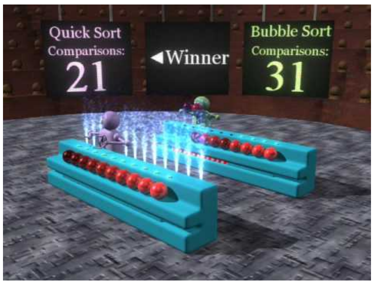
Ordenação: algoritmo Quicksort



Fonte:
https://www.youtube.com/watch?v=vxENKlcs2Tw/

PF 11
http://www.ime.usp.br/ pf/algoritmos/aulas/quick.html

Problema da separação

Problema: Rearranjar um dado vetor v[p : r] e devolver um índice q, $p \leq q < r$, tais que

$$v[p:q] \leq v[q] < v[q{+}1:r]$$

Entra:

	p										r
v	99	33	55	77	11	22	88	66	33	44	

Problema da separação

Problema: Rearranjar um dado vetor v[p : r] e devolver um índice q, $p \leq q < r$, tais que

$$v[p:q] \leq v[q] < v[q{+}1:r]$$

Entra:

	p										r
v	99	33	55	77	11	22	88	66	33	44	

Sai:

	p				q						r
v	33	11	22	33	44	55	99	66	77	88	

Separe

	p										r
v	99	33	55	77	11	22	88	66	33	44	

Separe

i	j									x
v	99	33	55	77	11	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i		j							x
v	33	99	55	77	11	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i			j						x
v	33	99	55	77	11	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i				j					x
v	33	99	55	77	11	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i				j					x
v	33	99	55	77	11	22	88	66	33	44
		i				j				x
v	33	11	55	77	99	22	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i				j					x
v	33	99	55	77	11	22	88	66	33	44
		i				j				x
v	33	11	55	77	99	22	88	66	33	44
			i				j			x
v	33	11	22	77	99	55	88	66	33	44

Separe

i		j								x
v	99	33	55	77	11	22	88	66	33	44
	i				j					x
v	33	99	55	77	11	22	88	66	33	44
		i				j				x
v	33	11	55	77	99	22	88	66	33	44
			i					j		x
v	33	11	22	77	99	55	88	66	33	44

Separe

i		j								x	
v	99	33	55	77	11	22	88	66	33	44	
	i				j					x	
v	33	99	55	77	11	22	88	66	33	44	
		i				j				x	
v	33	11	55	77	99	22	88	66	33	44	
			i						j	x	
v	33	11	22	77	99	55	88	66	33	44	

Separe

i		j								x	
v	99	33	55	77	11	22	88	66	33	44	
	i				j					x	
v	33	99	55	77	11	22	88	66	33	44	
		i				j				x	
v	33	11	55	77	99	22	88	66	33	44	
			i						j	x	
v	33	11	22	77	99	55	88	66	33	44	
				i						j	
v	33	11	22	33	99	55	88	66	77	44	

Separe

i		j								x	
v	99	33	55	77	11	22	88	66	33	44	
	i				j					x	
v	33	99	55	77	11	22	88	66	33	44	
		i				j				x	
v	33	11	55	77	99	22	88	66	33	44	
			i						j	x	
v	33	11	22	77	99	55	88	66	33	44	
				i						j	
v	33	11	22	33	99	55	88	66	77	44	
	p				q						r
v	33	11	22	33	44	55	88	66	77	99	

Função separe

Rearranja v[p : r] de modo que $p \le q < r$ e
v[p : q] $\le$ v[q] $<$ v[q+1 : r].
A função devolve q.

```
def separe (p, r, v):
1    i = p-1
2    x = v[r-1]
3    for j in range(p,r):   # *A*
4        if v[j] <= x:
5            i += 1
6            v[i], v[j] = v[j], v[i]
7    return i
```

Invariantes

Em *A* vale que

(i0) v[p : i+1] $\le$ x $<$ v[i+1 : j]

	p		i			j				x	r
v	33	11	22	77	99	55	88	66	33	44	

Consumo de tempo

Supondo que a execução de cada linha consome 1 unidade de tempo.
Qual o consumo de tempo da função separe em termos de $n := r - p$?

linha		consumo de todas as execuções da linha
1–2	=	?
3	=	?
4	=	?
5–6	=	?
7	=	?
total	=	?

