DATA STRUCTURES AND ALGORITHMS COLLECTION OF EXAMS

Albert Atserias

Amalia Duch

Albert Oliveras

Enric Rodríguez Carbonell

(Editors)

5 de febrer de 2021

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Taula de continguts

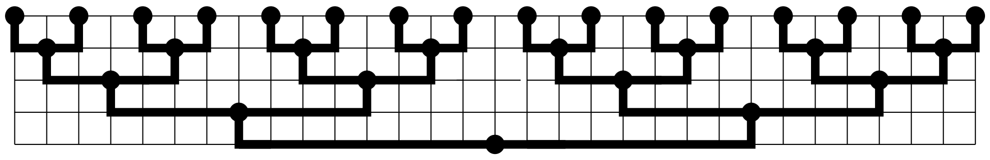
1

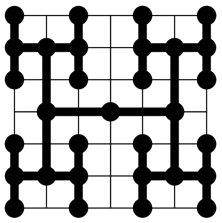
Mid Term Exams

Midterm EDA **Length: 2 hours** **24/3/2014**

Problem 1 **(3 points)**

As graphs, all trees are planar. In particular this means that they are all representable on a sufficiently large rectangular grid in such a way that the edges do not intersect (except at the vertices, obviously). Here are two planar representations of a full binary tree with 16 leaves:





(1 point) Following the recursive pattern of the **first** representation, give recurrences for the height $H(n)$ and width $W(n)$, respectively, of the grid that is needed to represent a full binary tree with 2^n leaves in this representation.

(1 point) Following the recursive pattern of the **second** representation, give the recurrence for the length $L(n)$ of the square grid that is needed to represent a full binary tree with 4^n leaves in this representation.

(1 point) If your goal is to minimize the area of the grid for a full binary tree with 4^n leaves, explain which of the two representations you will choose (to relate both representations note that $4^n = 2^{2n}$). You do not need to solve the recurrences exactly; an asymptotic estimation is enough.

Problem 2 **(2 points)**

Consider the following four pieces of code:

```
void f1 (int n) {
    int x = 2;
    int y = 1;
    while (y ≤ n) {
        y = y + x;
        x = x + 1;
}   }

void f2 (int n) {
    int x = 2;
    int y = 1;
    while (y ≤ n) {
        y = y + x;
        x = 2*x;
}   }

void f3 (int n) {
    int x = 2;
    int y = 1;
    while (y ≤ n) {
        y = y*x;
        x = 2*x;
}   }

void f4 (int n) {
    int x = 2;
    int y = 1;
    while (y ≤ n) {
        y = y*x;
        x = x*x;
}   }
```

What is the cost of $f1$? Answer: $\Theta(\boxed{})$

What is the cost of $f2$? Answer: $\Theta(\boxed{})$

What is the cost of $f3$? Answer: $\Theta(\boxed{})$

What is the cost of $f4$? Answer: $\Theta(\boxed{})$

A correct answer without proper reasoning **will not get any points**.

(0,5 points) Reasoning for $f1$:

(0,5 points) Reasoning for $f2$:

(0,5 points) Reasoning for $f3$:

(0,5 points) Reasoning for $f4$:

Problem 3 **(3 points)**

An *inversion* in a vector of integers $T[0 \ldots n-1]$ is a pair of positions storing values that are not ordered, that is, a pair (i, j) such that $0 \leq i < j < n$ and $T[i] > T[j]$.

(1.5 points) Let $T[0 \ldots n-1]$ be a vector of integers. Show that if there is an inversion (i, j) in T, then T has at least $j - i$ inversions.

(1.5 points) Let us fix a value N such that $N \geq 0$ (that is, N is a constant in this exercise). By using Divide & Conquer, implement in C++ a function

bool *search* (**int** x, **const vector**<**int**>& T)

which, given an integer x and a vector of integers T of size *n with at most N inversions*, determines if x is in T. The function should have worst-case cost $\Theta(\log n)$. Show that indeed it has this cost.

Hint: to solve question 2, use the fact from question 1. Besides, deal with subvectors of size $\leq 2N$ in the base case of the recursion, and with the rest of subvectors in the recursive case.

Problem 4 **(2 points)**

Four quickies:

1. Let $f(n) = n \log(\cos(n\pi) + 4))$. Then $f(n) = \Theta(\boxed{})$.
2. Which is the last digit of 7^{1024} written in decimal notation? Answer: $\boxed{}$.
3. What is the **recurrence** for the cost of Strassen's algorithm to multiply $n \times n$ matrices?
   Answer $T(n) = \boxed{}$.
4. An algorithm admits all 2^n vectors of n bits as possible inputs. On $2^n - 1$ of these inputs the cost of the algorithm is $\Theta(n^2)$, and on the remaining one the cost is $\Theta(n^4)$. Therefore, its worst-case cost is $\Theta(n^4)$ and its best-case cost is $\Theta(n^2)$. What is its average-case cost when the input is chosen uniformly at random (so each input has probability $1/2^n$)?
   Answer: $T(n) = \Theta(\boxed{})$.

Reasoning for questions 1, 2 and 4 (reasoning for question 3 is **not** required):

Midterm EDA **Length: 2.5 hours** **13/10/2014**

Problem 1 **(2 points)**

(1 point) Prove that $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

Hint: There are several ways to do it, one of them by proving O and Ω separately.

(1 punt) Let $f(n) = 2^{\sqrt{\log n}}$ and $g(n) = n$. Asymptotically, which one grows faster? Prove it.

Problem 2 **(3 points)**
Consider the procedure *mystery* which takes as inputs a vector A and a positive integer k such that all elements of A are between 0 and k, both included.

```
void mystery(const vector<int>& A, vector<int>& B, int k){
    int n = A.size ();
    vector<int> C(k+1, 0);
    for (int j = 0; j < n; ++j) ++C[A[j]];
    for (int i = 1; i ≤ k; ++i) C[i] += C[i−1];
    B = vector<int>(n);
    for (int j = n−1; j ≥ 0; −−j){
        B[C[A[j]] − 1] = A[j];
        −−C[A[j]];
    }
}
```

(a) (1 point) What is the content of B at the end of the execution of *mystery*?

(b) (1 point) If n is the length of the vector A and you are guaranteed that $k = O(n)$, what is the asymptotic cost of *mystery* as a function of n?

(c) (1 point) Without writing any code, describe an algorithm for the following problem: given a positive integer k, a vector A of n integers between 0 and k, both endpoints included, and a vector P of m pairs of integers $(a_0, b_0), \ldots, (a_{m-1}, b_{m-1})$, with $0 \le a_i \le b_i \le k$ for each $i = 0, \ldots, m-1$, you are asked to write the number of elements of A which are in the interval $[a_i, b_i]$ for $i = 0, \ldots, m-1$ in time $\Theta(m+n+k)$, and in particular time $\Theta(m+n)$ when $k = O(n)$.

Hint: using a vector C like the one in *mystery* will be useful.

Problem 3 **(3 points)**
A board game of chance has 64 possible positions $1, \dots, 64$. The rules of the game are such that, from any position $i \in \{1, \dots, 64\}$, in one step a checker will hop to any other position $j \in \{1, \dots, 64\}$ with probability $P_{i,j}$. Let $P = (P_{i,j})_{1 \le i,j \le 64}$ be this matrix of probabilities.

(a) (1 point) Remember that if $R = P^2$, then $R_{i,j} = \sum_{k=1}^{64} P_{i,k} P_{k,j}$ for each i, j. Taking into account that $P_{i,j}$ is the probability of hopping from i to j in one step, how do you interpret $R_{i,j}$?

(b) (2 points) Design an algorithm that, given a matrix of probabilities $P = (P_{i,j})_{1 \le i,j \le 64}$ and given a certain number of steps $t \ge 0$, computes the matrix $(Q_{i,j})_{1 \le i,j \le 64}$ where each $Q_{i,j}$ is the probability that the checker, starting at position i, ends at position j after exactly t steps.

```
typedef vector<double> Row;
typedef vector<Row> Matrix;

void probabilities (const Matrix& P, int t, Matrix& Q) {
```

}

Note 1: In order to get full grade your algorithm must run in $\Theta(\log t)$ time.
Note 2: If you need any auxiliary function, implement on the back.
Note 3: Justify the cost of your algorithm on the back.

Auxiliary functions (if needed):

Cost of the algorithm:

Problem 4 **(2 points)**

(a) (1 point) Assume that an algorithm receives as input any of the 2^n vectors of n bits with equal probability. Assume that the worst-case cost of the algorithm is $\Theta(n^2)$, and that its best-case cost is $\Theta(n)$. How many inputs are required to cause cost $\Theta(n^2)$ to be sure that the average-case cost of the algorithm is $\Theta(n^2)$?

Answer: $\Omega($ $\boxed{}$ $)$
Justification:

(b) (0.5 points) Consider the following algorithm that adds one unit to a natural number that comes represented by a vector **vector**<**int**> A of n decimal digits:

```
int i = n − 1;
while (i ≥ 0 and A[i] == 9) { A[i] = 0; −−i; }
if (i ≥ 0) ++A[i]; else cout << ``Overflow'' << endl;
```

If every digit $A[i]$ of the input is equally likely and independent of the rest (that is, each $A[i]$ is one of the 10 possible digits with probability $1/10$ independently of the other digits), what is the probability that this algorithm makes exactly k iterations when $0 \leq k \leq n-1$?

Answer: $\boxed{}$
Justification:

(c) (0.5 points) Use the master theorem for subtractive recurrences to solve the recurrence $T(n) = \frac{9}{10}T(n-1) + \Theta(1)$ with base case $T(0) = \Theta(1)$.

Answer: $T(n) = \Theta($ $\boxed{}$ $)$
Justification:

———————— END OF THE EXAM —————————-

(d) (**optional** extra bonus: 1 additional point to the grade) What does the recurrence $T(n)$ from section (c) compute about the algorithm from section (b)?

Midterm EDA Length: 2h30m 23/3/2015

Problem 1: Cost analysis (2 points)

The sieve of Eratostenes (276–194 BC) is a method to generate all prime numbers smaller or equal than a given n. The method goes as follows: traversing the sequence of numbers $2, 3, 4, \ldots, n$, look for the next number x that is not yet marked, mark all its multiples $2x, 3x, 4x, \ldots$ up to n, and repeat with the next x. When it finishes, those $x \geq 2$ that are not marked are the prime numbers. In C++:

```
vector<bool> M(n + 1, false);
for (int x = 2; x ≤ n; ++x) {
    if (not M[x]) {
        for (int y = 2∗x; y ≤ n; y += x) M[y] = true;
    }
}
```

For the first three questions below we ask for an exact (non-asymptotic) expression *as a function of n*. If needed, use the notation $\lfloor z \rfloor$ that rounds z to the maximum integer smaller or equal than z; for example, $\lfloor \pi \rfloor = \lfloor 3.14\ldots \rfloor = 3$.

(a) (0.33 points) How many times is $M[y]$ = **true** executed when x is 2?
Answer: Exactly ( ) many times.

(b) (0.34 points) How many times is $M[y]$ = **true** executed when x is 15?
Answer: Exactly ( ) many times.

(c) (0.33 points) How many times is $M[y]$ = **true** executed when x is 17?
Answer: Exactly ( ) many times.

Note: More questions on the back.

(d) (0.5 points) It is known that

$$\sum_{\substack{p=2 \\ p \text{ prime}}}^{n} \frac{1}{p} = \Theta(\log \log n).$$

Use this, in conjunction with the answers to the previous questions, to determine the cost of the algorithm as a function of n, in asymptotic notation. Answer: $\Theta($ ( ) $)$.

Justification:

(e) (0.5 points) An improvement consists in replacing the condition $x \leq n$ in the external loop by $x{*}x \leq n$. Would this improve the asymptotic cost?

Answer and justification:

Problem 2: Strassen & company **(2 points)**

The school algorithm to multiply two $n \times n$ matrices makes $\Theta(n^3)$ arithmetic operations. In 1969 Strassen found an algorithm that makes $\Theta(n^{2.81})$ arithmetic operations. Twenty one years later, Coppersmith and Winograd discovered a method that makes $\Theta(n^{2.38})$ operations.

Assuming (for simplicity) that the implicit constants in the Θ notation are 1, 10 and 100, respectively, and that they apply to every $n \geq 1$ (that is, the costs are n^3, $10n^{2.81}$ and $100n^{2.38}$, respectively, for every $n \geq 1$), compute the least n for which one of these algorithms makes less operations than another.

(a) (1 point) For $n \geq$ ☐ , Strassen improves the school method.

(b) (1 punt) For $n \geq$ ☐ , Coppersmith-Winograd improves Strassen.

Justifications:

Problem 3: Algorithm design **(3 points)**

Given a sequence of n non-empty intervals $[a_1, b_1], \ldots, [a_n, b_n]$, we want to compute their uni-on in time $O(n \log n)$. The output is represented by a sequence of disjoint intervals sorted by their left boundary. For example, if the intervals in the input were [17,19] [−3,7] [4,9] [18,21] [−4,15], then the intervals in the output would be [−4,15] [17,21].

(a) (1 point) Describe an algorithm that solves this problem. Explain the algorithm in words, without writing code, but clearly enough so that the algorithm can be implemented. Assume that the input is given by the vectors $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$, with $a_i \leq b_i$ for every $i = 1, \ldots, n$.

(b) (1 point) Now, in addition to the sequence of n intervals, we are given a sequence m different reals $p_1, \ldots, p_m$, and we want to determine how many fall in some interval of the union (only the number is needed; not which ones). Using the algorithm from the previous section, describe an algorithm that solves this problem in time $O(n \log n)$ when $m = n$. Assume that the input is given by the vectors a and b from the previous section, and the vector $(p_1, \ldots, p_m)$ with $p_i \neq p_j$ if $i \neq j$.

