Lista 4, 5 e 6 de Complexidade

P2 de 2010: http://www.mediafire.com/?lcm8na6ml3br2vb

Lista 4

1) Um jeito seria armazenar o valor de x e ir multiplicando enquanto multiplica e soma o valor.. É bem idiota e ingênuo e só to salvando de recalcular o x, mas o algoritmo de horner é ótimo segundo o wikipedia (http://en.wikipedia.org/wiki/Horner%27s_method), então não ia achar coisa muito melhor mesmo.

2)É uma busca binária, complexidade log n

```
BUSCA-BINÁRIA (V[], início, fim, e)
   i recebe o índice do meio entre início e fim
   se (v[i] = e) entao
      devolva o índice i    # elemento e encontrado
   fimse
   se (inicio = fim) entao
      não encontrou o elemento procurado
   senão
     se (V[i] vem antes de e) então
       faça a BUSCA-BINÁRIA(V, i+1, fim, e)
     senão
       faça a BUSCA-BINÁRIA(V, inicio, i-1, e)
     fimse
   fimse
```

3)Um algoritmo interessante seria ir pulando os numeros de x em x... quando o numero passar, você sabe que ele tá entre um número y que você tá agora e o número y-x. O problema é que a complexidade não deixa de ser n. Seria teto de (n/x) pra achar os limites e log x pra achar o número no intervalo. Como x é constante, seria n, e não tem jeito de não ser n. Algoritmo tão eficiente que ir de numero a numero até achar na complexidade
É mais eficiente que percorrer número a número até encontrar. Tome x = 1, a complexidade seria teto(n/1) + log 1 = n + 0 = n. Eis o algoritmo de percorrer um a um.

Agora para tome x = constante positiva. A complexidade fica teto(n/x) + log x que pertence a omegazão de (n). Só se não sei se ficaria do jeito que ela gosta (em função de n). Mas fica. x é constante, então log x é constante, então você ignora eu acho e-e. Verdade, então acho que complexidade final fica teto(n/x).

4) Pelo que entendi, o vetor está QUASE ordenado, de forma que A[i] pode ser no máximo igual a A[i + 1] + 1, ou seja, 0 1 0 2 2 3 2
porque quase ordenado? x < y e |A[i] - A[i+1]| <= 1 então o valor a frente é sempre maior ou igual não?
Deve estar ordenado sim, pois x <= z <= y, para x = A[1], y = A[n] e z = A[j] (j vc quer encontrar)
A restrição é de que o acréscimo entre 1 elemento e o seguinte seja de no máximo 1, podendo até ser nula. Então a sequência 0, 1, 1, 1, 2, 3, 4, 5 é válida
Na verdade é o absoluto da diferença... é |v[i]-v[i+1]|... então pode ser 0, 1, -1, -2...
Mas e a segunda restrição x <= z <= y, não garante a ordenação?
Não. Exemplo, x = 3 e y = 5. Posso ter um vetor 3 2 1 0 0 0 -1 -2 -1 0 1 2 3 4 5. O z é o valor que você quer, ou seja, o valor que você vai precisar achar tá entre x e y, mas não todo valor

Uma coisa que pensei:
o número que você quer, segundo o problema, (está entre 1 e n) tá entre x e y. Você pega o valor que está no meio do vetor. Se esse valor tá entre x e y, você sabe que metade o valor que você quer está, se o valor é maior que a metade, está na segunda metade, se for menor, está na primeira (nada impede que esteja nas duas, mas como o valor anda de 1 em 1, você tem certeza que pra chegar da metade até o y, tem que ter passado por todos os valores intermediários). Agora se o valor da metade é menor que x, você pode ter certeza que o valor que você quer tá na segunda metade, pois a segunda metade tem todos os valores entre esse numero do meio e y.

Você quer o índice j entre 1 e n.

5) Podemos ir varrendo o vetor inteiro a procura do elemento em questão e se não achar ele retorna -1, complexidade O(n) no pior caso.
A cota inferior = O(lg n) já que, mapeando para árvore de decisão binária temos que, o máximo possível de soluções é n elementos que serão as folhas da árvore. Considerando uma árvore de decisão T, sabemos que seu numero máximo de folhas é dado por 2^h, logo n <= 2^h -> h >= lg n -> h $\in$ Ω(lg n).
Para melhorar o nosso algoritmo temos que supor que o número que procuramos está entre os valores do primeiro e do último elemento, e não só isso, temos que ter a garantia que há uma certa sequência entre os números (como no 4, acho) em que |A[i] - A[i + 1]| <= 1 para que então possamos analisar da seguinte forma: pegamos o elemento do meio, se for o que queremos ok, senão vemos se ele é maior do que queremos, se for, então fazemos isso pra primeira metade senão pra segunda (busca binária). Ele tem a complexidade esperada pois temos a recorrência T(n) = T(n/2) -> O(lg n).

