Relatório para o Problema de Programação 2 - ARChitecture

Equipa:
N.º de Estudante: 2018285941 Nome: Cláudia Filipa Peres Saraiva de Campos
N.º de Estudante: 2018275530 Nome: Dário Filipe Torres Félix

1. Descrição do Algoritmo

$\boldsymbol{Function}\ arch(n, h, H)$
$\quad HMax \leftarrow min\big(h + \big((h-1) \times (n-1)\big), H\big) - (h-1)$
$\quad nMax \leftarrow min(1\ +\ (2 \times (HMax - 1)), n)$
$\quad M \leftarrow \boldsymbol{new}\ matrix[HMax, nMax]\ \boldsymbol{of\ integer}$

$\quad \boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ HMax - 1\ \boldsymbol{do}$ $\qquad\qquad$ $//\ Place\ 1st\ lego\ on\ the\ ground$
$\qquad M[i, 1] \leftarrow 0$
$\quad M[Hmax, 1] \leftarrow 1$

$\quad \boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ nMax\ \boldsymbol{do}$ $\qquad\qquad$ $//\ Up$
$\qquad M[HMax, j] \leftarrow 0$
$\qquad \boldsymbol{for}\ i \leftarrow HMax - 1\ \boldsymbol{downto}\ 1\ \boldsymbol{do}$
$\qquad\quad M[i, j] \leftarrow modAdd(M[i+1, j-1], M[i+1, j], 1000000007)$
$\qquad\quad \boldsymbol{if}\ i + h \leq Hmax\ \boldsymbol{then}$
$\qquad\qquad M[i, j] \leftarrow modSub(M[i, j], M[i+h, j-1], 1000000007)$

$\quad \boldsymbol{for}\ j \leftarrow 3\ \boldsymbol{to}\ nMax\ \boldsymbol{do}$ $\qquad\qquad$ $//\ Down$
$\qquad cache \leftarrow 0$
$\qquad \boldsymbol{for}\ i \leftarrow 2\ \boldsymbol{to}\ HMax\ \boldsymbol{do}$
$\qquad\quad cache \leftarrow modAdd(cache, M[i-1, j-1], 1000000007)$
$\qquad\quad \boldsymbol{if}\ i\ -\ h\ \geq 1\ \boldsymbol{then}$
$\qquad\qquad cache \leftarrow modSub(cache, M[i-h, j-1], 1000000007)$
$\qquad\quad M[i, j] \leftarrow modAdd(M[i, j], cache, 1000000007)$
$\qquad M[HMax, j] \leftarrow modAdd(M[HMax, j-1], M[HMax, j], 1000000007)$

$\quad \boldsymbol{return}\ M[Hmax, nMax]$

1.1. Programação Dinâmica Bottom-Up
1.1.1. Subproblema

Para obter a solução para o problema composto por n, h, H, necessitamos do subproblema da coluna anterior dado por $n - 1$, h, H até $n = 3$.

1.1.2 Memoization

Ao longo da execução, guardamos os resultados intermédios numa matriz que servirão para calcular os problemas seguintes.

1.2. Otimizações
1.2.1. $HMax$

Se o n for muito restrito, haverá casos em que não necessitaremos de todo o H, e, dependendo de h, fazemos um cálculo do H máximo ($HMax$)

necessário sem restrições: $h + \left((h-1) \times (n-1)\right)$, que é o cálculo estimado da altura máxima que a configuração dada por h e n pode atingir.

A primeira coluna tem sempre altura máxima h, uma vez que corresponde ao 1º bloco que se encontra junto ao solo.

Por cada coluna adicional $(n-1)$, somamos $h-1$, uma vez que é o máximo que um bloco pode subir partilhando uma célula com o bloco anterior.

Calculamos o mínimo entre este H máximo necessário e o H originalmente lido. Por fim, subtraímos $h-1$ ao valor resultante, para ignorar os valores abaixo da altura h, uma vez que nenhum bloco pode passar abaixo desta altura.

1.2.2. $nMax$

Se o H for muito restrito, haverá casos em que não necessitaremos de todo o n, e, fazemos um cálculo do n máximo necessário com restrições: $1 + (2 \times (HMax - 1))$, que é o cálculo estimado do número de blocos n que a configuração dada por $HMax$ permite.

O primeiro bloco é contabilizado (1 coluna) e por cada altura adicional até $HMax$ necessitaremos de 2 colunas.

Calculamos o mínimo entre este n necessário e o n originalmente lido.

1.2.3. Ciclos for's

Saltamos algumas iterações à frente (linhas ou colunas) uma vez que o seu cálculo é irrelevante para a solução do problema.

