Desenho e Análise de Algoritmos

Notas - Aula TP 9 e 10

Algoritmos com mecanismos de Aleatoriedade

Ana Maria de Almeida

DCTI

2022/2023

1

Plano para esta aula

- Finalização da análise do algoritmo da k-seleção para diferentes métodos da escolha de pivot.
- Análise de algoritmos com aleatoriedade:
  - *Métodos de Las Vegas*
  - *Métodos de Monte Carlo*
  - *Análise da complexidade esperada*

2022/2023

2

Análise de algoritmos recorrentes

continuação

- Vimos um algoritmo recursivo para a resolução da k-seleção, ou seja, devolver o k-ésimo menor elemento de uma sequência de n elementos.
- Podemos usar o MergeSort para ordenar por ordem crescente a sequência e devolver o k-ésimo elemento, a resolução, em termos de tempo computacional, no pior dos casos, seria O(n log(n)).
- Será que podemos melhorar?
- Uma idéia possível é a seguinte: escolher um pivot próximo da mediana e usar recursão num dos lados, após utilizar o pivot para efetuar uma separação de Hoare (em menores que o pivot para a esquerda e maiores para a direita).
  - Truque: Usar recursão na própria escolha do pivot!
  - **Conjectura**: Esta escolha de pivot leva tempo O(n)!

k-seleção: algoritmo final das medianas de 5 elementos

```
Select(A, k):
    //Input: sequência, A, e inteiro k
    //Output: o k-ésimo menor elemento de A

    if len(A) < 10:
        A = Mergesort(A)
        return A[k]
    p = choosePivot(A)
    L, m, R = Partitition(A, p)
    if len(L)==k-1: return A[m]
    else if len(L) > k-1: return Select(A, k)
    else if len(L) < k-1: return Select(R, k-Len(L)-1)
```

Diferentes escolhas de pivot:

a) escolha pela aproximação da mediana (medianas de 5 elementos)

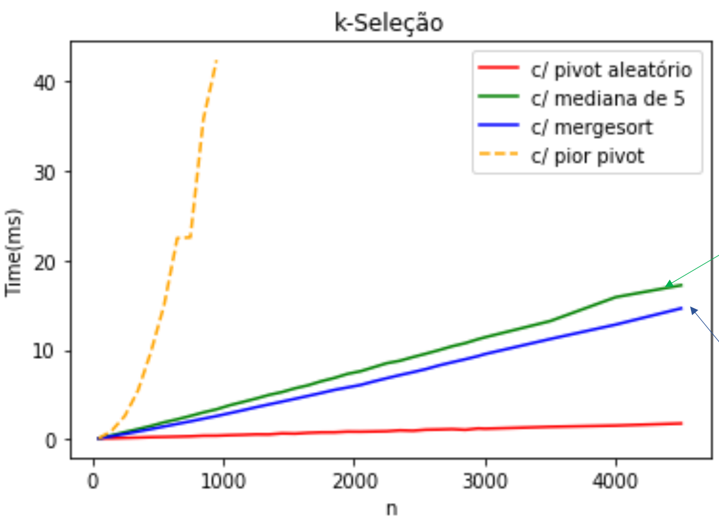
b) escolha aleatória

c) outras escolhas

```
choosePivot(A):
    Dividir A em m =⌈n/5⌉ grupos de tamanho <= 5 cada.
    for i=1, .., m:
        Calcular a mediana do grupo I, pi
    p = Select( [ p1, p2, p3, …, pm ] , m/2 )
    devolver o índice de p em A
```

Qual a ordem de complexidade deste algoritmo de seleção com esta escolha de pivot?

Testes empíricos (laptop)



k-Seleção com escolha informada de pivot

k_Seleção com ordenamento completo

k-seleção com escolha pela aproximação da mediana: análise de complexidade

- É possível provar que esta escolha de pivot consegue sempre separar em maiores (R) e menores (L) tal que

$$|L| \leq \frac{7n}{10} + 5 \text{ e } |R| \leq \frac{7n}{10} + 5$$

- A relação de recorrência para calcular o trabalho computacional do algoritmo da seleção (Select), **usando esta escolha de pivot**, será, então

trabalho da separação

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

o choosePivot() efetua uma chamada recursiva com as medianas para Select()

chamada recursiva a Select() **após** separação

- O teorema principal não pode ser aplicado nesta recorrência (porquê?).
- Deve ser usado um método alternativo: o método da substituição ou do fecho da recorrência.

