Lista 5

Victor Sena Molero - 8941317

April 6, 2016

Ex 8. Descreva um algoritmo que, dados n inteiros no intervalo de 1 a k, preprocesse sua entrada e então responda em $O(1)$ qualquer consulta sobre quantos dos n inteiros dados caem em um intervalo $[a..b]$. O preprocessammento efetuado pelo seu algoritmo deve consumir tempo $O(n+k)$.

Resposta. Para resolver o problema em tempo linear podemos, primeiro, inicializar um vetor c de contagem de tamanho $k+1$ (de 0 a k, inclusive) com todos os valores iguais a 0 em tempo $O(k)$. Depois, precisamos percorrer o vetor de entrada v e, para cada valor v_i, somar 1 a c_{v_i}, isso é feito em $O(n)$.
Agora, basta acumular o valor do vetor c nele mesmo, ou seja, percorrer o vetor c de 1 a k efetuando $c_i = c_{i-1} + c_i$, que também custa $O(k)$. Assim, nosso algortimo preprocessa o vetor de maneira conveniente em $O(k) + O(n) + O(k) = O(n+k)$.
Para responder a uma query (a, b) basta imprimir o valor de $c_b - c_{a-1}$. O fato do valor $a-1$ ser consultado justifica a posição 0 no vetor c. Além disso, se não houver garantia de que $1 \leq a, b \leq k$ basta executar, antes de calcular a resposta, $a = min(max(a, 1), k)$ e $b = min(max(b, 1), k)$. □

Algoritmo.

```
function PRE_PROCESSA
    i ← 0
    while i ≤ k do
        c[i] ← 0
        i ← i + 1
    end while
    i ← 1
    while i ≤ n do
        c[v[i]] ← c[v[i]] + 1
        i ← i + 1
    end while
    i ← 1
    while i ≤ k do
        c[i] ← c[i − 1] + c[i]
        i ← i + 1
    end while
end function
```

function CONSULTA(a, b)
  $a = max(min(a, k), 1)$
  $b = max(min(b, k), 1)$
  return $c[b] - c[a-1]$
end function

□

Ex 13. Mostre como multiplicar dois números complexos $a+bi$ e $c+di$ usando apenas três multiplicações reais. O seu algoritmo deve receber como entrada os números a, b, c e d e devolver os números $ac-bc$ (componente real do produto) e $ad+bc$ (componente imaginária do produto).

Algoritmo.
$r_1 \leftarrow (a+b) * (c-d)$
$r_2 \leftarrow b * c$
$r_3 \leftarrow a * d$
return $r_1 - r_2 + r_3, r_2 + r_3$

□

Ex 14. No SELECT-BFPRT, os elementos do vetor são divididos em grupos de 5. O algoritmo continua linear se dividirmos os elementos em grupos de 7? E em grupos de 3? Justifique sua resposta.

Reposta. As respostas são, respectivamente, sim e não.
Na análise do algoritmo SELECT-BFPRT padrão, chegamos em duas fórmulas importantes, $\lceil n/5 \rceil$, que representa a quantidade de intervalos de tamanho 5 ou menos nos quais o vetor original é dividido, e $\lceil 7n/10 \rceil + 3$.
A primeira é fácilmente adpatável se trocarmos a quantidade de elementos em cada intervalo, por exemplo, suponha que queiramos intervalos de k elementos, podemos seguramente substituir aquela fórmula por $\lceil n/k \rceil$.
Para generalizar a segunda, vamos pensar na quantidade mínima de elementos maiores que o pivô escolhido. Cada grupo que tem mediana maior que o pivô (pelo menos $\lfloor 1/2 \lceil n/k \rceil \rfloor$ grupos) contribui com $\lceil k/2 \rceil$ elementos, com exceção de um possível grupo com um só elemento que pode ser maior que o pivô, pelo qual são descontados $\lfloor k/2 \rfloor$ elementos da conta final. Além disso, temos mais $\lfloor k/2 \rfloor$ elementos maiores que o pivô no mesmo intervalo que ele. Assim, a segunda fórmula pode ser generalizada para:

$$n - 1 - (\lfloor 1/2 \lceil n/k \rceil \rfloor)(\lceil k/2 \rceil) \leq$$
$$\leq n - 1 - (1/2)(n/k + 1)((k+1)/2) =$$
$$= n - 1 - ((k+1)/2)(n/2k + 1/2)$$

Para resolver a recorrência para $k = 5$ e provar que $T(n) = O(n)$ chega um momento em que devemos mostrar que, com alguma constante α e outra β e um n suficientemente grande, temos

$$\alpha(\lceil n/5 \rceil) + \alpha(\lceil 7n/10 \rceil + 3) + \beta n < \alpha n$$

Generalizando isso, deveríamos mostrar que

$$\alpha(\lceil n/k \rceil) + \alpha(n - 1 - ((k+1)/2)(n/2k + 1/2)) + \beta n < \alpha n$$

Agora vamos separar os casos onde $k = 3$ e $k = 7$.
Se $k = 3$ temos que mostrar

$$\alpha(\lceil n/3 \rceil) + \alpha(n - 1 - 2(n/6 + 1/2)) + \beta n \leq$$

$$\leq \alpha(n/3 + 1 - 1 + 2n/3 + 1) + \beta n = \alpha(n+1) + \beta n < \alpha n$$

