Data Structures and Algorithms

Part 1.4

Werner Nutt

DSA, Part 1:

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

Sorting

- Sorting is a classical and important algorithmic problem.
  - For which operations is sorting needed?
  - Which systems implement sorting?
- We look at sorting arrays
  (in contrast to files, which restrict random access)
- A key constraint are the restrictions on the space:
  in-place sorting algorithms (no extra RAM).
- The run-time comparison is based on
  - the number of comparisons (C) and
  - the number of movements (M).

Sorting

- Sorting is a classical and important algorithmic problem.
  - For which operations is sorting needed?
  - Which systems implement sorting?
- We look at sorting arrays
  (in contrast to files, which restrict random access)
- A key constraint are the restrictions on the space:
  in-place sorting algorithms (no extra RAM).
- The run-time comparison is based on
  - the number of comparisons (C) and
  - the number of movements (M).

Sorting

- Simple sorting methods use roughly ***n * n*** comparisons
  - Insertion sort
  - Selection sort
  - Bubble sort

- Fast sorting methods use roughly ***n * log n*** comparisons
  - Merge sort
  - Heap sort
  - Quicksort

What's the point of studying those simple methods?

Example 2: Sorting

INPUT

sequence of *n* numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7

OUTPUT

a permutation of the input sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

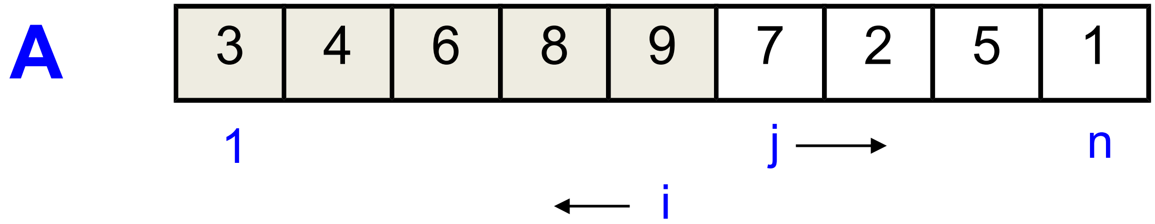
2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Insertion Sort



Strategy

- Start with one sorted card.
- Insert an unsorted card at the correct position in the sorted part.
- Continue until all unsorted cards are inserted/sorted.

44	**55**	**12**	**42**	**94**	**18**	**06**	**67**
44	55	**12**	**42**	**94**	**18**	**06**	**67**
12	44	55	**42**	**94**	**18**	**06**	**67**
12	42	44	55	**94**	**18**	**06**	**67**
12	42	44	55	94	**18**	**06**	**67**
12	18	42	44	55	94	**06**	**67**
06	12	18	42	44	55	94	**67**
06	12	18	42	44	55	67	94

Insertion Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

for j := 2 to n do // A[1..j-1] sorted
  key := A[j]; i := j-1;
  while i > 0 and A[i] > key do
    A[i+1] := A[i];  i--;
  A[i+1] := key
```

The number of comparisons during the jth iteration is

– at least 1: $C_{min} = \sum_{j=2}^{n} 1 = n - 1$

– at most j-1: $C_{max} = \sum_{j=2}^{n} j-1 = (n*n - n)/2$

Insertion Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

for j := 2 to n do // A[1..j-1] sorted
  key := A[j]; i := j-1;
  while i > 0 and A[i] > key do
    A[i+1] := A[i];  i--;
  A[i+1] := key
```

The number of comparisons during the jth iteration is

– at least 1: $C_{min} = \sum_{j=2}^{n} 1 = n - 1$

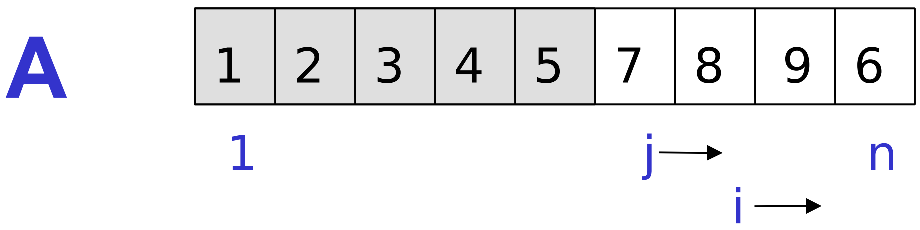
– at most j-1: $C_{max} = \sum_{j=2}^{n} j-1 = (n*n - n)/2$

