7SENG010W Data Structres & Algorithms

Week 4 Lecture

Stacks & Queues

Overview of Week 4 Lecture: Stacks & Queues

- *Preliminaries for Stacks & Queues*
  - Recap of *Arrays* & *Linked Lists*
  - *Static* versus *Dynamic* Data Structures
  - *Unrestricted Lists*
- *Stacks*
  - Definition & Operations
  - Implementations: Static & Dynamic
- *Queues*
  - Definition & Operations
  - Implementations: Static & Dynamic
- *.NET Generic Stack & Queue classes*
  - `Stack<T>` class
  - `Queue<T>` class
- *Analysis of Algorithms – the "Empirical Approach"*

Acknowledgements: these notes are partially based on those of P. Brennan.

PART I

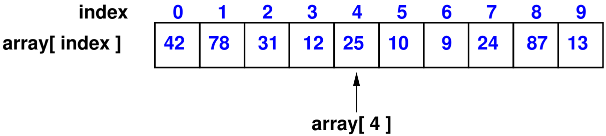
Preliminaries for Stacks & Queues

Recap – Arrays (Week 2) & Lists (Week 3)

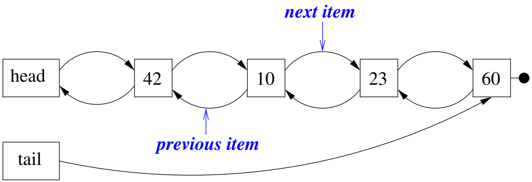
Two topics previously covered are relevant to the topics covered in this lecture:

- *Arrays*: a basic *sequential data structure* and its operations. .

Integer Array of 10 elements indexes from 0 to 9 (= 10–1)



- *Linked Lists*: a basic *sequential data structure* and its operations, e.g. a **doubly linked** list.



Static vs. Dynamic Implementations

Arrays (and structures or records) are **static** which means that the compiler allocates a **fixed amount of memory space** for storage of the array.

As we shall see **arrays** can be used to store data structures such as *stacks, queues, lists,* etc.

However, such **static implementations** have **disadvantages**:

- The size of the array is fixed, the maximum number of items which can be stored at a given time is constrained by the array size.
  Big disadvantage since the maximum size of the data structure may not be known in advance.

- The compiler **allocates memory space for all elements** of the array.
  If/when the data structure is **not full, memory space will be wasted**.

- Insert and delete **operations can be inefficient**.
  Due to having to “shunt” data around to accommodate new values in the array.

Dynamic Implementations & Memory Management

To avoid these disadvantages we can use a **dynamic implementation of dynamic data structures**.

So a general principle should be adopted:

> *The amount of memory used to store a data structure should change as the size of the data structure changes.*

Consequently, the amount of memory used by a data structure should be **directly proportional** to the amount of information stored in the structure at any time.

Restrictions on Lists

In a *pure list* data structure there are very few restrictions, e.g. *order of items in the list* is *not* specified; *insertion* & *deletion* can occur *anywhere* in a list.

However, most applications do not use a *pure list*, but one with additional constraints, such as on:

- its *structure*, e.g. limiting its minimum or maximum length.
- its *ordering* of the items, e.g. based on the value of an attribute of an item, such as *priority*, *size*, *order of insertion*, etc
- how its *operations can be performed*, e.g. limiting its maximum length, or only allowing insertions at the head, or tail of the list, etc.

Therefore, the *insert* and *delete* operations *must preserve* whatever additional constraints apply to a list when modifying it.

For example, the specific *ordering* used on the items in the list would have to be preserved, or where items can be inserted or deleted.

This week, we will look at two data structures that have additional restrictions & can be implemented using either arrays or lists these are *stacks* & *queues*.

PART II

Stacks

Stacks

A *stack* is a data structure containing a collection of items of the same data type that **can only be accessed at one end**, known as the *top* of the stack.

Items can be:

- "*pushed*" onto the stack – that is *added* onto the *top* of the stack,
- "*popped*" off the stack – that is *removed* from the *top* of the stack.

A *stack* is known as a *Last-In-First-Out (LIFO)* data structure[4].

This is because the *last item added to the stack* by *push*, will be the *first item removed from it* by *pop*.