O que é impossível, ou seja, pelos métodos apresentados pelo livro e com os bounds que conseguimos achar, não conseguimos mostrar que T é linear, isso não é uma prova de que T é não linear, mas eu não consegui provar isso, apenas consegui mostrar que, pelos métodos usados no livro, não conseguimos provar.
E, por outro lado, se $k = 7$ temos que mostrar

$$\alpha(\lceil n/7 \rceil) + \alpha(n - 1 - 4(n/14 + 1/2)) + \beta n \leq$$

$$\leq \alpha(n/7 + 1 - 1 + 10n/14 + 2) + \beta n =$$

$$= \alpha(12n/14) + \alpha(2) + \beta n < \alpha(n)$$

E para isso, basta escolher $\alpha = 14$ e $\beta = 1$ para obter

$$13n + 28 < 14n$$

O que é verdade quando $n > 28$, logo, T é linear. □

Ex 15. Descrever o algoritmo pedido no 18 (dando uma ideia de porque funciona e porque consome o tempo pedido).

Proof. Primeiro, temos um vetor de tamanho n com os valores a_i e b_i em cada índice i. Estes valores definem n retas $y = a_i x + b$.
Ordenamos este vetor crescentemente no valor de a_i e, no caso de empate, decrescentemente no tamanho de b_i. Agora podemos montar uma pilha de elementos que guarda, para cada elemento x os valores a_x, b_x e c_x representam uma reta com os valores a_x e b_x e, também, a intersecção entre esta reta e a reta que vem imediatamente abaixo dela na pilha, se esta for a reta mais baixa na pilha, $c_x = -\infty$.
Já que as retas vão ser inseridas ordenadamente na pilha, os valores de a estarão decrescentes na pilha (os menores vão estar mais em cima), podemos inserir nesta pilha de uma maneira esperta para conseguir remover as retas que não serão vistas quando ignoradas as retas que vêm depois da reta atual no vetor, com isso, naturalmente, no passo final, teremos todas as retas que serão vistas quando consideradas todas as retas dadas.
Para isso, basta observar que se a reta x e a reta y são tais que $a_x > a_y$, para todos os pontos à direita da intersecção de x com y, x está acima de y e nos pontos à esquerda da intersecção, y está acima de x. Ou seja, x é visível à partir deste ponto, mas não antes deste, não importa o que vem antes deste ponto, x não será visível lá e não importa o que venha depois deste ponto, y não será visível lá. Se houverem duas retas tais que $a_x = a_y$, então a

reta com b menor nunca será visível e poderá ser ignorada.
Agora, se houver uma reta y que só é visível à partir do ponto c_y e uma outra reta x tal que $a_x > a_y$ tal que a intersecção p entre x e y é tal que $p < c_y$, então y nunca é visível, pois x será maior que y em todos os pontos onde y antes era visível. Portanto, a reta y pode ser ignorada.
Ou seja, nosso algoritmo ordena as retas por ângulo em $O(n \lg n)$ e inicializa uma pilha vazia. Agora, ele percorre os elementos da pilha e vai colocando um por um na pilha, mas antes de inserir o elemento x na pilha, 4 casos diferentes são checados.
Se a pilha estiver vazia, $c_x = -\infty$
Se o topo y da pilha for tal que $a_x = a_y$, então x é ignorado e nunca é inserido (pois $b_x < b_y$, ja que no desempate o b é decrescente)
Se o topo y da pilha se intersectar com x no ponto p e p for tal que $p \leq c_y$, y é removido da pilha e tentamos inserir x novamente (checando essas condições novamente)
Se o topo y da pilha se intersectar com x no ponto p e p for tal que $p > c_y$, então $c_x = p$
Todas as retas x que estiverem na pilha ao final do algoritmo são visíveis do ponto c_x até o ponto c_{x+1}, onde $x+1$ é o elemento acima de x na pilha. Na falta de um elemento $x+1$ (ou seja, se x é o topo da pilha), ele é visível de c_x até ∞.
Vamos contar a quantidade de operações feitas na pilha. A cada tentativa de inserção é necessário buscar o topo da pilha, ou seja, no mínimo, são feitas $O(n)$ operações na pilha. Além disso, elementos são removidos da pilha, porém, cada elemento é removido no máximo uma vez da pilha, logo, são feitas até $O(n)$ remoções. Todas as outras operações estão atreladas a (acontecem se e somente se acontecem) inserções ou remoções e custam todas $O(1)$. Logo, todo o processo custa $O(n)$. Assim, o algoritmo envolve uma ordenação e um processamento $O(n)$, portanto, tem tempo final $O(n \lg n)$.
Existe também uma forma de se implementar isso numa árvore de busca binária em $O(n \lg n)$. Para cada inserção de uma reta x, basta buscar a reta y com a_y imediatamente inferior e ao a_x. Assim, encontra-se a posição de x na pilha, agora, basta checar para as duas retas adjacentes a x se a_x fica em cima delas em alguma parte do intervalo em que elas são visíveis. As operações ficam um pouco mais complicadas com mais corner-cases, mas isso é útil quando não se pode ordenar as retas dadas ao começo do algoritmo (por exemplo, quando não se sabe, à priori, todas as retas que serão dadas).

□