(c) (1 punt) If you happened to know that m is bounded by a small constant independent of n, say $m \leq 5$, would you still use the same algorithm? If not, which one would you use instead? If you choose a different algorithm, make its cost explicit.

Problem 4: Quickies **(3 points)**

- (0,5 points) Determine if they are equal ($=$) or different ($\neq$), and prove it:

$$\Theta(3^{\log_2(n)}) \ \boxed{} \ \Theta(3^{\log_4(n)}).$$

- (0,5 points) Compute $2^1 \cdot 2^2 \cdot 2^3 \cdot \cdots \cdot 2^{99} \cdot 2^{100} \mod 9$. This problem is not meant to be solved with the help of a calculator.

- (1 point) Sort the functions that follow increasingly according to their asymptotic growth rates: $n^4 - 3n^3 + 1, (\ln(n))^2, \sqrt{n}, n^{1/3}$. Exceptionally, you are not required to justify your answer.

- (1 point) Consider the following three alternatives for solving a problem:
  - A: divide an instance of size n into five instances of size $n/2$, solve each instance recursively, and combine the solutions in time $\Theta(n)$.

- B: given an intance of size n, solve two instances of size $n-1$ recursively, and combine the solutions in constant time.
- C: divide an instance of size n into nine instances of size $n/3$, solve each instance recursively, and combine the solutions in time $\Theta(n^2)$.

Write down and solve the corresponding recurrences. Which alternative is the most efficient?

Midterm EDA Exam **Length: 2.5 hours** **19/10/2015**

Problem 1 **(3.5 points)**

Fibonacci numbers are defined by the recurrence $f_k = f_{k-1} + f_{k-2}$ for $k \geq 2$, with $f_0 = 0$ and $f_1 = 1$. Answer the following exercises:

(a) (0.5 points) Consider the following function *fib1* which, given a non-negative integer k, returns f_k:

```
int fib1 (int k) {
  vector<int> f(k+1);
  f[0] = 0;
  f[1] = 1;
  for (int i = 2; i ≤ k; ++i)
    f[i] = f[i−1] + f[i−2];
  return f[k];
}
```

Describe the asymptotic cost *in time and space* of *fib1* (k) as a function of k as precisely as possible.

(b) (1 point) Show that, for $k \geq 2$, the following matrix identity holds:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k-1} = \begin{pmatrix} f_k & f_{k-1} \\ f_{k-1} & f_{k-2} \end{pmatrix}$$

(c) (1 point) Fill the gaps in the following code so that function *fib2* (k) computes f_k, given a $k \geq 0$.

```
typedef vector<vector<int>> matrix;

matrix mult(const matrix& A, const matrix& B) {
   // Pre: A and B are square matrices of the same dimensions
   int n = A.size ();
   matrix C(n, vector<int>(n, 0));
   for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
         for (int k = 0; k < n; ++k)
            [                                   ]
   return C;
}


matrix mystery(const matrix& M, int q) {
   int s = M.size ();
   if (q == 0) {
      matrix R(s, vector<int>(s, 0));
      for (int i = 0; i < s; ++i) [                  ]
      return R;
   }
   else {
      matrix P = mystery(M, q/2);
      if ( [                  ] ) return mult(P, P);
```

```
    else return [          ] ;
} }

int fib2 (int k) {
  if (k ≤ 1) return k;
  matrix M = { {1, 1}, {1, 0} };
  matrix P = mystery(M, k−1);
  return [          ] ;
}
```

(d) (1 point) Describe the asymptotic cost in time of *fib2* (k) as a function of k as precisely as possible.

Problem 2 **(3.25 points)**

Given a vector of integers v and an integer x, the function

```
int position (const vector<int>& v, int x) {
  int n = v.size ();
  for (int i = 0; i < n; ++i)
    if (v[i] == x)
      return i;
  return −1;
}
```

examines the $n = v.size()$ positions of v and returns the first one that contains x, or -1 if there is none.

(a) (0.75 points) Describe as a function of n the asymptotic cost in time of *position* in the best case as precisely as possible. When can this best case take place?

(b) (0.75 points) Describe as a function of n the asymptotic cost in time of *position* in the worst case as precisely as possible. When can this worst case take place?

(c) (0.75 points) Show that for any integer $n \geq 1$, the following identity holds:

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n}{2^n} - \frac{1}{2^{n-1}}.$$

(d) (1 point) Assume that we have a probability distribution over the input parameters. Namely, the probability that x is element $v[i]$ is $\frac{1}{2^{i+1}}$ for $0 \leq i < n-1$, and that it is element $v[n-1]$ is $\frac{1}{2^{n-1}}$. In particular, these probabilities add up 1, so that x is always one of the n values of v.

Describe as a function of n the asymptotic cost in time of *position* in the average case as precisely as possible.

Problem 3 **(3.25 points)**

In this exercise we tackle the problem of, given two positive integers a and b, to compute their greatest common divisor $\gcd(a,b)$. Recall that $\gcd(a,b)$ is, by definition, the only positive integer g such that:

1. $g \mid a$ (g divides a),
2. $g \mid b$,
3. if $d \mid a$ and $d \mid b$, then $d \mid g$.

(a) (1.25 points) Show that the following identities are true:

$$\gcd(a,b) = \begin{cases} 2\gcd(a/2,b/2) & \text{if } a,b \text{ are even} \\ \gcd(a,b/2) & \text{if } a \text{ is odd and } b \text{ is even} \\ \gcd((a-b)/2,b) & \text{if } a,b \text{ are odd and } a > b \end{cases}$$

Hint: you can use that, given two positive integers a and b such that $a > b$, we have $\gcd(a,b) = \gcd(a-b,b)$.

(b) (1 point) Write a function **int** gcd(**int** a, **int** b) in C++ which, by using divide and conquer and exercise (a), computes the greatest common divisor $\gcd(a,b)$ of two given positive integer numbers a, b.

Hint: you can also use that for any positive integer a, $\gcd(a,a) = a$.

(c) (1 point) Assuming that a and b are positive integers, each of which represented with a vector of n bits, describe the cost in time in the worst case of $gcd(a,\ b)$ as a function of n as precisely as possible. When can this worst case take place?

Assume that the cost of the following operations with integers of n bits: adding, subtracting, comparing, multiplying/dividing by 2 is $\Theta(n)$, and that computing the remainder modulo 2 takes time $\Theta(1)$.

Midterm EDA **Length: 2.5 hours** **31/03/2016**

Problem 1 **(1 point)**

Answer the following questions. In this exercise, you do **not** need to justify the answers.

1. (0.2 points) The cost of Strassen's algorithm for multiplying two matrices of size $n \times n$ is $\Theta(\boxed{})$

2. (0.6 points) The master theorem for subtractive recurrences states that given a recurrence $T(n) = aT(n-c) + \Theta(n^k)$ with $a, c > 0$ and $k \geq 0$ then:

$$T(n) = \begin{cases} \boxed{} & \text{if } a < 1, \\ \boxed{} & \text{if } a = 1, \\ \boxed{} & \text{if } a > 1. \end{cases}$$

3. (0.2 points) The mergesort sorting algorithm uses $\Theta(\boxed{})$ auxiliary space when sorting a vector of size n.

Problem 2 **(3 points)**

In this problem costs are only considered *in time*. Consider the following program:

```
void f(int m);
void g(int m);

void h(int n) {
  int p = 1;
  for (int i = 1; i ≤ n; i++) {
    f(i);
    if (i == p) {
      g(n);
      p *= 2;
} } }
```

(a) (1.5 points) Answer: if the cost of $f(m)$ as well as the cost of $g(m)$ is $\Theta(m)$, then the cost of $h(n)$ as a function of n is $\Theta(\boxed{})$

Justification:

(b) (1.5 points) Answer: if the cost of $f(m)$ is $\Theta(1)$ and the cost of $g(m)$ is $\Theta(m^2)$, then the cost of $h(n)$ as a function of n is $\Theta(\boxed{})$

Justification:

Problem 3 **(3 points)**

We say that a square matrix M of integer numbers of size $n \times n$ is *symmetric* if $M_{i,j} = M_{j,i}$ for all i, j such that $0 \leq i, j < n$.

Let us consider the following implementation of symmetric matrices with unidimensional vectors, which only stores the "lower triangle" to minimise the consumed space:

$$\begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & \dots & M_{0,n-1} \\ M_{1,0} & M_{1,1} & M_{1,2} & \dots & M_{1,n-1} \\ M_{2,0} & M_{2,1} & M_{2,2} & \dots & M_{2,n-1} \\ \vdots & & \ddots & & \\ M_{n-1,0} & M_{n-1,1} & M_{n-1,2} & \dots & M_{n-1,n-1} \end{pmatrix}$$

$$\Downarrow$$

$M_{0,0}$	$M_{1,0}$	$M_{1,1}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$\dots$	$M_{n-1,0}$	$M_{n-1,1}$	$M_{n-1,2}$	$\dots$	$M_{n-1,n-1}$

For example, the symmetric matrix 3×3

$$\begin{pmatrix} 1 & 2 & 0 \\ 2 & -2 & 3 \\ 0 & 3 & -1 \end{pmatrix}$$

would be implemented with the unidimensional vector

1	2	−2	0	3	−1

(a) (0.5 points) Answer: the cost in space of this representation for a symmetric matrix $n \times n$ as a function of n is $\Theta(\ \boxed{}\ )$.

Justification:

(b) (1 point) Given $n > 0$ and a value k, the function

pair<**int**,**int**> *row_column*(**int** n, **int** k);

j is the column of the coefficient pointed by k in a unidimensional vector that implements a symmetric matrix $n \times n$. If k is not a valid index, it returns $(-1,-1)$. For example, *row_column*(3,2) returns $(1,1)$, *row_column*(3,3) returns $(2,0)$, and *row_column*(3,6) returns $(-1,-1)$.

Complete the following implementation of *row_column*:

```
int p(int x) { return                ;}

int mystery(int k, int l, int r) {
  if (l+1 == r) return l;
  int m = (l+r)/2;
  if (p(m) ≤ k) return mystery(k,                 , r);
  else          return mystery(k, l,                 );
}

pair<int,int> row_column(int n, int k) {
  if (k < 0 or k ≥ p(n)) return                 ;
  int i = mystery(k, 0, n);
  return {i,                 };
}
```

(c) (1 point) Analyse the cost in time in the worst case of the implementation of *row_column*(**int** n, **int** k) of the previous exercise as a function of n.

(d) (0.5 points) Using the functions **double** *sqrt*(**double** *x*) and **int** *floor* (**double** *x*) which respectively compute $\sqrt{x}$ and $\lfloor x \rfloor$ in time $\Theta(1)$, give an alternative implementation of *row_column* with cost $\Theta(1)$.

Problem 4 **(3 points)**

Assume that n is a power of 2, that is, of the form 2^k for a certain $k \geq 0$.

A square matrix A of real numbers of size $n \times n$ is a *Monge matrix* if for all i, j, k and l such that $0 \leq i < k < n$ and $0 \leq j < l < n$, it holds that

$$A_{i,j} + A_{k,l} \leq A_{i,l} + A_{k,j}$$

In other words, whenever we take two rows and two columns of a Monge matrix and consider the four elements at the intersections of the rows and the columns, the sum of the elements at the upper left and lower right corners is less than or equal to the sum of the elements at the upper right and lower left corners. For example, the following matrix is a Monge matrix:

$$\begin{pmatrix} 10 & 17 & 13 & 28 \\ 16 & 22 & 16 & 29 \\ 24 & 28 & 22 & 34 \\ 11 & 13 & 6 & 6 \end{pmatrix}$$

(a) (1 point) Let $f_A(i)$ be the index of the column where the the minimum element of the i-th row appears (breaking ties if needed by taking the leftmost one). For example, in the matrix above $f_A(0) = 0$, $f_A(1) = 0$, $f_A(2) = 2$ i $f_A(3) = 2$.

Show that $f_A(0) \leq f_A(1) \leq \ldots \leq f_A(n-1)$ for any Monge matrix A of size $n \times n$.

(b) (1 point) In what follows a divide-and-conquer algorithm is described which computes the function f_A for all rows of a Monge matrix A:

(1) Build two square submatrices B_1 i B_2 of the matrix A of sizes $\frac{n}{2} \times \frac{n}{2}$ as follows: B_1 is formed by the rows of A with even index and the columns between 0 and $\frac{n}{2} - 1$, and B_2 is formed by the rows of A with even index and the columns between $\frac{n}{2}$ and $n - 1$.

(2) Recursively determine the column where the leftmost minimum appears for any row of B_1 and for any row of B_2.

(3) Find the column where the leftmost minimum appears for any row of A.

Explain how, by using the result of (2), one can accomplish (3) in time $\Theta(n)$.

(c) (1 point) Compute the cost as a function of n of the algorithm proposed in the previous exercise for computing the function f_A for all rows of a Monge matrix A of size $n \times n$. Assume that step (1) can be carried out in time $\Theta(n)$.

Midterm EDA exam **Length: 2.5 hours** **07/11/2016**

Problem 1 **(2 points)**

In this problem you do not need to justify your answers.

(a) (0.8 pts.) Fill the gaps in the following table (except for the cell marked with **Do not fill**) with the costs in time for sorting a vector of integers of size n using the indicated algorithms. Assume uniform probability in the average case.

	Best case	*Average case*	*Worst case*
Quicksort (with Hoare's partition)			
Mergesort			
Insertion		**Do not fill**	

(b) (0.2 pts.) The solution to the recurrence $T(n) = 2T(n/4) + \Theta(1)$ is

(c) (0.2 pts.) The solution to the recurrence $T(n) = 2T(n/4) + \Theta(\sqrt{n})$ is

(d) (0.2 pts.) The solution to the recurrence $T(n) = 2T(n/4) + \Theta(n)$ is

(e) (0.3 pts.) What does Karatsuba's algorithm compute? What is its cost?

