$$\sum_{n=1}^{\infty} \frac{1}{n \log n \log\log n} = +\infty$$

$$\sum_{i=0}^{\log(n)} 2^i (\log_n(-i))$$

$$\text{sum} \approx \sum_{p=0}^{\log N} \frac{N}{2^p}$$

Apresentação dos Exercícios Selecionados

$$T(n) = T(\frac{n}{n}) + \sum_{i=0}^{\log n - 1} \frac{c_2 n}{2^i} + \sum_{i=1}^{\log n} c_1$$

$$T(n) = T(1) + n * \sum_{i=0}^{\log n - 1} \frac{1}{2^i} + c_1 \log n$$

$$\begin{aligned} \sum_{i=1}^{n} log(i) &= log(n) + log(n-1) + ... + log(n/2) + ... + log(1) \\ &\geq log(n) + log(n-1) + ... + log(n/2) \\ &\geq log(n/2) + ... + log(n/2) \qquad ((n/2) \text{ times}) \\ &\geq (n/2) log(n/2) \\ &= \Omega(n log(n)) \end{aligned}$$

$$log(n!) = \sum_{i=1}^{n} log(i) \leq \sum_{i=1}^{n} log(n) = n log(n)$$

Alunos: Kizzy Terra
Otto Tavares

Exercício Selecionado 1

Uma subsequência contígua de uma lista S é uma subsequência composta por elementos consecutivos de S. Dê um algoritmo de tempo linear que dada uma lista de números S devolva a subsequência contígua de soma máxima.

Exercício 6.1 - Papadimitriou

Pseudocódigo

Entrada: lista de números a[1],a[2],...,a[n]
Saída: Subsequência contígua de soma máxima

```
startSoFar = 0
start_previous = 0
maxSoFar_index = 0
maxSoFar = a[1]
max_previous = a[1]
i = 2
Para cada a[i] em a[2], a[3],...,a[n]:
        se max_previous + a[i] >= a[i]:
                max_previous = max_previous + a[i]
        se não:
                start_previous = i
                max_previous = a[i]
        se max_previous >= maxSoFar:
                maxSoFar = max_previous
                maxSoFar_index = i
                startSoFar = start_previous
        i = i + 1
retornar a[startSoFar,..., maxSoFar_index]
```

Discussão

Complexidade:

- Tempo - O(n)
  - Iteramos apenas uma vez na lista e realizamos operações que não dependem do tamanho da entrada em cada iteração.
- Espaço - O(1)
  - Utilizamos um numero fixo variáveis auxiliares para fazer os cálculos independente do tamanho da entrada.

Discussão

Qual a diferença do algoritmo implementado para o algoritmo de divisão e conquista apresentado no artigo que foi discutido em sala?

Discussão

Qual a diferença do algoritmo implementado para o algoritmo de divisão e conquista apresentado no artigo que foi discutido em sala?

```
recursive function MaxSum(L, U)
    if L > U then     /* Zero-element vector */
         return 0.0
    if L = U then     /* One-element vector */
         return max(0.0, X[L])

M := (L + U)/2     /* A is X[L..M], B is X[M + 1..U] */
/* Find max crossing to left */
     Sum := 0.0; MaxToLeft := 0.0
     for I := M downto L do
          Sum := Sum + X[I]
          MaxToLeft := max(MaxToLeft, Sum)
/* Find max crossing to right */
     Sum := 0.0; MaxToRight := 0.0
     for I := M + 1 to U do
          Sum := Sum + X[I]
          MaxToRight := max(MaxToRight, Sum)
MaxCrossing := MaxToLeft + MaxToRight

MaxInA := MaxSum(L, M)
MaxInB := MaxSum(M + 1, U)
return max(MaxCrossing, MaxInA, MaxInB)
```

Algorithm 3. A divide-and-conquer algorithm

Fonte: Algorithm Design Techniques. Bentley, John (1984). Programming Pearls

Discussão

Solução que utiliza programação dinâmica apresentada no artigo

```
MaxSoFar := 0.0
MaxEndingHere := 0.0
for I := 1 to N do
    MaxEndingHere := max(0.0,
        MaxEndingHere + X[I])
    MaxSoFar := max(MaxSoFar,
        MaxEndingHere)
```

Algorithm 4. The linear algorithm

Fonte: Algorithm Design Techniques. Bentley, John (1984). Programming Pearls

Discussão

E se a subsequência não for contígua?

Podemos utilizar o algoritmo proposto para o exercício 6.1?