The "*top of a stack*" when it is implemented as:

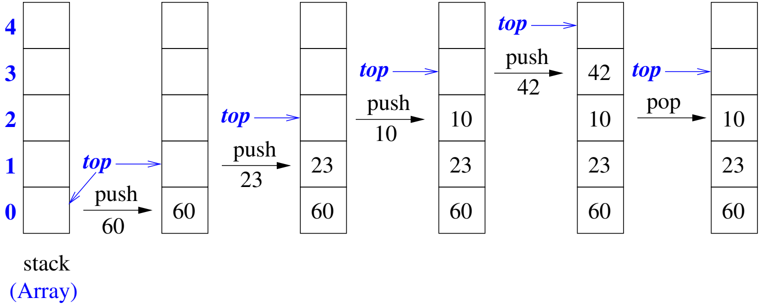
- an *array* is usually the highest **unused index**, but can also be the highest **used index**.
- a *list* is the *head of the list*.

[4] Can also be referred to as *First-In-Last-Out (FILO)*.

Examples of Stacks

A **stack** produced by:

```
push(60) ;
push(23) ;
push(10) ;
push(42) ;
pop() ;
```



Implemented using an array, with **top** as the highest **unused index**.

Stack – Static Implementation using an Array

As was illustrated on the previous slide, a **static implementation** of a stack can be achieved by using an array to store the stack items.

When this is done we need to keep track of the position of the **top** of the stack.

A variable might be used to store the index value of the array element which is the **next free space** at the top of the stack, e.g.

```
final int MAX_STACK_SIZE = 10 ;  // fixed constant size

int stack[ MAX_STACK_SIZE ] ;    // the stack


int topOfStack = 0 ;        // points to next unused space

// int topOfStack = -1 ;    // points to last used space
```

Stack Operations

The pseudo code for the main stack operations is as follows.

Initialising a stack as empty:

```
Initialise:
            topOfStack <-- 0   // Or <-- -1
```

When implementing a stack it is necessary for the programmer to determine if the **stack** is **empty** or **full** and sometimes what the value at the top of the stack is *without popping it*.

```
isEmpty:
         RETURN topOfStack equals 0   // Or -1


isFull:
         RETURN  topOfStack greater  than or equal to MAX_STACK_SIZE

top:
         RETURN  stack[ topOfStack -1 ]
```

The `isFull` and `top` alternatives are left as an exercise.

Stack Operations: Push, Pop

The **push** and **pop** operations must ensure that **no attempt** should be made to either:

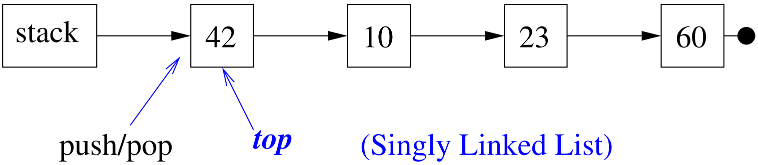
- **push** an item on to a **full stack**,
- **pop** an item from an **empty stack**.

```
Push:
      IF    isFull
      THEN
            ERROR
      ELSE
            stack[ topOfStack ] <-- pushedItem
            increment topOfStack
      ENDIF


Pop:
      IF    isEmpty
      THEN
            ERROR
      ELSE
            decrement topOfStack
            poppedItem <-- stack[ topOfStack ]
            stack[ topOfStack ] <-- NULL
      ENDIF
```

Stacks Dynamic Implementation using Lists

A **stack** produced by: `push(60); push(23); push(10); push(42);`



The method is similar to that used to provide a dynamic implementation of a list.

Since stack operations operate at just the top (**head**) of the stack (**list**) and there is usually no need to traverse the list in both directions then the implementation only requires:

- a **singly linked list**,
- only a link (reference/pointer) to the **head** of the list.

Stacks are Restricted Lists

A **stack** is a restricted form of list in the sense that:

- The insertion (push) and deletion (pop) are **only permitted to operate at the same end of the stack**, i.e. the top.

  Unlike a list where they can be performed anywhere.

  Either the **head** or **tail** of the list can be chosen as the **top** of the stack, but usually it is the **head**.

- In the vast majority of uses of stacks the **ordering** of items in a stack are based solely on the *order of insertion*.

PART III

Queues

Queues

A **queue** is a data structure containing a collection of values of the same data type which can be **accessed at both ends**:

- data items are *queued* (or inserted, added) at the **rear** (or tail) of the queue,
- data items are *dequeued* (or deleted, removed) from the **front** (or head) of the queue.

The concept of a queue data structure in computing mirrors that in real-life.