(f) (0.3 pts.) What does Strassen's algorithm compute? What is its cost?

Problem 2 **(3 points)**

Given $n \geq 2$, we say that a sequence of n integers $a_0, \ldots, a_{n-1}$ is *bi-increasing* if $a_{n-1} < a_0$ and there exists an index t (with $0 \leq t < n$) that satisfies the following conditions:

- $a_0 \leq \ldots \leq a_{t-1} \leq a_t$
- $a_{t+1} \leq a_{t+2} \leq \ldots \leq a_{n-1}$

For example, the sequence $12, 12, 15, 20, 1, 3, 3, 5, 9$ is bi-increasing (take $t = 3$).

(a) (2 pts.) Implement in C++ a function

bool *search* (**const vector**<**int**>& *a*, **int** *x*);

which, given a vector a that contains a bi-increasing sequence and an integer x, returns whether x appears in the sequence or not. If you use auxiliary functions that are not part of the C++ standard library, implement them too. The solution must have cost $\Theta(\log(n))$ in time in the worst case.

(b) (1 pt.) Justify that the cost in time in the worst case of your function *search* is $\Theta(\log(n))$. When does this worst case take place?

Problem 3 **(2 points)**

Consider the following function:

```
int mystery(int m, int n) {
  int p = 1;
  int x = m;
  int y = n;
  while (y ≠ 0) {
    if (y % 2 == 0) {
      x *= x;
      y /= 2;
    }
    else {
      y −= 1;
      p *= x;
  } }
  return p;
}
```

(a) (1 pt.) Given two integers $m,n \geq 0$, what does *mystery*(m,n) compute? You do not need to justify your answer.

(b) (1 pt.) Analyse the cost in time in the worst case as a function of n of *mystery*(m, n).

Problem 4 **(3 points)**

The *Fibonacci sequence* is defined by the recurrence $F(0) = 0$, $F(1) = 1$ and

$$F(n) = F(n-1) + F(n-2) \quad \text{if } n \geq 2.$$

(a) (0.5 pts.) Let $\phi = \frac{\sqrt{5}+1}{2}$ be the so-called *golden number*. Prove that $\phi^{-1} = \phi - 1$.

(b) (1.5 pts.) Prove that, for any $n \geq 0$, we have

$$F(n) = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

(c) (1 pt.) Prove that $F(n) = \Theta(\phi^n)$.

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

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 $\boxed{}$ and its cost is $O(\boxed{})$. 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).

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.

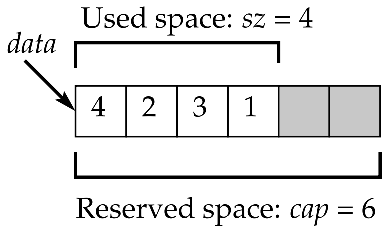
(b) (1.5 pts.) Implement a function **bool** *search* (**const vector**<**int**>& a, **int** x) in C++ which, given a non-empty vector a that contains a unimodal sequence and an integer x, returns whether x occurs in the sequence or not. If you use auxiliary functions that do not belong to the C++ standard 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.

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 ≥ 1
    return A*cap + B; }
```

Problem 4 **(3 points)**

We want to have a function

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

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:

Midterm EDA Exam **Length: 2.5 hours** **06/11/2017**

Problem 1 (3 points)

Answer the following questions. You do not need to justify your answers.

(a) (0.5 pts.) The cost in time of the following code snippet as a function of n:

```
int j = 0;
int s = 0;
for (int i = 0; i < n; ++i)
  if (i == j*j) {
    for (int k = 0; k < n; ++k) ++s;
    ++j;
  }
```

is $\Theta(\boxed{})$.

(b) (1 pt.) Given a vector of integers v and an integer x, the function

```
int position (const vector<int>& v, int x) {
  int n = v.size ();
  for (int i = 0; i < n; ++i)
    if (v[i] == x)
      return i;
  return −1;
}
```

examines the $n = v.size()$ positions of v and returns the first one that contains x, or -1 if there is none.

Assume that x occurs in vector v. The asymptotic cost in time of *position* in the worst case is $\Theta(\boxed{})$, and in the average case (assuming uniform probability) is $\Theta(\boxed{})$.

(c) (0.5 pts.) Give the order of magnitude of the following function in its simplest form: $5(\log n)^2 + 2\sqrt{n} + \cos(n^8) = \Theta(\boxed{})$.

(d) (0.5 pts.) The solution to the recurrence

$$T(n) = \begin{cases} 1 & \text{if } 0 \le n < 2 \\ 4 \cdot T(n/2) + 3n^2 + 2\log(\log n) + 1, & \text{if } n \ge 2 \end{cases}$$

is $T(n) = \Theta(\boxed{})$.

(e) (0.5 pts.) The solution to the recurrence

$$T(n) = \begin{cases} 1 & \text{if } 0 \le n < 2 \\ 4 \cdot T(n-2) + 3n^2 + 2\log(\log n) + 1, & \text{if } n \ge 2 \end{cases}$$

is $T(n) = \Theta(\boxed{})$.

Problem 2 **(2.5 points)**

Consider the following program, which reads a strictly positive integer m and a sequence of n integers $a_0, a_1, \ldots, a_{n-1}$ that are guaranteed to be between 1 and m:

```
int main() {

  int m;
  cin >> m;
  vector<int> a;
  int x;
  while (cin >> x)
    a.push_back(x);

  vector<int> b(m + 1, 0);
  int n = a.size ();
  for (int i = 0; i < n; ++i)
    ++b[ a[i] ];

  for (int j = 1; j ≤ m; ++j)
    b[j] += b[j−1];
}
```

(a) (0.5 pts.) Given y such that $0 \le y \le m$, what is the value of $b[y]$ at the end of *main* in terms of the sequence a?

(b) (1 pt.) Let us define the *median* of the sequence a as the value p between 1 and m such that $b[p] \ge \frac{n}{2}$ and $b[p-1] < \frac{n}{2}$, where b is the vector computed as in the code above.

Write in C++ a function

int *median*(**int** n, **const vector**<**int**>& b);

which, given n, the size of the sequence a, and b, computed as above, finds the median of a. Solutions with cost $\Omega(m)$ will not be considered as valid answers. If you use auxiliary functions, implement them too.

(c) (1 pt.) Analyze the cost of your function *median* of the previous exercise as a function of m.

Problem 3 **(2 points)**

Consider the type

```
struct complex {
  int real ;
  int imag;
};
```

for implementing complex numbers (with integer components), where *real* is the real part and *imag* is the imaginary part of the complex number.

For example, if z is an object of type *complex* that represents the number $2 + 3i$, then $z.real$ = 2 and $z.imag$ = 3.

(a) (1 pt.) Implement in C++ a function

```
complex exp(complex z, int n);
```

which, given a complex number z and an integer $n \geq 0$, computes the complex number z^n. The solution must have cost $\Theta(\log n)$ in time. If you use auxiliary functions, implement them too.

Note. Recall that the product of complex numbers is defined as follows:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

(b) (1 pt.) Justify that the cost in time of your function *exp* is $\Theta(\log n)$.

Problem 4 **(2.5 points)**

Consider the following recurrence:

$$T(n) = T(n/2) + \log n$$

(a) (1 pt.) Let us define the function U as $U(m) = T(2^m)$. Using the recurrence for T, deduce a recurrence for U.

(b) (0.5 pts.) Solve asymptotically the recurrence for U of your answer to the previous exercise.

(c) (1 pt.) Solve asymptotically the recurrence for T.

Midterm EDA exam **Length: 2.5 hours** **19/04/2018**

Problem 1 **(1.5 points)**

(a) (0.5 pts.) Consider the functions $f(n) = n^{n^2}$ and $g(n) = 2^{2^n}$.

Which of the two functions grows asymptotically faster? $\boxed{}$

Justification:

Note: Read n^{n^2} as $n^{(n^2)}$, and similarly 2^{2^n} as $2^{(2^n)}$.

(b) (0.5 pts.) Assume that the inputs of size n of a certain algorithm are of the following types:

- Type 1: for each input of this type, the algorithm takes time $\Theta(n^4)$. Moreover, the probability that the input is of this type is $\frac{1}{n^3}$.
- Type 2: for each input of this type, the algorithm takes time $\Theta(n^3)$. Moreover, the probability that the input is of this type is $\frac{1}{n}$.
- Type 3: for each input of this type, the algorithm takes time $\Theta(n)$. Moreover, the probability that the input is of this type is $1 - \frac{1}{n^3} - \frac{1}{n}$.

Then the cost of the algorithm in the average case is $\boxed{}$.

Justification:

(c) (0.5 pts.) Solve the recurrence $T(n) = 2T(n/4) + \Theta(\sqrt{n})$.

Answer: $T(n) = \Theta(\boxed{})$.

Justification:

Problem 2 **(2 points)**

In this problem we take n as a constant. Given a vector $x_0 \in \mathbb{Z}^n$ and a square matrix $A \in \mathbb{Z}^{n\times n}$, we define the sequence $x(0), x(1), x(2), \ldots$ as:

$$x(k) = \begin{cases} x_0 & \text{if } k = 0 \\ A \cdot x(k-1) & \text{if } k > 0 \end{cases}$$

For example, for $n = 3$, if

$$x_0 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad \text{and} \qquad A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

we have that

$$x(0) = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \qquad x(1) = \begin{pmatrix} 6 \\ 1 \\ 2 \end{pmatrix}, \qquad x(2) = \begin{pmatrix} 9 \\ 6 \\ 1 \end{pmatrix}, \qquad \ldots$$

(a) (1 pt.) Show by induction that $x(k) = A^k \cdot x_0$ for all $k \geq 0$.

(b) (1 pt.) Explain at a high level how you would implement a function

vector<**int**> *k_th*(**const vector**<**vector**<**int**>>& *A*, **const vector**<**int**>& *x0*, **int** *k*);

for computing the k-th term $x(k)$ of the sequence defined by the matrix A and the vector *x0*. Analyse also the cost in time as a function of k. The cost must be better than $\Theta(k)$.

Problem 3 **(3.5 points)**

Given a natural number x, let us consider its representation in base 3. Assume that the number of digits n in this representation is a power of 3. We split the sequence of digits in three parts of the same size x_2, x_1 and x_0, corresponding to the highest, middle and lowest digits of x.

For example, if $x = 102111212_3$, then $x_2 = 102_3$, $x_1 = 111_3$ and $x_0 = 212_3$.

(a) (0.5 pts.) Express x in terms of x_2, x_1 and x_0. No justification is needed.

$$x = \boxed{}$$

(b) (1 pt.) Let y be another natural number with n digits in its representation in base 3, and let us define y_2, y_1 and y_0 analogously. Express $x \cdot y$ in terms of x_2, x_1, x_0, y_2, y_1 and y_0. No justification is needed.

$$\begin{aligned} x \cdot y \;=\; & \boxed{} \cdot 3^{4n/3} \\ + & \boxed{} \cdot 3^{3n/3} \\ + & \boxed{} \cdot 3^{2n/3} \\ + & \boxed{} \cdot 3^{n/3} \\ + & \boxed{} \end{aligned}$$

(c) (1 pt.) Express $x \cdot y$ in terms of x_2, x_1, x_0, y_2, y_1 and y_0 in such a way that strictly less than 9 products are introduced. No justification is needed.

$$\begin{aligned} x \cdot y \;=\; & \boxed{} \cdot 3^{4n/3} \\ + & \boxed{} \cdot 3^{3n/3} \\ + & \boxed{} \cdot 3^{2n/3} \\ + & \boxed{} \cdot 3^{n/3} \\ + & \boxed{} \end{aligned}$$

Hint: Consider $(x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2)$.

(d) (1 pt.) Describe at a high level an algorithm for computing the product of two given natural numbers represented in base 3 with a number of digits that is a power of 3. Analyse the asymptotic cost in time.

Problem 4 **(3 points)**

(a) (1 pt.) Consider a recurrence of the form

$$T(n) = T(n/b) + \Theta(\log^k n)$$

where $b > 1$ and $k \geq 0$. Show that its solution is $T(n) = \Theta(\log^{k+1} n)$.

Justification:

Note: $\log^k n$ is short for $(\log(n))^k$.

Hint: do a change of variable.

(b) (1 pt.) Given an $n \geq 1$, 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 $t = 3$).

Fill the gaps in the following code so that the function

bool *search* (**const vector**<**int**>& *a*, **int** *x*),

given a non-empty vector *a* that contains a unimodal sequence and an integer *x*, returns whether *x* appears in the sequence or not. No justification is needed.

```
bool search (const vector<int>& a, int x, int l, int r) {
  if ( [          ] ) return x == a[l];
  int m = (l+r)/2;
  auto beg = a.begin ();
  if (a[m] < [          ] )
    return search (a, x, m+1, r) or binary_search (beg + l, beg + [        ] , x);
  else
    return search (a, x, l, m) or binary_search (beg + [        ] , beg + r + 1, x);
}
```

bool *search* (**const vector**<**int**>& *a*, **int** *x*) { **return** *search* (*a* , *x* , 0, $\boxed{}$); }

Note: Given an element x and iterators *first*, *last* such that the interval $[first, last)$ is sorted (increasingly or decreasingly), the function $binary\_search(first, last, x)$ returns **true** if x appears in the interval $[first, last)$ and **false** otherwise, in logarithmic time in the size of the interval in the worst case.

(c) (1 pt.) Analyse the cost in time in the worst case of a call *search* (*a* , *x*), where *a* is a vector of size $n > 0$ that contains a unimodal sequence and x is an integer. Describe a situation in which this worst case can take place.

Midterm EDA exam **Length: 2.5 hours** **05/11/2018**

Problem 1 **(2 points)**

For each of the following statements, mark with an X the corresponding cell depending on whether it is true or false.

Note: Each right answer will add 0.2 points; each wrong answer will subtract 0.2 points, except in the case that there are more wrong answers than right ones, in which the grade of the exercise will be 0.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
TRUE										
FALSE										

(1) The cost in time of quicksort in the worst case is $\Theta(n^2)$.

(2) The cost in time of quicksort in the average case is $\Theta(n^2)$.

(3) Any algorithm that computes the sum $x + y$ of two naturals x, y of n bits each, must have cost in time $\Omega(n)$.

(4) There exists an algorithm that computes the sum $x + y$ of two naturals x, y of n bits each, in time $O(n)$.

(5) Any algorithm that computes the product $x \cdot y$ of two naturals x, y of n bits each, must have cost in time $\Omega(n^2)$.

(6) There exists an algorithm that computes the product $x \cdot y$ of two naturals x, y of n bits each, in time $O(n^2)$.

(7) Any algorithm that, given an integer $k \geq 0$ and a matrix A of $n \times n$ integer numbers, computes the matrix A^k must do $\Omega(k)$ products of matrices.

(8) $2^{2n} \in O(2^n)$.

(9) $2n \in O(n)$.

(10) $\log(2n) \in O(\log n)$.

Problem 2 **(3 points)**

Given two matrices of Booleans A and B of size $n \times n$, we define their *logical product* $P = A \cdot B$ as the $n \times n$ matrix which at the coefficient of the i-th row and j-th column ($0 \leq i, j < n$) contains:

$$p_{ij} = \bigvee_{k=0}^{n-1} (a_{ik} \wedge b_{kj})$$

(a) (0.5 pts.) Consider the following function for computing the logical product:

```
vector<vector<bool>> product1(const vector<vector<bool>>& A,
                              const vector<vector<bool>>& B) {
  int n = A.size ();
  vector<vector<bool>> P(n, vector<bool>(n, false));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = 0; k < n; ++k)
        P[i][j] = P[i][j] or (A[i][k] and B[k][j]);
  return P;
}
```

What is the cost in time in the worst case in terms of n? The cost is $\Theta($ $)$.

Give an example of worst case.

(b) (1 pt.) Consider the following alternative for computing the logical product:

```
vector<vector<bool>> product2(const vector<vector<bool>>& A,
                              const vector<vector<bool>>& B) {
  int n = A.size ();
  vector<vector<bool>> P(n, vector<bool>(n, false));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = 0; k < n and not P[i][j]; ++k)
        if (A[i][k] and B[k][j]) P[i][j] = true;
  return P;
}
```

What is the cost in time in the best case in terms of n? The cost is $\Theta($ $)$.

Give an example of best case.

What is the cost in time in the worst case in terms of n? The cost is $\Theta($ $)$.

Give an example of worst case.

(c) (1.5 pt.) Explain at a high level how to implement a function for computing the logical product of matrices that is more efficient asymptotically in the worst case than those proposed at sections (a) and (b). What is its cost in time in the worst case?

Problem 3 **(2 points)**

Consider the following function:

```
int mystery_rec(int n, int l, int r) {
  if (r == l+1) return l;
  int m = (l+r)/2;
  if (m*m ≤ n) return mystery_rec(n, m, r);
  else         return mystery_rec(n, l, m);
}
```

```
int mystery(int n) {
  return mystery_rec(n, 0, n+1);
}
```

(a) (1 pt.) Given an integer $n \geq 0$, what does function *mystery* compute?

(b) (1 pt.) What is the cost in time of *mystery*(n) in terms of n? The cost is $\Theta(\quad)$

Justify your answer.

Problem 4 **(3 points)**

Consider the problem of, given a vector of integers (possibly with repetitions), sorting it in increasing order. The following costs are in time and are asked in terms of n, the size of the vector.

(a) (0.25 pts.) What is the cost of the insertion sort algorithm in the best case? The cost is $\Theta(\quad)$.

(b) (0.25 pts.) What is the cost of the insertion sort algorithm in the worst case? The cost is $\Theta(\quad)$.

(c) (0.25 pts.) What is the cost of the mergesort algorithm in the best case? The cost is $\Theta(\boxed{\qquad\qquad})$.

(d) (0.25 pts.) What is the cost of the mergesort algorithm in the worst case? The cost is $\Theta(\boxed{\qquad\qquad})$.

(e) (0.75 pts.) Fill the gaps of the function *my_sort* defined below, so that given a vector of integers v, it sorts it increasingly:

```
void my_sort(vector<int>& v) {
  int n = v.size ();
  double lim = n * log(n);
  int c = 0;
  for (int i = 1; i < n; ++i) {
    int x = v[i];
    int j;
    for (j = i; j > 0 and $\boxed{\qquad\qquad}$ ; --j) {
      v[j] = $\boxed{\qquad\qquad}$ ;
      ++c;
    }
    v[j] = $\boxed{\qquad\qquad}$ ;
    if (c > lim) {
      merge_sort(v);
      return;
    }
  }
}
```

The auxiliary function *merge_sort* is an implementation of the mergesort algorithm, and function *log* computes the neperian logarithm (that is, in base e).

(f) (1.25 pts.) What is the cost of *my_sort* in the best case? The cost is $\Theta(\boxed{\qquad\qquad})$.

What is the cost of *my_sort* in the worst case? The cost is $\Theta(\boxed{\qquad\qquad})$.

Justify your answers and give one example of best case and one of worst case.

Midterm EDA exam **Length: 2.5 hours** **25/04/2019**

Problem 1 **(3 points)**

In this problem you do not need to justify your answers.

(a) (1 pt.) Consider the functions $f(n) = n \log n$ and $g(n) = n^{\alpha}$, where α is a real parameter. Determine for which values of α:

(1) $f = O(g)$: $\boxed{}$

(2) $f = \Omega(g)$: $\boxed{}$

(3) $g = O(f)$: $\boxed{}$

(4) $g = \Omega(f)$: $\boxed{}$

(b) (0.5 pts.) The solution to the recurrence $T(n) = 2T(n-2) + \Theta(n)$ is asymptotically $T(n) = \Theta(\boxed{})$.

(c) (0.5 pts.) The solution to the recurrence $T(n) = 2T(n/2) + \Theta(n)$ is asymptotically $T(n) = \Theta(\boxed{})$.

(d) (0.5 pts.) The cost in the worst case of mergesort when sorting a vector of size n as a function of n is $\Theta(\boxed{})$.

(e) (0.5 pts.) The cost in the best case of mergesort when sorting a vector of size n as a function of n is $\Theta(\boxed{})$.

Problem 2 **(2 points)**

Consider the following program:

```
int partition (vector<int>& v, int l, int r) {
  int x = v[l];
  int i = l − 1;
  int j = r + 1;
  while (true) {
    while (x < v[−−j]);
    while (v[++i] < x);
    if (i ≥ j) return j;
    swap(v[i], v[j]);
  }
}


void mystery(vector<int>& v, int l, int r, int m) {
  if (l < r) {
    int q = partition (v, l, r);
    mystery(v, l, q, m);
    int p = q−l+1;
    if (p < m)
      mystery(v, q+1, r, m−p);
  }
}
```

(a) (1 pt.) Given a vector v and an integer $m \geq 0$, explain what a call *mystery*(v, 0, $v.size()-1$, m) does.

(b) (1 pt.) Suppose that v is a vector of different integers sorted in increasing order. Let $n = v.size()$. Analyse the cost of calling *mystery*(v, 0, $n-1$, n) as a function of n.

Problem 3 **(2 points)**

In this problem we want to multiply natural numbers that are arbitrarily large. A natural is implemented with a **vector**<**bool**> using its representation in bits, from the most to the least significant one. Thus, for example, the vector (1, 1, 0) represents the number 4 + 2 = 6.

Assume we have already implemented the following functions with the given costs:

```
// Computes x × 2.
// Cost: Θ(n), where n = x.size().
vector<bool> twice(const vector<bool>& x)

// Computes x ÷ 2 (integer division rounded down).
// Cost: Θ(n), where n = x.size().
vector<bool> half(const vector<bool>& x)

// Computes x + y.
// Cost: Θ(n), where n = max(x.size(),y.size()).
vector<bool> sum(const vector<bool>& x, const vector<bool>& y)
```

(a) (1.5 pts.) Using these functions, complete the implementation of the function

vector<**bool**> *prod*(**const vector**<**bool**>& *x*, **const vector**<**bool**>& *y*)

for computing the product of *x* and *y*:

```
vector<bool> prod(const vector<bool>& x, const vector<bool>& y) {

  if (x.size() == 0 or y.size() == 0) return [                    ]

  vector<bool> z = twice(twice(prod(half(x), half(y))));
  vector<bool> one = vector<bool>(1, 1);
  if      (x.back() == 0 and y.back() == 0) return [              ]
  else if (x.back() == 1 and y.back() == 0) return [              ]
  else if (y.back() == 1 and x.back() == 0) return [              ]
  else {
    [                                                              ]
} }
```

Note: *If v is a* **vector** *of the STL, then v.back() is equivalent to v[v.size() − 1].*

If $n = x.size() = y.size()$, justify that the cost of calling *prod*(*x*,*y*) as a function of n is $\Theta(n^2)$.

(b) (0.5 pts.) Give the name of a well-known algorithm for computing the product of two naturals x and y of n bits each which is more efficient that the previous function *prod*. What is the cost of this algorithm as a function of n? (you do not need to justify the cost)

Problem 4 **(3 points)**
A matrix of integers of dimensions $N \times N$ is *bisorted* if each column has the elements sorted in non-decreasing order from top to bottom, and each row has the elements sorted in non-decreasing order from left to right. For example, a matrix of this kind would be:

$$\begin{pmatrix} 1 & 4 & 9 \\ 9 & 9 & 9 \\ 12 & 14 & 15 \end{pmatrix}$$

We want, given an integer x and a bisorted matrix A, to tell if x occurs in A or not.

(a) (1.5 pts.) In this section we assume that N is a power of 2. Given an integer x, a bisorted matrix A, and three integers i,j,n such that n is a power of 2 and that $1 \leq n \leq N$ and $0 \leq i,j \leq N - n$, the function

```
bool search1(int x, const vector<vector<int>>& A, int i, int j, int n)
```

returns whether x occurs in the submatrix $n \times n$ of A that contains from row i to row $i+n-1$, and from column j to column $j+n-1$. Rows and columns are indexed from 0.

Complete **using recursivity** the following implementation of *search1*:

```
bool search1(int x, const vector<vector<int>>& A, int i, int j, int n) {
  if (n == 1) return                                          ;
  int mi = i + n/2 − 1;
  int mj = j + n/2 − 1;

  if (A[mi][mj] < x) return



  if (A[mi][mj] > x) return



  return                                          ;
}
```

Note: *at each box several recursive calls can be made, which can be combined with boolean operators.*

How can function *search1* be used to determine whether x occurs in A or not? Analyse the cost in the worst case of your algorithm for doing it as a function of N.

(b) (0.75 pts.) Consider the following function *search2* to tell if x occurs in A or not:

```
bool search2(int x, const vector<vector<int>>& A) {
  int i = A.size() − 1;
  int j = 0;
  while (i ≥ 0 and j < A.size())
    if      (A[i][j] > x) −−i;
    else if (A[i][j] < x) ++j;
    else return true;
  return false;
}
```

If it is correct, justify it. Otherwise, give a counterexample.

(c) (0.75 pts.) Analyse the cost in the worst case of *search2* as a function of $N = A.size()$.

Midterm EDA exam Length: 2h45min 11/11/2019

Problem 1 **(1 point)**

(a) (0.5 pts.) The solution to the recurrence $T(n) = 2T(n/4) + \Theta(\sqrt{n})$ is asymptotically $T(n) = \Theta(\boxed{})$. You do not have to justify your answer.

(b) (0.5 pts.) For which $X \in \{O, \Omega, \Theta\}$ does it hold that $\log_2(n) \in X(\log_2(\log_2(n^2)))$?

Problem 2 **(2.5 points)**

Given a natural number $n \geq 1$, any function $f : \{0,1,\dots,n-1\} \longrightarrow \{0,1,\dots,n-1\}$ can be represented as a vector of integers $[f(0), f(1), ..., f(n-1)]$.

For example, if $n = 5$ and $f(0) = 2, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 3$, the function f can be represented by the vector $[2,1,2,4,3]$. In what follows, we will use this representation of functions.

(a) (0.75 pts.) Consider the following code:

```
void mystery_aux(const vector<int>& f, const vector<int>& g,
                 int i, vector<int>& r) {
  if (i < f.size()) {
        r[i] = f[g[i]];
        mystery_aux(f,g,i+1,r);
    }
}

vector<int> mystery(const vector<int>& f, const vector<int>& g) {
  // Precondition: f and g have the same size and
  // they contain numbers between 0 and f.size() − 1
   vector<int> r(f.size ());
   mystery_aux(f,g,0,r);
   return r;
}
```

What does the function *mystery* return? You do not have to justify your answer.

If we denote by n the size of f, what is the cost of *mystery* as a function of n?

(b) (0.75 pts.) Let us now consider the following code:

```
vector<int> mystery_2(const vector<int>& f, int k) {
  if (k == 0) {
    vector<int> r(f.size ());
    for (int i = 0; i < f.size (); ++i) r[i] = i;
    return r;
  }
  else return mystery(f,mystery_2(f,k−1));
}
```

Assuming that $k \geq 0$, what does the function *mystery_2* return? You do not have to justify your answer.

What is the cost of *mystery_2* as a function of only *k*?

(c) (1 pt.) Complete the following function so that it returns the same vector as *mystery_2* but it is asymptotically more efficient. Analyze its cost as a function of *k*.

