Quicksort

```
method quicksort(a: array<int>)
{
    quicksort2(a, 0, a.Length-1);
}
method quicksort2(a: array<int>, lo: int, hi: int){
{
    if lo < hi
    {
        var pivot := partition(a, lo, hi);
        quicksort2(a, lo, pivot  - 1);
        quicksort2(a, pivot + 1, hi);
    }
}
```

Quicksort

```
method partition(a: array<int>, lo: int, hi: int)
         returns(pivot: int) {
    var i := lo;
    var j := lo;
    pivot := hi;
    while  j < hi
    {
        if a[j] < a[hi]
        {
            a[i], a[j] := a[j], a[i];
            i := i + 1;
        }
        j := j+1;
    }
    a[hi], a[i] := a[i], a[hi];
    pivot := i;
    return pivot;}
```

Predicados

```
predicate sorted(a: array<int>, lo: int, hi: int)
reads a
{
    forall i, j :: 0 <= lo <= i < j <= hi < a.Length ==>
              a[i] <= a[j]
}
```

```
predicate partitioned(a: array<int>, lo: int, hi: int)
reads a
{
    (forall i, j :: 0 <= i < lo <= j <= hi < a.Length
       ==> a[i] <= a[j])
    && (forall i, j :: 0 <= lo <= i <= hi < j < a.Length
        ==> a[i] <= a[j])
}
```

Pré-condições

- quicksort2:

  ```
  requires 0 <= lo <= hi + 1 <= a.Length
  ```

- partition:

  ```
      requires 0 <= lo <= hi < a.Length
      requires partitioned(a, lo, hi)
  ```

Pós-condições

- quicksort:

  ```
  ensures sorted(a, 0, a.Length-1)
  ensures multiset(a[..]) == multiset(old(a)[..])
  ```

- quicksort2:

  ```
  ensures sorted(a, lo, hi)
  ensures multiset(a[lo..hi+1]) == multiset(old(a)[lo..hi+1])
  ensures forall k :: 0 <= k < lo || hi < k < a.Length
           ==> a[k] == old(a[k])
  ensures partitioned(a, lo, hi)
  ```

- partition: (p is `pivot`)

  ```
  ensures lo <= pivot <= hi
  ensures forall k :: lo <= k < p ==> a[k] <= a[p]
  ensures forall k :: p < k <= hi ==> a[k] >= a[p]
  ensures multiset(a[lo..hi+1]) == multiset(old(a)[lo..hi+1])
  ensures forall k :: 0 <= k < lo || hi < k < a.Length ==>
       a[k] == old(a[k])
  ensures partitioned(a, lo, hi)
  ```

Invariantes

Para o `while` do `partition`. Notar que `pivot==hi`.

```
invariant lo <= i <= j <= hi
invariant forall k :: lo <= k < i ==> a[k] < a[pivot]
invariant forall k :: i <= k < j ==> a[k] >= a[pivot]
invariant multiset(a[lo..hi+1]) == multiset(old(a)[lo..hi+1])
invariant forall k :: 0 <= k < lo || hi < k < a.Length
     ==> a[k] == old(a[k])
invariant partitioned(a, lo, hi)
```

Objectos: `class`

```
class Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    constructor (cap: nat)
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate method isEmpty()
    {
        size == 0
    }
    predicate method  isFull()
    {
        size == elems.Length
    }
    method push(x : T)
    {
        elems[size] := x;
        size := size + 1;
    }
    function method  top() : T
    {
         elems[size-1]
    }
    method pop()
    {
         size := size-1;
    }
```

```
}
```

Objectos

- atributos/campos: `elems, size`
- constructores : chamados quando os objectos são criados
- métodos: `method`
- criação: `var s:= new Stack(3)`

```
Stack

method {:verify false} testStack()
{
    var s := new Stack(3);
    assert s.isEmpty();
    s.push(1);
    s.push(2);
    s.push(3);
    assert s.top() == 3;
    assert s.isFull();
    s.pop();
    assert s.top() == 2;
}
```

Objectos

- Classes podem estender outras classes
- Métodos podems er declados estáticos
- O objecto é referido por `this`
- Os constructores podem ter nomes
- O tipo de todas as classes é `object`
- Atributos imutáveis podem ser declarados com `const`
- podem incluir predicados ou funções

Invariante de classe/objecto

- predicado `Valid()` que descreve o invariante da classe
- este invariante é requerido (pré-condição) por todos os métodos (excepto os constructores)
- e verificado no fim da execução (pós-condição) por todos os constructores e modificadores (qualquer método)
- com o atributo `:autocontracts` o Dafny adiciona automaticamente `requires/ensures` do predicado `Valid()` e outros contractos necessários.

Invariant de `Stack`

```
predicate Valid()
reads this
{
size <= capacity == elems.Length
}

constructor (cap: nat)
    requires cap > 0
    ensures fresh(elems)
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }

predicate method  isEmpty()
    reads this
    {
        size == 0
    }

predicate method  isFull()
    reads this
    {
        size == elems.Length
    }

method  push(x : T)
    modifies this, elems
    requires Valid() && size < capacity
```

```
    {
        elems[size] := x;
        size := size + 1;
    }

function method  top(): T
    requires Valid() && size > 0
    reads this, elems
    {
         elems[size-1]
    }

method pop()
    requires Valid() && size > 0
    modifies this
    ensures Valid()
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}


:autocontracts

class{:autocontracts} Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    predicate Valid()
    reads this
    { size <= capacity == elems.Length
    }
    constructor (cap: nat)
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate method  isEmpty()
    {
        size == 0
    }
    predicate method isFull()
```

```
    {
        size == elems.Length
    }
    method push(x : T)
    requires size < capacity
    modifies this, elems
    ensures elems[..size]==old(elems[..size])+[x]
    {
        elems[size] := x;
        size := size + 1;
    }

    function method  top(): T
    requires size > 0
    {
         elems[size-1]
    }
    method  pop()
    requires size > 0
    modifies this
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}
```