Programming Concepts

Comparing algorithms

Topics

Code reuse

Example problem:

Analyse two arrays to see which has the largest element

Code reuse

Example problem:

Analyse two arrays to see which has the largest element

Algorithmic solution:

(1) find maximum in first array

(2) find maximum in second array

(3) compare the two maxima

(4) output name of appropriate array

Pseudo-code

```
Input: Arrays $A[1 \ldots n]$, $B[1 \ldots m]$
Output: name of array with largest item
$\textsc{ptr} \leftarrow 1$
$\textsc{aMax} \leftarrow A[\textsc{ptr}]$
while $\textsc{ptr} \neq n$ do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
    if $A[\textsc{ptr}] > \textsc{aMax}$ then $\textsc{aMax} \leftarrow A[\textsc{ptr}]$
$\textsc{ptr} \leftarrow 1$
$\textsc{bMax} \leftarrow B[\textsc{ptr}]$
while $\textsc{ptr} \neq m$ do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
    if $B[\textsc{ptr}] > \textsc{bMax}$ then $\textsc{bMax} \leftarrow B[\textsc{ptr}]$
if $\textsc{aMax} > \textsc{bMax}$ then return $A$ else return $B$
```

Pseudo-code

Duplication of code in lines 1-5, and 6-10

```
   Input: Arrays $A[1 \ldots n]$, $B[1 \ldots m]$
   Output: name of array with largest item
1  PTR ← 1
2  aMax ← $A$[PTR]
3  while PTR $\neq n$ do
4      PTR ← PTR + 1
5      if $A$[PTR] > aMax then aMax ← $A$[PTR]
6  PTR ← 1
7  bMax ← $B$[PTR]
8  while PTR $\neq m$ do
9      PTR ← PTR + 1
10     if $B$[PTR] > bMax then bMax ← $B$[PTR]
11 if aMax > bMax then return $A$ else return $B$
```

Pseudo-code with code reuse

```
**Input**: Arrays $A[1 \ldots n]$, $B[1 \ldots m]$
**Output**: name of array with largest item
$\textsc{aMax} \leftarrow$ Max (A,1,n)
$\textsc{bMax} \leftarrow$ Max (B,1,m)
if $\textsc{aMax} > \textsc{bMax}$ then return $A$ else return $B$
```

Pseudo-code with code reuse

```
Input: Arrays $A[1 \ldots n]$, $B[1 \ldots m]$
Output: name of array with largest item
$\textsc{aMax} \leftarrow$ Max (A,1,n)
$\textsc{bMax} \leftarrow$ Max (B,1,m)
if $\textsc{aMax} > \textsc{bMax}$ then return $A$ else return $B$
```

`Max`(A, s, f) means a call to a procedure

- named `Max`
- with input parameters A, s, f
- which calculates largest item in A between s and f

An example procedure

Procedure: $\boxed{\texttt{Max}(A, s, f)}$

Input: array A, indices $s \leq f$

Output: largest item in sub-array $A[s \ldots f]$

```
$\textsc{ptr} \leftarrow s$
$\textsc{currentMax} \leftarrow A[\textsc{ptr}]$
while $\textsc{ptr} \neq f$ do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
    if $A[\textsc{ptr}] > \textsc{currentMax}$ then
        $\textsc{currentMax} \leftarrow A[\textsc{ptr}]$
return $\textsc{currentMax}$
```

Code re-use

In programming languages:

- procedures
- subroutines
- methods
- …

Code re-use

In programming languages:

- procedures
- subroutines
- methods
- . . .

Programming issues

- How is input passed to procedures ?
- How are results returned from procedures ?

Code re-use

In programming languages:

- procedures
- subroutines
- methods
- …

Programming issues

- How is input passed to procedures ?
- How are results returned from procedures ?

See Dowek

How do we proceduralise an algorithm?

An example

An algorithm

Name: $\boxed{\text{Sequential search}}$

Input: Array $A[1 \dots n]$, item M

Output: true, if M in A, false otherwise

```
$\textsc{found} \leftarrow \textit{false}$
$\textsc{ptr} \leftarrow 1$
while not $\textsc{found}$ and $\textsc{ptr} \leq n$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow \textit{true}$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Corresponding procedure

Procedure: `seqSearch(`*A, s,f, M*`)`

Input: array *A*, indices *s*,*f*, item *M*

Output: *true* if *M* in $A[s \dots f]$, *false* otherwise

```
$\textsc{found} \leftarrow$ false
$\textsc{ptr} \leftarrow s$
while not $\textsc{found}$ and $\textsc{ptr} \leq f$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow$ true
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Corresponding procedure

Procedure: `seqSearch(A, s,f, M)`

Input: array *A*, indices *s*,*f*, item *M*

Output: *true* if *M* in $A[s \dots f]$, *false* otherwise

```
$\textsc{found} \leftarrow false$
$\textsc{ptr} \leftarrow s$
while $not\ \textsc{found}\ and\ \textsc{ptr} \leq f$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow true$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Call `seqSearch`$(A, 1, n, M)$ to search entire array $A[1 \dots n]$ for *M*.

Putting procedures to work

Improving performance of searching

Searching sorted arrays

Procedure: `seqSearch(A, s,f, M)`

Input: sorted array A, indices s, f, item M

Output: *true* if M in $A[s \dots f]$, *false* otherwise

```
$\text{PTR} \leftarrow s$
while $\text{PTR} \leq f$ and $A[\text{PTR}] < M$ do
    $\text{PTR} \leftarrow \text{PTR} + 1$
if $\text{PTR} > f$ then
    return false
else
    return $(A[\text{PTR}] = M)$
```

Improving performance of searching

Searching sorted arrays

Procedure: `seqSearch(`A, s, f, M`)`

Input: sorted array A, indices s, f, item M

Output: *true* if M in $A[s \dots f]$, *false* otherwise

