atual ou a atual, as remoções são feitas em $\mathcal{O}(1)$. Após resolver o problema recursivamente, precisamos recuperar as informações desta lista ligada ao início da iteração para podermos descobrir os valores de mínimo nas linhas ímpares daquela matriz,

Algoritmo 4.4 Algoritmo SMAWK

```
função SMAWK(A)
    se A tem uma linha então
        A é uma matriz 1 × 1 e a resposta é trivial
    senão
        Retiramos as linhas pares de A gerando A′
        A″ ← Reduce(A′)
        SMAWK(A″)
        para i linha ímpar de A faça
            l ← 1 e r ← m
            se i > 1 então
                l ← índice de mínimo da linha i − 1
            se i < n então
                r ← índice de mínimo da linha i + 1
            Busca o índice de mínimo da linha i entre l e r, inclusive
```

para isso, basta, ao começo de cada passo, criar uma cópia da lista ligada original, o que é feito em $\mathcal{O}(n)$ e não afeta a análise do tempo do algoritmo.

No início do algoritmo, precisamos gerar a lista ligada original e, caso haja mais colunas do que linhas, aplicar uma vez o algoritmo Reduce. Caso a quantidade de linhas seja maior do que a de colunas, precisamos criar uma nova matriz com colunas a mais do que a original. Já que nossas matrizes são representadas por funções, suponha que o algoritmo recebe uma função f e dois inteiros n, quantidade de linhas, e m, quantidade de colunas. Podemos criar uma função h definida, para todo $1 \leq i, j \leq n$ como $f(i, j)$ se $j \leq m$ e $+\infty$ caso contrário e substituir f por esta no restante do algoritmo.

A implementação em C++ do algoritmo apresentado, levando em conta as considerações acima, pode ser encontrada em `implementacao/SMAWK.cpp`.

4.6. **Aplicações.** O problema apresentado na Subseção 3.4 foi resolvido utilizando a técnica da divisão e conquista, porém, as matrizes para as quais aplicamos a técnica naquele exemplo são Monge convexas, portanto, totalmente monótonas convexas nas linhas, já que estavamos interessados em mínimos de linhas podemos aplicar o SMAWK ao invés da divisão e conquista para resolver estes problemas, conseguindo uma solução $\mathcal{O}(kn)$.

Como mencionado no início desta seção, a técnica apresentada aqui pode ser usada para resolver o problema de encontrar todos os pares de pontos mais distantes num polígono convexo em tempo $\mathcal{O}(n)$. Além disso, Aggarwal [3] mostrou a aplicação deste algoritmo em vários problemas de geometria computacional.

5. Otimização de Knuth-Yao

O problema da árvore de busca binária ótima [7] é um exemplo clássico de aplicação de programação dinâmica que é fácilmente resolvido em tempo $\mathcal{O}(n^3)$ com uma modelagem que associa o custo de cada subárvore a uma entrada de uma matriz, reduzindo o problema a calcular estas entradas. Aproveitando algumas propriedades da matriz, Knuth [10] apresentou uma solução que calcula suas entradas em tempo $\mathcal{O}(n^2)$, resolvendo o problema original nesta complexidade.

Mais tarde, a solução de Knuth foi estudada por Yao [11, 12] que mostrou que as propriedades observadas por Knuth eram consequência do fato de que a matriz de interesse era Monge convexa. Desta maneira, foi possível perceber que a otimização de Knuth poderia ser útil em vários outros problemas de programação dinâmica.

Bein, Golin, Larmore e Zhang [4] buscaram enfraquecer a condição encontrada por Yao e mostraram que as matrizes descritas pelos problemas agilizados com a otimização Knuth-Yao podem ser decompostas de 3 maneiras diferentes em matrizes totalmente monótonas. Esta introdução foi baseada no artigo citado neste parágrafo.

Vamos discutir os resultados observados por Yao, descrever a técnica desenvolvida por Knuth e aplicar este conhecimento para resolver um problema de programação dinâmica.

5.1. **Definições básicas.** Vamos apresentar a otimização de Knuth-Yao em termos de problemas de minimização, o que nos leva a trabalhar com convexidade. É fácil adaptar o conhecimento discutido nesta seção para problemas de maximização, porém, ao invés da convexidade, a concavidade deve ser usada para provar os resultados e modelar os problemas de interesse.