6)Tenho uma idéia, mas não provei a corretude aindplesmente varrer o vetor e somar todos os elementos e somar um(isso porque as n- 1 posição. O vetor possui n numeros reais. Se os numeros fossem positivos então, poderiamos simes podem ser 0 e ultima pode ter um valor diferente), com certeza esse número não estaria no vetor. Podemos fazer da seguinte maneira

então, percorremos o vetor e se o numero for negativo multiplicamos por -1 e somamos, caso contrario apenas somamos. Após percorrer o vetor todo some 1 apenas para garantir.

Funciona! A soma com certeza vai ser maior que qualquer elemento do vetor! Mas usando uma ideia similar, daria pra pegar o maior elemento e somar 1. O problema é provar a cota inferior, nem sei como fazer isso ainda.

Gente, não posso simplesmente percorrer o vetor e obter o valor máximo, e somar +1? ele certamente não pertence ao vetor, pois se estivesse certamente seria uma contradição quanto ao máximo .-.

Podemos projetar por indução e já dá a prova que tá certo:
CB: um elemento v, retorne seu valor v+1 já que v+1 != v
HI: para menos que n elementos sabemos encontrar um v que não está na sequencia
PI: Dados n elementos, retiramos o ultimo (seja u), e por HI sabemos encontrar um número que não está na sequência de 1..n-1, seja v; devolvemos u. Se u > v, então certamente u + 1 não está na sequência 1..n, pois v não estava em 1..n-1 e v < u < u + 1; senão, v >= u, neste caso certamente v + 1 não está na sequência, já que u <= v < v + 1 e u != v + 1. QED

7)Ordena os dois vetores. Coloca um contador em um e no outro. Aí anda com o contador do primeiro vetor enquanto o número dele for menor, e vice versa. Se encontrar um número igual, o conjunto não é vazio, caso contrário, é vazio. Complexidade de ordenação é n log n, a segunda etapa é 2n. Funciona pois, com os dois vetores ordenados, se um valor no primeiro vetor está selecionado, eu só preciso verificar os valores no segundo vetor enquanto o valor do segundo vetor é menor, quando passar, eu sei que aquele valor do primeiro vetor não está no segundo. A prova também vale do segundo para o primeiro vetor.
Minha solução:
Ordene o vetor A apenas, e percorra o vetor B chamando uma busca binária no vetor A para cada valor de B, se em algum momento a busca retornar um então a intersecção não é vazia, caso contrário a intersecção é vazia. Como a complexidade da busca binaria é logn e pode ser chamada no pior caso n vezes então, a complexidade é nlogn no pior caso.

8)Ordene os dois vetores. Começa com o primeiro valor de um vetor e o último valor do outro. Se A[i]+B[j]>x, j--. Senão i++. Se forem iguais achou. Funciona porque se você incrementar(decrementar no caso j, certo?) (não, j mesmo, tô falando, se fizer diferente não dá... sei que não é uma boa prova, mas tô tentando argumentar :P) j, o valor vai ficar ainda maior que x, se você decrementar i, vai ficar ainda menor. Todas as outras combinações que não serão testadas são irrevelantes (para cada i, só tem dois valores de j que compensam olhar: o que A[i]+B[j]<=x e o que A[i]+B[j+1]>=x, o resto ou vai ser obviamente maior ou menor... com esse algoritmo a gente tenta ficar entre esses valores de j).

9) Ah, sério mesmo? Só usar o intercala do mergesort com 3 vetores e-e.
Intercala com k-vetores, alguém sabe fazer? Forte candidato a cair na prova amanhã!!! (não da para usar o mesmo intercala do mergesort para k-vetores, creio que apenas a memoria adicional aumentaria)

é o 14

10) Nah

11) Estável - MergeSort, InsertionSort e CountingSort (Radix Sort, sub-ordenação estável obrigatória);
In place - QuickSort, Selection, Bubble Sort
Mas quais poderiam ser implementados de forma a serem estáveis, nenhum?
Acho que o quick não da... mas o selection/ insertion da sim

