Data Structures and Algorithms

Chapter 1.4

Werner Nutt

DSA, Chapter 1:

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

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?

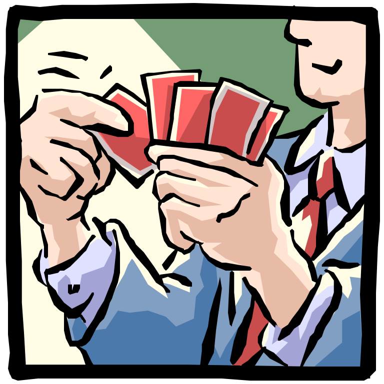
Example 2: Sorting

INPUT

sequence of *n* numbers

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

2 5 4 10 7



OUTPUT

a permutation of the input sequence of numbers

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

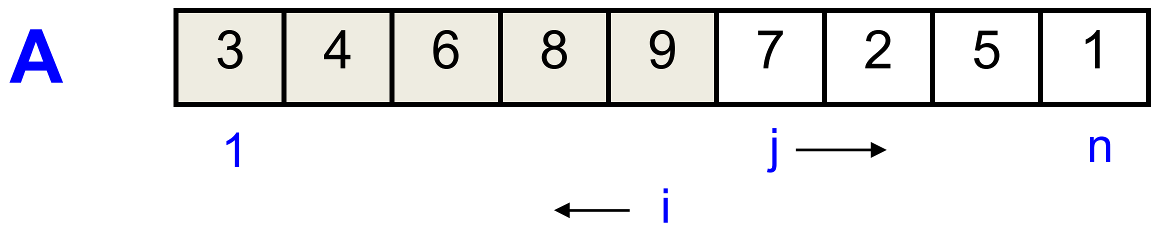
2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 \le b_2 \le b_3 \le \ldots \le b_n$
- $b_1, b_2, b_3, \ldots, b_n$ 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: Principles

- Idea: stepwise, increase sorted part.
  Initially, A[1..1] is sorted
- Control structure: increase stepwise from left to right
  => iteration
- Insertion into sorted part: check until position is found
  => while-loop
  Number to be inserted: key:= A[j]
  Move sorted part to right, until correct position found

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 *j*th iteration is

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

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

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 *j*th 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 on average: $C_{avg} = \sum_{j=2}^{n} j/2 = (n*n + n - 2)/4$

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

Ideas of Insertion Sort

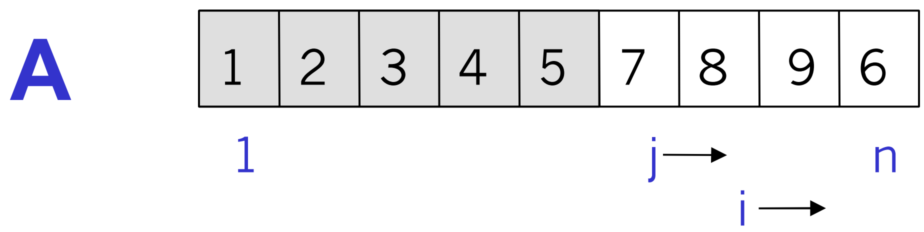
- Start with something that is a trivial partial solution
  - what is the initial (trivial) partial solution?
  - what could be another trivial partial solution?

- Stepwise extend each partial solution to a bigger partial solution
  … until it is full solution
  - in which way are the results of each (outer) iteration partial solutions?

Loop Invariants

- Which property (in terms of A and j) is true whenever the execution reaches the for-loop?
- Why is it true initially?
- Why does it continue to be true later on?
- What does this property mean when the for-loop is reached the last time?

Selection Sort



Strategy

- Start empty handed.
- Enlarge the sorted part by swapping the *least* element of the unsorted part with the *first* 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: Principles

- Idea: increase the sorted part by adding the minimum of the unsorted part.
- Initially, the empty segment A[1..0] is sorted and contains the 0 minimal elements
- Control structure: iteration over j,
  find min in A[j..n] and put it into position j
-

Selection Sort: Abstract Version

```
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] is sorted and contains the
   // j-1 minimal elements of the array
   minpos := findMinPos(A,j,n);
   swap(A,j,minpos)
```

Selection Sort: Principles

- Idea: increase the sorted part by adding the minimum of the unsorted part.
- Initially, the empty segment A[1..0] is sorted and contains the 0 minimal elements
- Control structure: iteration over j,
  find min in A[j..n] and put it into position j
- Inner loop: find the min in the rest A[j..n]
  Hypothesis: min is A[j], revise during inner loop.
  Control structure: iteration

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
  min := A[j]; minpos := j
  for i := j+1 to n do
    if A[i] < min then min := A[i]; minpos := i;
  A[minpos] := A[j]; A[j] := min
```

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
  min := A[j]; minpos := j
  for i := j+1 to n do
    if A[i] < min then min := A[i]; minpos := i;
  A[minpos] := A[j]; A[j] := min
```

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:

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

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

Bubble Sort

A									
	1	2	3	4	5	7	9	8	6
	1					j →	←		n

Strategy

- Start from the back and compare pairs of adjacent elements.
- Swap the elements if the larger comes before the smaller.
- In each iteration 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: Principles

- Idea: let small elements move down (= to left).
  Effect: initial array segment is sorted.
- Control structure: Initially, the empty array A[1..0] is sorted, then the sorted part grows by one element per round
  => Iteration
- Sinking down: lesser elements are swapped with greater ones
  => Iteration

Bubble Sort

```
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 downto j do
    if A[i-1] > A[i] then
      swap(A,i,i-1)
```

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 downto j do
    if A[i-1] > A[i] then
      val := A[i-1];
      A[i-1] := A[i];
      A[i]:= val
```

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	Yes	Yes	Yes
Selection Sort	No	Yes, if ...	No
Bubble Sort	No		No

Sorting Algorithms: Properties

Which algorithm has which property?

	Adaptive	Stable	Online
Insertion Sort	Yes	Yes	Yes
Selection Sort	No	Yes (if we select the first minimum)	No
Bubble Sort	No	Yes	No

Summary

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