Definição 5.1 (Recorrência de intervalos)**.** *Uma matriz $A \in \mathbb{Q}^{n\times n}$ é considerada uma recorrência de intervalos se existe uma matriz $C \in \mathbb{Q}^{n\times n}$ tal que, para todo $i, j \in [n]$,*

$$A[i][j] = \begin{cases} C[i][j] & \text{, se } i = j, \\ C[i][j] + \min\{A[i][k-1] + A[k][j] \mid i < k \le j\} & \text{, se } i < j \text{ e} \\ +\infty & \text{caso contrário.} \end{cases}$$

A matriz C é chamada matriz de custos de A.

É fácil resolver uma recorrência desta forma em tempo $\mathcal{O}(n^3)$.

Definição 5.2 (Matriz de cortes ótimos)**.** *Se $A \in \mathbb{Q}^{n\times n}$ é uma recorrência de intervalos com matriz de custos C, definimos a matriz de cortes ótimos P de A. Para todo $i \in [n]$, $P[i][i] = i$ e para todo $j \in [n]$ com $i < j$,*

$$P[i][j] = \min\{k \mid i < k \le j \text{ e } A[i][j] = C[i][j] + A[i][k-1] + A[k][j]\}.$$

Assim, a matriz P guarda, para cada $i < j$, o menor argumento para o qual a função de mínimo na definição de $A[i][j]$ atinge seu valor ótimo. Note que, enquanto descobrimos os valores da matriz A em tempo $\mathcal{O}(n^3)$, descobrimos também os valores de P.

Definição 5.3 (Knuth-Yao otimizável)**.** *Se $A \in \mathbb{Q}^{n\times n}$ é uma recorrência de intervalos e P é sua matriz de cortes ótimos. Dizemos que A é Knuth-Yao otimizável se, para todo $i, j \in [n]$ com $i < j$, vale $P[i][j-1] \le P[i][j] \le P[i+1][j]$.*

Vamos mostrar que se $A \in \mathbb{Q}^{n\times n}$ é Knuth-Yao otimizável, tanto A quanto sua matriz de cortes ótimos P podem ser calculados em $\mathcal{O}(n^2)$.

Algoritmo 5.4 Otimização Knuth-Yao

1: **função** KNUTHYAO(C, n)
2: $\quad A \in \mathbb{Q}^{n\times n}$ e $P \in \mathbb{N}^{n\times n}$
3: $\quad$ **para** i de 1 até n **faça**
4: $\quad\quad A[i][i] \leftarrow C[i][i]$
5: $\quad\quad P[i][i] \leftarrow i$
6: $\quad$ **para** d de 1 até $n-1$ **faça**
7: $\quad\quad$ **para** i de 1 até $n-d$ **faça**
8: $\quad\quad\quad j \leftarrow i + d$
9: $\quad\quad\quad A[i][j] \leftarrow +\infty$
10: $\quad\quad\quad$ **para** k de $P[i][j-1]$ até $P[i+1][j]$ **faça**
11: $\quad\quad\quad\quad v \leftarrow C[i][j] + A[i][k-1] + A[k][j]$
12: $\quad\quad\quad\quad$ **se** $v < A[i][j]$ **então**
13: $\quad\quad\quad\quad\quad A[i][j] \leftarrow v$
14: $\quad\quad\quad\quad\quad P[i][j] \leftarrow k$

5.2. **Técnica.** Seja $A \in \mathbb{Q}^{n\times n}$ uma matriz Knuth-Yao otimizável. Vamos calcular as entradas $A[i][j]$ onde $i \leq j$ em ordem crescente de $j - i$, ou seja, as entradas $A[i][i]$ serão calculadas para todo i, seguidas das $A[i][i+1]$, $A[i][i+2]$ e assim por diante. É possível calcular as entradas nesta ordem pois ela respeita as relações de dependência da matriz A, isto é, ao calcular uma entrada $A[i][j]$ qualquer, todas as entradas $A[i][k-1]$ e $A[k][j]$ com $i < k \leq j$ já estarão disponíveis e, portanto, será possível descobrir o valor de $A[i][j]$.

Se calcularmos também as entradas da matriz P enquanto calculamos as da A, poderemos aproveitar o fato de que A é Knuth-Yao otimizável para buscar o valor de uma entrada de A em um intervalo menor do que o trivial, isto é, podemos escrever, para todo $i, j \in [n]$ com $i < j$, a igualdade

$$A[i][j] = C[i][j] + \min\{A[i][k-1] + A[k][j] \mid i < k \leq j \text{ e } P[i][j-1] \leq k \leq P[i+1][j]\}. \tag{5.5}$$