12)Se o quicksort escolhe o último elemento como pivot, vai ser pior caso quando tá ordenado ou totalmente desordenado (ele vai dividir em um pedaço de n-1 e 1). Não consigo pensar em que outros casos... mas depende de como ele escolhe o pivot também
Um vetor em ordem inversa tb resultaria no pior caso do Quick

13)a) Testa cada jarro vermelho com cada jarro azul
b)
c)Não pensei bem nisso, mas uma ideia é ir "ordenando" a capacidade dos jarros assim que você vai despejando. Tipo, escolhe um jarro azul como "pivot". Aí compara com todos os jarros vermelhos dividindo entre os maiores e menores (e o igual). Aí escolhe o segundo jarro... aí comparando com um jarro de cada lado você consegue descobrir que lado ele tá e repetir o processo... vai ser meio que um quick sort, que a complexidade no caso médio é n log n (não fica meio que um quick sort????) É, achei meio QuickSort tbm Isso! Quick! Lol, errei, arrumei ae, vlw

14)Cria um vetor com contadores pra cada vetor. Percorre os k vetores, pega o menor, coloca no vetor da resposta e incrementa 1.
Agora a complexidade... é nk, eita
Um jeito de melhorar usando a dica, seria fazer um heap com os valores dos vetores que estão no contador... acho que pra arrumar um heap quando muda um elemento é log k, se for, é O(1) pra ver o menor elemento, e dá certinho n log k (por isso acho que deve ser log k, mas preciso conferir)

Lista 5

1) O algoritmo continuaria funcionando normalmente. Não há uma ordem específica dos elementos de A, e a única coisa que é usada como índice de C é o valor de A[j]. Sendo assim, o valor de j poderia variar de qualquer maneira portanto que passase uma vez por cada valor possível entre 1 e n.
Por outro lado, o algoritmo não é mais estável. Se há dois elementos iguais no vetor A, então o primeiro elemento, que aparecerá como A[j] antes, será colocado na última posição reservada

2) Só usar o radix sort

3)Radix sort. Complexidade de tempo é O(n) e de espaço também

4)n^2, pois pode cair todo mundo em um único bucket. Uma alteração seria usar o mergesort pra ordenar (eu nao poderia refinar novamente e quebra-lo em novos bucket's ?)

5) **(Alguém conseguiu resolver esta? Caiu num dos testes do ano passado!)**
Pega a mediana, usa como pivot e divide o vetor em duas partes. Agora veja se o indice k é maior ou menor que n/2. Se for menor, você faz isso recursivamente com a parte de baixo, até que n/2 seja igual a k. A compĺexidade disso que é complicado, acho que dá algo como 2n+2n/2+2n/4+...+1... isso dá um múltiplo de n como 4n será? Se for, fica em O(n) também

→ funciona no caso de voce querer o indice como se o vetor estivesse todo ordenado; se na verdade ela quiser o k-ésimo menor número na sequência não podemos ter essa garantia: contra exemplo: 1 2 2 4 5, o algoritmo daria 2 como o 3º menor sendo que seria o 4.

Como a gente tem um algoritmo caixa-preta que garante a mediana em tempo linear, a gente pode usar o mesmo algoritmo dos slides dela. A diferença é a seguinte: na hora de escolher o pivo em PARTICAO, use a caixa preta. Com isso teríamos a seguinte recorrência:

T(n) = 1, n <= 1
T(n) = T(n/2) + n, n > 1

6) Descreva um algoritmo de complexidade de pior caso O(n) que determina, para um conjunto de n elementos e um inteiro k, k ≤ n, os k elementos mais próximos da mediana do conjunto. Voce pode usar como "caixa-preta" o algoritmo linear para encontrar a mediana de um conjunto.

7)Precisamos achar a posição do primeiro e segundo vetor, onde o indice somado da n e os valores são os mais proximos possiveis (porque o valor de uma determinada posição de um

vetor não pode estar acima de nenhum outro valor acima do indice do outro vetor), e pegar o maior valor. Vê o indice do meio dos dois vetores. Pega o de menor valor, e escolhe o meio da metade de cima do vetor, e do outro escolhe o da metade de baixo. Faz isso até começo=fim. O maior valor é a mediana.

Lista 6

1) a solução ótima do problema inclui a solução ótima de subproblemas. O cálculo da solução através de recursão implica no recálculo de subproblemas.

