Last name(s) **Name** **ID**

Midterm EDA exam **Length: 2.5 hours** **20/04/2017**

- *The exam has 4 sheets, 8 sides and 4 problems.*
- *Write your full name and ID on every sheet.*
- *Write your answers to all problems in the exam sheets within the reserved space.*
- *Unless otherwise indicated, all your answers must be justified.*

Problem 1 **(2 points)**

(a) (0.5 pts.) Compute the average cost of insertion sort to sort a vector with n different integers when with probability $\frac{\log n}{n}$ one chooses a vector which is sorted backwards and with probability $1 - \frac{\log n}{n}$ one chooses a sorted vector.

(b) (0.5 pts.) Consider the following function:

```
bool mystery(int n) {
  if (n ≤ 1) return false;
  if (n == 2) return true;
  if (n%2 == 0) return false;
  for (int i = 3; i*i ≤ n; i += 2)
    if (n%i == 0)
      return false;
  return true;
}
```

Complete: function *mystery* computes ______ and its cost is $O($ ______ $)$. Do not justify the answer.

(c) (0.5 pts.) Show that $n! = O(n^n)$ by applying the definition (not using limits).

(d) (0.5 pts.) Show that $n! = \Omega(2^n)$ by applying the definition (not using limits).

Last name(s) **Name** **ID**

Problem 2 **(3 points)**

Given $n \geq 1$, a sequence of n integers $a_0, \ldots, a_{n-1}$ is *unimodal* if there exists t with $0 \leq t < n$ such that $a_0 < \ldots < a_{t-1} < a_t$ and $a_t > a_{t+1} > \ldots > a_{n-1}$. The element a_t is called the *top* of the sequence.

For example, the sequence $1, 3, 5, 9, 4, 1$ is unimodal, and its top is 9 (take $t = 3$).

(a) (1.5 pts.) Implement a function **int** *top*(**const** *vector*<**int**>& *a*) in C++ which, given a non-empty vector a that contains a unimodal sequence, returns the index of the top of the sequence. If you use auxiliary functions that do not belong to the C++ standard library, implement them too. The solution must have cost $\Theta(\log n)$ in time in the worst case. Justify the cost is indeed $\Theta(\log n)$ and give a situation in which this worst case takes place.

(b) (1.5 pts.) Implement a function **bool** *search* (**const** *vector* <**int**>& *a*, **int** *x*) in C++ which, given a non-empty vector *a* that contains a unimodal sequence and an integer *x*, returns whether *x* occurs in the sequence or not. If you use auxiliary functions that do not belong to the C++ standard library, implement them too. The solution must have cost $\Theta(\log n)$ in time in the worst case. Justify the cost is indeed $\Theta(\log n)$ and give a situation in which this worst case takes place.

Last name(s) **Name** **ID**

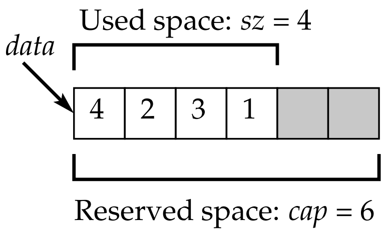
Problem 3 (2 points)

The following class *vect* is a simplification of the class *vector* of the STL:

```
class vect {
  int sz, *data, cap;
  int new_capacity();
  void reserve(int new_cap);

public:
  vect()                         { sz = cap = 0; data = nullptr; }
  int& operator[](int index)     { return data[index];           }
  int size()                     { return sz;                    }
  void push_back(int value)      {
    if (sz ≥ cap) reserve(new_capacity());
    data[sz] = value;
    ++sz;
  } };
```

In an object of class *vect*, field *sz* stores the number of elements that the vector currently contains. The reserved memory for the vector, pointed by field *data*, always has enough space to store the *sz* current elements, and possibly some more. The maximum number of elements that could be stored in the reserved memory is called capacity, and is the value of field *cap*. The diagram on the right illustrates a possible implementation of a vector with contents $(4, 2, 3, 1)$. Note shaded cells are reserved but unused.



The reason for this data structure is that calling the system to ask for more memory is an expensive operation that one does not want to perform often. Function **void** *reserve*(**int** *new_cap*), whose implementation is not detailed here, takes care of that: it asks for a new memory fragment that is big enough to store *new_cap* elements, copies the contents of the old vector there and updates conveniently the fields *sz*, *data* and *cap*.
Observe that every time function *push_back* is called, if there is not enough reserved space, function **int** *new_capacity*() is called which, using the current capacity, determines the new capacity for function *reserve*.