Data items are: **retrieved in the same order as they are added to the queue**, i.e. data is processed on a "*first come, first served*" basis.

A **queue** is known as a **First-In-First-Out (FIFO)** data structure[5].

This is because the *first item added to the queue* will be the *first item removed*.

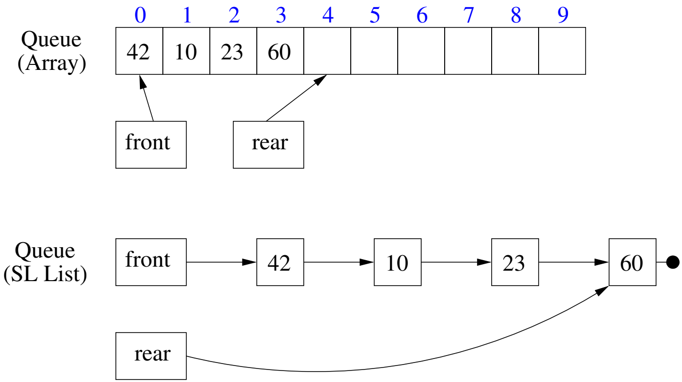
In other words, the **order of items** flowing through a queue is maintained.

[5] Can also be referred to as **Last-In-Last-Out (LILO)**.

Examples of Queues

A **queue** produced by:

```
queue(42) ;   queue(10) ;   queue(23) ;   queue(60) ;
```



Implemented first as an **array**, with **rear** pointing to the next free slot; second as a **singly linked list**.

Queue – Static Implementation using an Array

An **array** can be used to store the items in the **queue**.

As well as the array, two variables need to be used as “**pointers**” to:

- the **front** position of the queue,
- the **rear** position of the queue.

```
final int MAX_QUEUE_SIZE = 10 ;

int[] queue = new int[ MAX_QUEUE_SIZE ] ;

int front = 0 ;     // points to front
int rear  = 0 ;     // points to back, next free space

int numberInQueue = 0 ;
```

NOTE: A similar discussion as with the **top of the stack** can be had about what `rear` points to, i.e. either the *actual last item in the queue* or the *next free index value in the array*.

Using these declarations we can now give the queue operations algorithms in pseudo code.

Queue Operations

Main queue operations are: create queue, adding an item, removing an item, is queue empty or full, list queue contents.

```
initialisation:
               front <-- 0
               rear  <-- 0
               numberInQueue <-- 0
isEmpty:
        RETURN  numberInQueue equals 0

isFull:
        RETURN  numberInQueue equals MAX_QUEUE_SIZE

queueItem:
            queue[ rear ] <-- queuedItem
            rear          <-- (rear + 1) % MAX_QUEUE_SIZE
            increment numberInQueue

dequeueItem:
            dequeuedItem  <-- queue[ front ]
            front  <-- (front + 1) % MAX_QUEUE_SIZE
            decrement numberInQueue
```

Queue Implements a Circular Queue

The above array implementation of a queue implements a **circular queue**.

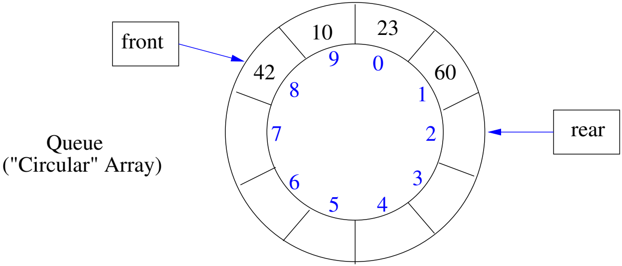
A **queue** produced by:

```
// front = 0, rear = 0, numberInQueue = 0,  insert 8 items: 1 - 8
queue(1) ;   queue(2) ;   ...   queue(8) ;

// front = 0 rear = 8, , numberInQueue = 8, remove 8 items
dequeue() ;   dequeue() ;   ...   dequeue() ;

// front = 8 rear = 8, numberInQueue = 0, insert 4 items
queue(42) ;   queue(10) ;   queue(23) ;   queue(60) ;

// front = 8 rear = 2, numberInQueue = 4
```



Queue Implements a Circular Queue

The above array implementation of a queue implements a **circular queue**.

As **insert** and **remove** operations are performed, the *portion of the array that contains the queue elements migrates through the array*.

When the "**queue portion**" reaches the end of the array we allow it to *wrap around to the beginning of the array*.