```
$\textsc{ptr} \leftarrow s$
while $\textsc{ptr} \leq f$ and $A[\textsc{ptr}] < M$ do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
if $\textsc{ptr} > f$ then
    return false
else
    return $(A[\textsc{ptr}] = M)$
```

Scan through array only as far as expected slot for M

Improving performance of searching

Searching sorted arrays

Procedure: `seqSearch(`A, s, f, M`)`

Input: sorted array A, indices s, f, item M

Output: *true* if M in $A[s \dots f]$, *false* otherwise

```
$\textsc{ptr} \leftarrow s$
while $\textsc{ptr} \leq f$ and $A[\textsc{ptr}] < M$ do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
if $\textsc{ptr} > f$ then
    return false
else
    return $(A[\textsc{ptr}] = M)$
```

Scan through array only as far as expected slot for M

May still have to search entire array

More efficient searching

Binary search of a sorted array $A[1 \dots n]$ for M:

- If array is empty, item is not *found*
- Calculate *middle* of list, m
- Compare M to $A[m]$
- If equal, item is *found*
- If less, search sub-array $A[1 \dots (m-1)]$
- If greater, search sub-array $A[(m+1) \dots n]$

More efficient searching

Binary search of a sorted array $A[1 \ldots n]$ for M:

- If array is empty, item is not *found*
- Calculate *middle* of list, m
- Compare M to $A[m]$
- If equal, item is *found*
- If less, search sub-array $A[1 \ldots (m-1)]$
- If greater, search sub-array $A[(m+1) \ldots n]$

Division:

Problem instance of size n

- decomposed in problem instance of size $n/2$

Binary search: looking for Ollie

Alice
Bob
Carol
David
Elaine
Fred
George
Harry
Irene
John
Kelly
Larry
Mary
Nancy
Ollie

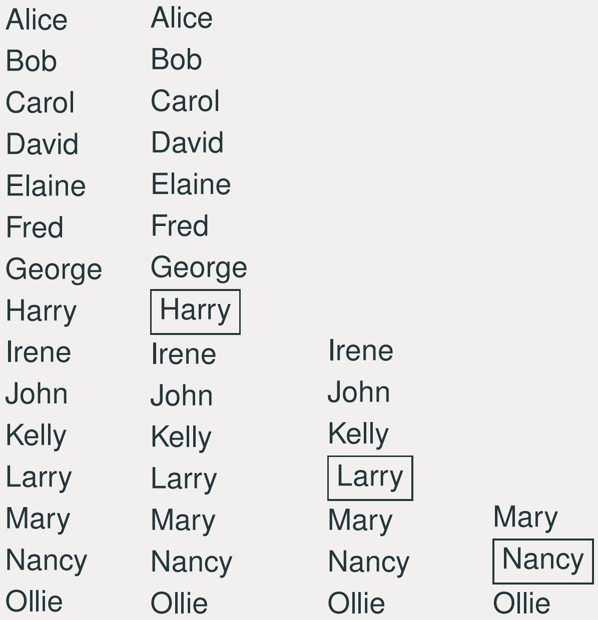
Binary search: looking for Ollie

Alice	Alice
Bob	Bob
Carol	Carol
David	David
Elaine	Elaine
Fred	Fred
George	George
Harry	$\boxed{\text{Harry}}$
Irene	Irene
John	John
Kelly	Kelly
Larry	Larry
Mary	Mary
Nancy	Nancy
Ollie	Ollie

Binary search: looking for Ollie

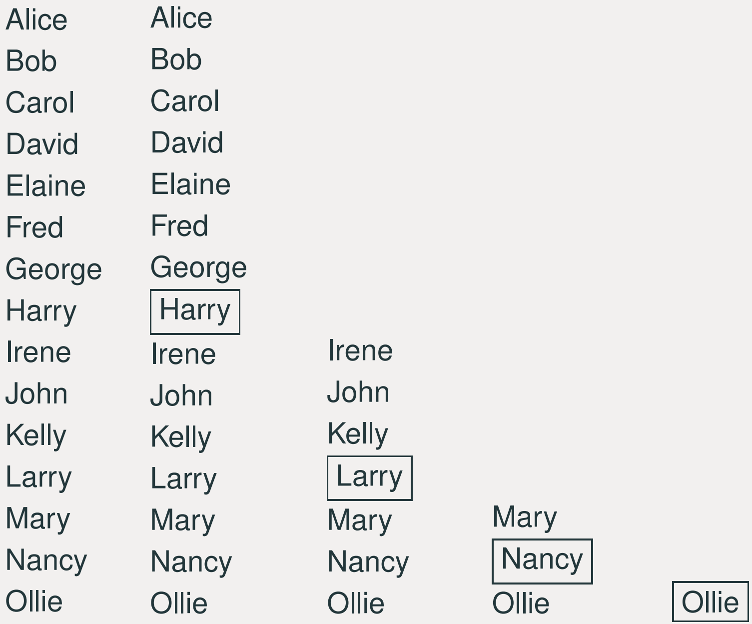
Alice	Alice	
Bob	Bob	
Carol	Carol	
David	David	
Elaine	Elaine	
Fred	Fred	
George	George	
Harry	$\boxed{\text{Harry}}$	
Irene	Irene	Irene
John	John	John
Kelly	Kelly	Kelly
Larry	Larry	$\boxed{\text{Larry}}$
Mary	Mary	Mary
Nancy	Nancy	Nancy
Ollie	Ollie	Ollie

Binary search: looking for Ollie



Alice	Alice		
Bob	Bob		
Carol	Carol		
David	David		
Elaine	Elaine		
Fred	Fred		
George	George		
Harry	$\boxed{\text{Harry}}$		
Irene	Irene	Irene	
John	John	John	
Kelly	Kelly	Kelly	
Larry	Larry	$\boxed{\text{Larry}}$	
Mary	Mary	Mary	Mary
Nancy	Nancy	Nancy	$\boxed{\text{Nancy}}$
Ollie	Ollie	Ollie	Ollie

Binary search: looking for Ollie



Alice	Alice			
Bob	Bob			
Carol	Carol			
David	David			
Elaine	Elaine			
Fred	Fred			
George	George			
Harry	Harry			
Irene	Irene	Irene		
John	John	John		
Kelly	Kelly	Kelly		
Larry	Larry	Larry		
Mary	Mary	Mary	Mary	
Nancy	Nancy	Nancy	Nancy	
Ollie	Ollie	Ollie	Ollie	Ollie

More efficient searching

Binary search of a sorted array $A[1 \dots n]$ for M:

- If array is empty, item is not *found*
- Calculate *middle* of list, m
- Compare M to $A[m]$
- If equal, item is *found*
- If less, search sub-array $A[1 \dots (m-1)]$
- If greater, search sub-array $A[(m+1) \dots n]$

More efficient searching

Binary search of a sorted array $A[1 \ldots n]$ for M:

- If array is empty, item is not *found*
- Calculate *middle* of list, m
- Compare M to $A[m]$
- If equal, item is *found*
- If less, search sub-array $A[1 \ldots (m-1)]$
- If greater, search sub-array $A[(m+1) \ldots n]$

Can we write this in pseudocode?

Procedure: `binarySearch(`A, s, f, M`)`

Input: sorted array A, indices s, f, item M
Output: *true* if M in $A[s \ldots f]$, *false* otherwise

Binary search

Procedure: `binarySearch(`A, s, f, M`)`

Input: sorted array A, indices s, f, item M

Output: *true* if M in $A[s \ldots f]$, *false* otherwise