(a) (0.5 pts.) Consider the following implementation of function *new_capacity*:

```
int new_capacity() {
  const int A; // A is a prefixed parameter such that A ≥ 1
  return cap + A; }
```

What is the exact value of *cap* in a vector that initially has *cap* = 0 and to which operation *reserve* has been applied m times? Let us denote this value by $C(m)$.

If we make n calls of *push_back* over a vector which is initially empty ($sz = cap = 0$), how many times **exactly** have we called *reserve* (in other words, which is the value of m such that $n \leq C(m)$ and $C(m-1) < n$)? Give also an asymptotic expression of this number that is as precise and simple as possible.

(b) (0.75 pts.) Show that the solution to the recurrence defined by $C(0) = 0$, $C(m+1) = AC(m) + B$, where $A > 1$ and $B \geq 1$, is $C(m) = \frac{BA^m - B}{A-1}$ for $m \geq 0$.

(c) (0.75 pts.) The same as in exercise (a), but with the following function *new_capacity*:

int *new_capacity* () {
  const int *A*, *B*; // *A*, *B* are prefixed parameters such that $A > 1$, $B \geq 1$
  return *A*∗*cap* + *B*; }

Last name(s) **Name** **ID**

Problem 4 **(3 points)**

We want to have a function

$$\textbf{int} \quad \textit{stable\_partition} \ \ (\textbf{int} \ x, \ \ \textit{vector}<\textbf{int}>\& \ a)$$

which, given an integer x and a vector of n different integers $a = (a_0, a_1, \ldots, a_{n-1})$, reorders the contents of the vector so that all elements of the subvector $a[0..k]$ are less than or equal to x, and all elements of the subvector $a[k+1..n-1]$ are greater than x, and also returns the index k ($k = -1$ if all elements of a are greater than x). Moreover, the original relative order of the elements of a is respected:

- if $a[i]$ and $a[j]$ with $i < j$ are both less than or equal to x, then in the final vector $a[i]$ will occur before $a[j]$, and both will be placed in the "part" $a[0..k]$ that contains the elements $\leq x$;
- if $a[i]$ and $a[j]$ with $i < j$ are both greater than x, then in the final vector $a[i]$ will occur before $a[j]$, and both will be placed in the "part" $a[k+1..n-1]$ that contains the elements $> x$.

For example, given $x = 2$ and the sequence $a = (1, 5, 3, 0, 4)$, we would like that the function updated a to $(1, 0, 5, 3, 4)$, and that it returned $k = 1$.

(a) (1 pt.) Implement the function *stable_partition* in C++. The cost in time must be $\Theta(n)$. Justify the cost. What is the cost in space of the auxiliary memory that your implementation is using?

(b) (0.5 pts.) What does the next function *mystery* do? Do not justify the answer.

```
void mystery_aux(vector<int>& a, int l, int r) {
  // Pre: 0 ≤ l ≤ r < a.size()
  for (int i = l, j = r; i < j; ++i, --j) swap(a[i], a[j]);
}

void mystery(vector<int>& a, int l, int p, int r) {
  // Pre: 0 ≤ l ≤ p ≤ r < a.size()
  mystery_aux(a, l, p);
  mystery_aux(a, p+1, r);
  mystery_aux(a, l, r);
}
```

(c) (1.5 pts.) Fill the gaps in the following alternative implementation of *stable_partition* and analyze its cost in time in the worst case. Assume that, in the worst case, at each recursive call of *stable_partition_rec* we have that $q - p = \Theta(r - l + 1)$.

```
int stable_partition (int x, vector<int>& a) {
  return stable_partition_rec (x, a, 0, a.size()−1);
}

int stable_partition_rec (int x, vector<int>& a, int l, int r) {
  if (l == r) {
    if (a[l] ≤ x) return l;
    else        return [          ] ;
  }
  int m = (l+r)/2;
  int p = stable_partition_rec (x, a, l, m);
  int q = stable_partition_rec (x, a, [          ] , r);
  mystery(a, [          ] , m, [          ] );
  return [          ] ; }
```

Analysis of the cost: