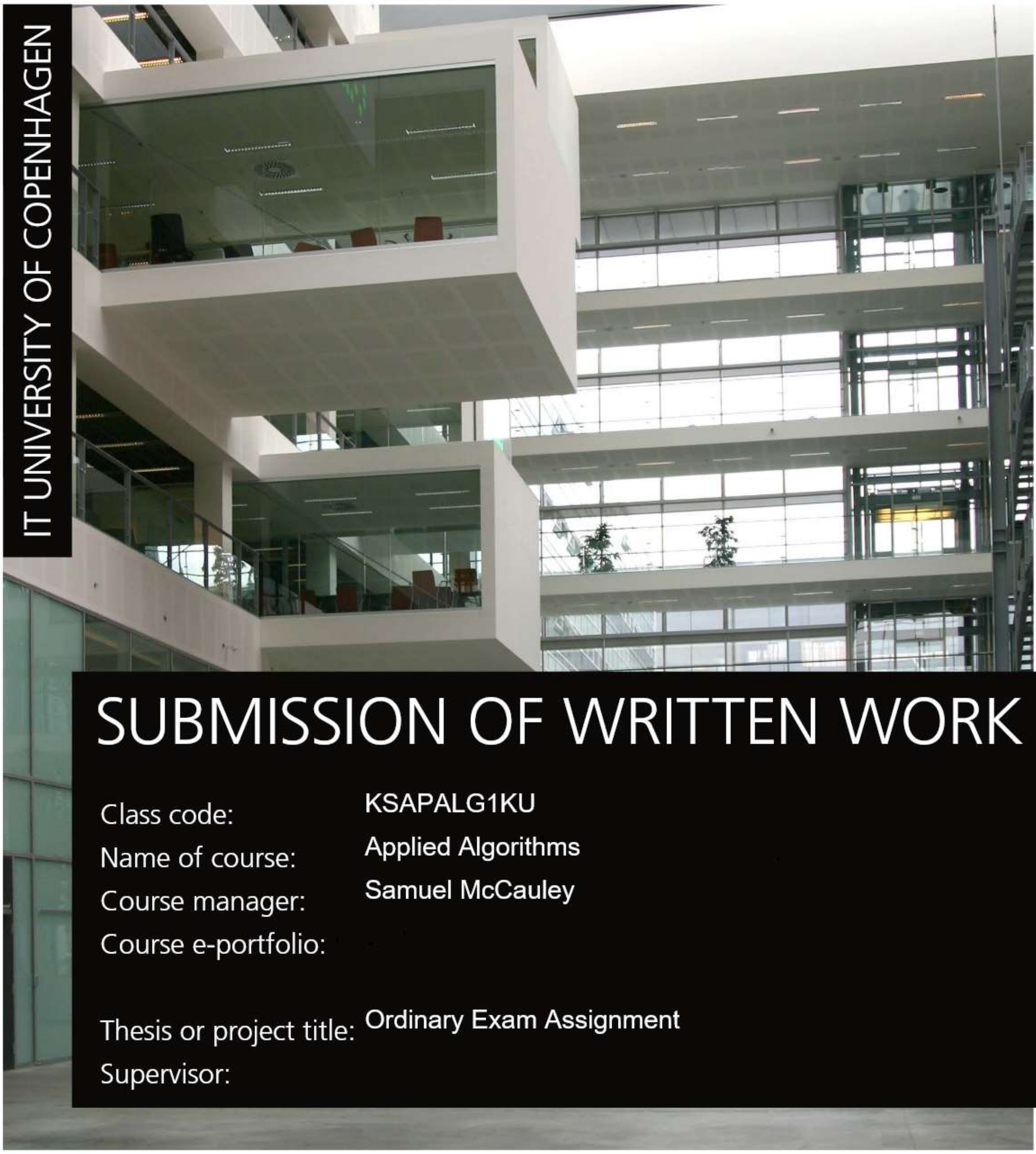

SUBMISSION OF WRITTEN WORK

Class code:	KSAPALG1KU
Name of course:	Applied Algorithms
Course manager:	Samuel McCauley
Course e-portfolio:	
Thesis or project title:	Ordinary Exam Assignment
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Emil Christian Lynegaard	17/07-1994	ecly@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk

Applied Algorithms

Emil Lynegaard

December 18, 2017

I hereby declare that I have answered the exam questions myself without any outside help.

