Fundação Getúlio Vargas
Estruturas de Dados e Algoritmos
Prof. Jorge Poco

Homework 4
Due:: April 24, 2020

Homework 4

1 Sorting in Place in Linear Time

(2pts) Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

(a) Give an algorithm that satisfies criteria 1 and 2 above.

Como no counting sort, primeiramente nós contamos a quantidade 0's (vamos chamá-la de L_0) e a quantidade 1's. Isso roda em $O(n)$. Depois criamos um array do mesmo tamanho do original e insertamos os elementos segundo o pseudocódigo abaixo, onde "R" será o nosso resultado e "A" nossa lista original:

```
k0 = 0
k1 = L_0
for i = 0 to n-1:
  if L[ i ] == 0:
    R[ k0 ] = L[ i ]
    k0++
  else
    R[ k1 ] = L[ i ]
    k1++
```

Logo, $T(n) = 2O(n) + O(1) = O(n)$.

(b) Give an algorithm that satisfies criteria 1 and 3 above.

Novamente, seja L a lista, faremos o seguinte algorítmo para a ordenação.

```
a = 0
b = n-1
while a < b:
  if L[ a ] == 0:
    a++
  if L[ b ] == 1:
```

```
  b–
if L[ b ] == 0 and L[ a ] == 1:
  exchange L[a] with L[b]
  a++
  b–
```

Exemplo de como o algorímo funciona:

a=0, b=4
(r1, 1), (r2, 0), (r3, 1), (r4, 0), (r5, 0)
(r5, 0), (r2, 0), (r3, 1), (r4, 0), (r1, 1)
a=1, b=3
(r5, 0), (r2, 0), (r3, 1), (r4, 0), (r1, 1)
a=2, b=3
(r5, 0), (r2, 0), (r4, 0), (r3, 1), (r1, 1)
fim

Em cada iteração, 'a' avança ou 'b' diminui. Logo, é óbvio que o número de iterações vai ser $< n$. Então o algorítmo roda em $O(n)$.

(c) Give an algorithm that satisfies criteria 2 and 3 above.

Insertion sort funciona nesse caso.

(d) Can any of your sorting algorithms from parts(a)–(c) be used to sort n records with b-bit keys using radix sort in $O(bn)$ time? Explain how or why not.

Sim. Usando, por exemplo, o algorítmo do item **a** que é estável e roda em tempo $O(n)$. Logo, o radix sort utilizando esse algorítmo funciona, pois ele é estável, e roda em tempo $bO(n) + c = O(bn)$.

(e) Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that the records can be sorted in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (Hint: How would you do it for $k = 3$?)

Primeiramente, reservamos um espaço de memória $O(k)$ e contamos quantas aparições cada chave teve. Sejam L a lista original e count a lista com as contagens das chaves. Faremos o seguinte algorítmo:

```
nc[ 0 ] = 0
for i = 1 to k-1:
  nc[ i ] = nc[ i - 1 ] + count[ i - 1 ]
nc[ k ] = n
ncfixo = copy(nc)
for i = 0 to k-1:
  while nc[ i ] < ncfixo[ i + 1 ]:
    if L[ nc[ i ] ] != i + 1:
      aux = L[ nc[ i ] ] - 1
      exchange L[ nc[ i ] ] with L[ nc[ aux ] ]
      nc[ aux ]++
    if L[ nc[ i ] ] == i + 1:
      nc[ i ]++
```

A memória utilizada além da lista original é $3k + 1 = O(k)$. E o tempo de execução que podemos conferir pelo número de iterações é de $n * O(1) = O(n)$; porém, primeiramente fizemos o counting que é $O(k)$. Logo, a complexidade do algorítmo todo é de $O(n + k)$.

2 Alternative Quicksort Analysis

(2pts) An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to QUICKSORT, rather than on the number of comparisons performed.

(a) Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = I\{i\text{th smallest element is chosen as the pivot}\}$. What is $E[X_i]$?

Com dos n elementos escolhe-se o pivô aleatoriamente, a chance de qualquer número ser escolhido é a mesma. Definindo X_i da mesma forma que no enunciado, e usando que a probabilidade se considerarmos todo o espaço amostral deve ser 1, temos que $\sum_{i=1}^{n} P(X_i = 1) = nP(X_i = 1) = 1$, ou seja, $P(X_i = 1) = \frac{1}{n}$ $\forall i$ tal que $1 \leq i \leq n$.

Logo, $E(X_i) = 0P(X_i = 0) + 1P(X_i = 1) = \frac{1}{n}$.

(b) Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] \tag{1}$$

Vamos supor que escolhemos o j-ésimo elemento. Logo, teremos $T(n) = T(j-1) + T(n-j) + \Theta(n)$. O $\Theta(n)$ é devido ao partition. Como nesse caso $X_j = 1$ e, dado um $i \neq j$ $X_i = 0$, temos $T(n) = \sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))$. Ou seja, vale (1).

(c) Show that equation 1 simplifies to

$$E[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \tag{2}$$

Usando a linearidade do valor esperado $E[T(n)] = \sum_{q=1}^{n} E[X_q(T(q-1) + T(n-q) + \Theta(n))]$.

