Quicksort

Algoritmos e Estruturas de Dados II

História

- 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

Algoritmo

- Dividir 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

Particionamento

- A parte mais delicada do quicksort é o processo de partição
- O vetor **v** é rearranjado por meio da escolha arbitrária de um *pivô* **p**
- O vetor **v** é particionado em dois:
  - Partição esquerda: chaves ≤ **p**
  - Partição direita: chaves ≥ **p**

Algoritmo para particionamento

- Escolha arbitrariamente o pivô **p**
- Percorra a partir da esquerda até que **v[e] ≥ p**
- Percorra a partir da direita até que **v[d] ≤ p**
- Troque **v[e]** com **v[d]**
- Repita os passos anteriores até que **e** e **d** se cruzarem (**d < e**)

Exemplo de particionamento

- Selecionando o pivô como **`v[(d+e)/2]`**

C	A	M	I	N	H	O
C	A	M	I	N	H	O
C	A	M	I	N	H	O
C	A	M	I	N	H	O
C	A	M	I	N	H	O
C	A	H	I	N	M	O
C	A	H	I	N	M	O

- Ao final do particionamento
  - O pivô está em sua posição final
  - Elementos na partição da esquerda são menores
  - Elementos na partição da direita são maiores que o pivô

Exemplo quicksort

C	A	H	I	N	M	O
C	A	H	I			
C	A	H	I			
C	A	H	I			
A	C	H	I			
A	C	H	I			
A	C	H	I			
A	C	H	I			
A	C	H	I	N	M	O
				N	M	O
				N	M	O
				M	N	O
				M	N	O
A	C	H	I	M	N	O

Quicksort

```
void quicksort(struct item *v, int e, int d) {
   if(d <= e) return;
   int p = particao(v, e, d);
   quicksort(v, e, p-1);
   quicksort(v, p+1, d);
}

void particao(struct item *v, int e, int d) {
   int i = e; int j = d; struct item pivo = v[(d+e)/2];
   while(1) {
     while(v[i].chave < pivo.chave) i++;
     while(pivo.chave < v[j].chave) j--;
     if(i >= j) break;
     troca(v[i], v[j]); i++; j--;
   }
   troca(v[i], v[d]);
   return i;
}
```

Não funciona quando tem elementos repetidos!

Quicksort – Análise

- Pior caso
  - Acontece quando o pivô é sempre o maior ou menor elemento (partições de tamanho desequilibrado)

```
void quicksort(struct item *v, int e, int d) {
   if(d <= e) return;
   int p = particao(v, e, d);   /* p = d; */
   quicksort(v, e, p-1);        /* quicksort(v, e, d-1); */
   quicksort(v, p+1, d);        /* quicksort(v, d+1, d); */
}
```

n comparações

n-1 comparações

zero comparações

$$C(n) = n + (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Quicksort – Análise

- Exemplo de pior caso

T	**V**	**Y**	**Z**	**S**	**X**	**U**
T	**V**	**Y**	**U**	**S**	**X**	**Z**
T	**V**	**X**	**U**	**S**	**Y**	**Z**
T	**V**	**S**	**U**	**X**	**Y**	**Z**
T	**U**	**S**	**V**	**X**	**Y**	**Z**
T	**S**	**U**	**V**	**X**	**Y**	**Z**
S	**T**	**U**	**V**	**X**	**Y**	**Z**

Quicksort – Análise

- Melhor caso
  - Acontece quando as partições têm sempre o mesmo tamanho (partições balanceadas)

```
void quicksort(struct item *v, int e, int d) {
   if(d <= e) return;
   int p = particao(v, e, d); /* p = (d+e)/2; */
   quicksort(v, e, p-1);      /* quicksort(v, e, (d+e)/2-1); */
   quicksort(v, p+1, d);      /* quicksort(v, (d+e)/2+1, d); */
}
```

n comparações

n/2 comparações

n/2 comparações

$$C(n) = 2C(n/2) + n \approx n \lg n$$

Quicksort – Análise

- Caso médio
  - Partições podem ser feitas em qualquer posição no vetor

$$C(n) = n+1+\frac{1}{n}\sum_{i=1}^{n} C(i-1)+C(n-i)$$

$$= n+1+\frac{2}{n}\sum_{i=1}^{n} C(i-1)$$

$$nC(n)-(n-1)C(n-1) = 2n+2C(n-1)$$

$$\frac{C(n)}{(n+1)} = \frac{C(n-1)}{n}+\frac{2}{n+1} \approx 2\ln n = 1{,}39\lg n$$

Quicksort – Análise

- Melhor caso

$$C(n) \approx n \lg n = O(n \lg n)$$

- Caso médio

$$C(n) \approx 1{,}39 n \lg n = O(n \lg n)$$

- Pior caso

$$C(n) = \frac{n(n+1)}{2} = O(n^2)$$

Vantagens e desvantagens

- Vantagens:
  - Melhor opção para ordenar vetores grandes
  - Muito rápido por que o laço interno é simples
  - Memória auxiliar para a pilha de recursão é pequena
  - Complexidade no caso médio é $O(n \lg(n))$

- Desvantagens:
  - Não é estável (não conhecemos forma eficiente para tornar o quicksort estável)
  - Pior caso é quadrático

Otimização – Mediana de três

- Para evitar o pior caso do quicksort, podemos escolher o pivô como a mediana de três elementos

T	**V**	**Y**	**Z**	**S**	**X**	**U**
T	**V**	**Y**	**Z**	**S**	**X**	**U**
T	**S**	**U**	**Z**	**Y**	**X**	**V**

  - Aumentar o número de elementos considerados na mediana, por exemplo pra 5 ou 9, não ajuda muito

Otimização – Partição menor primeiro

- Implementações não recursivas do quicksort precisam manter uma pilha auxiliar
- No pior caso, a pilha auxiliar pode ter tamanho n
- Para limitar o crescimento da pilha auxiliar basta ordenar a partição menor primeiro

T	V	Y	Z	S	X	U
T	V	Y	Z	S	X	U
T	S	U	Z	Y	X	V
T	S	U				

Otimização – Partições pequenas

- Quando a partição a ordenar é pequena, o método de ordenação por inserção é mais rápido que o quicksort
  - Não tem chamadas recursivas
  - Laço interno muito eficiente

T	**V**	**Y**	**Z**	**S**	**X**	**U**
T	**V**	**Y**	**Z**	**S**	**X**	**U**
T	**S**	**U**	**Z**	**Y**	**X**	**V**
T	**S**	**U**				

Pontos extras

- 1 ponto para os três primeiros alunos que apresentarem código do quicksort que usa mediana de três
- 1 ponto para os três primeiros alunos que apresentarem código do quicksort que usa inserção em partições pequenas
- 1 ponto para os três primeiros alunos que apresentarem quicksort não recursivo e que limita o tamanho da pilha auxiliar ordenando partições pequenas primeiro
- Cada aluno pode ganhar apenas 1 dos pontos acima
- A solução deve ser construída como extensão do código colocado na página do curso