Consumo de tempo

Supondo que a execução de cada linha consome 1 unidade de tempo.

Qual o consumo de tempo da função separe em termos de $n := r - p$?

linha		consumo de todas as execuções da linha
1–2	$=$	1
3	$=$	$n + 1$
4	$=$	n
5–6	$\leq$	n
7	$=$	1
total	$\leq$	$3n + 3 \quad = O(n)$

Conclusão

O consumo de tempo da função separe é proporcional a n.

O consumo de tempo da função separe é $O(n)$.

Quicksort

Rearranja $v[p : r]$ em ordem crescente.

```
def quick_sort (p, r, v):
1    if p < r-1:
2        q = separe(p,r,v)
3        quick_sort(p, q, v)
4        quick_sort(q+1, r, v)
```

	p										r
v	99	33	55	77	11	22	88	66	33	44	

Quicksort

Rearranja $v[p : r]$ em ordem crescente.

```
def quick_sort (p, r, v):
1    if p < r-1:
2        q = separe(p,r,v)
3        quick_sort(p, q, v)
4        quick_sort(q+1, r, v)
```

	p				q						r
v	33	11	22	33	44	55	88	66	77	99	

No começo da linha 3,

$$v[p : q] \leq v[q] < v[q+1 : r]$$

Quicksort

Rearranja $v[p : r]$ em ordem crescente.

```
def quick_sort (p, r, v):
1    if p < r-1:
2        q = separe(p,r,v)
3        quick_sort(p, q, v)
4        quick_sort(q+1, r, v)
```

	p				q						r
v	11	22	33	33	44	55	88	66	77	99	

Quicksort

Rearranja $v[p : r]$ em ordem crescente.

```
def quick_sort (p, r, v):
1    if p < r-1:
2        q = separe(p,r,v)
3        quick_sort(p, q, v)
4        quick_sort(q+1, r, v)
```

	p				q						r
v	11	22	33	33	44	55	66	77	88	99	

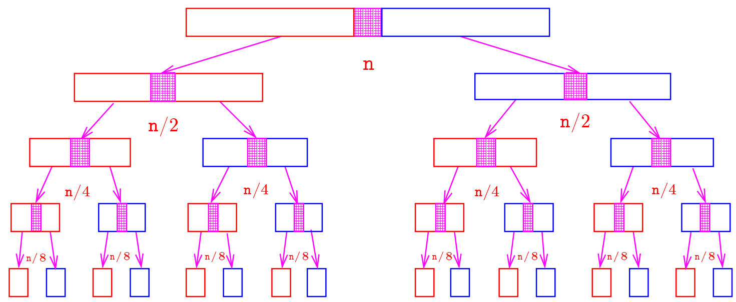
Quicksort

Rearranja $v[p:r]$ em ordem crescente.

```
def quick_sort (p, r, v):
    if p < r-1:
        q = separe(p,r,v)
        quick_sort(p, q, v)
        quick_sort(q+1, r, v)
```

Consumo de tempo?

Consumo de tempo: versão MAC0122



Consumo de tempo: versão MAC0122

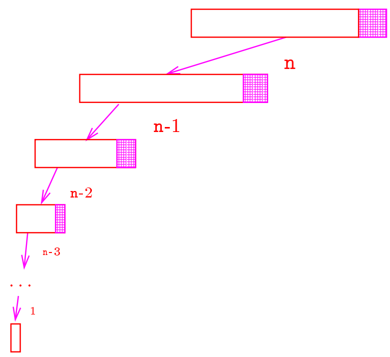
O consumo de tempo em cada nível da recursão é proporcional a n.