Discussão

Precisamos armazenar as soluções encontradas para todas as iterações anteriores, não apenas a solução encontrada para a iteração imediatamente anterior

Pseudocódigo

Entrada: lista de números a[1],a[2],...,a[n]
Saída: Subsequência de soma máxima

```
maxSoFar_index = 0

Para cada max[i] em max[1],..,max[n]:
        max[i] = 0
Para cada subsequence[i] em subsequence[1],.., subsequence[n]:
        subsequence[i] = 0

max[1] = a[1]
maxSoFar_index = 1
i = 2
Para cada a[i] em a[2], a[3],...,a[n]:
          Para cada max[j] em max[1],...,max[i-1]:
                se max[j] + a[i] >= a[i]:
                        subsequence[i] =j
                        max[i] = max[j] + a[i]
                se não:
                        max[i] = a[i]
          se max[i] > max[maxSoFar_index]:
                maxSoFar_index = i
          i = i + 1

k = subsequence[maxSoFar_index]
i = 0
Enquanto k != 0 :
          max_subsequence[i] = k
          k = subsequence[k]
          i = i + 1

retornar reverse(max_subsequence)
```

Discussão

Complexidade:

- Tempo - O(n^2)
  - Iteramos através de um índice i na lista de n elementos e para cada iteração de i, iteramos (i -1) vezes na lista que contém as soluções parciais.
- Espaço - O(n)
  - Utilizamos uma lista de n elementos para armazenar as soluções parciais.

Exercício Selecionado 2

Você recebe um conjunto de cidades com o padrão de estradas entre elas, na forma de um grafo não-direcionado G=(V,E). Cada estrada liga duas cidades e está representada por uma aresta ‘e’ do grafo. Além disso, cada estrada ‘e’ possui um comprimento L_e (em milhas) conhecido. Você quer sair da cidade ’s’ e chegar a cidade ’t’ . Entretanto, existe um problema: seu carro possui gás para andar apenas L milhas. Existem postos para reabastecer o gás em cada cidade, mas não entre as cidades. Portanto, você só pode pegar uma roa se todas as arestas que a compões possuem comprimento L_e menor ou igual a L.

Exercício 4.13 - Papadimitriou

Exercício Selecionado 2

a) Dada a limitação da capacidade de combustível do seu carro, mostre como determinar, em tempo linear, se existe uma rota possível entre 's' e 't'.

Exercício 4.13 - Papadimitriou

Pseudocódigo

Entrada: Grafo não direcionado (G=(V, E)), s (source), t (goal)
Saída: True / False

```
Para todo u em V (set of nodes)
        visited(u) = False
Q = [s]  #Q é uma fila
Enquanto Q não está vazia:
        u = eject(Q)
        para cada aresta e=(u,v):
                se Le <= L:
                        inject(Q,v)
        visited(u)
retornar visited(t)
```

Discussão

Complexidade:

- Tempo - O(V + E)
  - O algoritmo é uma modificação do BFS. Esta modificação acrescentou uma comparação ao algoritmo.

Exercício Selecionado 2

b) Agora você planeja comprar um novo carro e você quer saber a capacidade mínima necessária para viajar entre ’s‘ e ’t’. Implemente um algoritmo de tempo O((|M| + |E|)log|M|).

Exercício 4.13 - Papadimitriou

Pseudocódigo

Entrada: Grafo não-direcionado (G=(V,E)), source (s), goal (t), L (comprimento das arestas)
Saída: O comprimento da aresta de maior tamanho no menor caminho

```
Para todo u em V:
        dist(u) = inf
        prev(u) = nil
dist(s) = 0
node = nil
H = makequeue(v) #usando os valores do vetor dist como chaves
Enquanto H não está vazio e node != t:
        u = deletemin(H)
        para toda aresta e=(u,v) em E:
                se dist(v) > dist(u) + L(u,v):
                        dist(v) = dist(u) + L(u,v)
                        prev(v) = u
                        decreasekey(H,v)
                        node = v
max_length = 0
current_node = t
Enquanto current_node != s:
        v = prev(current_node)
        max_length = max(max_length, L(v, current_node))
        current_node = v
retornar max_length
```

Discussão

Complexidade:

- Tempo - O((|M| + |E|) log |M|)
  - O algoritmo possui dois laços principais: no primeiro, no pior caso, serão realizadas as mesmas operações do laço principal do Djisktra(O((|M| + |E|) log |M|)) e o segundo é linear no numero de vértices (O(V)).