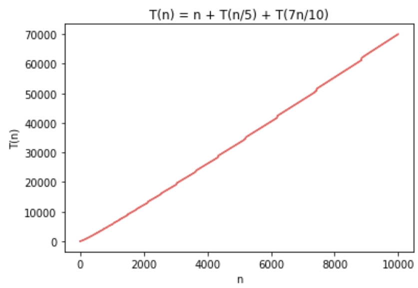
Outro modo de calcular T(n):

- Passo 1: Procurar perceber qual vai ser o comportamento de T(n):
- Por exemplo: Conjecturar (antever!) qual a possível conclusão (fecho da fórmula).

Dica: explorar usando testes empirícos:

- *Teste exploratório (laptop): n + T(n/5)+ T(7n/10)*
- *Parece ter um comportamento linear*

Será mesmo O(n)?



- Passo 2: Provar, **usando indução**, que a conjectura é válida.

iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

Provar que $T(n) \leq T(n/5) + T(7n/10) + cn$ é $O(n)$

- Para provar que $T(n) = O(n)$, temos de provar que

  existe algum $n_0 > 0$ e algum d> 0 tal que, $\forall n \geq n_0, T(n) \leq dn$

- Pelo **caso base**, temos $n_0 = 1$ e $T(1) = 1$.

  Logo, a constante d tem de ser: $d \geq 1$, para termos $T(1) \leq d$.

- Vamos agora construir a nossa **Hipótese indutiva**:

  Assumimos que a conjetura $T(n) \leq dn$ é verdadeira para $n < k$. Em seguida, queremos mostrar que, então, é verdadeira para $n = k$. Ou seja, vamos assumir que existe d tal que $\forall n_0 \leq n < k$, $T(n) \leq dn$.
  Para provar que também T(n) é limitado para $n = k$, temos de resolver a seguinte inequação:

Fórmula recursiva do trabalho neste caso:

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$ se $n > 1$

e $T(n) = 1$ se $n = 1$

$T(k) \leq T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right) + ck$ *substituindo pela fórmula fazendo n=k*

$\leq dk/5 + 7dk/10 + ck$ *aplicação hipótese indutiva, porque k/5 e 7k/10 são menores que k*

$= \left(c + \frac{d}{5} + \frac{7d}{10}\right)k$ *colocando k em evidência*

O resultado final pretendido é $T(k) \leq dk$. Assim, vamos agora resolver a inequação $\left(c + \frac{d}{5} + \frac{7d}{10}\right)k \leq dk$ e verificar se existe mesmo um d nestas condições:

Provar que $T(n) \leq T(n/5) + T(7n/10) + cn$ é $O(n)$ - continuação

Fórmula recursiva do trabalho neste caso:

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$ se $n > 1$

e $T(n) = 1$ se $n = 1$

Vamos agora resolver a inequação em ordem a d:

$\left(c + \frac{\boldsymbol{d}}{5} + \frac{7\boldsymbol{d}}{10}\right)k \leq \boldsymbol{d}k$

$\Leftrightarrow 9/10d + c \leq d \Leftrightarrow c \leq d/10 \Leftrightarrow d \geq 10c$

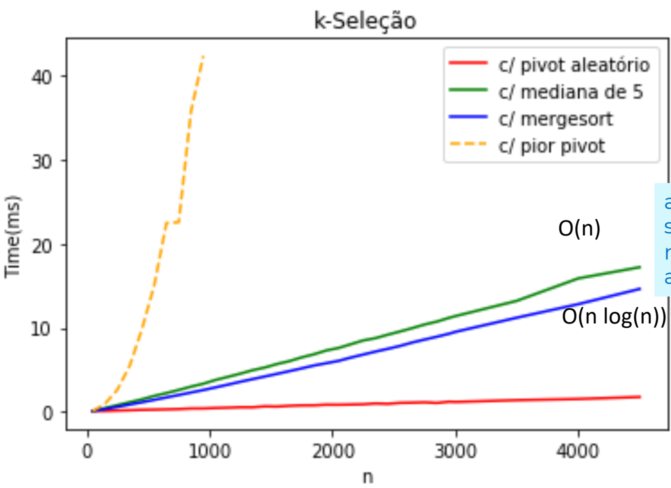
O que é sempre possível (relembro que o valor de c vem do processo separação, que é linear em n).