```
vector<int> mystery_2_quick(const vector<int>& f, int k) {
    if (k == 0) {
        vector<int> r(f.size ());
        for (int i = 0; i < f.size (); ++i) r[i] = i;
        return r;
    }



}
```

Cost analysis as a function of *k*:

Problem 3 **(3.25 points)**

Given a set S of $m = 2n$ different integers, we want to put them in pairs so that the sum of their products is maximum. That is, we look for the maximum expression of the form $x_0 * x_1 + x_2 * x_3 + \ldots + x_{2n-2} * x_{2n-1}$, where the x_i's are all the elements of S.

For example, if $S = \{5,6,1,3,8,4\}$, two possible expressions are $1*5+6*3+4*8$, that adds 55, and $5*4+1*8+3*6$, that adds 46. Between these two expressions, we prefer the first one. However, there are still better expressions.

The function *max_sum* computes the maximum sum of products in S:

```
int pos_max (const vector<int>& v, int l, int r) {
  int p = l;
  for (int j = l + 1; j ≤ r; ++j)
    if (v[j] > v[p]) p = j;
  return p;
}

int max_sum (vector<int>& S) {
  int sum = 0;
  int m = S.size ();
  for (int i = 0; i < m; ++i) {
    int p = pos_max(S,i,m−1);
    swap(S[i], S[p]);
    if (i%2 == 1) sum += S[i−1]*S[i];
  }
  return sum;
}
```

(a) (1 pt.) Analyze the worst-case cost of *max_sum* as a function of m, the number of elements in S.

(b) (1 pt.) Explain at a high level how you would implement a function that solves that same problem but it is asymptotically more efficient than *max_sum*. State precisely which is the resulting cost.

(c) (1.25 pts.) Prove that the function *max_sum* returns the maximum sum of products.

Hint: first prove that if x_0 and x_1 are the two largest elements in S, then an expression that contains the products $x_0 * y$ and $x_1 * z$, for some $y, z \in S$ cannot be maximum. Then, use this fact to prove the correctness of *max_sum* by induction on m.

Problem 4 **(3.25 points)**

An important multi-sport event will be held during the next $n \geq 3$ days. We know that there exists an important black market buying and selling tickets and we want to take profit from it. We know that we can always buy or sell a ticket and we also know the prices of the tickets for every day, given as a sequence $(p_0, p_1, \ldots, p_{n-1})$.

(a) (1.25 pts.) We have realized that the sequence of prices has a very particular form. There is a unique day $0 \leq d \leq n-1$ with minimum price p_d and we know that $p_0 > p_1 > \cdots > p_d$ and $p_d < p_{d+1} < \cdots < p_{n-1}$.

Our goal is to buy a ticket on day c and sell it on day v with $0 \leq c \leq v \leq n-1$ so that we maximize our profit. That is, we want $p_v - p_c$ to be maximum. Fill in the gaps in the following code so that function *max_profit* returns the pair $< c, v >$ in time $\Theta(\log n)$ and analyze why the resulting function has this cost.

```
int f(const vector<int>& p, int l,  int r){
    if (l + 1 ≥ r) return (p[l] ≤ p[r] ? l : r);
    else {
        int m = (l+r)/2;
        if ( [                    ] ) return f(p, [        ] , [        ] );
        else if ( [                    ] ) return f(p, [        ] , [        ] );
        else return m;
    }
}

pair<int,int> max_profit (const vector<int>& p) {
    return { [                         ] , [                         ] };
}
```

Note: the expression $(B \; ? \; E_T : E_F)$ is equivalent to E_T if the Boolean expression B is true, and is equivalent to E_F otherwise.

Cost analysis:

(b) (1 pt.) From now on, assume that the sequence p does not have the form mentioned in part a), but it is an arbitrary sequence of natural numbers.

Given a day k, in which we need to have a ticket, we want to discover which is the maximum profit we can take by buying the ticket in a certain day c and selling it in a certain day v, but making sure that we have the ticket on day k. That is, not every pair (c, v) is valid, we need that $0 \leq c \leq k \leq v \leq n-1$. Implement a function with cost $\Theta(n)$ that computes this maximum profit.

int *max_profit* (**const vector**<**int**>& *p*, **int** *k*) {

}

(c) (1 pt.) We finally tackle the general problem, in which the sequence p can have any form and we want to compute the maximum profit $p_v - p_c$ that corresponds to buying the ticket on day c and selling it afterwards on day v. Explain at a high level how you would implement a function that computes this maximum profit and analyze its cost. Solutions with cost $\Omega(n^2)$ will be given 0 points.

Hint: the function of part b) can be useful to implement a divider-and-conquer algorithm.

Midterm EDA Exam Length: 1h15min 23/04/2020

Problem 1 **(3 points)**

Quizz `https://jutge.org/problems/X71865_en` to be solved via Jutge.org. You have 20 minutes to submit your answers.

Problem 2 **(3 points)**
A problem chosen randomly among:

Jutge.org, Problem X35804: Cruises
(`https://jutge.org/problems/X35804_en`).

Jutge.org, Problem X22314: Donations
(`https://jutge.org/problems/X22314_en`).

Jutge.org, Problem X79163: Anniversaries
(`https://jutge.org/problems/X79163_en`).

Problem 3 **(4 points)**
A problem chosen randomly among:

Jutge.org, Problem X30043: Cool vector
(`https://jutge.org/problems/X30043_en`).

Jutge.org, Problem X74873: Cool vector
(`https://jutge.org/problems/X74873_en`).

Jutge.org, Problem X83303: Cool vector
(`https://jutge.org/problems/X83303_en`).

Jutge.org, Problem X90362: Cool vector
(`https://jutge.org/problems/X90362_en`).

Midterm EDA Exam **Length: 1h30min.** **06/11/2020**

Problem 1 **(2.5 pts.)**

Answer the following questions:

(a) (1.5 pts.) Given a vector v of integers sorted increasingly, and an integer x, we want to determine whether x appears in v. Instead of implementing binary search, someone suggests us to choose two elements that partition the vector in three equal parts and to determine in which of these three parts we have to look for x. Complete the following code so that it is a correct implementation of this idea:

```
bool tri_search (const vector<int>& v, int l, int r, int x) {
  if (l > r) return false;
  else {
    int n_elems = (r−l+1);
    int f = l + n_elems/3;
    int s = r − n_elems/3;

    if ( [                    ] ) return true;
    if ( [          ] ) return tri_search(v, [    ] , [    ] , x);
    if ( [          ] ) return tri_search(v, [    ] , [    ] , x);
    return tri_search (v, [    ] , [    ] , x);
  }
}

bool tri_search (const vector<int>& v, int x) {
  return tri_search (v,0,v.size()−1, x);
}
```

If n is the number of elements of vector v, analyze the worst-case cost of a call to *tri_search*(v, x) as a function of n.

(b) (1 pt.) Give two functions f and g with $f \notin \Theta(g)$ such that they both are $\Omega(n)$ and $O(n \log n)$, but they both are neither $\Theta(n)$ nor $\Theta(n \log n)$.

Problem 2 **(7.5 pts.)**

Given a vector v of n natural numbers, we want to determine whether there exists a *dominant* element, that is, an element that appears more than $n/2$ times. For example:

- If $v = \{5, 2, 5, 2, 8, 2, 2\}$, then 2 is the dominant element because it appears $4 > 7/2$ times.
- If $v = \{3, 2, 3, 3, 2, 3\}$, then 3 is the dominant element because it appears $4 > 6/2$ times.
- If $v = \{6, 1, 6, 1, 6, 2, 9\}$, there is no dominant element because no element appears more than $7/2$ times.

We want to write a C++ function that receives a vector v and returns the dominant element of v, or the number -1 if such an element does not exist.

(a) (2.5 pts.) A *PRO2* student suggests the following solution:

```
int dominant_pro2 (vector<int> v) {
  int n = v.size ();
  for (int i = 0; i < n; ++i) {
    if (v[i] ≠ −1) {
      int times = 0;
      int candidate = v[i];
      for (int j = i; j < n; ++j) {
        if (v[j] == candidate) {
          ++times;
          v[j] = −1;
          if (times > n/2) return candidate;
        } } } }
  return −1;
}
```

Analyze its worst-case cost as a function of n. Explain how to construct a vector of size n for which this worst-case scenario holds.

Analyze its best-case cost as a function of n. Explain how to construct a vector of size n for which this best-case scenario holds.

If we know that, for any n, vector v will always have at most 100 different natural numbers, would that change its worst-case cost?

(b) (2 pts.) Another *PRO2* student realizes that, if we first sort the vector, a simple algorithm exists:

```
int dominant_sort (vector<int> v) {
  int n = v.size ();
  own_sort(v.begin (), v.end ());
  int i = 0;
  while (i < n) {
    int times = 0, j = i;
    while (j < n and v[j] == v[i]){
      ++times;
      ++j;
    }
    if (times > n/2) return v[i];
    i = j;
  }
  return −1;
}
```

If *own_sort* corresponds to insertion sort, which are the worst and best-case costs of *dominant_sort* as a function of *n*?

If *own_sort* corresponds to *quicksort*, which are the worst and best-case costs of *dominant_sort* as a function of *n*?

(c) (3 pts.) Finally, a good, but not brilliant, *EDA* student provides us with a divide-and-conquer solution. However, parts of the code have been lost and we ask you to complete the following function:

```
int times (const vector<int>& v, int l, int r, int x) {
  if (l > r) return 0;
  return (v[l] == x) + times(v,l+1,r,x);
}

int dominant_divide (const vector<int>& v, int l, int r) {
  if (l == r) return v[l];
  int n_elems = (r−l+1);
  int m = (l+r)/2;
  int maj_left  = dominant_divide(v, [      ] , [      ] );
  if ( maj_left  ≠ −1 and times( [            ] ) > n_elems/2) return [        ] ;
  int maj_right = dominant_divide(v, [      ] , [      ] );
  if (maj_right ≠ −1 and times( [            ] ) > n_elems/2) return [        ] ;
  return −1;
}

