Applied Programming and Computer Science, DD2325/appcs15

PODF, Programmering och datalogi för fysiker, DA7011

Autumn 2015

Lecture 2, Complexity and Sorting

A. Maki, C. Edlund

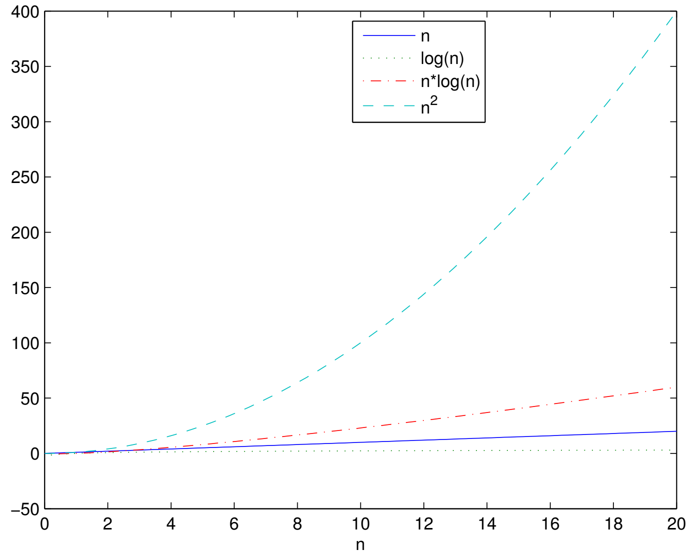
Complexity

We need to know how the elapsed time, T of an algorithm, varies with the number of elements n.

This leads us to complexity analysis, i.e. to find a mathematical expression of ordo-type which gives a measure of the time needed to complete an algorithm.

The sequential search has elapsed time $O(n)$ while the binary search has $O(log(n))$.

Complexity



Sorting Algorithms

An algorithm that puts elements of a list in a certain order.

The output is:

- in nondecreasing order
- a permutation (reordering) of the input

The properties:

- complexity
- stable/unstable
- memory usage, ...

The choice of sorting algorithm depends on n, and application.

1. Selection sort

view the list as two parts; sorted and unsorted

Algorithm

```
* find the smallest number in the unsorted part of the list
* swap the first element with the smallest in the unsorted part
* the sorted part of the list expands with one element,
  the unsorted reduces with one element
* if the unsorted part isn't empty, use the algorithm again
```

- inefficient on large lists

Selection sort

```
function v = selection(v)
for i = 1:length(v)-1
    minimum = v(i);
    index = i;
    for j = i+1:length(v)
        if minimum > v(j)
            minimum = v(j);
            index = j;
        end % if
    end % for
    tmp = v(i);
    v(i) = minimum;
    v(index) = tmp;
end % for
end % selection
```

2. Insertion sort

Algorithm

```
* take elements from the list one by one
* insert them in their correct position into a new sorted list
```

- efficient for small lists and mostly sorted lists
- often used as part of other algotihms

Divide and conquer

Algorithm

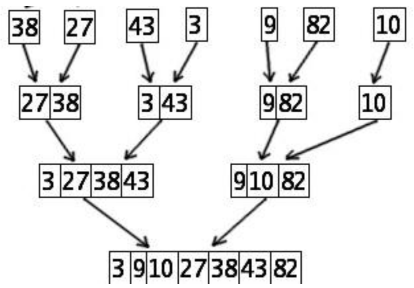
```
program Sort(list)

if length of list is 1
   * the list is sorted
if length of list is greater than 1
   * Partition the list into a lower and a higher list
   * Sort the lower list
   * Sort the higher list
   * Combine the sorted lower and sorted higher lists
```

Mergesort and quicksort use the above algorithm.

3. Mergesort (von Neumann, 1945)

Divide the list into two sublists of equal size and sort them separately.
Merge the two sorted lists into one.
Combining the sorted lists needs some work.



4. Quicksort (Hoare, 1960)

Choose a pivot element.

Elements less than the pivot element comes in the lower list.
Elements greater than the pivot element comes in the higher list.

NB. There are variations.