No melhor caso, em cada chamada recursiva, q é $\approx (p+r)/2$.

Nessa situação há cerca de $\lg n$ níveis de recursão.

nível	consumo de tempo (proporcional a)
1	$\approx n$
2	$\approx n/2 + n/2$
3	$\approx n/4 + n/4 + n/4 + n/4 + n/4$
…	…
$\lg n$	$\approx 1 + 1 + 1 + 1 \cdots + 1 + 1 + 1 + 1$
Total	$\approx n \lg n = O(n \lg n)$

Consumo de tempo: versão MAC0122



Consumo de tempo: versão MAC0122

No pior caso, em cada chamada recursiva, o valor de q devolvido por `separe` é $\approx p$ ou $\approx r$.

Nessa situação há cerca de n níveis de recursão.

nível	consumo de tempo (proporcional a)
1	$\approx n$
2	$\approx n - 1$
3	$\approx n - 2$
4	$\approx n - 3$
…	…
n	≈ 1
Total	$\approx n(n-1)/2 = O(n^2)$

Quicksort no melhor caso

No melhor caso, em cada chamada recursiva q é aproximadamente $(p+r)/2$.

> O consumo de tempo da função `quick_sort` no melhor caso é proporcional a $n \log n$.

> O consumo de tempo da função `quick_sort` no melhor caso é $O(n \log n)$.

Quicksort no pior caso

O consumo de tempo da função `quick_sort` no pior caso é proporcional a n^2.

O consumo de tempo da função `quick_sort` no pior caso é $O(n^2)$.

O consumo de tempo da função `quick_sort` é $O(n^2)$.

Discussão geral

Pior caso, melhor caso, todos os casos?!?!

Dado um algoritmo $\mathcal{A}$ o que significam as expressões:

- $\mathcal{A}$ é $O(n^2)$ no pior caso.
- $\mathcal{A}$ é $O(n^2)$ no melhor caso.
- $\mathcal{A}$ é $O(n^2)$.

Análise experimental

Algoritmos implementados:

mergeR	`merge_sort` recursivo.
mergeI	`merge_sort` iterativo.
quick	`quick_sort` recursivo.
sort	método `sort` do Python.

Análise experimental

A plataforma utilizada nos experimentos foi um computador rodando Ubuntu GNU/Linux 3.19.0-33

Python:
`Python 3.4.3.`

Computador:

```
 model name:  Intel(R) Core(TM)2 Quad CPU Q6600 @
2.40GHz
cpu MHz :  1596.000
cache size:  4096 KB
MemTotal :  3354708 kB
```

Estudo empírico (aleatório)

n	mergeR	mergeI	quick	sort
1024	0.01	0.01	0.00	0.00
2048	0.01	0.01	0.01	0.00
4096	0.03	0.03	0.02	0.00
8192	0.06	0.06	0.05	0.00
16384	0.12	0.12	0.10	0.01
32768	0.26	0.25	0.20	0.02
65536	0.55	0.54	0.45	0.03
131072	1.17	1.15	0.98	0.07
262144	2.49	2.47	2.09	0.17
524288	5.30	5.21	4.51	0.38
1048576	11.19	11.09	9.44	0.85

Tempos em segundos.

Estudo empírico (decrescente)

n	mergeR	mergeI	quick	sort
1024	0.01	0.01	*	0.00
2048	0.01	0.01	*	0.00
4096	0.03	0.03	*	0.00
8192	0.06	0.05	*	0.00
16384	0.12	0.11	*	0.00
32768	0.25	0.24	*	0.00
65536	0.53	0.51	*	0.00
131072	1.12	1.08	*	0.00
262144	2.35	2.27	*	0.01
524288	4.99	4.85	*	0.01
1048576	10.29	9.86	*	0.03