```
FOUND ← $false$
if $s \leq f$ then
    $m \leftarrow (s + f)$ div 2 // middle of array
    if $A[m] = M$ then
        FOUND ← $true$
    else
        if $M < A[m]$ then
            FOUND ← binarySearch($A,s,(m-1),M$)
        else
            FOUND ← binarySearch($A,(m+1),f,M$)
return FOUND
```

How much faster is binary search?

Remember:

Sequential search: worst case *n* steps

Binary search: halves every time

Log to the base 2

Chopping arrays in half:

How many times can you chop an array of size *n* in half?

Log to the base 2

Chopping arrays in half:

How many times can you chop an array of size *n* in half?

Size	log
15	4
100	7
1000	10
1 million	20
1 billion	30
a billion billion	60

Log to the base 2

Chopping arrays in half:

How many times can you chop an array of size n in half?

Size	log
15	4
100	7
1000	10
1 million	20
1 billion	30
a billion billion	60

sequential search	binary search

Log to the base 2

Chopping arrays in half:

How many times can you chop an array of size *n* in half?

Size	log
15	4
100	7
1000	10
1 million	20
1 billion	30
a billion billion	60

sequential search	binary search

A worst case analysis

Divide and Conquer

Algorithm design technique:

- Decompose problem instance into smaller subinstances
- Solve subinstances successively and independently
- Combine subsolutions to obtain solution to original problem instance

Divide and Conquer

Algorithm design technique:

- Decompose problem instance into smaller subinstances
- Solve subinstances successively and independently
- Combine subsolutions to obtain solution to original problem instance

Most successful:

when problem instance of size n is decomposed into subinstances of size $n/2$

Divide and conquer in action

Divide and conquer for sorting ⇝ binary search

Divide and conquer for searching ⇝ merge sort

Divide and Conquer: Sorting

Mergesort:

Basic algorithm:

- Input array *A* to be sorted
- If size of *A* less than 2 it is sorted
- Otherwise
  - Divide *A* into two subarrays *L* and *R*
  - Sort *L*
  - Sort *R*
  - Merge sorted *L* and sorted *R* into a new, sorted array

Divide and Conquer: Sorting

Mergesort:

Basic algorithm:

- Input array *A* to be sorted
- If size of *A* less than 2 it is sorted
- Otherwise
  - Divide *A* into two subarrays *L* and *R*
  - Sort *L*
  - Sort *R*
  - Merge sorted *L* and sorted *R* into a new, sorted array

Based on fact that merging sorted arrays is a simple operation

Merging sorted arrays

3	7	10	14	17	…
↑					

5	6	12	15	…
↑				

Merged array:

-	-	-	-	-	-
↑					

Merging sorted arrays

3	7	10	14	17	…
	↑				

5	6	12	15	…
↑				

Merged array:

3	-	-	-	-	-
	↑				

Merging sorted arrays

3	7	10	14	17	…
	↑				

5	6	12	15	…
	↑			

Merged array:

3	5	-	-	-	-
		↑			

Merging sorted arrays

3	7	10	14	17	…
	↑				

5	6	12	15	…
		↑		

Merged array:

3	5	6	-	-	-
			↑		

Merging sorted arrays

3	7	10	14	17	…
		↑			

5	6	12	15	…
		↑		

Merged array:

3	5	6	7	-	-
				↑	

Merging sorted arrays

3	7	10	14	17	…
			↑		

5	6	12	15	…
		↑		

Merged array:

3	5	6	7	10	-
					↑

Merging sorted arrays

3	7	10	14	17	…
			↑		

5	6	12	15	…
		↑		

Merged array:

3	5	6	7	10	-
					↑

Needs only one scan through arrays

Merge sort

Mergesort: the idea

1. Input array *A* to be sorted
2. If size of *A* less than 2 it is sorted
3. Otherwise
    - 3.1 Divide *A* into two subarrays *L* and *R*
    - 3.2 Sort *L*
    - 3.3 Sort *R*
    - 3.4 Merge sorted *L* and sorted *R* into a new, sorted array

Can we write this as pseudocode?

Merge sort

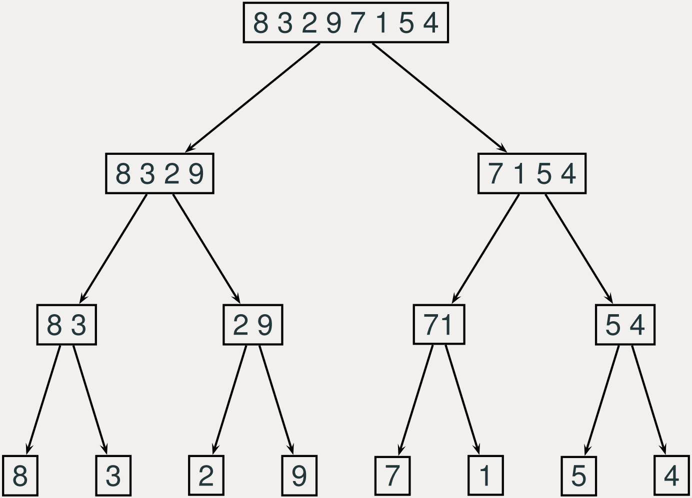
Procedure: `mergeSort(A)`

Input: array $A[1 \dots n]$

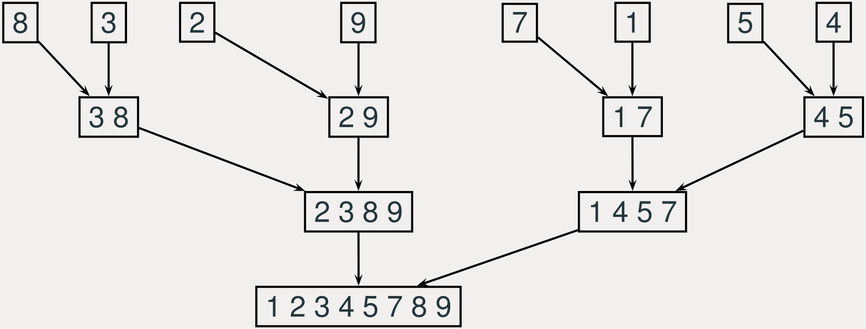
Result: array $A[1 \dots n]$ sorted

