Estruturas de Dados / Programação 2
Algoritmos de Ordenação - Parte 1

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Last class...

```
for (i = 0; i < n; i++) {
  min = i;
  for (j = i + 1; j < n; j++)
    if (s[j] < s[min]) min = j;
  swap(&s[i], &s[min]);
}
```

$O(n^2)$

What does it do?

```
void selection_sort(int s[], int n)
{
  int i, j, min;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
      if (s[j] < s[min]) min = j;
    swap(&s[i], &s[min]);
  }
}
```

S E L E C T I O N S O R T
C E L E S T I O N S O R T
C E L E S T I O N S O R T
C E E L S T I O N S O R T
C E E I S T L O N S O R T
...

Many sorting algorithms!

Bubble Sort

Watch the video!

Bubble Sort

- We need two nested loops
  - First: takes care of the “bubble”
  - Second: makes the first repeat to take care of other “bubbles”
- Notice that when reaching the end of the array for the first time, the biggest value will be sorted

```
void bubble_sort(int *v, int size)
{
  for (j = 1; j <= size; j++) {
    for (i = 0; i < size - 1; i++) {
      if (v[i] > v[i+1])
        swap(&v[i], &v[i+1]);
    }
  }
}
```

How to improve this algorithm version?

- The sorted elements are being traversed again…
- In this case, we do not need to traverse them…
  - “j” varies from 1 to the size (5 elements)
  - But 2 elements are already sorted!

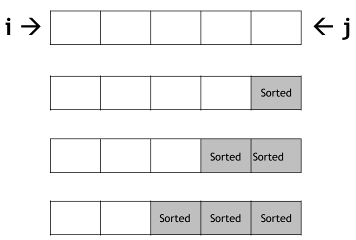
```
for (j = 1; j <= size; j++)
  for (i = 0; i < size - 1; i++)
```

2	3	1	4	5
0	1	2	3	4

Not traversing sorted elements

- Two indexes: the first decreasing and the second increasing

```
for (j = size - 1; j >= 0; j--)
  for (i = 0; i < j; i++)
```



Bubble Sort (Version 2)

- Now we do not traverse sorted elements…

```
void bubble_sort(int *v, int size) {
  for (j = size - 1; j >= 0; j--) {
    for (i = 0; i < j; i++) {
      if (v[i] > v[i+1])
        swap(&v[i], &v[i+1]);
    }
  }
}
```

Execution: Bubble Sort

6 5 3 1 8 7 2 4

Efficiency (Version 1)

- Inner loop executes "n – 1" times (from 0 to n-1)
- This happens "n" times (outer loop)

```
for (j = 1; j <= size; j++)
  for (i = 0; i < size - 1; i++)
```

i	Inner loop iterations
0	n - 1
1	n - 1
…	n - 1

$n(n - 1) = O(N^2)$

Efficiency (Version 2): try yourself!

- Be careful! The inner loop **depends on** the outer loop index

```
for (j = size - 1; j >= 0; j--)
  for (i = 0; i < j; i++)
```

i	Inner loop iterations
0	n - 1
1	n - 2
…	…
n - 1	1

$$\sum_{i=1}^{n-1} i = O(N^2)$$

Quick Sort

Quick Sort

- First step: we pick an element called pivot in each step
- Rearrange the array in such a way that:
  - Elements larger than the pivot appear on the right side of the pivot
  - Elements smaller than the pivot appear on the left side of the pivot
- In all subsequent iterations, the pivot position remains unchanged, because it has been put in its correct position

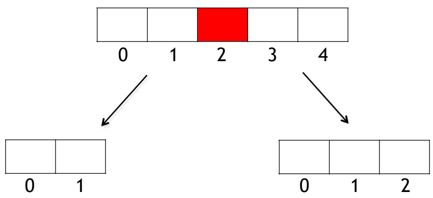
Quick Sort

- Suppose the pivot is v[2]

0	1	2	3	4

- After the first iteration, the array is rearranged

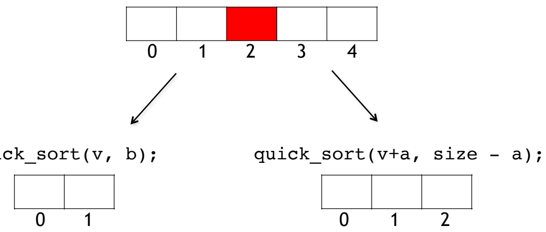
Then, we call the quick sort function recursively…




```
void quick_sort(int *v, int size);
```

How to divide the array “v”?

- a = b = 2




```
void quick_sort(int *v, int size);
```

```
void quick_sort(int *v, int size)
{
  if (size <= 1) {
    return;
  } else {
    int pivot = v[size / 2];
    int a = 0;
    int b = size - 1;

    while (a < b) {
      while (v[a] < pivot) a++;
      while (v[b] > pivot) b--;
      if (a < b)
        swap(&v[a], &v[b]);
    }
    quick_sort(v, b);
    quick_sort(v+a, size - a);
  }
}
```

There is something wrong with this algorithm version. Fix it!

Quick Sort (recursive version in Haskell)

```
quickSort :: [Int] -> [Int]
quickSort [] = []
quickSort (a:as) = quickSort [x | x <-as, x < a]
                   ++ [a] ++
                   quickSort [x | x <-as, x >= a]
```

Execution: Quick Sort

6 5 3 1 8 7 2 4

Efficiency

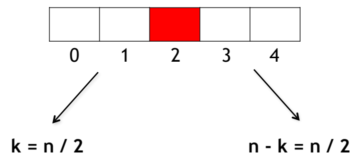
- As we mentioned, we divide the array in two parts
  - One part has size “k”
  - The other one has size “n – k”
  - Both of these parts still need to be sorted
- To rearrange the array, we have O(n)
  - Suppose “c.n”, where “c” is a constant
- T(n), to sort “n” elements is:

$$T(n) = T(k) + T(n - k) + c.n$$

Recurrence equation

Efficiency: best case

- The pivot divides the array into two exactly equal parts in every step

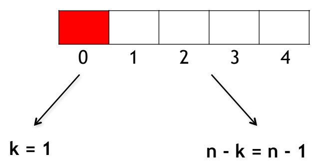


$$T(n) = 2T(n/2) + n$$

$$O(n \log n)$$

Efficiency: worst case

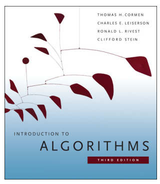
- The pivot is the smallest (or largest) element of the array in every step...



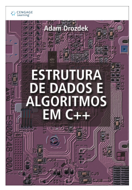
$$T(n) = T(1) + T(n - 1) + n$$

$$O(n^2)$$

References



Chapter 7



Chapter 9