Para n=1024 `quick_sort` apresentou `RuntimeError`

Estudo empírico (crescente)

n	mergeR	mergeI	quick	sort
1024	0.01	0.01	*	0.00
2048	0.01	0.01	*	0.00
4096	0.03	0.03	*	0.00
8192	0.06	0.05	*	0.00
16384	0.12	0.11	*	0.00
32768	0.25	0.24	*	0.00
65536	0.54	0.51	*	0.00
131072	1.13	1.08	*	0.00
262144	2.35	2.27	*	0.01
524288	4.91	4.74	*	0.01
1048576	10.24	9.84	*	0.03

Para n=1024 `quick_sort` apresentou `RuntimeError`

Consumo de tempo: outra versão

Quanto tempo consome a função `quick_sort` em termos de $n := r - p$?

linha		consumo de todas as execuções da linha
1	=	?
2	=	?
3	=	?
4	=	?
total	=	????

Consumo de tempo: outra versão

Quanto tempo consome a função `quick_sort` em termos de $n := r - p$?

linha		consumo de todas as execuções da linha
1	=	$O(1)$
2	=	$O(n)$
3	=	$T(k)$
4	=	$T(n-k-1)$
total	=	$T(k) + T(n-k-1) + O(n+1)$

$$0 \leq k := q - p \leq n - 1$$

Recorrência: outra versão

$T(n)$:= consumo de tempo máximo quando $n := r - p$

$T(0) = O(1)$

$T(1) = O(1)$

$T(n) = T(k) + T(n-k-1) + O(n)$ para $n = 2, 3, 4, \ldots$

Recorrência: outra versão

$T(n)$:= consumo de tempo máximo quando $n := r - p$

$T(0) = O(1)$

$T(1) = O(1)$

$T(n) = T(k) + T(n-k-1) + O(n)$ para $n = 2, 3, 4, \ldots$

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + O(n)$$

$T(n)$ é $O(???)$.

Recorrência: outra versão

$T(n)$:= consumo de tempo máximo quando $n := r - p$

$T(0) = O(1)$

$T(1) = O(1)$

$T(n) = T(k) + T(n-k-1) + O(n)$ para $n = 2, 3, 4, \ldots$

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + O(n)$$

$T(n)$ é $O(n^2)$.

Demonstração: ...

Recorrência cuidadosa: ...

$T(n)$:= consumo de tempo máximo quando
$n = r - p$

$$T(0) = O(1)$$
$$T(1) = O(1)$$
$$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + O(n)$$
$$\text{para } n = 2, 3, 4, \ldots$$

$T(n)$ é $O(n^2)$.

Demonstração: ...

quick_sort: versão iterativa

Na versão iterativa devemos administrar uma pilha que simula a pilha da recursão.

A pilha armazenará os índices que delimitam segmentos do vetor que estão à espera de ordenação.

Na implementação utilizaremos a classe Pilha():

- Pilha(): cria uma pilha;
- vazia(): método que retorna True se e só se a pilha não está vazia;
- empilha(): método que põe um intervalo no topo da pilha;
- desempilha(): método que tira e retorna o intervalo no topo da pilha;

quick_sort: versão iterativa

```
def quick_sort (v):
    n = len(v)
    p = 0 # inicio segmentos
    r = n # fim segmentos
    # crie e inicialize a pilha
    pilha = Pilha()
    pilha.empilha([p,r])
```

quick_sort: versão iterativa

```
while not pilha.vazia():
    p, r = pilha.desempilha()
    if p < r-1:
        q = separe(p,r,v);
        # segmento inicial
        pilha.empilha([p,q])
        # segmento final
        pilha.empilha([q+1,r])
```

Exercícios

Qual a diferença das duas versões?

Qual o comportamento da pilha se $v[0:n]$ é crescente?

Ver exercícios em

http://www.ime.usp.br/~pf/algoritmos/aulas/quick.html

k-ésimo menor elemento

x é o k-**ésimo menor elemento** de um vetor $v[0:n]$ se em um rearranjo crescente de v, x é o valor na posição v[k-1].

Problema: encontrar k-ésimo menor elemento de um vetor v[0:n-1], supondo $1 \le k \le n$.