Esta observação induz o Algoritmo 5.4 para calcular as entradas das matrizes A e P. Perceba que na linha 10 a variável k varia de $P[i][j-1]$ até $P[i+1][j]$ e não de $\max(i+1, P[i][j-1])$ até $\min(j, P[i+1][j])$ como indica a igualdade (5.5). Primeiramente, perceba que $P[i+1][j] \leq j$, portanto $\min(j, P[i+1][j]) = P[i+1][j]$. Além disso, sabemos $P[i][j-1] \geq i$, separamos em dois casos, no primeiro $P[i][j-1] > i$ e vale $\max(i+1, P[i][j-1]) = P[i][j-1]$. No outro, onde $P[i][j-1] = i$, a iteração onde $k = i$ ocorre indevidamente. Note que quando $k = i$, $i > k-1$ e $A[i][k-1] = +\infty$, portanto $C[i][j] + A[i][k-1] + A[k][j] = +\infty$ e esta iteração será irrelevante para a resposta final do algoritmo, o que nos permite realizar a iteração $k = i$ sem problemas.

5.3. **Análise.** Vamos analisar a complexidade do Algoritmo 5.4. Podemos escrever a quantidade de iterações do laço da linha 10 como

$$\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=P[i][i+d-1]}^{P[i+1][i+d]} 1 = \sum_{d=1}^{n-1}\sum_{i=1}^{n-d} P[i+1][i+d] - P[i][i+d-1] + 1, \tag{5.6}$$

com um d fixo, a soma $\sum_{i=1}^{n-d} P[i+1][i+d] - P[i][i+d-1]$ é uma soma telescópica e tem valor igual a $P[n-d+1][n] - P[1][1+d] = \mathcal{O}(n)$, com isso, escrevemos (5.6) como $\sum_{d=1}^{n-1} \mathcal{O}(n) + n - 1 = \mathcal{O}(n^2)$.

5.4. **Quebrando strings.** Para exemplificar a otimização de Knuth e apresentar a relação das matrizes Monge com as definições da Subseção 5.1, iremos resolver um outro problema clássico de programação dinâmica [7, Exercício 15-9] disponível no juíz online SPOJ em `http://www.spoj.com/problems/BRKSTRNG/`.

Considere uma linguagem de processamento de strings que consegue quebrar uma string s de tamanho $m > 1$ em qualquer posição $t \in [m-1]$, ou seja, gerar duas strings $s[1\,..\,t]$ e $s[t+1\,..\,m]$. Um programador quer usar esta linguagem para separar uma string n vezes, nas posições $p_1 < p_2 < \cdots < p_n$, porém, para quebrar uma string de tamanho m em qualquer posição, a linguagem gasta tempo m. Queremos descobrir qual é a melhor ordem para realizar estes cortes.

Suponha, por exemplo, que estamos interessados em quebrar uma string `stringdeexemplo` de tamanho 15 nas posições 6 e 8 para gerar as strings `string`, `de` e `exemplo`, isso pode ser realizado de duas maneiras. Uma maneira é quebrar primeiro na posição 8 gerando as strings `stringde` e `exemplo` e depois na 6, gerando as 3 strings desejadas e a outra maneira é quebrar primeiro na posição 6 gerando `string` e `deexemplo` e depois na posição 8. A primeira opção tem custo $15 + 8$ enquanto a segunda tem custo $15 + 9$, o que faz a resposta ótima ser a primeira alternativa.

Dados os valores n, m e os pontos $p_1 < p_2 < \cdots < p_n$ dos cortes desejados, chamamos de s a string que desejamos separar e definimos, por conveniência, $p_0 = 0$ e $p_{n+1} = m$. Assim, se $A \in \mathbb{Q}^{n\times n}$ guarda em toda posição $A[i][j]$ com $i \leq j$ a melhor solução para o subproblema que recebe a string $s[p_{i-1} + 1\,..\,p_{j+1}]$ e as posições de corte $p_i, p_{i+1}, \ldots, p_j$ como entrada, podemos concluir facilmente que A é uma matriz de recorrência de intervalos com matriz de custo C onde $C[i][j] = p_{j+1} - p_{i-1}$. O valor de $A[1][n]$ nos dará o tempo mínimo de concluir a tarefa desejada e a ordem ótima das quebras pode ser reconstruída através da matriz de cortes ótimos de A.

Como observado anteriormente, se A é uma recorrência de intervalos ela pode ser calculada em tempo $\mathcal{O}(n^3)$. Pretendemos aproveitar propriedades da matriz C e alguns resultados provados por Yao [11] para concluir que A é Knuth-Yao otimizável e aplicar o Algoritmo 5.4 para calcular A em tempo $\mathcal{O}(n^2)$. Vamos começar provando que C é uma matriz Monge convexa.