Exercício Selecionado 3

Supõe-se que esteja fazendo uma consultoria a um banco, que está preocupado em detectar possíveis fraudes, e esse banco surge com o seguinte problema:

O banco possui uma coleção de N cartões, os quais foram confiscados, pois o banco suspeita que tais cartões estejam sendo usados em uma fraude. Cada cartão de crédito é um pequeno pedaço de plástico, contendo uma tira magnética com um dado criptografado. Cada conta pode possuir uma série de cartões de crédito, e vamos dizer que dois cartões são equivalentes se eles correspondem à mesma conta. É muito dificil ler o número criptografado, mas o banco possui uma tecnologia que faz o seguinte teste:

Toma dois cartões, faz algumas contas e determina se os cartões são equivalentes.

Isto posto, a pergunta é a seguinte: Entre a coleção de N cartões, existe um conjunto maior que N/2 cartões que acusarão equivalência no teste? Considera-se a hipótese de a única operação possível que se possa fazer para testar os cartões seja pegar dois deles, plugar na máquina, e dai verificar a equivalência. Mostre como decidir a resposta dessa questão com O(n log n) chamadas do teste de equivalência.

Exercício 5.3 - Tardos

Pseudocódigo

verifyCards(A, k = length(A) / 2)
Entrada: Lista de cartões A[1],..,A[n]; n> 1
Saída: True / False

```
count_equals = 0
count_equals = count(A[0:n/2]) + count(A[n/2:n])
se count_eq > k:
        retornar “True”
se não:
        retornar “False”
```

count(L)
Entrada: Lista de cartões L[1],..,L[n]
Saída: inteiro

```
n = length(L)
se n == 2:
        retornar equivalence_test(L[0], L[1])
se não:
        retornar count(L[0:n/2]) + count(L[n/2:n])
```

equivalence_test(m1, m2):
Input: Dois cartões m1, m2
Output: inteiro

```
se m1 == m2:
        retornar 1
se não:
        retornar 0
```

Discussão

Complexidade:

- Tempo - O(n log n)
  T(n) = 2T(n/2) + O(n) (Teorema Mestre)

Exercício Selecionado 4

The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs. They've broken the overall computation into n distinct jobs, labeled J1, J2,..., Jn, which can be performed completely independently of one another.

Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let's say that job Ji needs pi seconds of time on the supercomputer, followed by fi seconds of time on a PC. Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallelall the jobs can be processed at the same time.

However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

Exercício 4.7 - Tardos

Exercício Selecionado 4 - Resumo

- Existe um conjunto de N jobs (J[1],J[2],...,J[n]) a serem executados por N PCs
- Porém, antes de serem executados pelos PCs os jobs devem ser pré-processados em um único supercomputador
- O supercomputador só pode pré-processar um job de cada vez
- Cada job J[i] leva um tempo p[i] para ser finalizado pelo supercomputador
- Após este tempo p[i], cada job J[i] é encaminhado para um PC
- Cada job J[i] leva um tempo f[i] para ser finalizado por um PC

Pergunta-se: Qual a melhor forma de alocar os trabalhos dos supercomputadores e dos PCs de modo a obter o menor tempo necessário para finalizar todos os jobs.

Ideia

Executar os trabalhos em ordem decrescente de f[i] (tempo de processamento no PC)

O ponto principal desse exercício está na demonstração da otimalidade do algoritmo guloso, através da propriedade de “stays ahead”.

Prova da otimalidade

- Se p[i] é constante e f[i] é constante: A ordem de execução dos jobs é indiferente. Pois, se p[i] é um job do supercomputador, considerando que os jobs realizados pelo supercomputador não podem ser executados em paralelo, p[i] pode estar em qualquer posição que o tempo final não será alterado.

- Se p[i] varia e f[i] é constante: Uma vez que devemos minimizar o tempo total de trabalho, a ordem p[i] é indiferente, pois não pode haver sobreposição de jobs do supercomputador

- Caso geral (p[i] e f[i] variam): Pelo fato de p[i] não admitir sobreposição, o algoritmo guloso criado A, ordena o tempo dos jobs f[i] dos PCs em ordem decrescente de seu fim. A partir da suposição de que exista um outro algoritmo O, ótimo (que não ordena os jobs em ordem decrescente de f[i]) vamos mostrar por indução que o algoritmo O não pode ser ótimo e que o algoritmo A segue a propriedade “stays ahead” de algoritmos gulosos, provando sua otimalidade.

Exercício Selecionado 5

STINGY SAT é o nome dado ao seguinte problema: Dado um conjunto de cláusulas (cada cláusula é uma disjunção de literais) e um inteiro ‘k’, encontre uma solução que satisfaça o conjunto de cláusulas com no máximo k variáveis iguais a true, se esta solução existir.

Prove que STINGY SAT é NP-completo.

Exercício 8.3 - Papadimitriou

Ideia