Exempo: **33** é o 4o. menor elemento de:

	0										n
v	99	33	55	77	11	22	88	66	33	44	

k-ésimo menor elemento

x é o **k-ésimo menor elemento** de um vetor v[0 : n] se em um rearranjo crescente de v, x é o valor na posição v[k-1].

Problema: encontrar k-ésimo menor elemento de um vetor v[0:n-1], supondo $1 \leq k \leq n$.

Exempo: **33** é o 4o. menor elemento de:

	0										n
v	99	33	55	77	11	22	88	66	33	44	

pois: no vetor ordem crescente temos

	0										n
v	11	22	33	**33**	44	55	66	77	88	99	

Solução inspirada em selecao()

Algoritmo baseado em ordenação por seleção.
Ao final o k-ésimo menor elemento está em v[k-1].

```
def k_esimo (k, n, v):
1   for i in range(k):
2       min = i
3       for j in range(i+1,n):
4           if v[j] < v[min]:  min = j
5       v[i], v[min] = v[min], v[i]
```

Invariantes

Relações **invariantes** chaves dizem que em /*A*/ vale que:

♡ (i0) $v[0:i]$ é crescente e
$v[0:i] \leq v[i:n]$

0				i							n
10	20	38	44	75	50	55	99	85	50	60	

Supondo que a invariantes valem.
Correção do algoritmo é evidente.

No início da última iteração das linhas 1–5 tem-se que $i = k$.
Da invariante conclui-se que $v[0:k]$ é crescente.
e que $v[k-1] \leq v[k:n]$.

Mais invariantes

Na linha 1 vale que: (i1) $v[i] \leq v[i+1:n]$;
Na linha 3 vale que: (i2) $v[min] \leq v[i:j]$

0				i	min		j				n
10	20	38	44	75	50	55	99	85	50	60	

invariantes (i1),(i2)
+ condição de parada do for da linha 3
+ troca linha 5 $\Rightarrow$ validade (i0)

Verifique!

Consumo de tempo

Se a execução de cada linha de código consome 1 unidade de tempo o consumo total é:

linha	todas as execuções da linha			
1	$=$	$k+1$	$=$	$O(k)$
2	$=$	k	$=$	$O(k)$
3	$=$	$n+(n-1)+\cdots+(n-k+1)$	$=$	$O(kn)$
4	$=$	$(n-1)+(n-2)+\cdots+(n-k+1)$	$=$	$O(kn)$
5	$=$	k	$=$	$O(k)$
total	$=$	$O(2kn+3k)$	$=$	$O(kn)$

Conclusão

O consumo de tempo do algoritmo k_esimo no pior caso e no no melhor caso é proporcional a kn.

O consumo de tempo do algoritmo k_esimo é $O(kn)$.

Solução inspirada em quick_sort()

Ao final o k-ésimo menor elemento está em v[k-1].
Primeira chamada: k_esimo(k,0,n,v).

```
def k_esimo (k, p, r, v):
    q = separe(p, r, v)
    if q == k-1:  return None
    if q >= k   :  k_esimo(k, p   , q, v)
    if q <  k-1:  k_esimo(k, q+1, r, v)
```

Consumo de tempo?

k_esimo: versão iterativa

```
def k_esimo (k, n, v):
    p = 0
    r = n
    q = separe(p,r,v)
    while q != k-1:
        if q >= k:  r = q
        if q <  k:  p = q + 1
        q = separe(p, r, v)
```

Exercícios

Qual o consumo de tempo no melhor caso do algoritmo k_esimo inspirado em quick_sort?

Qual o consumo de tempo no pior caso do algoritmo k_esimo inspirado em quick_sort?

Tente determinar experimentalmente o consumo de tempo do algoritmo k_esimo inspirado em quick_sort.

Sob *hipóteses razoáveis* é possível mostrar que o consumo de tempo esperado do algoritmo k_esimo inspirado no quick_sort é proporcional a n.