This is achieved by using **modulo** arithmetic: $x \% y$ on the two pointers **front** and **rear** when an item is *queued* and *dequeued* respectively:

```
queueItem:    rear   <-- (rear + 1)  % MAX_QUEUE_SIZE

dequeueItem:  front  <-- (front + 1) % MAX_QUEUE_SIZE
```

Where: modulo arithmetic $x \% y$ is defined as the **remainder** after x is **integer divided** by y, e.g. $23 \% 10 = 3$, $(9 + 1) \% 10 = 0$.

Queue Dynamic Implementation using Lists

As was illustrated previously, a **queue** is a natural fit for being implemented using a dynamic data structure such as a list.

The approach is similar to that used to provide a dynamic implementation of a list.

Since queue operations operate at both the front (**head**) and rear (**tail**) of the queue (**list**) and there is again usually no need to traverse the list in both directions then the implementation only requires:

- a **singly linked list**,
- a link (reference/pointer) to both the **head** and **tail** of the list.

Queues are Restricted Lists

Just like a stack, a **queue** is a restricted form of list in the sense that:

- The insertion (queue) and deletion (dequeue) are **only permitted to operate at the rear and front of the queue** respectively.

  Unlike a list where they can be performed anywhere.

  Either the **head** or **tail** of the list can be chosen as the **front** or **rear** of the queue **as long as they operate at opposite ends**.

- As with stacks in the vast majority of cases the **ordering** of items in a queue are based solely on the *order of arrival/insertion*.

Finally, we shall return to queues in Week 8, in the form of *priority queues*, these are data structures that store a collection of *(key, data)* values either in ascending or descending *key* order.

PART IV

C♯/.NET Stack & Queue Classes

`System.Collections.Generic`

`Stack<T>`

`Queue<T>`

C♯ Generic Stack Class: `Stack<T>`

- This is the C♯ *generic stack* class.
- `Stack<T>` is a generic class & `T` is its *type parameter* that specifies the type of elements in the stack.
- It is *variable sized* and is a "*last-in-first-out" (LIFO)* collection of instances of the same specified type.
- `Stack<T>` is implemented as an array.
- Three main operations can be performed on a `Stack<T>` and its elements:
  - *Push* – *inserts* an element at the *top of* the `Stack<T>`.
  - *Pop* – *removes* an element from the *top of* the `Stack<T>`.
  - *Peek* – *returns* an element that is at the top of the `Stack<T>`, but **does not remove it** from the `Stack<T>`.
- The capacity of a `Stack<T>` is the number of elements the `Stack<T>` can hold; as elements are added its capacity is automatically increased.
- See the `Stack<T>` class documentation for details & example programs.

The `Stack<T>` class API (1/2)

Property:

`Count` – *gets* the *number of elements* contained in the `Stack<T>`.

Constructors

Initialise a new instance of the `Stack<T>` class:

`Stack<T>()` – is empty & with default capacity.

`Stack<T>(Int32)` – is empty & has the *specified initial capacity* or the default capacity, whichever is greater.

`Stack<T>( IEnumerable<T> )` – contains elements copied from the specified collection & has sufficient capacity to accommodate the number of elements copied.

The `Stack<T>` class API (2/2)

Examples of the `Stack<T>` class's methods:

```
public void Push( T item ) ;
// Inserts item at the top of the Stack<T>

public T Pop() ;
// Removes & returns the object at the top of the Stack<T>

public T Peek() ;
// Returns the object at the top of the Stack<T> without removing it

public bool Contains( T item ) ;
// Tests if the item is in the Stack<T>, returns
// true if it is; false otherwise

public void Clear() ;
// Removes all objects from the Stack<T>

public T[] ToArray() ;
// Copies the Stack<T> elements into a new array T[]
```

See the [`Stack<T>`] class for a full list of methods.