int dominant_divide (const vector<int>& v) {
  return dominant_divide(v,0,v.size()−1);
}
```

Analyze the worst-case cost of *dominant_divide* as a function of n.

2

Lab Exams

Lab Exam EDA Length: 2 hours 13/12/2010

Shift 1

Problem 1
Jutge.org, Problem P39846: Treasures in a map (4)
(`https://www.jutge.org/problems/P39846_en`).

Problem 2
Jutge.org, Problem P71701: Peaceful kings
(`https://www.jutge.org/problems/P71701_en`).

Shift 2

Problem 1
Jutge.org, Problem P84415: Bag of words
(`https://www.jutge.org/problems/P84415_en`).

Problem 2
Jutge.org, Problem P22295: The travelling tortoise
(`https://www.jutge.org/problems/P22295_en`).

Lab Exam EDA Length: 2 hours 19/5/2011

Problem 1
Jutge.org, Problem P69865: Picking up coins
(`https://www.jutge.org/problems/P69865_en`).

Problem 2
Jutge.org, Problem P98123: Filling the bag
(`https://www.jutge.org/problems/P98123_en`).

Lab Exam EDA Length: 2 hours 13/12/2011

Problem 1
Jutge.org, Problem P90766: Treasures in a map (3)
(`https://www.jutge.org/problems/P90766_en`).

Problem 2
Jutge.org, Problem P67329: DNA
(`https://www.jutge.org/problems/P67329_en`).

Lab Exam EDA Length: 2 hours 14/05/2012

Problem 1
Jutge.org, Problem P47386: Gossip
(`https://www.jutge.org/problems/P47386_en`).

Problem 2
Jutge.org, Problem P87462: Pacman
(`https://www.jutge.org/problems/P87462_en`).

Lab Exam EDA Length: 2hours 29/11/2012

Problem 1
Jutge.org, Problem P90861: Queues of a supermarket (1)
(`https://www.jutge.org/problems/P90861_en`).

Problem 2
Jutge.org, Problem P14952: Topological sort
(`https://www.jutge.org/problems/P14952_en`).

Lab Exam EDA Length: 2 hours 22/5/2013

Problem 1
Jutge.org, Problem X07174: Nombres sense prefixos prohibits
(`https://www.jutge.org/problems/X07174_ca`).

Problem 2
Jutge.org, Problema X34032: El cavall afamat
(`https://www.jutge.org/problems/X34032_en`).

Lab Exam EDA Length: 2 hours 4/12/2013

Problem 1
Jutge.org, Problem P60796: Treasures in a map (2)
(`https://www.jutge.org/problems/P60796_en`).

Problem 2
Jutge.org, Problem X21319: Circuit Value Problem
(`https://www.jutge.org/problems/X21319_en`).

Lab Exam EDA Length: 2 hours 19/5/2014

Problem 1
Jutge.org, Problem P81453: Shortest path
(`https://www.jutge.org/problems/P81453_en`).

Problem 2
Jutge.org, Problem P89318: Forbidden words
(`https://www.jutge.org/problems/P89318_en`).

Lab Exam EDA Length: 2 hours 22/12/2014

Problem 1
Jutge.org, Problem X41530: Forest
(`https://www.jutge.org/problems/X41530_en`).

Problem 2
Jutge.org, Problem X92609: Two coins of each kind (3)
(`https://www.jutge.org/problems/X92609_en`).

Lab Exam EDA Length: 2.5 hours 25/05/2015

Problem 1
Jutge.org, Problem P86108: LOL
(`https://www.jutge.org/problems/P86108_ca`).

Problem 2
Jutge.org, Problem P27258: Monstres en un mapa
(`https://www.jutge.org/problems/P27258_ca`).

Lab Exam EDA Length: 2.5 hours 22/12/2015

Problem 1
Jutge.org, Problem P43164: Treasures in a map (5)
(`https://www.jutge.org/problems/P43164_en`).

Problem 2
Jutge.org, Problem P31389: Rooks inside a rectangle
(`https://www.jutge.org/problems/P31389_en`).

Lab Exam EDA Length: 2.5 hours 19/05/2016

Problem 1
Jutge.org, Problem P12887: Minimum spanning trees
(`https://www.jutge.org/problems/P12887_en`).

Problem 2
Jutge.org, Problem P54070: Rightmost position of insertion
(`https://www.jutge.org/problems/P54070_en`).

Lab Exam EDA Length: 2.5 hours 15/12/2016

Shift 1

Problem 1
Jutge.org, Problem P76807: Sudoku
(`https://www.jutge.org/problems/P76807_en`).

Problem 2
Jutge.org, Problem P99753: Bi-increasing vector
(`https://www.jutge.org/problems/P99753_en`).

Shift 2

Problem 1
Jutge.org, Problem P18679: Balance beam (1)
(`https://www.jutge.org/problems/P18679_en`).

Problem 2
Jutge.org, Problem P74219: Fibonacci numbers (2)
(`https://www.jutge.org/problems/P74219_en`).

Lab Exam EDA **Length: 2.5 hours** **25/05/2017**

Problem 1
Jutge.org, Problem P49889: Consonants and vowels (1)
(`https://www.jutge.org/problems/P49889_en`).

Problem 2
Jutge.org, Problem P96413: Erdos number (2)
(`https://www.jutge.org/problems/P96413_en`).

Lab Exam EDA **Length: 2.5 hours** **11/12/2017**

Problem 1
Jutge.org, Problem P70756: Partitions
(`https://www.jutge.org/problems/P70756_en`).

Problem 2
Jutge.org, Problem P71496: Cuts
(`https://www.jutge.org/problems/P71496_en`).

Lab Exam EDA **Length: 2.5 hours** **28/05/2018**

Problem 1
Jutge.org, Problem P29033: Two colors
(`https://www.jutge.org/problems/P29033_en`).

Problem 2
Jutge.org, Problem X39049: Powers of permutations
(`https://www.jutge.org/problems/X39049_en`).

Lab Exam EDA **Length: 2.5 hours** **03/12/2018**

Problem 1
Jutge.org, Problem X82938: Search in a unimodal vector
(`https://www.jutge.org/problems/X82938_en`).

Problem 2
Jutge.org, Problem X19647: Cheapest Routes
(`https://www.jutge.org/problems/X19647_en`).

Lab Exam EDA Length: 2.5 hours 20/05/2019

Problem 1
Jutge.org, Problem X64801: Is it cyclic?
(`https://www.jutge.org/problems/X64801_en`).

Problem 2
Jutge.org, Problem P11655: Equal sums (3)
(`https://www.jutge.org/problems/P11655_en`).

Lab Exam EDA Length: 2.5 hours 02/12/2019

Problem 1
Jutge.org, Problem P65553: Boardgames
(`https://jutge.org/problems/P65553_en`).

Problem 2
Jutge.org, Problem P98259: Searching for the telecos
(`https://jutge.org/problems/P98259_en`).

Lab Exam EDA Length: 2.5 hours 10/06/2020

Problem 1
One problem chosen randomly among:

Jutge.org, Problema X39187: Buying shares
(`https://jutge.org/problems/X39187_en`).

Jutge.org, Problema X46137: Compensated words
(`https://jutge.org/problems/X46137_en`).

Jutge.org, Problema X57029: Up and down
(`https://jutge.org/problems/X57029_en`).

Problem 2
One problem chosen randomly among:

Jutge.org, Problema X14417: Film searcher
(`https://jutge.org/problems/X14417_en`).

Jutge.org, Problema X39759: Knight's game
(`https://jutge.org/problems/X39759_en`).

Lab Exam EDA Length: 2.5 hours 11/01/2021

Shift 1

Problem 1
Jutge.org, Problem X40596: Balanced sequences
(`https://jutge.org/problems/X40596_en`).

Problem 2
Jutge.org, Problem X34137: Intermediate vertices
(`https://jutge.org/problems/X34137_en`).

Shift 2

Problem 1
Jutge.org, Problem X41088: Sequences with no wells
(`https://jutge.org/problems/X41088_en`).

Problem 2
Jutge.org, Problem X50299: Short roads
(`https://jutge.org/problems/X50299_en`).

3

Final Exams

Final exam EDA Length: 3 hours 20/6/2014

Problem 1: Four quickies (from the midterm) (2 points)

1. Let $f(n) = n\log(\cos(n\pi) + 4))$. Then $f(n) = \Theta(\boxed{})$.
2. Which is the last digit of 7^{1024} written in decimal notation? Answer: $\boxed{}$.
3. What is the **recurrence** for the cost of Strassen's algorithm to multiply $n \times n$ matrices? Answer $T(n) = \boxed{}$.
4. An algorithm admits all 2^n vectors of n bits as possible inputs. On $2^n - 1$ of these inputs the cost of the algorithm is $\Theta(n^2)$, and on the remaining one the cost is $\Theta(n^4)$. Therefore, its worst-case cost is $\Theta(n^4)$ and its best-case cost is $\Theta(n^2)$. What is its average-case cost when the input is chosen uniformly at random (so each input has probability $1/2^n$)?
   Answer: $T(n) = \Theta(\boxed{})$.

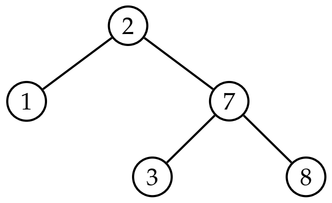
Reasoning for questions 1, 2 and 4 (reasoning for question 3 is **not** required):

Problem 2: Binary search trees and AVLs (3 points)

For this problem, use the following representation of BSTs:

```
struct node {
    Elem x;       // Information of the node
    node* lc ;    // Pointer to left child
    node* rc ;    // Pointer to right child
    node(Elem x, node* lc , node* rc ) : x( x ), lc ( lc ), rc ( rc ) {}
    ˜node() { delete lc ; delete rc ; }
};
typedef node* bst ; // A BST is a pointer to the root (null if empty)
```

(a) (1.0 points) Implement a C++ function *bst redo*(**const vector**<*Elem*> &*v*) that reconstructs a binary search tree from its preorder traversal *v*. For example, on input *v* = [2 1 7 3 8], the function should produce the following tree:



You may assume that *v* is the correct preorder of some BST. If you need so, you may implement an auxiliary function.

(b) (1.0 points) Implement a C++ function *bst build_AVL*(**const vector**<*Elem*> &*v*) that reconstructs a BST that satisfies the AVL property from a sorted vector of elements *v*. If you need so, you may implement an auxiliary function.

(c) (1.0 points) If we add a field *n.height* to denote the height of node *n* in the BST where it sits (with leaves at height 0 and therefore *NULL*s would be at height -1), complete the following code so that the result indicates if the BST *T* is an AVL:

```
bool is_AVL(bst T) {
    if (T == NULL) return true;
    else if (T−>lc == NULL or T−>rc == NULL) return (T−>height ≤ 1);
    else {

} }
```

Problem 3: Hamiltonian paths in tournament graphs (3 points)

Let us represent directed graphs with adjacency matrices:

typedef vector<**vector**<**bool**>> *Graph*;

A *tournament graph* is a directed graph in which there is exactly one arc between every pair of different vertices, and that does not have any arc from any vertex to itself.

a) (0.5 points) Write a C++ function that tells whether a given graph with n vertices is a tournament graph in $\Theta(n^2)$ time in the worst case.

bool *is_tournament*(**const** *Graph*& *G*) {

}

b) (1.0 points) Prove, by induction on the number of vertices, that every tournament graph has a Hamiltonian path, that is, a path that visits every vertex exactly once (hint: show that a new vertex can be inserted in a path of $n-1$ vertices to get a path of n vertices).

c) (1.0 points) Using the previous proof, give an algorithm that returns a Hamiltonian path of a tournament graph (clarity will be more valued than efficiency):

```
// Pre: G is a tournament graph
// Post: the result is a list of vertices that forms a Hamiltonian path in G
list <int> path(const Graph &G) {

}
```

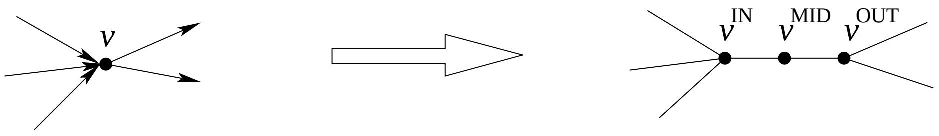
d) (0.5 points) Analize the cost of your solution.

Problem 4: Reductions **(1 point)**

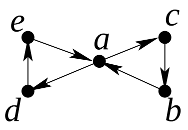
The problem of Hamiltonian cycles in directed graphs is:

> DHAM: Given a directed graph, determine whether there is a cycle that visits all vertices exactly once.

The problem of Hamiltonian cycles in undirected graphs UHAM is the same, but where the input graph is undirected. Steffy knowns that DHAM can be reduced to UHAM by simply applying the following transformation to each of the vertices v of the directed graph:



Roy and Johnny say that vertex v^{MID} is not actually needed: they say connecting v^{IN} — v^{OUT} and ignoring v^{MID} is enough. But Steffy replies: "Guys, take a look at this input to DHAM and then we talk again":



What did Steffy mean? Is she right? (be concise but precise, please)

Problem 5: Bellman-Ford (*competència transversal*) (1 point)

Recall (from the practical work of the *competència transversal*) that Bellman-Ford algorithm allows one to find minimum-weight paths in a weighted directed graph, even when weights are negative.

(a) (0.5 points) Which is the worst-case cost of Bellman-Ford algorithm on a graph with $|V|$ vertices and $|E|$ edges? Answer: Θ($\boxed{}$)

(b) (0.5 points) Steffy says that the statement at the beginning of this exercise cannot be completely right, since otherwise one could efficiently find a Hamiltonian cycle in a directed graph $G = (V, E)$ by simply assigning weight -1 to each edge in G, choosing any vertex s, and applying Bellman-Ford to check if the minimum-weight path from s to one of the vertices that have an edge towards s has weight $-(|V| - 1)$. How can you explain this paradox? (or have we proved that P = NP?)

Final exam EDA Length: 3 hours 16/1/2015

Problem 1: Asymptotic analysis (the same as in the midterm!) (2 points)

(1 point) Prove that $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

Hint: There are several ways to do it, and one of them is to consider the O and the Ω separately.

Note: Second part in the back.

(1 point) Let $f(n) = 2^{\sqrt{\log n}}$ and $g(n) = n$. Which one is asymptotically faster? Prove it.

Problem 2: Binary search trees and AVLs **(3 points)**

With the usual representation of binary search trees (this also holds for AVLs), some algorithms may be inefficient. For instance, function *smaller* $(T,\ x)$ that given a tree T and a key x, returns the number of elements in T that have a key $\leq x$.

The following modification in the representation

```
struct node {
  node* lft ;
  node* rgt ;
  Key k;      // generic type
  Value v;    // generic type
  int size ;  // size of the subtree rooted at this node ***** NEW *****
};
typedef node* bst ;
```

allows, between others, an implementation of *smaller* $(T,\ x)$ with cost $\Theta(h)$, where h is the height of T (which would be $\Theta(\log n)$ on average for a random BST and $\Theta(\log n)$ in the worst case for an AVL, where n is the size of T).

(1,5 points) Write in C++ an efficient implementation of the function

int *smaller* (*bst* T, **const** *Key*& x)

Note: Next question in the back.

(1,5 points) Write in C++ the modification of the *insert* (T,k,v) procedure that inserts a new pair $\langle k,v\rangle$ in a tree T. This procedure has as a precondition that k is not in T. Your implementation of *insert* () must have the same cost it would have under the usual representation of BSTs but, of course, somehow it must update the *size* attribute. Note that you are not asked to maintain the AVL property, nor that T is an AVL.

// Precondition: T does not contain k
void *insert* (*bst*& T, **const** *Key*& k, **const** *Value*& v)

Problem 3: Graphs and NP-completeness **(3 points)**

A mate of your is preparing the EDA lab exam and gets stuck in a problem in the "Graph Algorithms" list of the Judge. After a bit of thought, she realizes that this problem essentially asks to determine whether an undirected graph represented with adjacency lists is 2-colorable. That is, is it is possible to assign a *red* or *blue* color to each vertex so that no two adjacent vertices are painted with the same color?

Unfortunately, the Judge responds Time Limit Exceeded to all her submissions! At this moment, your mate decides to look for some more efficient code in the Internet and finds the following implementation in a web forum:

```
bool two_col_aux(const vector<vector<int>>& g, int u, vector<bool>& col,
                 vector<bool>& marked, bool is_red) {
  col[u] = is_red;
  marked[u] = true;
  for (int i = 0; i < g[u].size(); ++i) {
    int v = g[u][i];
```

```
      if (not marked[v]) {
        if (not two_col_aux(g, v, col, marked, not is_red)) return false;
      }
    }
    return true;
  }

  bool two_colourable(const vector<vector<int>>& g) {
    int n = g.size();
    vector<bool> marked(n, false);
    vector<bool> col(n);
    for (int u = 0; u < n; ++u) {
      if (not marked[u]) {
        if (not two_col_aux(g, u, col, marked, false)) return false;
      }
    }
    return true;
  }
```

Now, she expects that the code will be efficient enough, since after a bit of analysis, she knows it takes $O(|V| + |E|)$ time. After sending it to the Judge, the verdict is... Wrong Answer!

(2 points) Write a correct version of *two_col_aux* so that *two_colorable* () properly determines if a graph is 2-colorable within $O(|V| + |E|)$ time.