```
if $n \leq 1$ then
    return $A$
else
    $\textsc{m} \leftarrow n \operatorname{div} 2$
    copy $A[1 \dots \textsc{m}]$ to $B[1 \dots \textsc{m}]$
    copy $A[(\textsc{m}+1) \dots n]$ to $C[1 \dots (n-\textsc{m})]$
    mergeSort($B$)
    mergeSort($C$)
    merge($B$,$C$,$A$)
    return $A$
```

Merge sort in operation



Merging the sub-arrays



8 3 2 9 7 1 5 4

3 8 | 2 9 | 1 7 | 4 5

2 3 8 9 | 1 4 5 7

1 2 3 4 5 7 8 9

Efficiency of Merge sort

Divide and Conquer:

Problem of size n decomposed into

- two sub-problems of size $n/2$
- plus a merge of size about n

Efficiency of Merge sort

Divide and Conquer:

Problem of size n decomposed into

- two sub-problems of size $n/2$
- plus a merge of size about n

Merge sort:

Much more efficient than

- Bubble sort
- Insertion sort
- Selection sort

Efficiency: what we couold ask

Given a problem . . .

- Does there exist an algorithm for solving it?
- Does there exist an *efficient* algorithm?
- Can I *improve* on a published algorithm for the problem?

Given an algorithm . . .

- How efficient is it?
- Does a more efficient algorithm exist for the same problem?
- Which algorithm for a given problem is the *best*?

Two algorithms for summing numbers

Name: Summing numbers

Input: positive integer *n*

Output: sum of first *n* positive numbers

```
$\textsc{sum} \leftarrow 0$
$\textsc{iter} \leftarrow 1$
while $\textsc{iter} \leq n$ do
    $\textsc{sum} \leftarrow \textsc{Sum} + \textsc{iter}$
    $\textsc{iter} \leftarrow \textsc{iter} + 1$
return $\textsc{sum}$
```

Two algorithms for summing numbers

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
$\textsc{sum} \leftarrow 0$
$\textsc{iter} \leftarrow 1$
while $\textsc{iter} \leq n$ do
    $\textsc{sum} \leftarrow \textsc{Sum} + \textsc{iter}$
    $\textsc{iter} \leftarrow \textsc{iter} + 1$
return $\textsc{sum}$
```

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
return $n(n+1)/2$
```

Two algorithms for summing numbers

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
$\textsc{sum} \leftarrow 0$
$\textsc{iter} \leftarrow 1$
while $\textsc{iter} \leq n$ do
    $\textsc{sum} \leftarrow \textsc{Sum} + \textsc{iter}$
    $\textsc{iter} \leftarrow \textsc{iter} + 1$
return $\textsc{sum}$
```

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
return $n(n+1)/2$
```

Which is more efficient?

Measuring efficiency of algorithms

- Cost of running an algorithm depends on size of input (usually)

Measuring efficiency of algorithms

- Cost of running an algorithm depends on size of input (usually)
- Efficiency of an algorithm is expressed as a cost function on the size of input
- Comparison between algorithms difficult to see when size of inputs are small

Measuring efficiency of algorithms

- Cost of running an algorithm depends on size of input (usually)
- Efficiency of an algorithm is expressed as a cost function on the size of input
- Comparison between algorithms difficult to see when size of inputs are small
- Differences in efficiency become apparent as size of input get very large

Measuring efficiency of algorithms

We need

1. a measure on the size of inputs
2. a measure of cost of running an algorithm

Measuring efficiency of algorithms

We need

1. a measure on the size of inputs
2. a measure of cost of running an algorithm

Size of inputs:

Dependent on type of data

- arrays: number of items
- lists: number of items
- numbers: ? often size of binary representation

Measuring efficiency of algorithms

We need

1. a measure on the size of inputs
2. a measure of cost of running an algorithm

Size of inputs:

Dependent on type of data

- arrays: number of items
- lists: number of items
- numbers: ? often size of binary representation

Cost of running algorithm:

- time taken
- space required –less important these days
- energy consumed –becoming important

Measuring efficiency of algorithms

We need

1. a measure on the size of inputs
2. a measure of cost of running an algorithm

Size of inputs:

Dependent on type of data

- arrays: number of items
- lists: number of items
- numbers: ? often size of binary representation

Cost of running algorithm:

- time taken – we concentrate on this
- space required
- energy consumed

Measuring running time

- Difficult to use a stop watch

Measuring running time

- Difficult to use a stop watch
- Should not be influenced by speed of computer

Measuring running time

- Difficult to use a stop watch
- Should not be influenced by speed of computer
- Should not be influenced by choice of programming language

Measuring running time

- Difficult to use a stop watch
- Should not be influenced by speed of computer
- Should not be influenced by choice of programming language
- Should not be influenced by ability of programmer

Measuring running time

- Difficult to use a stop watch
- Should not be influenced by speed of computer
- Should not be influenced by choice of programming language
- Should not be influenced by ability of programmer
- We count the number of significant actions

Measuring running time

- Difficult to use a stop watch
- Should not be influenced by speed of computer
- Should not be influenced by choice of programming language
- Should not be influenced by ability of programmer
- We count the number of significant actions

Significant actions:

Depends on problem area:

- sorting: count comparisons between items
- searching: count comparisons between items
- summing: count arithmetic operations

Sometimes difficult to decide

Here we only care about the worst case complexity

Counting the cost

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
$\textsc{sum} \leftarrow 0$
$\textsc{iter} \leftarrow 1$
while $\textsc{iter} \leq n$ do
    $\textsc{sum} \leftarrow \textsc{Sum} + \textsc{iter}$
    $\textsc{iter} \leftarrow \textsc{iter} + 1$
return $\textsc{sum}$
```

```
Name: Summing numbers
Input: positive integer $n$
return $n(n+1)/2$
```