2)

3)Questão 5 da prova tem um similar

4)Esse problema é na verdade similar ao da soma de subconjunto. Você pode usar o algoritmo da mochila, igualando o custo com o peso do objeto, e ele vai encontrar a resposta correta.

b) =x

5) O último da prova

6)c={9,5,5} w={8,5,5}. Ele vai pegar primeiro o de custo 9, aí não cabe mais nada, mas ele poderia ter pego o 5 e 5 e ficado com 10 ao invés de 9

Se o que ela pediu foi pra ordenar os itens por ordem crescente de c/w, então c={1,10} e w={10,1} já resolve. Vai pegar primeiro o que tem 1/10 e então o de 10/1, mas o contrário seria vantajoso

c) No problema da mochila fracionária, podemos pegar o maximo possível do item de maior custo por peso, até acabar o item ou não sobrar mais espaço. Sendo assim, não há possibilidade de a soma de dois item com c/w menor dar maior que você pegar um item de c/w maior.

7)Pegue sempre o menor item da sequencia até terminar.
Prova de que funciona:
Vamos considerar que há um outro conjunto C' que funciona e não é igual ao C criado usando esse algoritmo. Esse conjunto C' terá pelo menos um item que não está em C. Removemos esse item, e colocamos no lugar um item de C que tem menor peso. Esse item de C, como tem menor peso, terá maior custo, e assim você consegue um conjunto C'' melhor que C'. Contradição.

8)

9)

10)

P2 (2011)

1. Considere um vetor A de n inteiros com a propriedade que
$|A[i] - A[i + 1]| \leq 1$ para todo $1 \leq i < n$.
Suponha que A[1] = x, A[n] = y e que x < y. Queremos projetar um algoritmo de busca otimo para encontrar um índice j tal que A[j] = z para um dado valor $x \leq z \leq y$.

(a) Prove que a cota inferior para este problema no modelo de arvore de decisao binaria é $\Omega(\log n)$.

Se considerarmos que o algoritmo de decisão binária deve decidir entre as duas metades do vetor, e considerando que índice do vetor é uma possível solução para o problema, temos que a árvore terá n folhas. O pior caso é o caminho mais longo até o fim da árvore. Como é uma árvore binária, se ela tem n folhas, então tem altura log n.

(agora isso só prova para algoritmos nesse estilo... deve ser o suficiente, já que o de ordenação ela também só fala de algoritmos baseados em comparação)

(b) De um algoritmo de divisao e conquista para este problema que seja otimo, isto é, de complexidade O(log n).
(c) Justifique por que seu algoritmo funciona e tem a complexidade esperada.

Para todos esse /\, ver o exercício 4 da lista 4, só não está provada a cota inferior (e não sei provar, mas deve ser o mesmo que a prova do busca binaria)

2. Considere um conjunto de 3 vetores ordenados A1 , A2 , A3 de tamanhos n1, n2, n3 , respectivamente, tais que n1 + n2 + n3 = n. De um algoritmo de complexidade O(n) que retorne um unico vetor ordenado A que contenha os n elementos destes vetores passados na entrada. Considere agora que na entrada sao dados d vetores ordenados A1, A2 , . . . , Ad , cada um de tamanho ni, tais que E ni = n (i.e., n é o numero total de elementos nos d vetores). De um algoritmo de complexidade O(n log d) que retorne um unico vetor ordenado A que contenha os n elementos destes vetores passados na entrada. Em ambos os casos, justifique por que seu algoritmo funciona e tem a complexidade requerida.

Ver 14 da lista 4

3. Dois alunos da disciplina de Algoritmos e Complexidade conversavam sobre o que haviam aprendido. La pela tantas, um aluno disse: “Fiquei maravilhado ao saber que era possıvel implementar o quicksort de modo que ele ordene um vetor de n numeros reais arbitrarios em tempo O(n log n) no pior caso”. O outro aluno retrucou: “Voce ficou louco? E´ impossıvel
implementar o quicksort de modo que ele tenha complexidade de pior caso inferior a $O(n^2)$.” Quem esta com a razao? Justifique cuidadosamente a sua resposta.

Resposta: Bom, se são valores arbitrários (acho que vc está confuso com o significado de valores arbitrários, não quer dizer que o aluno escolhe o melhor pivot sempre, mas que o vetor está preenchido arbitariamente - ela colocou isso para induzir a pensar que vetor não ordenado nunca terá complexidade de pior caso. Porém como sabemos, no campo teórico sempre podemos pegar um pivot que separe um elemento, de todos os outros), o primeiro aluno poderia simplesmente escolher valores que ele saberia que teriam a complexidade esperada. Por isso, em partes, ele tem razão.
Agora o segundo aluno também tem razão quando diz que o pior caso é n^2. Se a escolha do

pivot acontecer de forma que o vetor se divida entre duas partes de 1 e n-1, ocorre o pior caso. Foi um problema na comunicação dos alunos se pá.
Se não me engano dá pra garantir quicksort em O(n lg n) no pior caso usando o algoritmo que acha a mediana em O(n) logo o 2º aluno estaria errado ao dizer que é impossível.