Example of `Stack<T>` Class

Using an instance of `Stack<T>` with `T` as `string`, to represent putting luggage into a car's boot, first item in, is the last item out, alternatively the last item in is the first item out[6].

```
Stack<string> carsBoot = new Stack<string>() ;

carsBoot.Push("BlueSuitcase") ;    // hard luggage first
carsBoot.Push("RedSuitcase") ;
carsBoot.Push("Holdall") ;
carsBoot.Push("SportsBag") ;
carsBoot.Push("Rucksack") ;

// List luggage in boot, starting from the last item added
foreach( string luggage in carsBoot ) {
     Console.WriteLine( luggage );
}

Console.WriteLine( "{0} items of luggage in the car's boot.",
                   carsBoot.Count ) ;

Console.WriteLine("Is the camera bag in the boot {0}",
                   ( carsBoot.Contains( "CameraBag" ) ? "yes" : "NO!!" ) ) ;

Console.WriteLine("Got out the {0} from the boot",   carsBoot.Pop() ) ;
Console.WriteLine("Next item to get out is the {0}", carsBoot.Peek()) ;
Console.WriteLine() ;
```

[6] Full version on the module Blackboard site.

C♯ Generic Queue Class: `Queue<T>`

- `Queue<T>` is generic & its queue elements are of type `T`.
- It is a "*first-in-first-out" (FIFO)* collection of elements of the same type & implemented as a *variable sized circular array*.
- Objects stored in a `Queue<T>` are *inserted at one end* & *removed from the other end*.
- Three main operations can be performed on a `Queue<T>` & its elements:
  - *Enqueue* – *adds* an element to the *end of* the `Queue<T>`.
  - *Dequeue* – *removes* the *oldest element from the start of* the `Queue<T>`.
  - *Peek* – *returns* the *oldest element from the start of* the `Queue<T>`, but **does not remove it** from the `Queue<T>`.
- The capacity of a `Queue<T>` is the number of elements the `Queue<T>` can hold; as elements are added its capacity is automatically increased.
- See the `Queue<T>` class documentation for details & example programs.

The `Queue<T>` class API (1/2)

Property:

`Count` – *gets* the *number of elements* contained in the `Queue<T>`.

Constructors

Initialise a new instance of the `Queue<T>` class:

`Queue<T>()` – is empty & with default capacity.

`Queue<T>(Int32)` – is empty & has the *specified initial capacity* or the default capacity, whichever is greater.

`Queue<T>( IEnumerable<T> )` – contains elements copied from the specified collection & has sufficient capacity to accommodate the number of elements copied.

The `Queue<T>` class API (2/2)

Examples of the `Queue<T>` class's methods:

```
public void Enqueue( T item ) ;
// Adds an object to the end of the Queue<T>

public T Dequeue() ;
// Removes & returns the object at the beginning of the Queue<T>

public T Peek() ;
// Returns the object at the beginning of the Queue<T> without removin

public bool Contains( T item ) ;
// Tests if the item is in the Queue<T>, returns
// true if it is; false otherwise

public void Clear() ;
// Removes all objects from the Queue<T>

public T[] ToArray() ;
// Copies the Queue<T> elements into a new array T[]
```

See the `Queue<T>` class for a full list of methods.

Example of Queue<T> Class

Using an instance of `Queue<T>` with `T` as `string`, to represent a shop's checkout queue of people[7].

```
// first  2 customers waiting to check out
string[] shoppers = { "Jim", "Sue" } ;

// Create a shop check out queue of 2 people
Queue<string> checkout = new Queue<string>( shoppers ) ;

checkoutQueue.Enqueue("Tom") ;  // people join the end of the queue
checkoutQueue.Enqueue("Mia") ;
checkoutQueue.Enqueue("Zoe") ;

// print people waiting in the queue
foreach ( string person in checkoutQueue ) {
    Console.WriteLine( person ) ;
}

// first person (Jim) leaves queue to be served
Console.WriteLine("Now serving customer: {0}", checkoutQueue.Dequeue() ) ;

Console.WriteLine("Next to be served is: {0}", checkoutQueue.Peek() ) ;

Console.WriteLine("{0} customers waiting to be served", checkoutQueue.Count);
```

[7] Full version on the module Blackboard site.

PART V

Analysis of Algorithms

– the "Empirical Approach"

Recap of Algorithm Analysis

- *Algorithm analysis* estimates the "*resource*" (e.g. computation/execution time, storage space) consumption of an algorithm.
- Allows comparison of the *relative costs* of a group of algorithms for solving the same task, e.g. sorting, searching, etc.
- Algorithm analysis measures the *efficiency of an algorithm* or its implementation as a *program* as the: "*size of the input data increases*".
- Typical types of analysis are:
  - calculate the "*order of complexity*" of an algorithm to complete its task, or
  - estimate or measure the *execution time* for a program to complete its task, or
  - estimate the storage space required for a data structure.
- Complexity analysis measures an algorithm's execution time for a given problem size N by using a growth-rate function $T(N)$.
- 3 complexity analysis measures for *amount of "work"* an algorithm or program requires for a problem of size N:
  - *Worst-case* analysis (Big-O) – *maximum amount of work*.
  - *Average-case* analysis (Big-Θ) – *expected amount of work*.
  - *Best-case* analysis (Big-Ω) – *least amount of work*.