Counting the cost

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive
        numbers
SUM ← 0
ITER ← 1
while ITER $\leq n$ do
    SUM ← SUM + ITER
    ITER ← ITER + 1
return SUM
```

```
Name: Summing numbers
Input: positive integer $n$
return $n(n+1)/2$
```

Number of operations:

??	3

Counting the cost

```
Name: Summing numbers
Input: positive integer $n$
Output: sum of first $n$ positive numbers
SUM $\leftarrow 0$
ITER $\leftarrow 1$
while ITER $\leq n$ do
    SUM $\leftarrow$ SUM $+$ ITER
    ITER $\leftarrow$ ITER $+ 1$
return SUM
```

```
Name: Summing numbers
Input: positive integer $n$
return $n(n+1)/2$
```

Number of operations:

more than n	3

Estimating growth functions

Constants don't count:

Function	F	G	G larger than F after
	$100n^2$	$2n^3$	$n > 50$
	$1000n^2$	$3n^3$	$n > 350$
	$1000n^3$	n^4	$n > 1000$

Estimating growth functions

Constants don't count:

Function	F	G	G larger than F after
	$100n^2$	$2n^3$	$n > 50$
	$1000n^2$	$3n^3$	$n > 350$
	$1000n^3$	n^4	$n > 1000$

The tyranny of large numbers

Estimating growth functions

Small terms get swamped:

Function	Insignificant after
$2n^2 + \underline{10n + 6}$	$n > 10$
$n^4 + \underline{100n^2 + 5n}$	$n > 10$
$2n^5 + \underline{1000n^4}$	$n > 500$

Estimating growth functions

Small terms get swamped:

Function	Insignificant after
$2n^2 + \underline{10n + 6}$	$n > 10$
$n^4 + \underline{100n^2 + 5n}$	$n > 10$
$2n^5 + \underline{1000n^4}$	$n > 500$

The tyranny of large numbers

Approximating growth functions

Big Theta notation:

$\Theta(f(n))$: all functions which *grow* at the same rate as $f(n)$

Approximating growth functions

Big Theta notation:

$\Theta(f(n))$: all functions which *grow* at the same rate as $f(n)$

$g(n)$ is in $\Theta(f(n))$ if

- there is a constant K_g such that $g(n) \leq K_g * f(n)$, once n gets sufficiently large
- there is a constant K_f such that $f(n) \leq K_f * g(n)$, once n gets sufficiently large

Approximating growth functions

Big Theta notation:

$\Theta(f(n))$: all functions which *grow* at the same rate as $f(n)$

$g(n)$ is in $\Theta(f(n))$ if

- there is a constant K_g such that $g(n) \leq K_g * f(n)$, once n gets sufficiently large
- there is a constant K_f such that $f(n) \leq K_f * g(n)$, once n gets sufficiently large

which means: once n gets big enough

- $k_1 * f(n) \leq g(n) \leq k_2 * f(n)$
- (and vice-versa)

Important complexity classes

constant	$\Theta(1)$
logarithmic	$\Theta(\log n)$
linear	$\Theta(n)$
n-log-n	$\Theta(n \log n)$
quadratic	$\Theta(n^2)$
cubic	$\Theta(n^3)$
polynomial	$\Theta(n^k)$, for some $k \geq 1$
exponential	$\Theta(2^n)$

Examples

- $2 * n + 6$ is in $\Theta(n)$
- $4 * n^2 + 10 * n + 6$ is NOT in $\Theta(n)$

Examples

- $2 * n + 6$ is in $\Theta(n)$
- $4 * n^2 + 10 * n + 6$ is NOT in $\Theta(n)$

- $234 * n^2 + 658 * n + 200$ is in $\Theta(n^2)$
- $234 * n^2 + 658 * n + 200$ is NOT in $\Theta(n^3)$

Examples

- $2 * n + 6$ is in $\Theta(n)$
- $4 * n^2 + 10 * n + 6$ is NOT in $\Theta(n)$

- $234 * n^2 + 658 * n + 200$ is in $\Theta(n^2)$
- $234 * n^2 + 658 * n + 200$ is NOT in $\Theta(n^3)$

- $78 * 10^n + 34 * n^{27}$ is in $\Theta(2^n)$

Examples

- $2 * n + 6$ is in $\Theta(n)$
- $4 * n^2 + 10 * n + 6$ is NOT in $\Theta(n)$

- $234 * n^2 + 658 * n + 200$ is in $\Theta(n^2)$
- $234 * n^2 + 658 * n + 200$ is NOT in $\Theta(n^3)$

- $78 * 10^n + 34 * n^{27}$ is in $\Theta(2^n)$

Dominant exponent always wins out in the end

Orders of Growth

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
10	3.3	10	33	100	$1,000$	1,000
100	6.6	100	660	10^4	10^6	$1.3 \cdot 10^{30}$
1,000	10	1,000	10,000	10^6	10^9	
10,000	13	10,000	130,000	10^8	10^{12}	
100,000	17	100,000	1.7 million	10^{10}	10^{15}	
1 million	20	1,000,000	20 million	10^{12}	10^{18}	

Orders of Growth

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
10	3.3	10	33	100	$1,000$	1,000
100	6.6	100	660	10^4	10^6	$1.3 \cdot 10^{30}$
1,000	10	1,000	10,000	10^6	10^9	
10,000	13	10,000	130,000	10^8	10^{12}	
100,000	17	100,000	1.7 million	10^{10}	10^{15}	
1 million	20	1,000,000	20 million	10^{12}	10^{18}	

Estimated age of the universe: 10^{14} seconds

Why buying a new computer will not help

Running an exponential algorithm

Current computer: 10,000 instructions per second

problem size	time
10	0.1 sec
20	2 mins approx.
30	$>$ 24 hours
38	$>$ one year

Why buying a new computer will not help

Running an exponential algorithm

New computer: 100 times faster

problem size	time
10	0.1 sec
20	1 minute approx
37	$>$ 24 hours
45	$>$ one year

Why buying a new computer will not help

Running an exponential algorithm

New computer: 100 times faster

problem size	time
10	0.1 sec
20	1 minute approx
37	$>$ 24 hours
45	$>$ one year

Only minimal increase in effectiveness:

time T	problem size
old computer	n
new computer	n+7

Why buying a new computer will not help

Running an exponential algorithm

New computer: 100 times faster

problem size	time
10	0.1 sec
20	1 minute approx
37	$>$ 24 hours
45	$>$ one year

Only minimal increase in effectiveness:

time T	problem size
old computer	n
new computer	n+7

Algorithms v. fast computers

More efficient algorithms better than faster computers

Setup A:

- slow sorting algorithm: $\Theta(n^2)$
- crafty programmer: constant factor 2
- fast machine: 1 billion instructions per second

Setup B:

- fast sorting algorithm: $\Theta(n \log(n))$
- rubbish programmer: constant factor 50
- slow machine: 10 million instructions per second (100 times slower)

Algorithms v. fast computers

More efficient algorithms better than faster computers

Setup A:

- slow sorting algorithm: $\Theta(n^2)$
- crafty programmer: constant factor 2
- fast machine: 1 billion instructions per second

Setup B:

- fast sorting algorithm: $\Theta(n \log(n))$
- rubbish programmer: constant factor 50
- slow machine: 10 million instructions per second (100 times slower)

setup	1 million numbers	10 million numbers
A	$>$ 30 mins	$>$ 2 days
B	$<$ 2 mins	$<$ 20 mins

Comparing complexity classes

Double size of input	Extra time required
$\Theta(1)$	none
$\Theta(\log n)$	marginal increase
$\Theta(n)$	double
$\Theta(n \log n)$	double + tiny
$\Theta(n^2)$	four times longer
$\Theta(n^3)$	eight times longer
$\Theta(2^n)$	square of time

Examples: how to compute complexity

Remember we only care about the worse case

Example

