

Section 2.2 Complexity of Algorithms

Time Complexity: Determine the approximate number of operations required to solve a problem of size n .

Space Complexity: Determine the approximate memory required to solve a problem of size n .

Time Complexity

- Use the Big-O notation
- Ignore house keeping
- Count the expensive operations only

Basic operations:

- searching algorithms - key comparisons
- sorting algorithms - list component comparisons
- numerical algorithms - floating point ops. (flops) - multiplications/divisions and/or additions/subtractions

Worst Case: maximum number of operations

Average Case: mean number of operations assuming an input probability distribution

Examples:

- Multiply an $n \times n$ matrix A by a scalar c to produce the matrix B :

```
procedure (n, c, A, B)
  for i from 1 to n do
    for j from 1 to n do
      B(i, j) = cA(i, j)
    end do
  end do
```

Analysis (worst case):

Count the number of floating point multiplications.

n^2 elements requires n^2 multiplications.

time complexity is

$O(n^2)$

or

quadratic complexity.

- Multiply an $n \times n$ *upper triangular* matrix A

$$A(i, j) = 0 \text{ if } i > j$$

by a scalar c to produce the (upper triangular) matrix B .

```
procedure (n, c, A, B)
/* A (and B) are upper triangular */
  for i from 1 to n do
    for j from i to n do
      B(i, j) = cA(i, j)
    end do
  end do
```

Analysis (worst case):

Count the number of floating point multiplications.

The maximum number of non-zero elements in an $n \times n$ upper triangular matrix

$$= 1 + 2 + 3 + 4 + \dots + n$$

or

- remove the diagonal elements (n) from the total (n^2)
- divide by 2
- add back the diagonal elements to get

$$(n^2 - n)/2 + n = n^2/2 + n/2$$

which is

$$n^2/2 + O(n).$$

Quadratic complexity but the leading coefficient is $1/2$

-
- Bubble sort: L is a list of elements to be sorted.
 - We assume nothing about the initial order
 - The list is in ascending order upon completion.

Analysis (worst case):

Count the number of list comparisons required.

Method: If the j th element of L is larger than the $(j + 1)$ st, swap them.

Note: this is not an efficient implementation of the algorithm

```
procedure bubble ( $n, L$ )
/*
  -  $L$  is a list of  $n$  elements
  - swap is an intermediate swap location
*/

for  $i$  from  $n - 1$  to  $1$  by  $-1$  do
  for  $j$  from  $1$  to  $i$  do
    if  $L(j) > L(j + 1)$  do
       $swap = L(j + 1)$ 
       $L(j + 1) = L(j)$ 
       $L(j) = swap$ 
    end do
  end do
end do
```

- Bubble the largest element to the 'top' by starting at the bottom - swap elements until the largest in the top position.

- Bubble the second largest to the position below the top.

- Continue until the list is sorted.

n-1 comparison on the first pass

n-2 comparisons on the second pass

.

.

.

1 comparison on the last pass

Total:

$$(n - 1) + (n - 2) + \dots + 1 = O(n^2)$$

or

quadratic complexity

(what is the leading coefficient?)

- An algorithm to determine if a function f from A to B is an injection:

Input: a table with two columns:

- Left column contains the elements of A .
- Right column contains the images of the elements in the left column.

Analysis (worst case):

Count comparisons of elements of B .

Recall that two elements of column 1 cannot have the same images in column 2.

One solution:

- Sort the right column

Worst case complexity (using Bubble sort)

$$O(n^2)$$

- Compare adjacent elements to see if they agree

Worst case complexity

$$O(n)$$

Total:

$$O(n^2) + O(n) = O(n^2)$$

Can it be done in linear time?