Quicksort

Universidade Federal do Ceará - Campus de Crateús

Roberto Cabral
rbcabral@crateus.ufc.br

30 de Outubro de 2017

Estrutura de Dados

Introdução

- Algoritmo proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado (ou pelo menos deveria ser).

Algoritmo

- Como o mergesort, é um algoritmo de divisão e conquista.
- Basicamente divide o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- As partições são combinadas para produzir a solução final.

O problema da separação

- O núcleo do algoritmo Quicksort é o seguinte problema da separação, que, grosseiramente, pode ser descrito como:

Problema da separação

rearranjar um vetor $v[p..r]$ de modo que todos os elementos pequenos fiquem na parte esquerda do vetor e todos os elementos grandes fiquem na parte direita.

O problema da separação

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos c.
- Os elementos do vetor que forem maiores que c serão considerados grandes e os demais serão considerados pequenos.
- É importante escolher c de tal modo que as duas partes do vetor rearranjado sejam estritamente menores que o vetor todo.
- A dificuldade está em resolver o problema da separação de maneira rápida sem usar muito espaço de trabalho.

O problema da separação

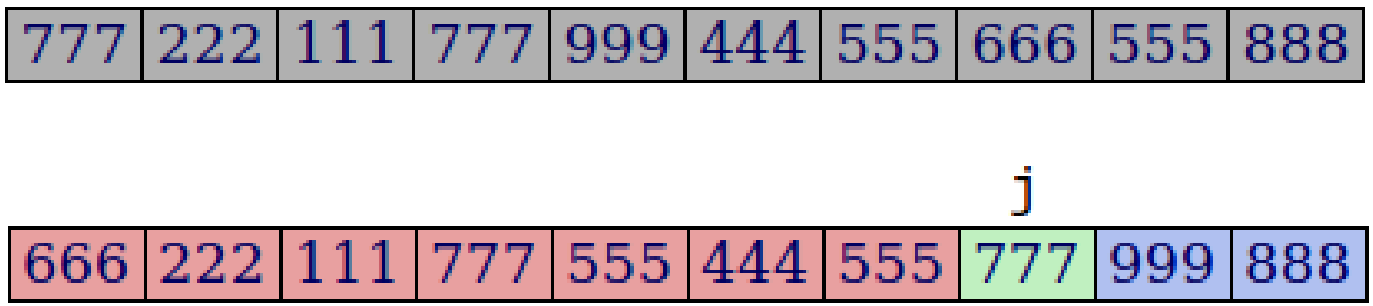
- O problema da separação admite muitas formulações concretas. Ex.:
  - rearranjar $v[p..r]$ de modo que tenhamos:

$$v[p..j] \leq v[j+1..r] \tag{1}$$

    - para algum j em $p..r-1$. (Nessa formulação, o pivô não é explícito.)
  - rearranjar $v[p..r]$ de modo que tenhamos:

$$v[p..j-1] \leq v[j] < v[j+1..r] \tag{2}$$

    - para algum j em $p..r$. (Aqui, v[j] é o pivô.)



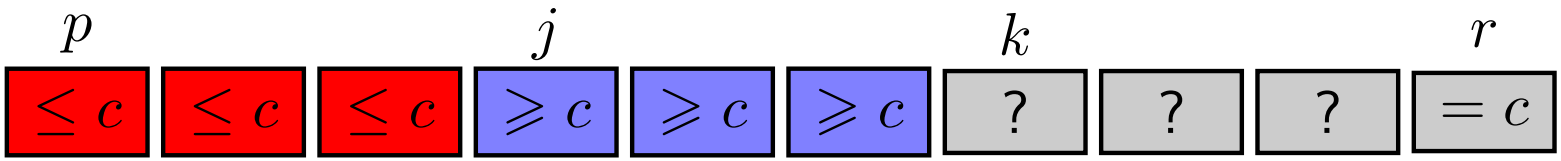
O algoritmo da separação

- A seguinte função é uma solução eficiente da formulação (2) do problema da separação.
- Ela começa por escolher um "pivô" c que definirá o significado de *pequeno* e *grande*.

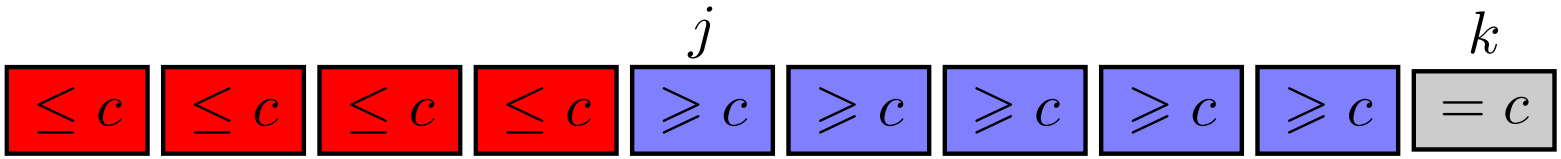
```
int separa (int p, int r, int *v){
        int c, j, k, t;
        c = v[r]; j = p;
        for(k = p; k < r; k++){
                if(v[k] <= c){
                        t = v[j]; v[j] = v[k]; v[k] = t;
                        j++;
                }
        }
        v[r] = v[j]; v[j] = c;
        return j;
}
```

