



Functions, Complexity of Algorithms

Section 2.1, 2.3 Complexity of Algorithms

Algorithm Complexity

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

Time Complexity

- **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

Time Complexity

■ Running Scenarios

- **Best Case:** minimum number of operations
- **Worst Case:** maximum number of operations
- **Average Case:** mean number of operations assuming an input probability distribution

Case Study: Numerical

- **A Numerical Algorithm**

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.

Case Study: Numerical

- **A Numerical Algorithm**

Multiply an $n \times n$ *upper triangular* matrix A $A(i, j) = 0$ 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

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

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

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

Case Study: Sorting

■ **Sorting Algorithms**

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.

■ **Method:**

- Bubble the largest element to the 'top' by starting at the bottom - swap elements until the largest is in the top position.
- Bubble the second largest to the position below the top.
- Continue until the list is sorted.

Note: this is not an efficient implementation of the algorithm.

Case Study: Sorting

■ Sorting Algorithms

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

■ Analysis:

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?)

Case Study: Search

- **Search algorithm:** locate an element x in a list of distinct elements, or determine that it is not in the list.
- **Linear search/sequential search algorithm:** The linear search algorithm begins by comparing x and a_1 . If $x = a_1$, the solution is the location of a_1 . If $x \neq a_1$, compare x with a_2 . Continue this process, until a match is found; unless no match is found.

Case Study: Search

Procedure linear_search (x : integer, a_1, a_2, \dots, a_n :
distinct integers)

$i=1$

while ($i \leq n$ and $x \neq a_i$)

$i=i+1$

if $i \leq n$, **then** location = i

else location = 0

(location 0 means no match found)

Case Study: Search

- *Worst case:*

Need to go through all of the n element in the list to find the match (or no match found).

Each step: two comparisons are performed

Finally, one more comparison is needed.

Worst case: $2n+1$ comparison

Worst-case analysis tells us the number of operations required to guarantee a solution.

Case Study: Search

- *Best case*: find the match on the 1st element.
- *Average case*:

X is the *i*-th element in the list

i Number of comparison

1 3

2 5

3 7

...

n $(2n+1)$

The average number of comparison used,

$$\frac{3 + 5 + 7 + \dots + (2n + 1)}{n} = n + 2$$