Deakin University

Data Structures and Algorithms

OnTrack Submission

# Problem solving: Search and Stack

*Submitted By:*
Peter Stacey
pstacey
2020/08/27 15:55

*Tutor:*
Maksym Slavnenko

| **Outcome** | **Weight** |
|---|---|
| Complexity | ♦♦♦◊◊ |
| Implement Solutions | ♦♦♦◊◊ |
| Document solutions | ♦♦♦◊◊ |

This task aligns with all three learning outcomes as it involves evaluating the complexity of different design possibilities for algorithms, the space complexity of different data structures (ULO1). It involves not only evaluating from design, but also implementing the designs to confirm the assumptions and assessment (not a mandatory component, but the best way to check) which aligns with ULO2; and it involves documenting the solution and constraints and trade-offs (ULO3).

August 27, 2020

# *Task 6.1*

Student Name: Peter Stacey

Student ID: 219011171

# Question 1

*Design a $\theta(n \log n)$ time algorithm that, given a set $S$ of $n$ integer numbers and another integer $x$, determines whether or not there exist two elements in $S$ whose sum is exactly $x$.*

**Approach**

Given a set S of n integers:

1. Sort S from smallest to largest (largest to smallest is also ok)
2. Create two pointer variables (eg. I and j) pointing at the start of sorted S (variable i) and the end of sorted S (variable j)
3. While the value at S[j] + S[i] > sum, j—
4. While the value at S[j] + S[i] < sim, i++
5. If S[j] + S[i] equals sum, return true
6. Return false if no matching sum is found

**Assumptions**

- Sort is conducted no worse than *O(n log n)*

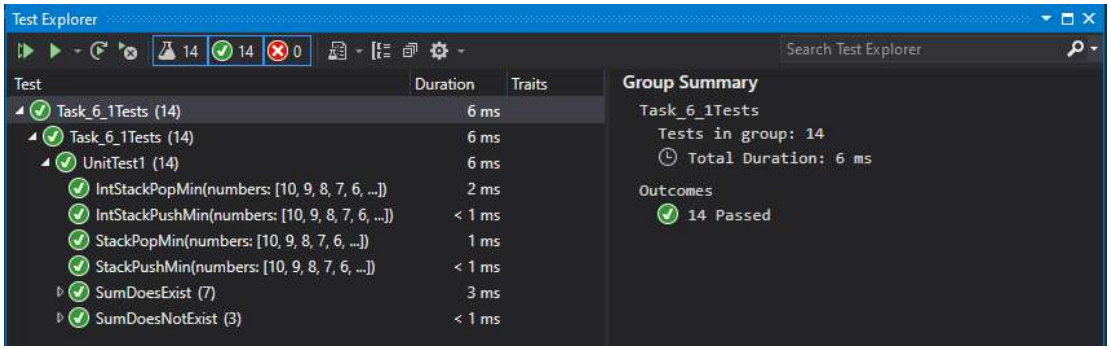
**C# Code**

```
using System;

namespace Task_6_1
{
    2 references
    public class SumExists
    {
        2 references
        public static bool Check(int[] numbers, int sum)
        {
            if (numbers is null) throw new ArgumentNullException();    // 1 operation
            Array.Sort(numbers);                                       // O(n log n)
            int i = 0;                                                 // 1 operation
            int j = numbers.Length - 1;                                // 1 operation
            while (i < j)                                              // n in worst case
            {
                if (numbers[i] + numbers[j] == sum) return true;       // n - 1 in worst case
                else if (numbers[i] + numbers[j] < sum) i++;           // 2(n - 1) in worst case
                else j--;                                              // n - 1 in worst case
            }
            return false;                                              // 1 operation
        }
    }
}