```
  bool two_col_aux(const vector<vector<int>>& g, int u, vector<bool>& col,
                   vector<bool>& marked, bool is_red)
```

(1 point) Thanks to your help (hopefully!), your mate could fix the code and get it accepted. Irritated, she continues reading the forum. The author of the code affirms there that his program can be easily extended to solve the 3-colorability problem with the same asymptotic complexity. Do you believe him? Why or why not?

Problem 4: Algorithmic culture **(2 point)**

Given a directed graph whose arcs have costs that can be negative, we wish to determine if it contains one or more negative cost cycles (just if they exist, not how many there are, nor which are they).

(0,5 points) Give the proper name of a famous algorithm that helps solving this problem with just a little adaptation.

(0,5 points) Briefly explain (but with enough details to implement it) how this algorithm works.

(0,5 points) What is the cost of this algorithm? Express it using the number of vertices $|V|$ and the number of arcs $|E|$ in the input graph.

(0,5 point) Explain how to adapt the general algorithm to our particular problem (negative cycle detection).

Final exam EDA Length: 3 hours 12/6/2015

One from the midterm, one from that thing, and one more (2 points)

(1 point) Determine if they are equal (=) or different ($\neq$), and prove it:

$$\Theta(3^{\log_2(n)}) \; \square \; \Theta(3^{\log_4(n)}).$$

(0.5 points) What is the algorithm from the *competencia transversal* useful for? Choose only one (reasoning not requested):

- To compute the most likely philogenetic tree.
- To compute the optimal price of a long option over a finantial product.
- To fit all 9 Beethoven symphonies into a single CD.
- To solve recurrences that do not match the form of the master theorems.

(0.5 points) The SUBSETSUM problem is the following: Given a list of natural numbers $a_1, \ldots, a_m$ and a target b, all written in decimal notation, determine if there exists a subset $S \subseteq \{1, \ldots, m\}$ such that $\sum_{i \in S} a_i = b$. If d is the number of digits of the largest natural number in the input, indicate which is the only correct assertion (reasoning not requested):

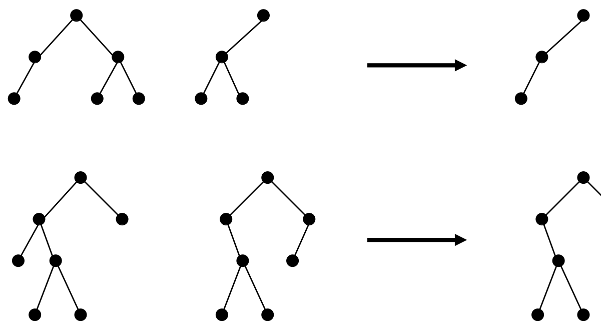
- The problem is NP-complete and, therefore, there does not exist any algorithm of cost $O(m \cdot d \cdot 10^d)$ unless P = NP.
- The problem is NP-complete and, therefore, there does not exist any algorithm of cost polynomial in m and d unless P = NP.

- The problem is NP-complete but nonetheless a known algorithm exists whose cost is polynomial in both m and d.

- The problem can be solved in time $O(d \cdot m \cdot \log m)$, and hence cost polynomial in the size of the input, as follows: first, all m numbers are added to a max-heap; then, the numbers are extracted one by one until either the sum exceeds b, or all numbers are extracted without reaching b, or the sum of the extracted numbers is exactly b.

Pointers **(2 points)**

Given two binary trees, create a new binary tree that is their intersection (and leave the given trees untouched). For example:



(1.5 points) Using the type and header as given, solve this problem.

```
struct Node {        // nodes do not store data; just pointers to children
  Node* left ;       // pointer to left child
  Node* right;       // pointer to right child
};

typedef Node* Tree;

Tree intersection (Tree t1, Tree t2) {
```

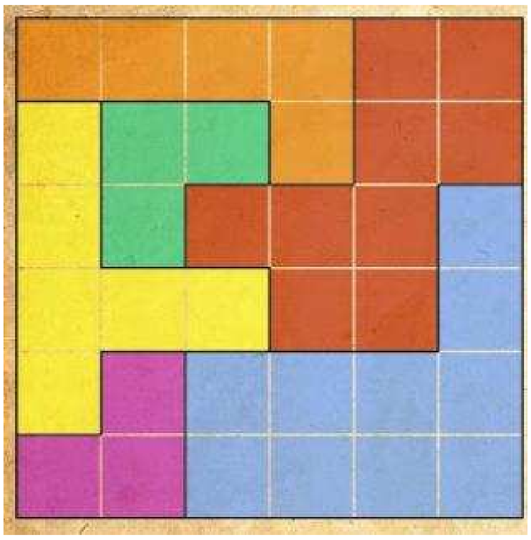
}

(0.5 points) What is its cost, in the worst case, as a function of the sizes $n1$ and $n2$ of *t1* and *t2*?

Backtracking fill-in-the-gaps **(2 points)**

Onions is a logical puzzle[1] in which we are given a square matrix $n \times n$ divided into n regions, each region painted with a different color.

[1]The original puzzle is called *Alberi* (tree in Italian) and the apps (both iOS and Android) for this game were tremendously successful a few years ago.



The goal of the puzzle is to plant n onions in such a way that

- each row contains exactly one onion,
- each column contains exactly one onion,
- each region contains exactly one onion,
- there are no two horizontally, vertically, or diagonally adjacent onions; in other words, all cells surrounding an onion must be free.

Using the following definition

```
struct Cell {
  int color;         // the color (between 0 and n-1) of its region
  bool has_onion;  // indicates if there is an onion in it or not
}

typedef vector<vector<Cell> > OnionBoard;
```

fill in the gaps in the C++ code below for the corresponding *backtracking* algorithm to find one solution, or determining that no solution exists. If there is more than one solution, any solution is good.

```
class Onions {
  int n;
  bool solution_found;
  OnionBoard board;                  // current partial solution
  vector<int> col;                   // col[i] = column of the onion at row i
  vector<bool> onion_in_col;         // onion_in_col[i] = there is an onion in column i
  vector<bool> onion_in_region; // onion_in_region[i] = there is an onion in region i

public:
  Onions(const OnionBoard& initial_board) {
    board = initial_board;
    n = board.size();
    col = vector<int>(n);
    onion_in_col = vector<bool>(n, false);
    onion_in_region = vector<bool>(n, false);
    solution_found = false;
    backtrack(0);
  }

void backtrack(int k) {
  if (k == n) { solution_found = true; return; }
  for (int j = 0; j < n and not solution_found; ++j) {
    int cell_col = [                                    ]
    if (not onion_in_col[j] and not onion_in_region[ cell_col ]
      and not onion_in_neighborhood(k, j)) {
      col[k] = j;
      [                                                 ]
      backtrack(k+1);
      [                                                 ]
} } }

// returns true iff the partial solution board has an onion in a cell adjacent to (i,j);
// you are not asked to implement it
bool onion_in_neighborhood(int i, int j);
```

Tweets **(4 points)**

A social network of the Twitter type has different users. These maintain a 'follow' relationship with other users (for instance, Salvador follows Anna, Ivet and Leo Messi, but Leo Messi does not follow anyone). Users can add or remove other users from their list of followed users. Moreover, at any moment, a user can tweet a message. Also, at any time, a user can obtain the k most recent tweets of the users she follows. (That is, if she follows n users, she gets at most k tweets, not nk).

The dimensions of such a system are what one would imagine:

- The number of users is huge; the number of tweets is even bigger.
- The number of users followed by some users may be big, but not huge.
- Some users are followed by many other users.
- The number of tweets per user has a high variability.
- The value of k is relatively small; something between 10 and 1000, say.

Users are represented by a *User* struct that contains its personal data (including a *string* with its name that identifies it in a unique way) and that tweets are represented with a struct *Tweet* that contains the message and its posting time:

```
struct User {
    ...                          // personal data
    string name;                 // different users have different names
};
```

```
struct Tweet {
    string message;              // body of the message
    int post_time ;              // number of miliseconds since some epoch
};
```

Explain which data structures and algorithms you would use to store the described information and to implement the following operations in the most efficient way: adding and removing follow relationships, tweeting a message, and obtaining the *k* most recent tweets from the users that are followed by a given user. Do not explain how users are created/deleted. Analyse the cost of your operations.

The expected answer is open but be specific defining the *SocialNetwork* data structure and describe (without code) how to implement the following operations:

```
struct SocialNetwork { ... };

// u1 follows u2 (if she didn't already); users exist.
void add_follower (SocialNetwork& sn, string u1, string u2);

// u1 unfollows u2 (if she did); users exist.
void remove_follower (SocialNetwork& sn, string u1, string u2);

// u tweets the message m; user exists.
void tweet_message (SocialNetwork& sn, string u, Tweet m);

// Returns a list with the k most recent tweets of the users followed by u; user exists.
list <Tweet> k_most_recent_tweets (const SocialNetwork& sn, string u, int k);
```

Final EDA Exam **Length: 3 hours** **07/01/2016**

Problem 1 **(3 pts.)**

Answer the following questions:

(a) (1 pt.) Consider the following decision problems:

- SAT: given a propositional formula, to determine whether it is satisfiable.
- COL: given a graph $G = (V, E)$ and a natural number k, to determine whether G can be painted with k colors so that the extremes of edges are painted with different colors.
- SOR: given an array of n different integers, to determine whether it is sorted in increasing order.

For each of the statements that follow, mark with an 'X' the corresponding column, depending on whether the statement is true, false, or is an open problem and it is still unknown if it is true or false. No justification is needed.

	Cert	**Fals**	**Obert**
SOR is in class P			
SOR is in class NP			
SOR is NP-hard			
SAT is in class P			
SAT is in class NP			
SAT is NP-hard			
SOR can be reduced polynomially to COL			
COL can be reduced polynomially to SOR			
SAT cannot be reduced polynomially to SOR			
COL can be reduced polynomially to SOR, and SAT cannot be reduced polynomially to SOR			

(b) (1 pt.) Given a vector of integers v and an integer x, the function

```
int position (const vector<int>& v, int x) {
  int n = v.size ();
  for (int i = 0; i < n; ++i)
    if (v[i] == x) return i;
  return −1;
}
```

returns the first position of v that contains x, or -1 if there is none.

Consider a probability distribution over the input parameters such that the probability that x is element $v[i]$ is $\frac{1}{n}$ for $0 \leq i < n$.

Answer: the average-case cost in time of *position* as a function of n is $\Theta(\boxed{})$.

Justification:

(c) (1 pt.) What does Floyd-Warshall's algorithm compute? What is its cost in time and space in the worst case? (no justification is needed)

Problem 2 **(2 pts.)**

Let V be a **vector**<**int**> with n different integers. Consider this code snippet:

```
set<int> S;
    for (int i = 0; i < n; ++i) {
        S.insert(V[i]);
        cout << *S.begin() << ' ';
    }
```

(a) (0.4 pts.) What does the code snippet write if $n = 6$ and the elements of V are 42 100 23 12 20 24 (in this order)?

(b) (0.4 pts.) Give a high-level explanation on which values are written by the code snippet with an arbitrary V with n different integers.

(c) (0.4 pts.) Justify what is the asymptotic cost in time of the code snippet in the worst case as a function of n.

Hint: You can use Stirling's formula: $\log(n!) = \Theta(n \log n)$.

(d) (0.4 pts.) If the elements of V were repeated, could the code snippet compute something different from what was answered in exercise (b)? If the answer is yes, show an example. If the answer is no, explain why.

(e) (0.4 pts.) Write with full detail an alternative program to the given code snippet that computes the same for an arbitrary vector V with n different integers ($n \geq 1$), but with asymptotic cost in time strictly less than that computed in exercise (c). Justify what is this asymptotic cost.

Problem 3 **(2 pts.)**

Consider the problem of, given a directed acyclic graph $G = (V, E)$, generating all its topological orderings. The next program solves it (assume $V = \{0, 1, ..., n-1\}$):

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> AdjList;
typedef vector<AdjList> Graph;

// implementation not given here
Graph read_graph ();
void print_solution (const vector<int>& sol);

bool ok(const Graph& G, const vector<int>& sol) {
  int n = sol.size ();
  vector<bool> mar(n, false);
  for (int i = 0; i < n; ++i) {
    int x = sol[i];
    if (mar[x]) return false;
    mar[x] = true;
    for (int y : G[x])
      for (int j = 0; j < i; ++j)
        if (sol[j] == y) return false;
  }
  return true;
}

void top_sorts_rec (int k, const Graph& G, vector<int>& sol) {
  int n = sol.size ();
  if (k == n) {
    if (ok(G, sol))
      print_solution (sol);
  }
  else
    for (int x = 0; x < n; ++x) {
      sol[k] = x;
      top_sorts_rec (k+1, G, sol);
    }
}

void top_sorts (const Graph& G, vector<int>& sol) {
   top_sorts_rec (0, G, sol);
}
int main() {
  Graph G = read_graph ();
  vector<int> sol(G.size ());
  top_sorts (G, sol);
}
```

However, it is too slow. Rewrite function *top_sorts* to solve the problem more efficiently, without changing *main*. Implement also the auxiliary functions that you use, except for *print_solution* and the functions of the standard C++ library.

Problem 4 **(3 pts.)**

Let G be the adjacency matrix of a directed graph with n vertices. The *closure matrix* of (the graph represented by) G, which we will represent with G^*, is an $n \times n$ matrix defined as follows: given u, v such that $0 \leq u, v < n$, the coefficient in row u and column v is

$$G^*_{uv} = \begin{cases} 1 & \text{if there is a (possibly empty) path from } u \text{ to } v \text{ in the graph} \\ 0 & \text{otherwise} \end{cases}$$

where the vertices of the graph are identified with the integers from 0 to $n-1$ as usual.