O quick sort possui análise de **caso médio**, que supõe escolha sucessivas de pivot's que separem T(i - 1) + T(n - i). Mas acho que está questão não é nem tanto sobre provar n^2 ou n log n, mas um "dialogue sobre o quicksort e vou julgar de 0 a 2,5, boa sorte" rs

4. Explique o que é um heap de maximo e de um exemplo de heap com 10 elementos. Descreva um algoritmo para construir um heap e explique como o heap é utilizado no algoritmo de ordenacao Heapsort. Justifique por que a complexidade de pior caso do Heapsort é O(n log n).

Heap máximo é onde o pai é sempre maior que os filhos. Um heap maixmo poderia ser

pos:	1	2	3	4	5	6	7	8	9	10
Valor:	10	9	8	7	6	5	4	3	2	1

A complexidade no pior caso é n log n, porque é necessário ajustar o heap para que o maior elemento fique na primeira posição para cada valor do vetor do heap. Para ajustar, é necessário tempo proporcional a altura do heap, para fazer as comparações entre pai e filho (ou seja, log n), e isso é feito n vezes, um pra cada posiçao do vetor

5. Use o algoritmo de programacao dinamica visto em aula para encontrar o valor da solucao ótima para o problema da mochila com 5 itens de custos c = (5, 9, 4, 10, 7), tamanhos w = (4, 1, 2, 6, 3) e mochila de capacidade W = 10. Apresente a tabela z de programacao dinamica toda preenchida para a entrada dada, bem como a relacao de recorrencia utilizada para preenche-la. Com base na tabela preenchida, determine quais itens sao selecionados na solucao ótima.

k/d	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5
2	0	9	9	9	9	14	14	14	14	14	14
3	0	9	9	13	13	14	14	18	18	18	18
4	0	9	9	13	13	14	14	19	19	23	23
5	0	9	9	13	16	16	20	20	21	23	26

Pra saber os itens selecionados:
Começa do ultimo elemento da tabela z(5,10) =26
Faz a comparação: “é diferente?” com o elemento exatamente acima z(4,10) = 23
Se sim, vê a linha que você está (k=5), portanto é o quinto elemento no vetor de custo(c[k]=c[5]=7), portanto o objeto de custo 7 está no resultado. Se não, pula pra linha de cima e repete a comparação.
Agora estamos no z(4,10) = 23, contamos 3 pra esquerda (pois w[k]=w[5] = 3) e repetimos a comparação: z(4,7) = 19 é diferente da linha acima? z(3,7) = 18 no caso, então sim. Com isso, estamos na linha k=4. Selecionando o quarto elemento no vetor de custos(c[k]=c[4]=10), temos o objeto de custo 10 na mochila também. Agora estamos no (3,7) = 18. Andamos 6 pra esquerda (w[k]=w[4]=6). Chegamos em z(3,1) = 9. Compara com a linha de cima, é diferente? Não, então pula pra linha de cima (2,1). Compara com a linha de cima, é diferente? Sim, estamos na linha 2, portanto o segundo objeto no vetor de custos(c[k]=c[2]=9), no caso o 9, está na mochila. Com isso, atingimos o valor final da mochila em termos de tamanho (1 + 6 + 3 = 10). A resposta final é 9,10,7.

6. Vimos em aula um algoritmo guloso que resolve o problema de encontrar, dado um conjunto de atividades e seus instantes de início e termino, um subconjunto de tamanho maximo com atividades duas a duas compatıveis. Mas nem toda escolha gulosa para esse problema leva a um algoritmo correto. Encontre instancias desse problema em que as seguintes escolhas gulosas nao levam a um algoritmo correto:

(a) selecionar dentre as atividades compatıveis com as pre-selecionadas aquela com menor duracao;
Atividades:
1-8, 9-16, 17- 24, 7-10, 15-18.
Ele vai selecionar o 7-10 e 15-18. Mas seria melhor pegar as outras 3

(b) selecionar dentre as atividades compatıveis com as pre-selecionadas aquela com menor tempo de início;
1-10, 2-3, 3-4, 4-5, 5-6, 6-7
Ele pega o 1-10 mas tinha 5 tarefas que ele poderia ter pego