```

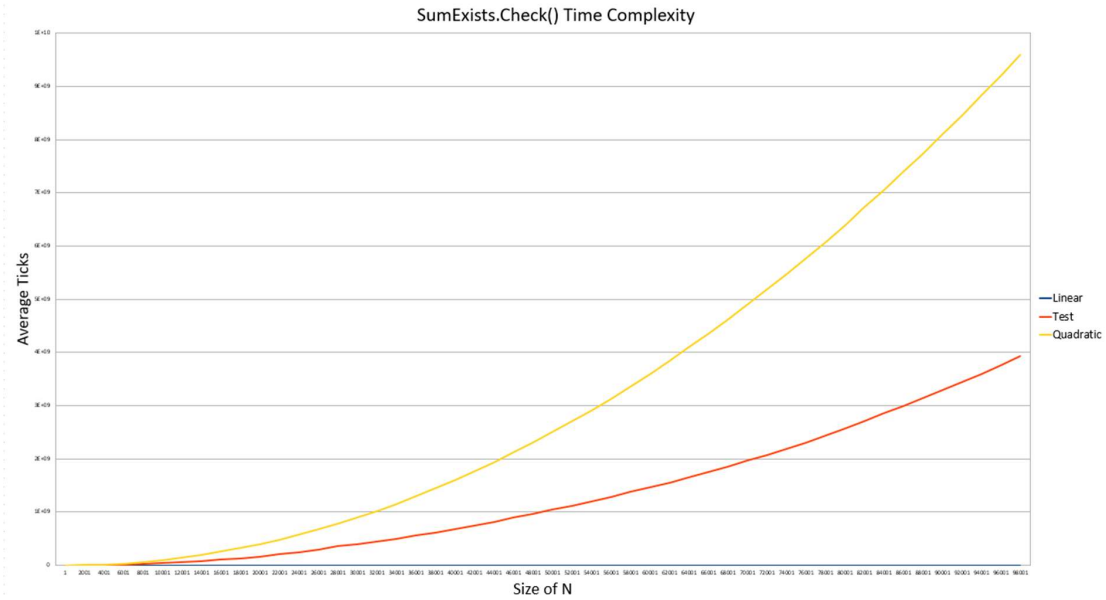
**Tests**

```
int[] numbers = {10, 5, 6, 8, 2, 19, 9, 11, 15, 1, -8, 50};

[Theory]
[InlineData(-2)]   // 6 + -8
[InlineData(0)]    // 8 + -8
[InlineData(1)]    // 9 + -8
[InlineData(15)]   // 10 + 5
[InlineData(26)]   // 15 + 11
[InlineData(42)]   // 50 + -8
[InlineData(69)]   // 50 + 18
0 references
public void SumDoesExist(int sum)
{
    Assert.True(SumExists.Check(numbers, sum));
}

[Theory]
[InlineData(-100)]
[InlineData(50)]
[InlineData(100)]
0 references
public void SumDoesNotExist(int sum)
{
    Assert.False(SumExists.Check(numbers, sum));
}
```

**Test Results**




**Performance**




# Question 2

*A Stack data structure provides Push and Pop, the two operations to write and read data, respectively. Using the Stack as a starting point design a data structure that, in addition to these two operations, also provides the Min operation to return the smallest element of the stack. Remember that the new data structure must operate in a constant $\theta(1)$ time for all three operations.*

**Approach**

There are a couple of different approaches immediately obvious, to ensure Θ(1) Push, Pop and Min methods.

These involve:

1. A stack for a pre-defined data type
2. A stack for a Generic data type

In developing a Stack for a pre-defined data type (eg. int), only one additional variable is required, to hold the current minimum value in the stack.

In developing a Stack for a generic data type, the generic type needs to implement IComparable<T> in order for values in the Stack to be compared, and a separate collection created to hold the set of minimum values.

**Assumptions**

- For a pre-defined data type stack, + and – operations can be applied to the data type
- For a generic data type stack, the generic type implements IComparable<T>
- For a generic data type stack, the use of O(n) additional space is acceptable

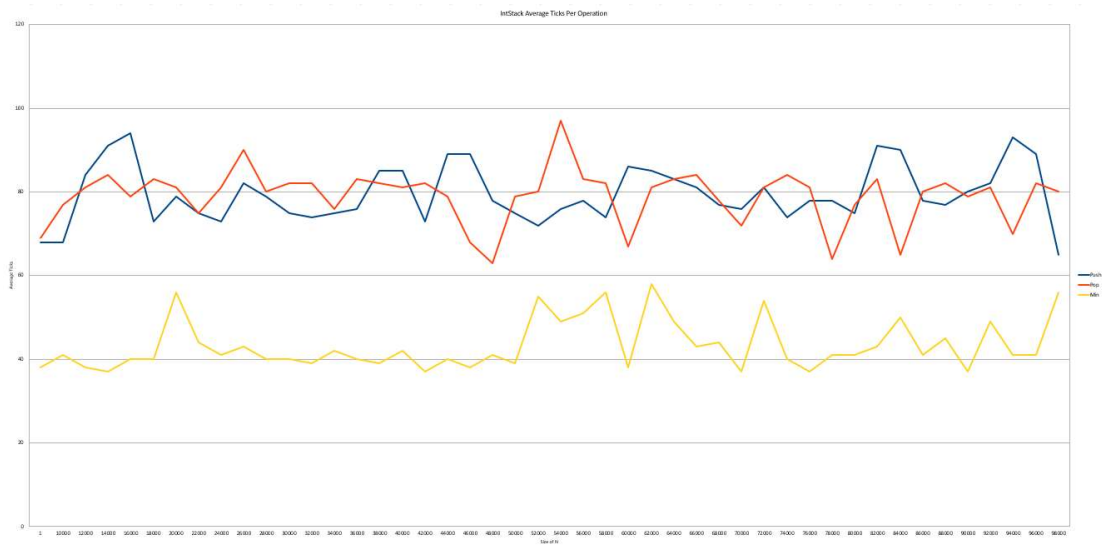
**Option 1: Stack of integers (could also be applied to other numeric types)**

```
using System;
using System.Collections.Generic;

namespace Task_6_1
{
    8 references
    public class IntStack
    {
        const int DEFAULT_MAX_SIZE = 100;
        private int[] _stack;
        private int _top;
        private int _min;

        2 references | 2/2 passing
        public IntStack()
        {
            _stack = new int[DEFAULT_MAX_SIZE];
            _top = -1;
        }

        1 reference
        public IntStack(int size)
        {
            if (size < 0) throw new ArgumentOutOfRangeException();
            _stack = new int[size];
            _top = -1;
        }

        3 references
        private bool IsEmpty() => _top is -1 ? true : false;
        1 reference
        private bool IsFull() => _top == _stack.Length - 1 ? true : false;

        3 references | 2/2 passing
        public void Push(int value)
        {
            if (IsFull()) throw new StackOverflowException();
            if (IsEmpty())
            {
                _stack[++_top] = value;
                _min = value;
                return;
            }

            if (value < _min)
            {
                _stack[++_top] = (2 * value - _min);
                _min = value;
            }
            else
            {
                _stack[++_top] = value;
            }
        }

        1 reference | 1/1 passing
        public int Pop()
        {
            if (IsEmpty()) throw new InvalidOperationException();
            int result = _stack[_top];

            if (result < _min)
            {
                int temp = (2 * _min) - result;
                result = _min;
                _min = temp;
            }

            _top --;
            return result;
        }

        3 references | 2/2 passing
        public int Min()
        {
            if (IsEmpty()) throw new InvalidOperationException();
            return _min;
        }
    }
}
```