- Portanto, se escolhermos $d = \max(1, 10c)$, a hipótese é verdadeira para qualquer valor de $k \geq 1$, logo, para qualquer valor de n. Ou seja,

  existe $n_0 = 1 > 0$ e algum $d = \max(1, 10c) > 0$ tais que, $\forall n \geq 1, T(n) \leq dn$.

- Concluimos que, pela definição de limite superior assintótico, o trabalho/tempo de execução do algoritmo da k-Seleção com escolha de pivot pela aproximação da mediana é, no pior dos casos, O(n).

Como explicar que um método O(n) tem este comportamento relativamente a um método O(n log n)?



a constante escondida em O(n) é 20 vezes superior à constante em O(n log n), logo, só com n bastante grande é que a linha azul ultrapassa a linha verde

Ver exemplo comparativo entre as duas linhas de crescimento no notebook de apoio à aula.

Já no caso da versão com **escolha aleatória** de pivot, o comportamento, ainda que também linear, é bastante mais rápido que a aproximação pela mediana de cinco.

Análise de algoritmos com aleatoriedade

continuação

Algoritmos com aleatoriedade
(aka, *randomized algorithms*)

- Algoritmos que usam mecanismos/métodos com geração aleatória de elementos.

- Para além do input, usa um gerador de números aleatórios que permite alguma escolha aleatória durante a execução
- Vai implicar que o tempo de execução **será uma variável aleatória**, pois o comportamento irá variar, mesmo mantendo o input.

16

Aplicações (algoritmos com mecanismos de aleatoriedade)

- Algoritmos baseados em aritmética de inteiros:
  * teste de primalidade (Monte Carlo)
- Estruturas de Dados:
  * ordenamento, k-seleção, pesquisa, geometria computacional
- Identidades algébricas:
  * identidade de polinómios ou de matrizes, sistemas de prova interativos
- Machine learning
- …

17

Algoritmos com aleatoriedade: tipo **Las Vegas**:

- Algoritmos do tipo **Las Vegas**:
  * para cada input, devolve sempre a resposta correta;
  * o trabalho (número de operações esperado) é descrito por uma variável aleatória, cujo valor esperado (esperança) é limitada.

- Ou seja, o output é determinístico mas o tempo de execução é aleatório.
- Exemplos:
  - Quicksort com escolha aleatória de pivot (randomized quicksort)
  - K-seleção com escolha aleatória de pivot (quick selection)

Randomized Quicksort

- Mais simples de implementar que o Mergesort.
- As constantes escondidas pela notação assintótica são pequenas.
- Pior dos casos: $O(n^2)$
- **Tempo expectável de execução: O(n log n)**

Quickselect

- k-Seleção com escolha aleatória de pivot.
- Esta é a escolha mais simples de implementar.
- As constantes escondidas pela notação assintótica são pequenas.
- Pior dos casos (pior pivot): $O(n^2)$
- **Tempo expectável de execução**: O(n)

- Quer no k-seleção, quer no quicksort, o processo de escolha de pivot e consequente separação **tem o maior peso computacional** no total do algoritmo.

- No quicksort, sabemos que a "regra" de implementação é usar a escolha aleatória de pivot. A razão prende-se com o facto de que, devido a esta aleatoriedade, é possível **esperar** que o pior dos casos seja O(n log n) para qualquer input de tamanho n. Mas, para isso é necessário calcular:

a ordem de complexidade esperada.

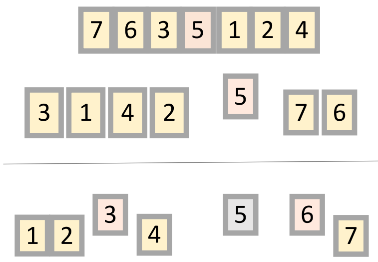
Tempo esperado

- O análise do caso médio tenta perceber o que é típico no input, i.e., qual a distribuição esperada para o input e o que isso implica em termos de trabalho computacional.
- Já a **análise do tempo esperado** é uma medida para a quantidade de trabalho expectável de um **algoritmo com aleatoriedade**, ou seja, como podemos esperar que ele se comporte, independentemente da distribuição do input.

- O uso de um mecanismo de aleatoriedade num algoritmo tenta assegurar que o comportamento no pior dos casos será altamente improvável.