```
Name: Sequential search
Input: Array $A[1 \ldots n]$, item $M$
Output: true, if $M$ in $A$, false otherwise
$\textsc{found} \leftarrow false$
$\textsc{ptr} \leftarrow 1$
while not $\textsc{found}$ and $\textsc{ptr} \leq n$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow true$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Example

Name: Sequential search
Input: Array $A[1 \ldots n]$, item M
Output: true, if M in A, false otherwise

```
FOUND ← false
PTR ← 1
while not FOUND and PTR ≤ n do
    if A[PTR] = M then
        FOUND ← true
    PTR ← PTR + 1
return FOUND
```

Running time:

- dependent on size of input

Example

```
Name: Sequential search
Input: Array $A[1 \ldots n]$, item $M$
Output: true, if $M$ in $A$, false otherwise
FOUND ← false
PTR ← 1
while not FOUND and PTR ≤ $n$ do
    if $A[\text{PTR}] = M$ then
        FOUND ← true
    PTR ← PTR + 1
return FOUND
```

Running time:

- dependent on size of input
- dependent on actual input

Example

Name: Sequential search
Input: Array $A[1 \dots n]$, item M
Output: true, if M in A, false otherwise

```
$\textsc{found} \leftarrow$ false
$\textsc{ptr} \leftarrow 1$
while not $\textsc{found}$ and $\textsc{ptr} \leq n$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow$ true
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Running time:

- dependent on size of input
- dependent on actual input

No. of comparisons:

- best possible: 1
- worst possible: n
- average: difficult to understand

Example

Name: Sequential search

Input: Array $A[1 \ldots n]$, item M

Output: true, if M in A, false otherwise

```
$\textsc{found} \leftarrow false$
$\textsc{ptr} \leftarrow 1$
while $not\ \textsc{found}\ and\ \textsc{ptr} \leq n$ do
    if $A[\textsc{ptr}] = M$ then
        $\textsc{found} \leftarrow true$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{found}$
```

Running time:

- dependent on size of input
- dependent on actual input

No. of comparisons:

- best possible: 1
- worst possible: n
- average: difficult to understand

We only deal with *worst case analysis*

Improved searching?

First sort the list:

- Not necessary to scan all the array
- stop when an item *greater than or equal to* M is encountered

Improved searching?

First sort the list:

- Not necessary to scan all the array
- stop when an item *greater than or equal to* M is encountered

Procedure:

```
seqSearch($A, s, f, M$)
$\textsc{ptr} \leftarrow s$
while $\textsc{ptr} \leq f$ and $A[\textsc{ptr}] < M$
do
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
if $\textsc{ptr} > f$ then return false
else
    return $(A[\textsc{ptr}] = M)$
```

Improved searching?

First sort the list:

- Not necessary to scan all the array
- stop when an item *greater than or equal to* M is encountered

Procedure:

```
seqSearch($A, s, f, M$)
$\text{PTR} \leftarrow s$
while $\text{PTR} \leq f$ and $A[\text{PTR}] < M$ do
    $\text{PTR} \leftarrow \text{PTR} + 1$
if $\text{PTR} > f$ then return false
else
    return $(A[\text{PTR}] = M)$
```

No. of comparisons:

- best possible: 1
- worst possible: n
- average: probably an improvement

Analysis of binary search

```
Procedure: binarySearch(A, s,f, M)
FOUND ← false
if $s \leq f$ then $m \leftarrow (s + f)$ div 2 // middle of array

if $A[m] = M$ then FOUND ← true else
    if $M < A[m]$ then
        FOUND ← binarySearch(A,s,(m-1),M)
    else
        FOUND ← binarySearch(A,(m+1),f,M)
return FOUND
```

Analysis of binary search

```
Procedure: binarySearch(A, s,f, M)
FOUND ← false
if $s \leq f$ then $m \leftarrow (s+f) \operatorname{div} 2$ // middle of array

if $A[m] = M$ then FOUND ← true else
    if $M < A[m]$ then
        FOUND ← binarySearch(A,s,(m-1),M)
    else
        FOUND ← binarySearch(A,(m+1),f,M)
return FOUND
```

Worst case:

- procedure called $\log n$ times
- each call gives 1 comparison

Analysis of binary search

```
Procedure: binarySearch(A, s,f, M)
FOUND ← false
if $s \leq f$ then $m \leftarrow (s + f)$ div 2 // middle of array

if $A[m] = M$ then FOUND ← true else
    if $M < A[m]$ then
        FOUND ← binarySearch(A,s,(m-1),M)
    else
        FOUND ← binarySearch(A,(m+1),f,M)
return FOUND
```

Worst case:

- procedure called $\log n$ times
- each call gives 1 comparison

worst case analysis: $\log n$

Sequential v. binary search

Array size	sequential search	binary search
15	15	4
100	100	7
1000	1000	10
1 million	1 million	20
1 billion	1 billion	30
1 billion billion	1 billion billion	60

Calculating *spread* of an array

```
Input: Array $A[1 \dots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

Calculating *spread* of an array

```
Input: Array $A[1 \ldots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

```
Input: Array $A[1 \ldots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        C

return $\textsc{cSpread}$
```

Calculating *spread* of an array

```
Input: Array $A[1 \dots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

```
Input: Array $A[1 \dots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        C

return $\textsc{cSpread}$
```

Worst case:

- inner loop executed *n* times
- each time *C* is executed: some *constant* no. of actions
- outer loop executed *n* times

Calculating *spread* of an array

```
Input: Array $A[1 \ldots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

```
Input: Array $A[1 \ldots n]$
Output: The spread of $A$
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        C

return $\textsc{cSpread}$
```

Worst case:

- inner loop executed *n* times
- each time *C* is executed: some *constant* no. of actions
- outer loop executed *n* times

Improved *spread* algorithm?

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i + 1)$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

Improved *spread* algorithm?

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i + 1)$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i + 1)$ to $n$ do
        $C$

return $\textsc{cSpread}$
```

Improved *spread* algorithm?

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$

return $\textsc{cSpread}$
```

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        $C$

return $\textsc{cSpread}$
```

Worst case:

- inner loop executed
  - first $(n-1)$ times, then $(n-2)$ times , then $(n-3)$ times, ...
- each time C is executed: some *constant* no. of actions
- outer loop executed n times

Improved *spread* algorithm?

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        $\textsc{d} \leftarrow diff(A[i], A[j])$
        if $\textsc{cSpread} < \textsc{d}$ then
            $\textsc{cSpread} \leftarrow \textsc{d}$
return $\textsc{cSpread}$
```

```
$\textsc{cSpread} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        $C$
return $\textsc{cSpread}$
```

Worst case:

- inner loop executed
  - first $(n-1)$ times, then $(n-2)$ times , then $(n-3)$ times, ...
- each time C is executed: some *constant* no. of actions
- outer loop executed n times

No. times C executed: $(n-1) + (n-2) + \ldots + 1 + 0$

Improved *spread* algorithm?

```
CSPREAD ← 0
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        D ← $diff(A[i], A[j])$
        if CSPREAD < D then
            CSPREAD ← D
return CSPREAD
```

```
CSPREAD ← 0
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow (i+1)$ to $n$ do
        $C$
return CSPREAD
```

Worst case:

- inner loop executed
    - first $(n-1)$ times, then $(n-2)$ times , then $(n-3)$ times, . . .
- each time C is executed: some *constant* no. of actions
- outer loop executed n times

Worst-case analysis: $\Theta(n^2)$

An improved spread algorithm

```
$\textsc{cMax} \leftarrow A[1]$
$\textsc{cMin} \leftarrow A[1]$
for $i \leftarrow 2$ to $n$ do
    if $A[i] < \textsc{cMin}$ then
        $\textsc{cMin} \leftarrow A[i]$
    else
        if $A[i] > \textsc{cMax}$ then
            $\textsc{cMax} \leftarrow A[i]$

$\textsc{cSpread} \leftarrow \textit{diff}(\textsc{cMax}, \textsc{cMin})$
return $\textsc{cSpread}$
```

An improved spread algorithm

```
$\textsc{cMax} \leftarrow A[1]$                      $C_1$
$\textsc{cMin} \leftarrow A[1]$                      for $i \leftarrow 2$ to $n$ do
for $i \leftarrow 2$ to $n$ do                          $C_2$
    if $A[i] < \textsc{cMin}$ then
        $\textsc{cMin} \leftarrow A[i]$                  $C_3$
    else
        if $A[i] > \textsc{cMax}$ then
            $\textsc{cMax} \leftarrow A[i]$