**Tests**

```
[Theory]
[InlineData(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)]
0 references
public void IntStackPushMin(params int[] numbers)
{
    IntStack stack = new IntStack();
    for(int i = 0; i < numbers.Length; i++)
    {
        stack.Push(numbers[i]);
        Assert.Equal((numbers.Length - 1) - i, stack.Min());
    }
}

[Theory]
[InlineData(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)]
0 references
public void IntStackPopMin(params int[] numbers)
{
    IntStack stack = new IntStack();
    for(int i = 0; i < numbers.Length; i++)
    {
        stack.Push(numbers[i]);
    }

    for(int i = 0; i < numbers.Length - 1; i++)
    {
        stack.Pop();
        Assert.Equal(i + 1, stack.Min());
    }
}
```

**Performance**



**Option 2: Generic Stack**

```
using System;
using System.Collections.Generic;

namespace Task_6_1
{
    6 references
    public class Stack<T> where T : IComparable<T>
    {
        private const int MAX_STACK_SIZE = 100;
        private T[] _stack;
        private int _count;
        private T[] _min;
        private int _minCount;

        2 references | 2/2 passing
        public Stack()
        {
            _stack = new T[MAX_STACK_SIZE];
            _count = 0;
            _min = new T[MAX_STACK_SIZE];
            _minCount = 0;
        }

        0 references
        public Stack(int size)
        {
            if (size < 0) throw new ArgumentOutOfRangeException();
            _stack = new T[size];
            _count = 0;
            _min = new T[size];
            _minCount = 0;
        }

        3 references
        private bool IsEmpty() => _count is 0 ? true : false;
        1 reference
        private bool isFull() => _count >= _stack.Length - 1 ? true : false;

        1 reference | 1/1 passing
        public T Pop()
        {
            if (IsEmpty()) throw new InvalidOperationException();
            T value = _stack[_count];
            _count--;
            if (_min[_minCount - 1].Equals(value))
            {
                _min[_minCount - 1] = default(T);
                _minCount--;
            }
            return value;
        }

        2 references | 2/2 passing
        public void Push(T value)
        {
            if (isFull()) throw new StackOverflowException();
            if (IsEmpty()) _min[_minCount++] = value;

            if (value.CompareTo(_min[_minCount - 1]) < 0)
            {
                _min[_minCount++] = value;
            }
            _stack[++_count] = value;
        }

        2 references | 2/2 passing
        public T Min()
        {
            if (IsEmpty()) throw new InvalidOperationException();
            return _min[_minCount - 1];
        }
    }
}
```

## Tests

```
[Theory]
[InlineData(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)]
0 references
public void StackPushMin(params int[] numbers)
{
    Stack<int> stack = new Stack<int>();
    for(int i = 0; i < numbers.Length; i++)
    {
        stack.Push(numbers[i]);
        Assert.Equal((numbers.Length - 1) - i, stack.Min());
    }
}

[Theory]
[InlineData(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)]
0 references
public void StackPopMin(params int[] numbers)
{
    Stack<int> stack = new Stack<int>();
    for(int i = 0; i < numbers.Length; i++)
    {
        stack.Push(numbers[i]);
    }

    for(int i = 0; i < numbers.Length - 1; i++)
    {
        stack.Pop();
        Assert.Equal(i + 1, stack.Min());
    }
}
```

## Performance