Report for Programming Problem 2 - ARChitecture

Team:

Student ID: 2018285164 Name: Eduardo Cruz

Student ID: 2018298209 Name: Rodrigo Sobral

1. Algorithm description

No nosso algoritmo efetuamos n-4 chamadas (3<=k<=n) a uma função *arcs_for_k_blocks(...)*, onde, para cada k blocos, efetuamos o cálculo de combinações possíveis a partir das combinações obtidas para k-1 blocos. Este cálculo é efetuado através da iteração, 2 vezes, de um array onde estão descritos blocos cujo sentido do movimento é ascendente (UP), e é também percorrido, 1 vez, um array que contém informação (a detalhar em 2.) relativo a blocos cujo sentido do movimento é descendente (DOWN). Da primeira iteração do array UP, resulta um novo array com informação relativa à quantidade e altura de blocos no sentido ascendente que resultam dos blocos descritos nesse array percorrido. Depois é percorrido o array DOWN e os respetivos blocos que resultam destes, em sentido descendente, são calculados e a sua informação é armazenada num novo array. Por fim, voltamos a percorrer o array UP para calcularmos os blocos em sentido descendentes que podem resultar destes em sentido ascendente. Este cálculo é feito sucessivamente para cada k blocos, através de uma soma cumulativa dos valores lidos, e o *counter* de combinações possíveis distintas vai sendo incrementado de acordo com os resultados obtidos. Além disso, em cada chamada da função *arcs_for_k_blocks(...)* é também calculada, inicialmente, a altura máxima possível que a base de um bloco pode assumir, dada a distância a que se encontra de *n*, de modo a que seja preservada a condição obrigatória de 'fechar' o arco, ou seja, é calculada o limite de altura a partir da qual, todos os blocos com uma altura superior a esse limite, não são, nem produzem blocos, com soluções possíveis. Assim conseguimos limitar o tamanho, que percorremos, dos arrays mencionados acima.

2. Data structures

Como foi mencionado em 1. utilizámos 2 arrays, em que cada índice i representa uma altura i+1 da base de um bloco, em que cada valor armazenado no índice i representa representa a quantidade de blocos existentes nessa altura i+1. Na realidade, ao nível da implementação, foi criado apenas um array a que chamamos *dp*, que, durante o programa, foi acedido e lido como se de uma matriz se tratasse, mas, na realidade podemos considerar que se trata de 2 arrays, um que permite armazenar, para cada altura, a respectiva quantidade de blocos em movimento descendente e outro array que armazena essa mesma informação para os blocos em movimento ascendente.
De notar que para cada chamada da função não-recursiva *arcs_for_k_blocks(...),* é passada como argumento o array dp e um array new_dp. Este array new_dp é exatamente igual à estrutura descrita acima, e é nesta estrutura que vão ser guardados os valores resultantes da leitura da

array dp passada. Desta função resultará a new_dp, preenchida para k blocos, que é depois utilizada como dp, na chamada da função seguinte, para k+1 blocos, da qual resultará uma nova new_dp. Estes *arrays* permitem-nos seguir uma abordagem de *memoization*. Para além destas estruturas usamos também variáveis tais como inteiros, para armazenarmos alguns cálculos intermédios como os cálculos de delimitações (inferior e superior) dos blocos ou o cálculo do limite de altura máxima, por exemplo.

3. Correctness

Nas primeiras submissões ficámos presos nos 140 porque a nossa solução não só era recursiva mas também porque o nosso algoritmo concentrava-se muito na ideia inicial dum array que armazena as alturas dos blocos na construção, no entanto, este algoritmo falhava redondamente em situações extremas, nas quais tínhamos os parâmetros extremos (n=500, h=500, H=60000). Daqui partimos também para uma solução iterativa, já com as estruturas que descrevemos em 2., no entanto, o cálculo a partir da *dp* tinha uma complexidade temporal de O(n^2), dado que percorríamos a *dp* e para cada altura do índice lido, calculávamos os limites inferiores e superiores e através de um ciclo for, incrementávamos na new_dp, a quantidade de blocos para essa altura, o que, para parâmetros de teste maiores, continuava a demostrar-se pouco eficiente.
A nossa solução foi capaz de chegar aos 200 pontos no momento em que passámos a ter uma abordagem de complexidade linear, no cálculo da new_dp. As operações realizadas são computacionalmente leves (maioritariamente lógicas/aritméticas e atribuições), e a solução completamente iterativa. O algoritmo que nos permitiu obter 200 pontos é extremamente otimizado, principalmente para parâmetros de teste elevados, comparativamente com os algoritmos explorados por nós antes desta solução final.

4. Algorithm Analysis

Para cada k, percorremos o *array dp* 3 vezes, ou seja, tendo em conta que n é o tamanho percorrido do array, temos que a complexidade temporal algorítmica é de, O(3n) (o que é muito inferior comparativamente à complexidade O(n^2) adquirida com algoritmos anteriores). De notar que que este n varia de k para k dado que, como já mencionámos, para cada k calculamos um limite para a altura máxima, ou seja, o n assume esse limite.
Tendo em conta as estruturas descritas no ponto 2. é possível concluir que a complexidade espacial é dada por O(2*2n). Uma vez utilizamos sempre a *dp* e a *new_dp* temos um ponteiro para cada um desses arrays *array* e desta forma conseguimos reaproveitá-los, apenas substituindo-os entre si.

5. References

Para este trabalho recorremos aos slides teóricos da disciplina, disponibilizados pelo professor, relativos ao conteúdo de Programação Dinâmica.