Tempo esperado para o Quicksort

- Dado que vamos escolher aleatoriamente o pivot (trabalho constante), só precisamos de contar o número de comparações que o algoritmo efetua.



Porquê só as comparações?
Porque as trocas/movimentos são (sempre) lineares

Todos os elementos são comparados com o primeiro pivot (5) uma vez no primeiro passo e nunca mais!

No segundo passo, só os elementos da chamada recursiva à esquerda (ou à direita) são comparados com o seu pivot e não entre si.

- Portanto, o número de vezes que cada par de elementos, i e j, será alguma vez comparado é ou 0 (zero) ou 1 (uma) vez.

Variável aleatória no quicksort

- Ou seja, a possibilidade de i e j serem comparados é uma variável aleatória:

$$X_{i,j} = \begin{cases} 1 & se\ i\ e\ j\ são\ comparados\ alguma\ vez \\ 0 & se\ i\ e\ j\ nunca\ são\ comparados \end{cases}$$

  - Neste exemplo, $X_{1,5} = 1$, porque o elemento 1 é comparado com o (pivot) 5.
  - Mas $X_{3,6} = 0$, porque 3 e 6 não vão ser (nunca) comparados.

- E esta variável (os seus valores) **depende**(m) **do pivot que for escolhido!**

Contagem de comparações

- O **número total de comparações** no algoritmo será, então (assumindo que, s.p.d.g., i < j):

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}$$

- E o **número expectável de comparações** é dado pelo valor esperado da soma total:

$$E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[\,X_{i,j}]$$

devido à lineariedade da esperança.

- E qual é o valor de $E[\,X_{i,j}\,]$?

$$E[\,X_{i,j}\,] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(X_{i,j} = 1)$$

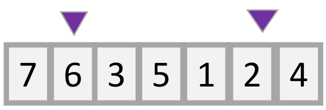
Usando a definição de valor esperado (se não se lembra, clique aqui):

$E[\,X_{i,j}\,] = P(\,X_{i,j} = 1\,) \cdot 1 + P(\,X_{i,j} = 0\,) \cdot 0 = P(X_{i,j} = 1)$

Contagem de comparações

- Donde, necessitamos de calcular

$P(X_{i,j} = 1)$ = probabilidade de i e j serem comparados alguma vez



Se i = 2 e j= 6 qual é a probabilidade de serem comparados alguma vez?

- Se o 3, ou o 4 ou o 5) for escolhido primeiro, o 2 e o 6 ficariam separados e **nunca** seriam comparados.
- Logo, a probabilidade de 2 ou 6 serem comparados alguma vez é igual à probabilidade de serem escolhidos para ser pivot (,de entre todos os números entre 2 e 6).

- Logo, pensando que i e j estão a ser escolhidos de entre i, i+1, ..., j,

$P(\, X_{i,j} = 1\,)$ =

= probabilidade de i, j serem alguma vez comparados

= probabilidade de um deles ser comparado entre os j – i +1 números que estão entre eles

= probabilidade de um deles ser o primeiro pivot escolhido

= $\frac{2}{j-i+1}$

Número de comparações esperado

$E\left[\sum_{a=1}^{n}\sum_{b=a+1}^{n} X_{a,b}\right]$ é, portanto, o número total de comparações esperado.

$= \sum_{a=1}^{n}\sum_{b=a+1}^{n} E[\,X_{a,b}]$ E, pela propriedade linear do valor esperado E(.), vem:

$= \sum_{a=1}^{n}\sum_{b=a+1}^{n} P(\,X_{a,b} = 1)$ Pela definição de valor esperado, temos

$= \sum_{a=1}^{n}\sum_{b=a+1}^{n} \frac{2}{b\ - a + 1} \leq 2n \sum_{k=1}^{n} \frac{1}{k} = 2n \log n$

(a série acima é a série harmónica)

que é da forma geral de escolhas de pivots em intervalos de tamanho k

Logo, o tempo esperado valor o quicksort com escolha aleatória de pivot é O(n log n)

Algoritmos com aleatoriedade: tipo Monte Carlo

- Algoritmo do tipo **Monte Carlo**:
  * executa um número fixo de passos;
  * produz um **resultado com probabilidade** de estar correto de, pelo menos 1/3
  * as probabilidades dizem respeito apenas às escolhas aleatórias feitas pelo algoritmo (são independentes do input).