O algoritmo da separação

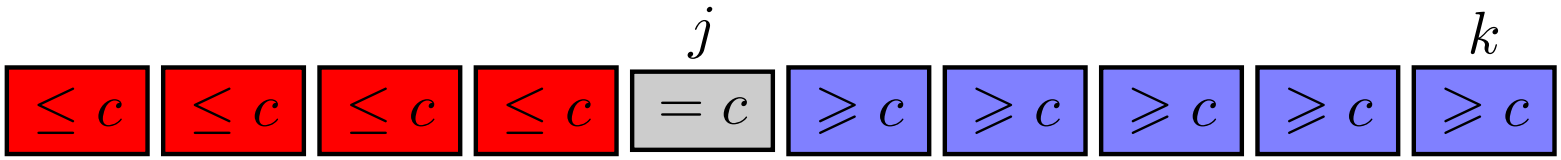
- No início de cada iteração valem os seguintes invariantes:
  - $v[p..r]$ é uma permutação do vetor original,
  - $v[p..j-1] \leq c < v[j..k-1]$, $v[r] = c$ e
  - $p \leq j \leq k \leq r$.



Início de uma iteração



Última passagem pelo ponto A.



Última passagem pelo ponto A.

O algoritmo da separação - Desempenho

- O consumo de tempo da função `separa` é proporcional ao número de iterações.
- Como o número de iterações é $r - p + 1$, podemos dizer que o consumo de tempo é proporcional ao número de elementos do vetor.

Algoritmo Quicksort básico

- Agora que resolvemos o problema da separação, podemos cuidar do Quicksort propriamente dito.
- O algoritmo usa a estratégia da divisão e conquista e tem a aparência de um Mergesort "ao contrário":

```
void quickSort (int p, int r, int *v)
{
   int j;                              // 1
   if (p < r) {                        // 2
      j = separa (p, r,v);            // 3
      quicksort (p, j-1,v);           // 4
      quicksort (j+1, r,v);           // 5
   }
}
```

O desempenho do Quicksort

- O tempo de execução do quicksort depende do particionamento ser balanceado ou não ser balanceado.
- Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto o mergesort.
- Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente tão lento quanto a ordenação por inserção.

Particionamento no pior caso

- O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com $n - 1$ elementos e um com 0 elementos.
- O particionamento custa o tempo $\theta(n)$.
- Se considerarmos que esse particionamento não balanceado ocorre em cada chamada recursiva, temos a seguinte recorrência para o tempo de execução:

$$T(n) = T(n-1) + T(0) + \theta(n).$$

- Intuitivamente, se somarmos os custos incorridos em cada nível da recursão, obtemos uma série aritmética, cujjo valor chega a $\theta(n^2)$.

Particionamento no pior caso

- Assim, se o particionamento é maximamente não balanceado em todo nível recursivo do algoritmo.
- O tempo de execução é $\theta(n^2)$.
- Além disso, o tempo de execução $\theta(n^2)$ ocorre quando o vetor de entrada já está completamente ordenado.
- Uma situação comum na qual a ordenação por inserção é executada no tempo de $O(n)$.

Particionamento no melhor caso

- Na divisão mais equitativa possível, `separa` produz dois subproblemas, cada um de tamanho não maior que $n/2$.
- Nesse caso, teríamos a seguinte recorrência para o tempo de execução:

