Tópicos Especiais em Algoritmos
Algoritmos Gulosos
Editorial

Daniel Saad Nogueira Nunes

Seletiva UnB 2019: Espetinho do Barbosinha

Este problema pode ser resolvido via uma abordagem gulosa.

Utilizamos dois vetores, S e F, que armazenam respectivamente os tempos (em segundos) de início e fim das estadias de cada cliente.

Após isso, cada um dos vetores é ordenado, independentemente, em ordem crescente.

Em seguida, os vetores S e F são varridos da esquerda para a direita através dos índices i e j:

- Se $S[i] \leq F[j]$, então existe uma sobreposição da pessoa i com a e pessoa j (possivelmente $i = j$). i deve ser incrementado e o número de sobreposições é incrementado. Se for o caso o número de sobreposições global é atualizado.

- Caso contrário, o número de sobreposições é decrementado e o contador de j é incrementado.

A justificativa é a seguinte. Suponha que sabemos o tamanho da sobreposição atual e o tamanho da sobreposição máxima. Caso $S[i] <= F[j]$, então o i-ésimo ponto de início está se sobrepondo com o j-ésimo ponto do fim e o tamanho da sobreposição atual deve ser incrementada, caso ela supere o tamanho da sobreposição global, a última deve ser ajustada. Caso $S[i] > F[j]$, então não há sobreposição entre o intervalo com início em $S[i]$ e o intervalo com fim em $F[j]$, portanto o número de sobreposições atual é decrementado.

Complexidade

Esta solução possui complexidade $\Theta(N \lg N)$ pois é dominada pelos passos de ordenação.

Maratona de Programação contra o COVID-19: Maratonando Cursos

Primeiramente os cursos são ordenados em ordem decrescente pelo conhecimento e, em caso de empate, pela quantidade de semanas de gratuidade do curso, dando preferência pelo curso com mais semanas de gratuidade.

Após esta ordenação, podemos criar um vetor **booleano** de tamanho $M + 1$, que representa a disponibilidade de Patrícia em cada semana. Inicialmente, todos os elementos são verdadeiros.

De acordo com a ordem criada, sejam g_i e v_i a quantidade de semanas de gratuidade e o conhecimento do i-ésimo curso. Caso seja possível enquadrar o i-ésimo curso na semana g_i, isto é feito, caso contrário, tentamos enquadrá-lo na semana $g_i - 1$ e assim sucessivamente. Caso tenha sido possível enquadrar o curso em alguma semana, adicionamos v_i à soma total de conhecimento e atualizamos o vetor de disponibilidade com falso na semana em que foi possível enquadrar o curso. Sempre que tentamos enquadrar um curso em uma semana é preciso verificar se aquela semana já não está ocupada.

Seguindo esta estratégia gulosa, estamos dando prioridade para os cursos que propiciam maior conhecimento e estamos postergando ao máximo estes cursos para que, caso haja um curso com menor tempo de gratuidade mas com valor significativo de conhecimento, ele possa ser *maratonado*.

Complexidade

O algoritmo descrito possui custo $O(N \lg N + NM)$.

Codeforces 124: Lexicographically Maximum Subsequence

O problema pode ser resolvido ao imprimir todos os símbolos z, em seguida todos os y, e assim por diante. Para respeitar a restrição de *subsequência* é necessário que os símbolos y que forem impressos ocorram todos à direita dos símbolos z, os símbolos x que forem impressos ocorram à direita de todos os y e assim em diante.

Para cada símbolo $c \in \{a, \ldots, z\}$, podemos armazenar um vetor V_c com as suas posições de ocorrência na string. Gulosamente, os vetores V_c são

inspecionados com c variando de z a a. Imprime-se todos os z's e guarda-se a última posição de ocorrência de um símbolo z. Em seguida, utilizamos uma busca binária em V_y para verificar se existe algum y que ocorra a direita do último símbolo z na string, os y's que atendem essa propriedade são impressos. O mesmo raciocínio deve ser seguido para os demais símbolos.

Complexidade

O algoritmo descrito possui complexidade de $O(n)$, haja vista que o alfabeto tem tamanho constante e que as buscas binárias sobre todas as sequência V_c tem custo $\Theta(\lg n)$.

AtCoder 167: Bracket Sequencing

A primeira observação que se pode fazer é a seguinte: para uma sequência de parênteses S_i, sempre que um parênteses '(' possui um parênteses ')' à direita, eles podem ser cancelados sem mudar a propriedade da string de ser (ou não) uma *Bracket Sequence*. Ao simplificar (virtualmente) cada uma das strings, a string resultante possui o seguinte formato: $)^a(^b$, isto é, uma sequência de a parênteses de fechamento (com a possivelmente nulo) seguida por uma sequência de b parênteses de abertura (com b potencialmente nulo). São exemplos de strings que seguem este formato:

- string vazia.
- (((((((
-))))))
-))((((

Para cada string S_i, armazenamos o par (a, b) correspondente. Em seguida, ordenamos os pares de acordo com o seguinte critério:

- Ordem crescente de a com $b - a \geq 0$.
- Ordem decrescente de b com $b - a \leq 0$.

Quando $b - a \geq 0$, significa que o número de parênteses de abertura é pelo menos o de parênteses de fechamento, quando $b - a \leq 0$, temos que o número de parênteses de abertura não é maior que o de parênteses de fechamento. Assim, ao selecionarmos as sequências com $b - a \geq 0$ em ordem crescente de a, temos certeza que estamos fazendo o possível para que o

número de parênteses de fechamento não supere o de abertura, em seguida, após todos as sequências deste tipo se esgotarem, procuramos concatenar aquelas com mais parênteses de fechamento, mas em ordem decrescente de b, visando minimizar o impacto dos parênteses de fechamento.

Seja s o número de parênteses de abertura menos o de fechamento com $s = 0$. Para cada par (a, b) na ordem estabelecida, se $s - a < 0$, então a resposta é negativa, caso contrário, s é atualizado como $s + b - a$.

A resposta será afirmativa ao final do processo se e somente se $s = 0$.

Complexidade

Para cada *string* S_i, os valores (a, b) podem ser calculados em $O(|S_i|)$ utilizando uma pilha. Inicialmente $(a, b) = (0, 0)$. S_i é inspecionado da esquerda para a direita e:

- Cada parênteses de abertura é inserido na pilha.
- Ao encontrar um parênteses de fechamento, caso a pilha não esteja vazia, um parênteses de abertura é removido da pilha. Caso contrário, o valor de a é incrementado.

Ao final do processo o valor de b corresponde exatamente ao tamanho da pilha.

A ordenação, que é o passo dominante do processo, leva tempo $O(n \lg n)$, o que justifica este como sendo o custo do algoritmo.