5. Radixsort

Sort the numbers:

```
7 2 13 4 23 18 1
```

Place the numbers in boxes (0 - 9) with respect to the last digit:

```
box  0:                    box  5:
     1: 1                       6:
     2: 2                       7: 7
     3: 13, 23                  8: 18
     4: 4                       9:
```

Put the sublists together. Start with box 0 and keep the internal order!

```
1 2 13 23 4 7 18
```

Radixsort

Place the numbers in boxes (0 - 9) with respect to the next but last digit:

```
box  0: 1, 2, 4, 7         box  5:
     1: 13, 18                  6:
     2: 23                      7:
     3:                         8:
     4:                         9:
```

Put the sublists together. Start with box 0 and keep the internal order!
The sorted list becomes:

```
1 2 4 7 13 18 23
```

6. Bubble sort (Demuth, 1956)

Algorithm

```
* scan the unsorted list from start to end
* if two adjacent numbers aren't in order, swap
* the largest number in the unsorted list will bubble
  up to the sorted list
* the sorted list expands with one element,
  the unsorted reduces with one element
* if the unsorted list isn't empty, use the algorithm again
```

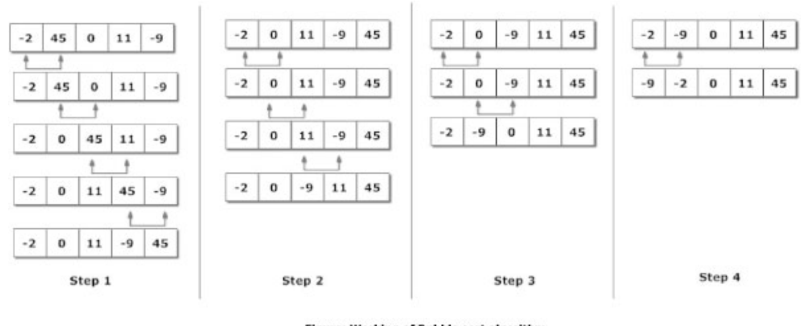


Figure: Working of Bubble sort algorithm

Popular sorting algorithms?

Efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithm for the overall sort with insertion sort for small lists at the bottom of a recursion.

- small data n ($n < 50$) insertion sort fast enough, but $O(n^2)$!
- large data n ($n > 1000$)
  - quicksort 1.5 to 2 times faster than the others
  - heapsort (in place)

(numerical Recipes in C, Press et al. 2nd Edition, 1992)

Interesting comparisons available at:
http://www.sorting-algorithms.com/

Required for good programming

- If code is repeated write a function (subprogram), i.e. *m*odularization into functions.
- Use/comment *preconditions* and *postconditions* (see Lecture 1 for a few examples).

Good programming

- Comments. The code should be written so the reader understands *how* it works. Comments should explain *why* it works and *what* it does.

Good programming (cont.)

- Use descriptive names on variables, functions/subprograms
- Write the code in a simple way, avoid too smart solutions that are hard to follow
- Use global variables only when necessary
- Use constants or variables instead of using the value repeatedly

Programming technique

Write programs that are user friendly! The program should

- be easy to use
    - instructions if necessary

      Press *h* for help
    - give example or default values

      Birthday (yyyymmdd)?
    - relevant, correct and polite communication
- have error messages that are
    - easy to understand

      Your birthday is not correct!
    - instructive

      Write on format: yyyymmdd

User friendly program?

```
Welcome to your horoscope teller! Fill in your personal data and the
horoscope teller will cast your personal horoscope.
Do you want to continue? Yes
What's your name? Alice
Do you want to continue? Yes
Birthday? 19000403
Fill in correct!
Do you want to continue? Yes
Birthplace? Stockholm
Do you want to continue? Yes
When were your parents born? 18720404
Do you want to continue? Yes

************** YOUR HOROSCOPE *******************
Your horoscope couldn't be cast due to lack of information.
```

Program code

```
% checks if there is no element then if it's one element
% and at last if it's more than one element

function [x, y] = pop(z)
if length(z)==0 %
   x = [];
elseif length(z)==1  % check if length equals one
   x = z(1);
   y = [];
else                 % fix the rest
   x = z(1);
   y = z(2:end);
end
```

Compare with the pop function and its description in lecture 1.

easy to read?

```
disp('Exchange sek to euro and vice versa')
kr = input('The amount is: ');
sek = input('The currency is: ','s');

if strcmp(sek, 'sek')
   disp(kr*0.1002)
else
   disp(kr*(1/0.1002))
end
```

`TF = strcmp(s1,s2)` compares two strings for equality.
The strings `s1,s2` are considered to be equal if the size and content of each are the same (the comparison is case sensitive).
The function returns a scalar logical 1/0 for equality/inequality.