Demonstração. Sejam $i, j \in [n-1]$ quaisquer. Temos

$$\begin{aligned} C[i][j] + C[i+1][j+1] &= p_{j+1} - p_{i-1} + p_{j+2} - p_i \\ &= p_{j+1} - p_i + p_{j+2} - p_{i-1} \\ &= C[i+1][j] + C[i][j+1]. \end{aligned}$$

Com isso vale que $C[i][j] + C[i+1][j+1] \leq C[i+1][j] + C[i][j+1]$ e usamos o Teorema 2.12 para concluir que C é Monge convexa. □

Definição 5.7 (Monótona nos intervalos). *Uma matriz $C \in \mathbb{Q}^{n\times n}$ é monótona nos intervalos se para todo $i, i', j, j' \in [n]$ onde $i \leq i' \leq j \leq j'$, vale*

$$C[i'][j] \leq C[i][j'].$$

A Definição 5.7 relaciona a distância entre os índices de linha e coluna da matriz com o valor da matriz. Aplicando ao problema discutido nesta subseção, dizer que a matriz C é monótona nos intervalos é equivalente a dizer que quanto maior a string que está sendo cortada, mais caro o corte. Vamos provar que esta propriedade vale.

Demonstração. Sejam $i, i', j, j' \in [n]$ tais que $i \le i' \le j \le j'$, vale $C[i'][j] = p_{i'+1} - p_{j-1}$, já que $p_{i'+1} \ge p_{i+1}$ e $-p_{j-1} \ge -p_{j'-1}$, temos $p_{i'+1} - p_{j-1} \ge p_{i+1} - p_{j'-1} = C[i][j']$. $\square$

Lema 5.8. *Se $A \in \mathbb{Q}^{n\times n}$ é uma recorrência de intervalos com matriz de custos C Monge convexa e monótona nos intervalos, então A é Monge convexa.*

Demonstração. Sejam A e B matrizes que respeitam as condições do enunciado e P a matriz de cortes ótimos de A. Sejam ainda i, i', j e $j' \in [n]$ onde $i \le i'$ e $j \le j'$. Queremos mostrar que sempre vale a desigualdade de Monge, isto é $A[i][j] + A[i'][j'] \le A[i][j'] + A[i'][j]$. Definimos $l = j' - i$ e usaremos indução em l. Se $l < 0$, vale $j > i'$ e, portanto $A[i'][j] = +\infty$ o que faz valer a desigualdade.

Fixamos $l \ge 0$ assumindo que a desigualdade vale nos casos onde $j' - i < l$. Note que se $i = i'$ ou $j = j'$ a desigualdade vale trivialmente, bem como no caso onde $j > i'$, analisado acima. Podemos assumir $i < i' \le j < j'$ e separamos isso em dois casos, quando $i' = j$ e quando $i' < j$.

Olhamos para o caso onde $i' = j$. Seja $x = P[i][j']$ o ponto de corte ótimo do estado $A[i][j']$, ou seja, $A[i][j'] = C[i'][j] + A[i'][x-1] + A[x][j]$. Assumimos que $x \le j$ e teremos

$$
\begin{aligned}
A[i][j] + A[i'][j'] - A[i'][j] &= A[i][j] + A[j][j'] - A[j][j] \\
&\le C[i][j] + A[i][x-1] + A[x][j] + A[j][j'] - A[j][j] && (5.9)\\
&\le C[i][j] + A[i][x-1] + A[x][j'] && (5.10)\\
&\le C[i][j'] + A[i][x-1] + A[x][j'] && (5.11)\\
&= A[i][j']
\end{aligned}
$$

onde (5.9) vale pois $i < x \le j$ e $A[i][j] = C[i][j] + \min_{i<k\le j}(A[i][k-1] + A[k][j])$, a desigualdade (5.10) vale pois temos $x \le j \le j \le j'$ e $j' - x < l$, portanto, pela hipótese de indução, vale $A[x][j] + A[j][j'] \le A[x][j'] + A[j][j]$ e, finalmente, também vale (5.11) pois C é monótona nos intervalos. Com isso, provamos $A[i][j] + A[i'][j'] \le A[i'][j] + A[i][j']$. No caso onde $x > j$, basta utilizar $A[j][j'] \le C[j][j'] + A[j][x-1] + A[x][j']$ ao invés do que foi utilizado em (5.9) e adaptar os passo (5.10) para aplicar a hipótese de indução em $i \le j \le j \le x-1$.