Contents

1 Independent Set in Interval Graphs

1.1 Implement

Implementation is seen in attached file `independent_set.go` but can also be seen in the appendix, Section 3.1.

1.2 Questions to Discuss

(a) The implementation has a running time of $O(n^2)$.

(b) We only use A_r and keep track of the right endpoint (r) of the element most recently added to the independent set. Now we can remove the head of A_r one by one, discarding elements with left endpoints (l) where $l \leq r$, and adding elements with $l > r$. This way we only have to run through A_r once, giving us $O(n)$ after sorting.

(c) Since both lists are sorted in increasing order, albeit on different properties, by searching from the start of the list we have a very good chance of finding it quickly. In the example on Figure 1 from the assignment, we will for example find the first interval [-2, -1] as the second element when we try to remove it from the list sorted by interval start.

(d) We use $l_2 \leq r$ for the inner loop since we want to discard all intervals that start before or at the same time of the end of the interval that we just added to the independent set.

(e) If A_r and A_l were priority queues, other than the regular insert and pull, we would need a `remove(elem)` method to remove a specific element somewhere in the queue.

2 Summing Triples of Integers

2.1 Implement

Implementation is seen in attached file `triples.go` but can also be seen in the appendix, Section 3.2.

2.2 Questions to Discuss

(a) Each buckets is represented as a slice[1] of integers, which is convenient as slices are dynamically-sized and we hash an arbitrary number of elements to each bucket. To store all the buckets we also use a slice, which in this case is convenient as slices have constant time random access due to their underlying arrays. The only inconvenience a slice may impose is that we can only have `Integer` keys, but since our `hash` returns an `Integer`, this is ideal.

(b) If we were to make the constant 16 into a 1, we would have an average bucket size of 1, in which case we will have to compare a lot of buckets all of which will have very few elements and generally bad cache efficiency. On the other hand, if our constant becomes too large, we end up in the worst case with a single bucket, causing us to naively try all combinations within that

[1]https://blog.golang.org/go-slices-usage-and-internals

bucket. Since we have no interest in neither of these properties, we instead look for a moderate number of buckets which affects the average size of each bucket, and reduces the total number of comparisons of $a + b == c$. The high-level concept that may benefit from this is the I/O model and thereby cache efficiency. We want the average size of each bucket to be of a size that leads to our `naiveCompare` method having a natural cache efficiency, meaning a minimal number of cache misses. This means the we are looking to balance the ratio between bucket size and the total number of buckets, to use our cache efficiently while minimizing the total number of comparisons needed.

If we ported this to a new machine, it will probably have a different CPU and that CPU may have different cache size causing a different bucket size to be optimal in terms of minimizing cache misses during our naive comparison.

3 Appendix

3.1 independent_set.go

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "sort"
    "strconv"
    "strings"
)

// Interval struct representing the beginning and end of and interval.
type Interval struct {
    From int
    To   int
}

// Utiliy function to remove an Interval from a slice of Intervals
func remove(intervals *[]Interval, elem Interval) {
    for i := 0; i < len(*intervals); i++ {
        if (*intervals)[i] == elem {
            *intervals = append((*intervals)[:i], (*intervals)[i+1:]...)
        }
    }
}

// Finds the maximum independent set in a list of Intervals
func maxIndependentSet(intervals []Interval) []Interval {
    // Make a copy of input and sort in increasing order of Interval.To
    endsFirst := make([]Interval, len(intervals))
    copy(endsFirst, intervals)
    sort.SliceStable(endsFirst, func(i, j int) bool {
        return endsFirst[i].To < endsFirst[j].To
```

```go
	})