$\textsc{cSpread} \leftarrow \mathit{diff}(\textsc{cMax}, \textsc{cMin})$
return $\textsc{cSpread}$
```

An improved spread algorithm

```
cMax ← $A[1]$                                      $C_1$
cMin ← $A[1]$
for $i \leftarrow 2$ to $n$ do                     for $i \leftarrow 2$ to $n$ do
    if $A[i] <$ cMin then                              $C_2$
        cMin ← $A[i]$                              $C_3$
    else
        if $A[i] >$ cMax then
            cMax ← $A[i]$

cSpread ← diff(cMax, cMin)
return cSpread
```

Worst case:

- loop executed $(n - 1)$ times
- each C_i: constant no. of actions

An improved spread algorithm

```
$\textsc{cMax} \leftarrow A[1]$                                  $C_1$
$\textsc{cMin} \leftarrow A[1]$                                  for $i \leftarrow 2$ to $n$ do
for $i \leftarrow 2$ to $n$ do                                       $C_2$
    if $A[i] < \textsc{cMin}$ then
        $\textsc{cMin} \leftarrow A[i]$                          $C_3$
    else
        if $A[i] > \textsc{cMax}$ then
            $\textsc{cMax} \leftarrow A[i]$

$\textsc{cSpread} \leftarrow \mathit{diff}(\textsc{cMax}, \textsc{cMin})$
return $\textsc{cSpread}$
```

worst case analysis: $\Theta(n)$

Analysis of selection sort

```
Name: Selection sort
Input: Array $A[1 \dots n]$
Output: Array $A$ sorted
for $i \leftarrow 1$ to $(n - 1)$ do
    // maintains $A[1 \dots i]$ sorted
    $\textsc{minptr} \leftarrow i$
    for $j \leftarrow (i + 1)$ to $n$ do
        // finds smallest item in $A[i \dots n]$
        if $A[j] < A[\textsc{minptr}]$ then
            $\textsc{minptr} \leftarrow j$

    $A[i] \leftrightarrow A[\textsc{minptr}]$ // swap items

return $A$
```

Analysis of selection sort

```
Name: Selection sort
Input: Array $A[1 \ldots n]$
Output: Array $A$ sorted
for $i \leftarrow 1$ to $(n - 1)$ do
    $C_1$
    for $j \leftarrow (i + 1)$ to $n$ do
        $C_2$
    $C_3$
```

Analysis of selection sort

```
Name: Selection sort
Input: Array $A[1 \dots n]$
Output: Array $A$ sorted
for $i \leftarrow 1$ to $(n-1)$ do
    $C_1$
    for $j \leftarrow (i+1)$ to $n$ do
        $C_2$
    $C_3$
```

- inner loop
  - first executed $(n-1)$ times, then $(n-2)$ times, ...
  - each time C_2 is executed
  - each C_3: *constant* no. of comparisons

Analysis of selection sort

```
Name: Selection sort
Input: Array $A[1 \dots n]$
Output: Array $A$ sorted
for $i \leftarrow 1$ to $(n-1)$ do
    $C_1$
    for $j \leftarrow (i+1)$ to $n$ do
        $C_2$
    $C_3$
```

- inner loop
  - first executed $(n-1)$ times, then $(n-2)$ times, ...
  - each time C_2 is executed
  - each C_3: *constant* no. of comparisons

worst case analysis: $\Theta(n^2)$

Analysis of Merge sort

Procedure: `mergeSort(A)`

Input: array $A[1 \ldots n]$

Result: array $A[1 \ldots n]$ sorted

```
if $n > 1$ then
    $\textsc{m} \leftarrow n \operatorname{div} 2$
    copy $A[1 \ldots \textsc{m}]$ to $B[1 \ldots \textsc{m}]$
    copy $A[(\textsc{m} + 1) \ldots n]$ to $C[1 \ldots (n - \textsc{m})]$
    mergeSort(B)
    mergeSort(C)
    merge(B,C,A)
```

Analysis of Merge sort

```
Procedure: mergeSort(A)
$M \leftarrow n \text{ div } 2$
mergeSort(left half of A)
mergeSort(right half of A)
merge(halves into whole)
```

Analysis of Merge sort

```
Procedure: mergeSort(A)
$\textsc{m} \leftarrow n \text{ div } 2$
mergeSort(left half of A)
mergeSort(right half of A)
merge(halves into whole)
```

Each call to *mergeSort* leads to

- two sub-calls to *mergeSort*
- one call to *merge*

Each new call halves the array to be treated

Analysis of Merge sort

Procedure: `mergeSort(A)`

```
$\textsc{m} \leftarrow n \operatorname{div} 2$
mergeSort(left half of A)
mergeSort(right half of A)
merge(halves into whole)
```

Each call to *mergeSort* leads to

- two sub-calls to *mergeSort*
- one call to *merge*

Each new call halves the array to be treated

Worst case:

- in total *around* $2\log n$ calls
- each call uses one *merge*
- each *merge* uses *around* n comparisons

Analysis of Merge sort

```
Procedure: mergeSort(A)
M ← n div 2
mergeSort(left half of A)
mergeSort(right half of A)
merge(halves into whole)
```

Each call to *mergeSort* leads to

- two sub-calls to *mergeSort*
- one call to *merge*

Each new call halves the array to be treated

worst case analysis: $\Theta(n \log n)$

Summary

Class	Name	Characteristics
$\Theta(1)$	constant	few interesting algorithms
$\Theta(\log n)$	logarithmic	result of cutting problem size in half each time round a loop
$\Theta(n)$	linear	Algorithms which scan an array/list
$\Theta(n \log n)$	n-log-n	Many divide-and-conquer algorithms
$\Theta(n^2)$	quadratic	Algorithms with two embedded loops
$\Theta(n^3)$	cubic	Algorithms with three embedded loops
$\Theta(2^n)$	exponential	Many important problems