The Empirical Approach (1/2)

- We can get evidence regarding the *complexity of an algorithm* by sampling the *execution times* of its implementation on a variety of inputs.
- This is an example of the "*empirical approach*" used throughout science.
- **Advantage:** it is easy to achieve given an implementation of the algorithm is available & suitable sample inputs are available.
- **Some caveats:**
  - Depends on *chosen inputs*:
    - Many problems have easy special cases with *lower complexity*, e.g. find the value looked for straightaway in a binary search.
    - Need to ensure that our sample inputs are representative, e.g. randomised & not only close to the *best case* inputs.
  - Depends on *implementation details*:
    - You are essentially testing the implementation.
    - This can help find implementation errors if you know what the complexity of the algorithm should be, e.g. the Big-O (*Worst-case*), Big-Θ (*Average-case*), Big-Ω (*Best-case*) values, & it does not correspond to these values.
  - Depends on *hardware* (memory, cache) , amount of CPU used by other processes, . . .

The Empirical Approach (2/2)

- **Basic idea:** if the *complexity of an implementation* is –
  - *Logarithmic* $O(\log_2(N))$:
    then repeatedly doubling the input size will always:
    increase the execution time by the same amount.
  - *Linear* $O(N)$, *Quadratic* $O(N^2)$, *Cubic* $O(N^3)$:
    then repeatedly doubling the input size will always:
    multiply the execution time by $2 = 2^{\mathbf{1}}$, $4 = 2^{\mathbf{2}}$, $8 = 2^{\mathbf{3}}$, respectively.
  - *Exponential* $O(2^N)$:
    then repeatedly increasing the input size by a fixed amount will always:
    multiply the execution time by some fixed amount.
- These *relations* will usually only be **approximate**, i.e. $\approx$ rather than $=$.

  Usually it oscillates around a particular value & only converges to the value when the input size N gets very large.
- Only a valid approach if **enough data points** are used, e.g. a large enough sample of inputs, to draw any conclusions.

Empirical Approach – Example 1

The execution times $T(N)$ of a particular algorithm *Alg-1*:

Input Size N	**Execution Time** $T(N)$	**Ratio** $T(2N)/T(N)$	**Ratio Value**
1000	7	–	–
2000	13	13 / 7	1.857
4000	27	27 / 13	2.077
8000	52	52 / 27	1.926
16000	103	103 / 52	1.981
32000	208	208 / 103	2.019

- The *input size doubles* from row to row, N to $2N$.
- Dividing each execution time $T(2N)$ by the previous one $T(N)$, e.g. $T(2N)/T(N)$, gives an approximately *doubling ratio*, e.g. $208/103 \approx 2$.
- The $T(N)$'s are *approximately doubling*, thus the ratio is converging towards 2, and $2 = 2^{\mathbf{1}}$ which means order of complexity is $O(N^{\mathbf{1}}) = O(N)$
- This is evidence that *Alg-1*'s complexity is *linear*, i.e. $O(N)$.

Empirical Approach – Example 2

The execution times $T(N)$ of a particular algorithm *Alg-2*:

Input Size N	Execution Time $T(N)$	Ratio $T(2N)/T(N)$	Ratio Value	Difference $T(2N) - T(N)$	Diff Value
100	17	–	–	–	–
200	22	22 / 17	1.294	22 - 17	5
400	28	28 / 22	1.272	28 - 22	6
800	33	33 / 28	1.178	33 - 28	5
1600	39	39 / 33	1.182	39 - 33	6
3200	44	44 / 39	1.128	33 - 28	5

- Dividing each execution time by the previous one $T(2N)/T(N)$, gives a *doubling input ratio* almost equal to 1, i.e. $1 < 2$ so **cannot** be a linear (2), quadratic (4), etc, complexity.
- Taking the differences between successive execution times $T(2N) - T(N)$ gives a *roughly constant difference*, e.g. 5 or 6.
- So they do not change except for small fluctuations.
- This is evidence that *Alg-2*'s complexity is *logarithmic*, i.e. $O(\log_2(N))$.