    logo, repetições independentes do algoritmo diminuem a probabilidade de errar.
- Ou seja, o output pode variar (aleatório) mas o tempo de execução é determinístico.
- Exemplos:
  - "Teste de igualdade"
  - Algoritmos baseados em aritmética de inteiros: teste de primalidade

Exemplo de um algoritmo de Monte Carlo

- "Teste de igualdade": sejam dados dois números **binários** X e Y. Decidir se X=Y.
  - Os tamanhos de X e Y podem ser muito grandes (Terabytes).
  - A igualdade ponto a ponto é linear (O(n)).
  - Vamos construir um algoritmo com aleatoriedade para tentar atingir (melhorar) o desempenho computacional mais efeiciente.
  - Seja n =len(X) = len(Y) a quantidade de dígitos dos números.

```
Algoritmo para teste de igualdade:
    //Input: números binários x e y
    //Output: True se x == y; False, caso contrário
    - se len(y) = len(x):
        n= len(x)
        escolher, aleatoriamente, um número primo p ≤ n^2 tal que len(p) ≤ log(n^2)
        devolver (x mod p) == (y mod p)
    - senão: devolver False
```

Algoritmo de Monte Carlo para testar igualdade

- Mas, será a resposta correta?
  - se (x mod p) ≠ (y mod p) então, certamente, x ≠ y e a resposta é correta.
  - se (x mod p) = (y mod p) então

    podemos ter x ≠ y ou x = y, logo, a resposta pode ser correta , ou não!
  - Qual a probabilidade de estar correta?
  - ou qual a probabilidade de estar incorreta?
- **Teorema dos números primos**: Em $\{1, 2, 3, 4, \dots, m\}$ existem cerca de $m/\log m$ números primos.
- Logo, em $\{1, 2, \dots\ n^2\}$ existem $n^2/\log n^2 \ = \ n^2/2\log n$ primos.
- E quantos primos em $\{1, 2, \dots\ n^2\}$ satisfazem (x mod p) = (y mod p) quando x ≠ y ?

Algoritmo de Monte Carlo para testar igualdade

- (x mod p) = (y mod p) ⇔ |x-y| é um múltiplo de p, ou seja, p é um divisor de |x-y|.
- Como estamos a usar números binários com n dígitos, então: |x-y| < 2^n e, portanto, este módulo tem, **no máximo,** n divisores (caso contrário seria maior que 2^n).
- Logo, de todos os primos que podemos escolher, no máximo, n são "maus" e a probabilidade de o algoritmo devolver uma resposta errada é inferior a

$$P(erro) \leq \frac{n}{\frac{n^2}{2\log n}} = \frac{2\log n}{n}$$

  - Exemplo: quando $n \ = n^{14}$, temos $P(erro) \leq \ 0{,}00000000000644$
  - E executando duas vezes, fazemos menos de 200 comparações e obtemos

    $(P(erro))^2 \leq \ 0{,}000000000000000000000000215$

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

Para reter das aulas anteriores

- Finalização da análise do algoritmo da k-seleção para diferentes métodos da escolha de pivot.
- Algoritmos com aleatoriedade:
  - *Métodos de Las Vegas*
  - *Métodos de Monte Carlo*
  - *Análise da complexidade esperada de um algoritmo com aleatoriedade*
    - *exemplo com a análise da complexidade do Quicksort*

45

QuickSort *versus* MergeSort

	QuickSort (random pivot)	MergeSort (determinístico)
Running time	• Worst-case: $O(n^2)$ • Expected: O(n log(n))	Worst-case: O(n log(n))
Used by	• Java para tipos primitivos • C - qsort • Unix • g++	• Java para objectos • Perl
In-Place (usando O(log(n)) memória extra)	Sim (facil)	Não é fácil* se quisermos manter a estabilidade e o tempo de execução. (mas muito fácil se sacrificar o tempo de execução).
Estável	Não	Sim
Prós (outros)	Boa localidade de cache se implementado com arrays	Merge muito eficiente com listas

Para saber

Só para conhecer

* procurar "Block Sort" na Wikipedia!

46

Para estudar

- Cap. 7 e secções 5.1, 5.2, 5.3 – *Introduction to Algorithms*. Cormen, Leiserson et al., 4th ed. MIT Press, 2009
- https://amaankhatib232.medium.com/randomized-algorithms-4e4a7f9b1f93
- https://en.wikipedia.org/wiki/Randomized_algorithm