	// Make a copy of input and sort in increasing order of Interval.From
	startsFirst := make([]Interval, len(intervals))
	copy(startsFirst, intervals)
	sort.SliceStable(startsFirst, func(i, j int) bool {
		return startsFirst[i].From < startsFirst[j].From
	})

	maxSet := make([]Interval, 0)
	for len(endsFirst) > 0 {
		fst := endsFirst[0]
		maxSet = append(maxSet, fst)
		endsFirst = endsFirst[1:]
		remove(&startsFirst, fst)

		snd := startsFirst[0]
		for snd.From <= fst.To {
			remove(&endsFirst, snd)
			startsFirst = startsFirst[1:]
			if len(startsFirst) == 0 { // avoid index out of range
				break
			}
			snd = startsFirst[0]
		}
	}

	return maxSet
}

// Reads input as seen on CodeJudge to a slice of Intervals
func readIntervals(filename string, amount int) []Interval {
	intervals := make([]Interval, 0, amount)
	if file, err := os.Open(filename); err == nil {
		defer file.Close()
		scanner := bufio.NewScanner(file)
		for scanner.Scan() {
			words := strings.Fields(scanner.Text())
			from, _ := strconv.Atoi(words[0])
			to, _ := strconv.Atoi(words[1])
			intervals = append(intervals, Interval{from, to})
		}
	}
	return intervals
}

func main() {
	filename := os.Args[1]
	amount, _ := strconv.Atoi(os.Args[2])
	intervals := readIntervals(filename, amount)
	independentSet := maxIndependentSet(intervals)
	for _, interval := range independentSet {
		fmt.Printf("%d %d\n", interval.From, interval.To)
```

```
    }
}
```

3.2 triples.go

```go
package main

import (
    "bufio"
    "fmt"
    "math"
    "math/rand"
    "os"
    "strconv"
)

// Bucket stores a subset of the input
// that hashed to this specific bucket in a list of buckets.
type Bucket []int

// Returns true if there exists a triple a+b==c in the input
// Otherwise returns false
func findTriples(input []int, amount int) bool {
    numBuckets := amount / 16
    shift := uint(math.Log2(float64(numBuckets)))
    seed := rand.Int()
    buckets := make([]Bucket, numBuckets)
    for i := 0; i < len(buckets); i++ {
        buckets[i] = Bucket{}
    }
    for _, x := range input {
        h := hash(x, seed, shift)
        buckets[h] = append(buckets[h], x)
    }
    for i := 0; i < len(buckets); i++ {
        for j := i; j < len(buckets); j++ {
            k1 := (i + j) % numBuckets
            if naiveCompare(buckets[i], buckets[j], buckets[k1]) {
                return true
            }
            k2 := (i + j + 1) % numBuckets
            if naiveCompare(buckets[i], buckets[j], buckets[k2]) {
                return true
            }
        }
    }

    return false
}

// Given hash function in Golang
```

```go
func hash(input int, seed int, shift uint) int {
    return ((input * seed) >> (64 - shift)) & ((2 << (shift - 1)) - 1)
}

// Naively check all combinations of a+b==c.
// Runs in O(n^3)
func naiveCompare(A Bucket, B Bucket, C Bucket) bool {
    for i := 0; i < len(A); i++ {
        a := A[i]
        for j := 0; j < len(B); j++ {
            b := B[j]
            for k := 0; k < len(C); k++ {
                if a+b == C[k] {
                    return true
                }
            }
        }
    }
    return false
}

// Reads input as seen on CodeJudge to a slice of ints
func readInput(filename string, amount int) []int {
    ints := make([]int, 0, amount)
    if file, err := os.Open(filename); err == nil {
        defer file.Close()
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
            i, _ := strconv.Atoi(scanner.Text())
            ints = append(ints, i)
        }
    }
    return ints
}

func main() {
    filename := os.Args[1]
    amount, _ := strconv.Atoi(os.Args[2])
    input := readInput(filename, amount)
    if findTriples(input, amount) {
        fmt.Println(1)
    } else {
        fmt.Println(0)
    }
}
```