1.2.4. Soma dos últimos $nMax - 2$ elementos da última linha da matriz

A cada chamada do algoritmo este tem de calcular a soma de todos os arcos possíveis com até $nMax$ blocos. Para evitar percorrer novamente a matriz, reaproveitamos o segundo ciclo *for* imbricado para calcular a soma cumulativa da e na sua última linha. Bastando apenas devolver o último elemento da matriz.

1.2.5. Memória Dinâmica

Decidimos optar pelo *malloc* ao invés do *vector* da biblioteca standard do C++ pelas seguintes razões:

Não necessitamos de inicializar a matriz com valores nulos normalizados.

Tirar partido do princípio da localidade espacial do *array*, porque a matriz vai ser armazenada na memória de forma contínua, evitando *cache misses* durante a execução no CPU.

1.2.6. Remoção do 3.º ciclo for utilizando a semi-soma dos elementos vizinhos

Inicialmente necessitávamos de um terceiro ciclo for que calculava a soma dos $h-1$ elementos da coluna anterior para um determinado elemento da matriz. No entanto, esta solução causava bastante *overhead* desnecessariamente, por este motivo, desenvolvemos uma otimização que permite este cálculo em tempo constante:

Considerando a descida (raciocínio análogo para a subida):

Para calcular o elemento $M[i][j]$, somamos o elemento anterior da mesma coluna, $M[i-1][j]$, com o elemento anterior da coluna anterior, $M[i-1][j-1]$. Dado que o elemento $M[i-1][j]$ já contém a soma dos $h-1$ elementos contíguos da coluna anterior a esse, necessitamos de subtrair o $h-ésimo$

elemento anterior da coluna anterior, $M[i-h][j-1]$, para manter o mesmo intervalo de valores $(h-1)$.

2. Estruturas de Dados

A principal estrutura de dados utilizada no algoritmo desenvolvido é uma matriz $H-(h-1)$ por n representada por um vector unidimensional alocado na *heap*. Considerando um elemento qualquer da matriz, $M[i][j]$:

Caso $M[i][j]$ seja igual a zero, significa que o $j-ésimo$ bloco não atinge a altura $i+h$. Caso contrário, se $M[i][j]$ corresponder a um inteiro positivo, significa que o $j-ésimo$ bloco atinge a altura $i+h$, $M[i][j]$ vezes.

3. Verificação do Algoritmo (*Correctness*)

A prova por contradição da propriedade subestrutura ótima:
Dada a sequência ótima $S=(s_1,\dots,s_n)$ para o problema $arch(n,h,H)$. Se s_n for removido de $arch(n,h,H)$ obtemos 1) a solução para o problema $arch(n-1,h,H)$. Vamos provar a afirmação 1:

1. (assunção) Assumir que $arch(n,h,H)$ constrói uma sequência que termina com s_n, que é solução óptima do problema.
2. (negação) Assumimos que $|arch(n-1,h,H)| > |arch(n,h,H)\backslash\{s_n\}|$
3. (consequência) Então adicionar s_n a $arch(n-1,h,H)$ vai originar uma sequência maior que $arch(n,h,H)$, isto é, $|arch(n-1,h,H)\cup\{s_n\}| > |arch(n,h,H)|$
4. (contradição) Porém, isto leva à contradição da afirmação 1

Ou seja, concluímos que $arch(n,h,H)\backslash\{s_n\}$ tem de ser igual a $arch(n-1,h,H)$.

4. Análise Algorítmica

4.1. Complexidade Espacial

Tendo em conta que n' e H' correspondem ao número optimizado de colunas e linhas da matriz, respetivamente, e, portanto, alocamos memória necessária para armazenar uma matriz n' por $H'-(h-1)$. Apesar das optimizações implementadas, no pior dos casos, H' corresponde a H e n' corresponde a n, daí concluímos que a complexidade espacial do algoritmo é $O(n\times(H-h))$.

4.2. Complexidade Temporal

Analisando o algoritmo desenvolvido em detalhe podemos verificar que o algoritmo realiza 2 passagens pela matriz. Tanto a primeira passagem como a segunda têm uma complexidade de $O(n\times(H-h))$.

Somando estas complexidades obtemos $O(2\times n\times(H-h))$. Como na notação O grande apenas consideramos as funções que contenham a menor ordem possível e sem qualquer constante associada, por isso concluímos que a complexidade temporal do algoritmo desenvolvido corresponde a $O(n\times(H-h))$.

5. Referências

Slide da disciplina de Estratégias Algorítmicas. 😉