Agora, vamos resolver o caso onde $i' < j$. Definimos os pontos ótimos de corte $x = P[i][j']$ e $y = P[i'][j]$ e seguimos um raciocício parecido com o caso anterior. Assumindo que $x \le y$, temos

$$
\begin{aligned}
A[i][j] + A[i'][j'] &\le C[i][j] + C[i'][j'] + A[i][x-1] + A[x][j] + A[i'][y-1] + A[y][j'] && (5.12)\\
&\le C[i][j] + C[i'][j'] + A[x][j'] + A[y][j] + A[i][x-1] + A[i'][y-1] && (5.13)\\
&\le C[i][j'] + A[i][x-1] + A[x][j'] + C[i'][j] + A[i'][y-1] + A[y][j] && (5.14)\\
&= A[i][j'] + A[i'][j]
\end{aligned}
$$

onde (5.12) se justifica pois $i < x \le j$ e $i' < y \le j'$, a hipótese de indução é aplicada em (5.13) pois $x \le y \le j \le j'$ e $j' - x < l$ e, por fim, o passo (5.14) é válido pelo fato de que C é Monge convexa. O caso onde $x > y$ é similar, basta observar que vale $i' < x \le j'$ e $i < y \le j$ o que implica em $A[i][j] + A[i'][j'] \le C[i][j] + C[i'][j'] + A[i][y-1] + A[y][j] + A[i'][x-1] + A[x][j']$ e substituir o passo (5.12) por essa desigualdade, adaptando o passo (5.13) de acordo, usando indução em $i \le i' \le y-1 \le x-1$. $\square$

Com o Lema 5.8 percebemos que a matriz A que representa o nosso problema atual é Monge convexa.

Lema 5.15. *Uma recorrência de intervalos A Monge convexa é Knuth-Yao otimizável.*

Demonstração. Seja $A \in \mathbb{Q}^{n \times n}$ uma matriz Monge convexa que é uma recorrência de intervalos com matriz de cortes ótimos P. Queremos mostrar que para todo $i, j \in [n]$ com $i < j$ vale $P[i][j-1] \leq P[i][j] \leq P[i+1][j]$. Observe que o caso onde $i + 1 = j$ é trivial e assuma $i + 1 < j$.

Sejam $x, y \in [n]$ tais que $x < y$. Vamos mostrar que se $A[i][x-1] + A[x][j] \leq A[i][y-1] + A[y][j]$ então $A[i][x-1] + A[x][j-1] \leq A[i][y-1] + A[y][j-1]$. Assumindo a primeira desigualdade, vale $A[i][x-1] - A[i][y-1] \leq A[y][j] - A[x][j]$ e já que $x < y$ e $j - 1 < j$, usando a Monge convexidade de A, temos $A[x][j-1] - A[y][j-1] \leq A[x][j] - A[y][j]$ e, com isso, obtemos $A[i][x-1] + A[x][j-1] \leq A[i][y-1] + A[y][j-1]$. O que conclui a prova da implicação proposta. Com isso, existir algum par $i, j \in [n]$ onde $i < j$ tal que $y = P[i][j-1] > P[i][j] = x$ implica em $A[i][x-1] + A[x][j] \leq A[i][y-1] + A[y][j]$ e $A[i][x-1] + A[x][j-1] > A[i][y-1] + A[y][j-1]$, contrariando o que foi provado neste parágrafo. Concluímos que $P[i][j-1] \leq P[i][j]$.

De forma simétrica, para mostrar que $P[i][j] \leq P[i+1][j]$ basta perceber que se $x, y \in [n]$ e $x > y$, será verdade que $A[i][x-1] + A[x][j] < A[i][y-1] + A[y][j]$ implica $A[i+1][x-1] + A[x][j] < A[i+1][y-1] + A[y][j]$, o que contradiz $P[i][j] > P[i+1][j]$, provando $P[i][j] \leq P[i+1][j]$. □

Com os Lemas 2.11 e 5.15 vale que A é Knuth-Yao otimizável e podemos aplicar a técnica de Knuth-Yao para resolver o problema em tempo $\mathcal{O}(n^2)$.

Referências

[1] https://www.quora.com/What-is-divide-and-conquer-optimization-in-dynamic-programming, May 2017.

[2] http://codeforces.com/blog/entry/8219, May 2017.

[3] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.

[4] Wolfgang Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The knuth-yao quadrangle-inequality speedup is a consequence of total monotonicity. *ACM Trans. Algorithms*, 6(1):17:1–17:22, December 2009.

[5] Peter Brucker. Efficient algorithms for some path partitioning problems. *Discrete Applied Mathematics*, 62(1):77 – 85, 1995.

[6] Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of monge properties in optimization. *Discrete Applied Mathematics*, 70(2):95 – 161, 1996.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[9] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49 – 76, 1992.

[10] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[11] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 429–435, New York, NY, USA, 1980. ACM.

[12] F Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540, 1982.