Insertion Sort/3

- The number of comparisons during the jth iteration is:
  - j/2 average: $C_{avg} = \sum_{j=2}^{n} j/2 = (n*n + n - 2)/4$

- The number of movements is Ci+1:
  - $M_{min} = \sum_{j=2}^{n} 2 = 2*(n-1)$,
  - $M_{avg} = \sum_{j=2}^{n} j/2+1 = (n*n + 5n - 6)/4$
  - $M_{max} = \sum_{j=2}^{n} j = (n*n + n - 2)/2$

Selection Sort



Strategy

- Start empty handed.
- Enlarge the sorted part by swapping the first element of the unsorted part with the smallest element of the unsorted part.
- Continue until the unsorted part consists of one element only.

44	55	12	42	94	18	06	67
06	55	12	42	94	18	44	67
06	12	55	42	94	18	44	67
06	12	18	42	94	55	44	67
06	12	18	42	94	55	44	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Selection Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤A[2]≤…≤A[n]

for j := 1 to n-1 do // A[1..j-1] sorted and minimum
  key := A[j]; ptr := j
  for i := j+1 to n do
    if A[i] < key then key := A[i]; ptr := i;
  A[ptr] := A[j]; A[j] := key
```

The number of comparisons is independent of the original ordering (this is a less natural behavior than insertion sort):

$$C = \sum_{j=1}^{n-1} (n-j) = \sum_{k=1}^{n-1} k = (n*n - n)/2$$

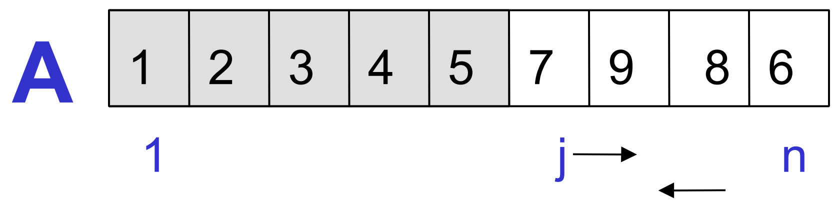
Selection Sort/3

The number of movements is:

$$M_{min} = \sum_{j=1}^{n-1} 3 = 3*(n-1)$$

$$M_{max} = \sum_{j=1}^{n-1} n\text{-}j+3 = (n*n - n)/2 + 3*(n-1)$$

Bubble Sort



Strategy

- Start from the back and compare pairs of adjacent elements.
- Swap the elements if the larger comes before the smaller.
- In each step the smallest element of the unsorted part is moved to the beginning of the unsorted part and the sorted part grows by one.

44	55	12	42	94	18	06	67
06	44	55	12	42	94	18	67
06	12	44	55	18	42	94	67
06	12	18	44	55	42	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94

Bubble Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤… ≤A[n]

for j := 2 to n do // A[1..j-2] sorted and minimum
  for i := n to j do
    if A[i-1] > A[i] then
      key := A[i-1];
      A[i-1] := A[i];
      A[i]:= key
```

The number of comparisons is independent of the original ordering:

$$C = \sum_{j=2}^{n} (n - j + 1) = (n*n - n)/2$$

Bubble Sort/3

The number of movements is:

$$M_{min} = 0$$

$$M_{max} = \sum_{j=2}^{n} 3(n-j+1) = 3*n*(n - 1)/2$$

$$M_{avg} = \sum_{j=2}^{n} 3(n-j+1)/2 = 3*n*(n - 1)/4$$

Properties of a Sorting Algorithm

- Efficient: has low (worst case) runtime
- In place: needs (almost) no additional space (fixed number of scalar variables)
- Adaptive: performs little work if the array is already (mostly) sorted
- Stable: does not change the order of elements with equal key values
- Online: can sort data as it receives them

Sorting Algorithms: Properties

Which algorithm has which property?

	Adaptive	Stable	Online
Insertion Sort			
Selection Sort			
Bubble Sort			

Summary

- Precise problem specification is crucial.
- Precisely specify Input and Output.
- Pseudocode, Java, C, … is largely equivalent for our purposes.
- Recursion: procedure/function that calls itself.
- Sorting: important problem with classic solutions.