$$T(n) = 2T(n/2) + \theta(n).$$

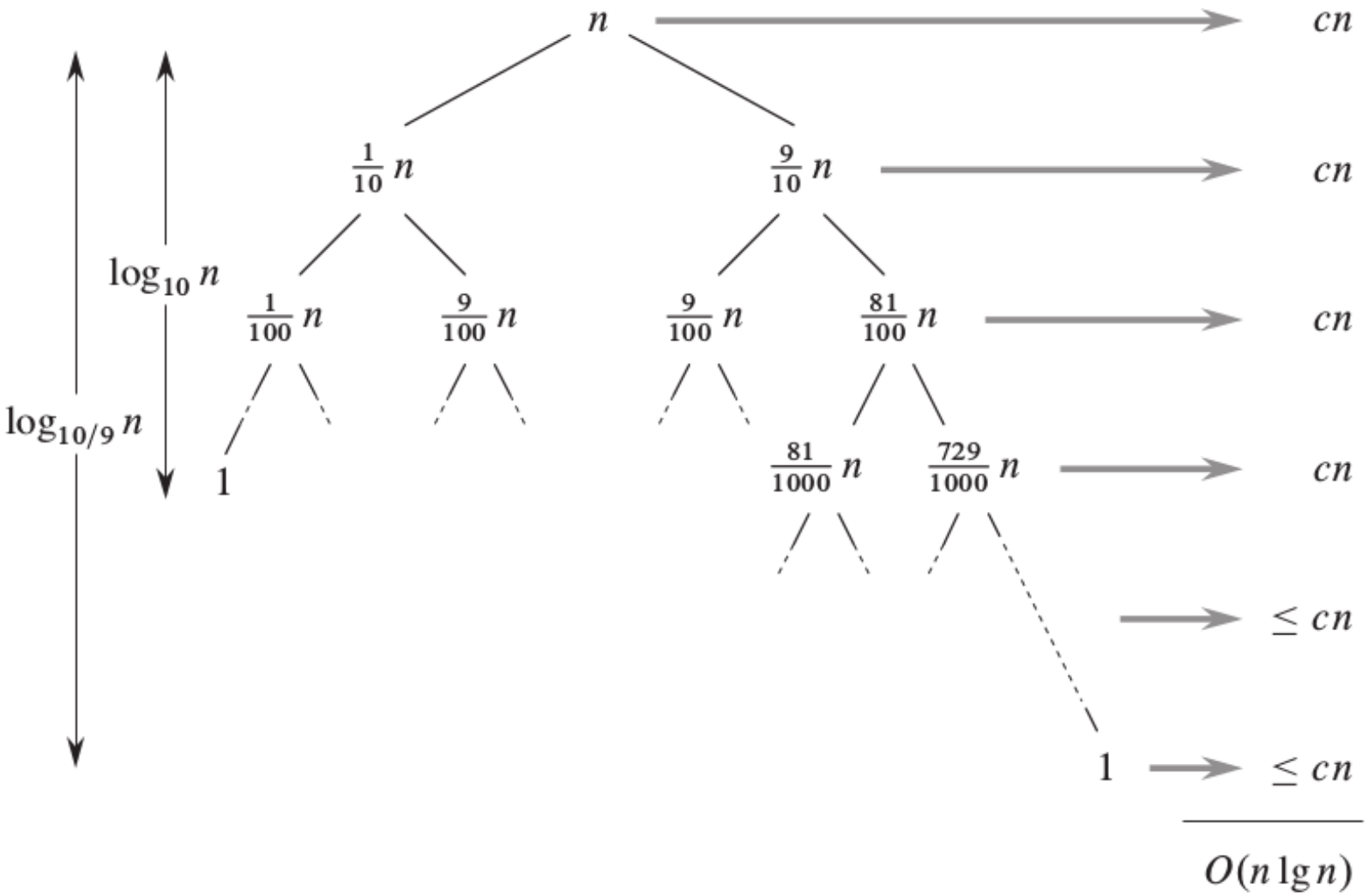
- Resolvendo a recorrência, temos que $T(n) = \theta(n \lg n)$.
- Balanceando igualmente os dois lados da partição em todo nível da recursão, obtemos um algoritmo assintoticamente mais rápido.

Particionamento balanceado

- O tempo de execução do caso médio do quicksort é muito mais próximo do melhor caso do que do pior caso.
- A chave para entender porque é entender como o equilíbrio do particionamento é refletido na recorrência que descreve o tempo de execução.
- Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que à primeira vista parece bastante desequilibrada
- Obteríamos a seguinte relação de recorrência:

$$T(n) = T(9n/10) + T(n/10) + cn.$$

Particionamento balanceado



Particionamento balanceado

- Todo nível da árvore tem custo cn até a recursão alcançar uma condição de contorno à profundidade $\log_{10} = \theta(\lg n)$.
- Daí em diante, os níveis tê no máximo o custo cn.
- A recursão termina na profundidade $\log_{10/9} = \theta(\lg n)$.
- Portanto, o custo total do quicksort é $O(n \log n)$.

Uma versão aleatorizada do quicksort

- Como poderíamos aleatorizar o vetor de modo a tentarmos não cair no pior caso do algoritmo?

```
int separaAleatorio(int p, int r, int *v){
  int i,t;
  i = random(p,r);
  t = v[r]; v[r] = v[i]; v[i] = t;
  return separa(p,r,v);
}
```

```
/*Cuidado! Função útil apenas para fins didáticos*/
int random(int p, int r){
  int a=0;
  srand(0);
  while(a < p)
    a = rand() % (r+1);
  return a;
}
```

Uma versão aleatorizada do quicksort

- Como poderíamos aleatorizar o vetor de modo a tentarmos não cair no pior caso do algoritmo?

```
void quickSortAleatorio (int p, int r, int *v)
{
   int j;                              // 1
   if (p < r) {                        // 2
      j = separaAleatorio(p, r,v);          // 3
      quicksortAleatorio(p, j-1,v);         // 4
      quicksortAleatorio(j+1, r,v);         // 5
   }
}
```

Quicksort

Universidade Federal do Ceará - Campus de Crateús

Roberto Cabral
rbcabral@crateus.ufc.br

30 de Outubro de 2017

Estrutura de Dados