Usando a independência de X_q em relaçao as outras variáveis, temos

$$\begin{aligned}
E[T(n)] &= \sum_{q=1}^{n} E[X_q]E[(T(q-1) + T(n-q) + \Theta(n))] \\
&= \sum_{q=1}^{n} \frac{1}{n}E[(T(q-1) + T(n-q) + \Theta(n))] \\
&= \frac{1}{n}\sum_{q=1}^{n} E[(T(q-1)] + \frac{1}{n}\sum_{q=1}^{n} E[T(n-q)] + \frac{1}{n}\sum_{q=1}^{n} E[\Theta(n)] \\
&= \frac{1}{n}\sum_{q=1}^{n} E[(T(q-1)] + \frac{1}{n}\sum_{q=1}^{n} E[T(n-q)] + \frac{1}{n}n\Theta(n) \\
&= \frac{1}{n}\sum_{q=1}^{n} E[(T(q-1)] + \frac{1}{n}\sum_{q=1}^{n} E[T(n-q)] + \Theta(n)
\end{aligned}$$

Trocando no primeiro somatório q por $q+1$ e no segundo q por $n-q$, temos

$$
\begin{aligned}
E[T(n)] &= \frac{1}{n}\sum_{q=0}^{n-1} E[(T((q+1)-1)] + \frac{1}{n}\sum_{q=0}^{n-1} E[T(n-(n-q))] + \Theta(n) \\
&= \frac{1}{n}\sum_{q=0}^{n-1} E[(T(q)] + \frac{1}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \\
&= \frac{2}{n}\sum_{q=0}^{n-1} E[(T(q)] + \Theta(n)
\end{aligned}
$$

Como queríamos provar.

(d) Show that

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \tag{3}$$

(Hint: Split the summation into two parts, one for $k = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n-1$.)

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &\leq \int_1^n x \log x dx \\
&= \left[\frac{1}{2}x^2 \log x\right]_1^n - \int_1^n \frac{1}{2}x^2 \frac{1}{x} dx \\
&= \frac{1}{2}n^2 \log n - \int_1^n \frac{1}{2} x dx \\
&= \frac{1}{2}n^2 \log n - \left[\frac{1}{4}x^2\right]_1^n \\
&= \frac{1}{2}n^2 \log n - \frac{1}{4}n^2 + \frac{1}{4}
\end{aligned}
$$

Tomando $n \geq 2$ temos

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &\leq \frac{1}{2}n^2 \log n - \frac{1}{4}n^2 + \frac{1}{4} \\
&\leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2
\end{aligned}
$$

Como queríamos demonstrar.

(e) Using the bound from equation 3, show that the recurrence in equation 2 has the solution $E[T(n)] = \Theta(n \lg n)$. (Hint: Show, by substitution, that $E[T(n)] \leq an \log n - bn$ for some positive constants a and b.)

Caso base é trivial.

Passo indutivo:

Suponha que para todo k com $1 \leq k \leq n-1$ temos $E[T(k)] \leq a_k k \log k - b_k k$ com a_k e b_k positivos.

Seja $a = max\{a_k\}$ e $b = max\{b_k\}$.

Logo, utilizando a recorrência 2 e a desigualdade 3, temos

$$\begin{aligned}
E[T(n)] &= \frac{2}{n}\sum_{k=0}^{n-1} E[(T(k)] + \Theta(n) \\
&\leq \frac{2}{n}\sum_{k=0}^{n-1}(a_k k \log k - b_k k) + cn \\
&\leq \frac{2}{n}\sum_{k=0}^{n-1}(a_k k \log k - b_k k) + cn \\
&\leq \frac{2}{n}a\sum_{k=0}^{n-1} k \log k - \frac{2}{n}b\sum_{k=0}^{n-1} k + cn \\
&\leq \frac{2}{n}a(\frac{1}{2}n^2 \log n - \frac{1}{8}n^2) - \frac{2}{n}b\frac{n(n-1)}{2} + cn \\
&\leq an \log n - a\frac{1}{4}n - (n-1)b + cn \\
&= an \log n + (-a\frac{1}{4} - b + c)n + b
\end{aligned}$$

Ou seja, para todo n existem a e b tais que $E[T(n)] \leq an \log n - bn$.

Concluímos então que $E[T(n)] = \Theta(n \log n)$.

3 Red-Black Trees

I am attaching a binary tree source code (`bst-0.0.cpp`) with the methods insert, delete, and print. Your job would be to implement a Red-Black Tree, adapting the functions insert, remove, and print as convenient. To test your code, you can follow the examples described in the document `anexo1.pdf`. Also, you might be interested in the document `anexo2.pdf` for a more detailed description of this tree, there is also some Java code that might be useful. However, your code must be in C++ and modifying the attached code. The grading will be as follow:

(a) **(2pts)** insert

(b) **(2pts)** remove

An example of the main function is:

```
int main() {
  // this constructor must call the function insert multiple times
  // respecting the order
  RBTree tree(41, 38, 31, 12, 19, 8);
  tree.print();

  // testing the remove function
  tree.remove(8);
  tree.print();
}
```

4 Radix Sort

(2pts) Your job is to implement the radix sort algorithm described in class. You must use python for this task. The following code is going to be used to test your implementation. You have to submit a notebook with your code.

```
def radix_sort(A, d, k):
  # A consists of n d-digit ints, with digits ranging 0 -> k-1
  #
  # implement your code here
  # return A_sorted


# Testing your function
A = [201, 10, 3, 100]
A_sorted = radix_sort(A, 3, 10)
print(A_sorted)
# output: [3, 10, 100, 201]
```