(a) (1 pt.) Give a high-level description of how to implement a function

vector<**vector**<**int**>> *closure*(**const vector**<**vector**<**int**>>& *G*);

which, given the adjacency matrix G of a directed graph, returns its closure. Analyze the asymptotic cost in time in the worst case.

(b) (1 pt.) Let us denote by I the $n \times n$ identity matrix, that is, the matrix where all coefficients are 0 except for those in the diagonal, which are 1. Show that, for any $k \geq 0$, there is a path in the graph from u to v with at most k edges if and only if the coefficient in row u column v of the matrix $(I+G)^k$ (the matrix resulting from multiplying k matrices $I+G$) is different from 0, i.e., $(I+G)^k_{uv} \neq 0$.

(c) (1 pt.) Give a high-level description of an algorithm for, given the adjacency matrix of a directed graph G, computing its closure G^* in time strictly better than $\Theta(n^3)$ in the worst case. Give the cost in the worst case and justify it. Assume that the cost of arithmetic operations with integers is constant.

Hint: if there is a path from u to v, there is at least one with at most $n - 1$ edges.

Final EDA Exam **Length: 3 hours** **07/01/2016**

Problem 1 **(1.5 pts.)**

Answer the following questions:

(a) (0.9 pt.) The master theorem for dividing recurrences claims that if we have a recurrence of the form $T(n) = aT(n/b) + \Theta(n^k)$ with $a > 0$, $b > 1$ and $k \geq 0$, then, letting $\alpha = \log_b a$,

$$T(n) = \begin{cases} \boxed{} & \text{if } \alpha < k, \\ \boxed{} & \text{if } \alpha = k, \\ \boxed{} & \text{if } \alpha > k. \end{cases}$$

(b) (0.3 pt.) The smallest k such that the following statement is true:

"For all directed graph $G = (V, E)$, it holds that $|E| = O(|V|^k)$"

is $\boxed{}$.

(c) (0.3 pt.) Class *stack*<*T*> of STL has two member functions *top*:

T& *top*();
const *T*& *top*() **const**;

On the other hand, class *priority_queue* <*T*> has one function *top* only:

const *T*& *top*() **const**;

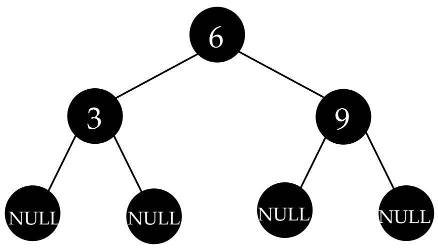
What is the problem with class *priority_queue* <*T*> having a function *T*& *top*()?

Problem 2 **(1.5 pts.)**

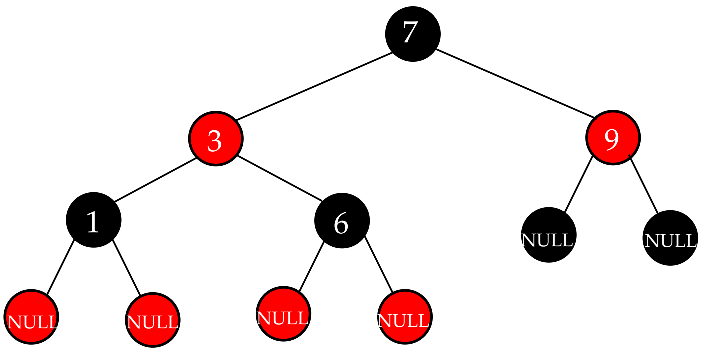
In this exercise, shaded nodes represent red nodes.

Wrong answers penalize with the same grade as right ones reward.

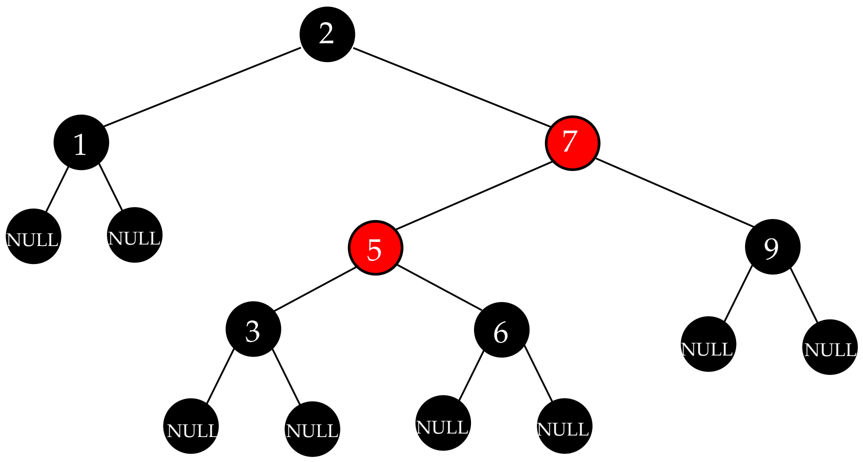
(a) The following binary tree is a red-black tree (answer **Yes** or **No**):



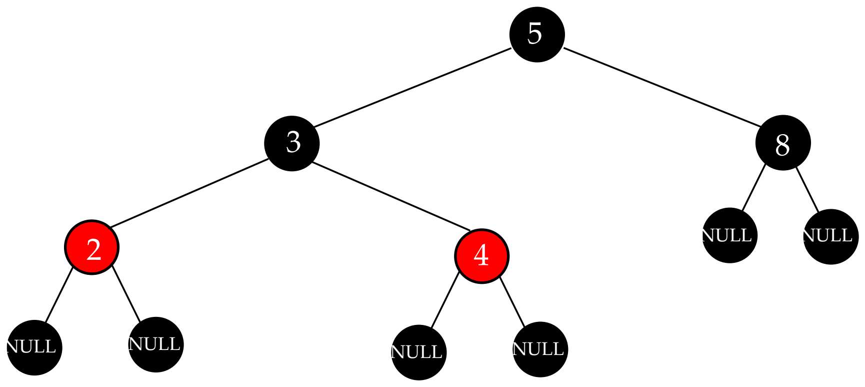
(b) The following binary tree is a red-black tree (answer **Yes** or **No**):



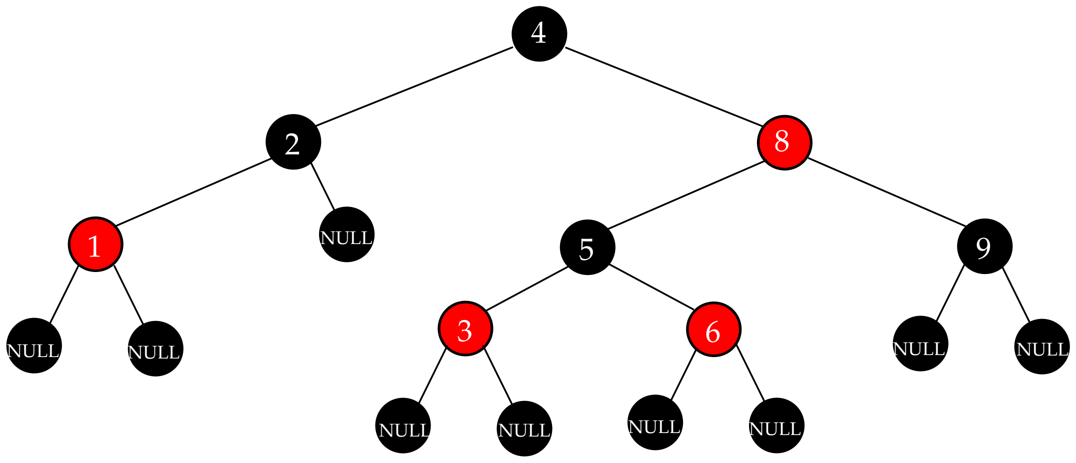
(c) The following binary tree is a red-black tree (answer **Yes** or **No**):



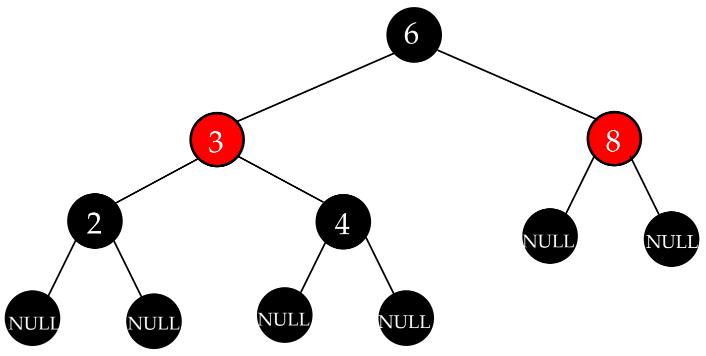
(d) The following binary tree is a red-black tree (answer **Yes** or **No**):



(e) The following binary tree is a red-black tree (answer **Yes** or **No**): $\boxed{}$



(f) The following binary tree is a red-black tree (answer **Yes** or **No**): $\boxed{}$



Problem 3 **(2 pts.)**

Let $s \in \mathbb{R}$ be such that $0 < s < 1$.

Consider the following variant of binary search which, given:

- an element x,

- a vector v that is sorted in non-decreasing order, and
- two positions l and r,

returns a position between l and r, both included, in which x appears in v (or -1 if there is none):

```
double s;   // 0 < s < 1

int position (double x, const vector<double>& v, int l, int r) {
  if (l > r) return −1;
  int p = l + int(s∗(r−l));
  if (x < v[p]) return position (x, v,   l, p−1);
  if (x > v[p]) return position (x, v, p+1, r  );
  return p;
}
```

(a) (0.75 pt.) According to the value of the parameter s, give a recurrence that describes the cost in time in the worst case of *position* as a function of $n = r - l + 1$.

(b) (0.5 pt.) According to the value of s, when does that worst case take place?

(c) (0.75 pt.) According to the value of s, solve the recurrence of the answer of exercise (a) and give the asymptotic cost in the worst case.

Problem 4 **(2 pts.)**

A *deque*<*T*> (acronym of *d*ouble-*e*nded *que*ue, pronounced: "dèk") is a sequential container of objects of type *T* that allows inserting and deleting elements both at the beginning as well as at the end, by means of the following functions:

void *push_front* (**const** *T*& *x*);
void *pop_front* ();
void *push_back* (**const** *T*& *x*);
void *pop_back* ();

respectively. Also, one can recover the element at the beginning with the functions:

T& *front* ();
const *T*& *front* () **const**;

All these functions have cost in time $\Theta(1)$.

(a) (1 pt.) A weighted directed graph $G = (V, E)$ is *binary* if for any edge $e \in E$, its weight is 0 or 1. By using a *deque*, implement in C++ a function

int *minimum_cost*(**const vector**<**vector**<*pair*<**int**,**int**>> >& G, **int** x, **int** y);

that, given a binary graph G and two vertices x and y of G, returns the minimum cost of going from x to y (or -1 if it is not possible to go from x to y).

Assume vertices are represented with integers between 0 and $n-1$, where n is the number of vertices; and that the graph is represented with adjacency lists with a **vector**<**vector**<*pair*<**int**,**int**>
where the first component of the *pair*s represents the successor vertex and the second one, the weight of the edge.

(b) (0.5 pt.) Analyze the cost in time of your function *minimum_cost* in the worst case as a function of the number of vertices $|V|$ and edges $|E|$.

(c) (0.5 pt.) Which cost in time in the worst case would *minimum_cost* have if we used the version of Dijkstra's algorithm that implements the priority queue with a binary heap (like the ones seen at class)?

Problem 5 **(3 pts.)**

The problem of DISEQUALITIES consists in, given an interval of integers and a set of disequalities ($\neq$) between variables, to determine whether there is a solution to all disequalities with values in the interval. More formally, given:

- a (finite) interval $[l, u] \subset \mathbb{Z}$,
- a set of variables V, and
- a set D of disequalities of the form $x \neq y$, where $x, y \in V$,

we have to determine where there is $s : V \to \mathbb{Z}$ such that:

- for any $x \in V$ we have $s(x) \in [l, u]$, and
- for any $x \neq y \in D$ it holds $s(x) \neq s(y)$.

We will consider that the set of variables is of the form $V = \{x_0, x_1, \dots, x_{n-1}\}$ for a certain $n > 0$, and will identify the variables with integers between 0 and $n-1$. Besides, we will represent the inputs for the problem of DISEQUALITIES by means of the type *inp_DISEQUALITIES* defined as follows:

```
struct inp_DISEQUALITIES {
  int l, u, n;
  set<pair<int,int>> D; // each pair (i,j) is a disequality x_i ≠ x_j
};
```

(a) (1 pt.) Complete the following implementation of the function

```
bool has_solution (const inp_DISEQUALITIES& e);
```

that solves the problem of DISEQUALITIES:

```
bool ok(const vector<int>& s, const set<pair<int,int>>& D) {
  for (auto d : D)
    if ( [                              ] )
      return false;
  return true;
}

bool has_solution (int k, vector<int>& s, const inp_DISEQUALITIES& e) {
  if (k == e.n) return ok(s, e.D);
  for (int v = [        ] ; v ≤ [        ] ; ++v) {
    [                    ]
    if ( has_solution (k+1, s, e)) return true;
  }
  return false;
}

bool has_solution (const inp_DISEQUALITIES& e) {
  vector<int> s(e.n);
  return has_solution ( [        ] , s, e);
}
```

(b) (0.75 pt.) Prove that the cost in time in the worst case of the previous implementation of *has_solution* is $\Omega((u-l+1)^n)$.

(c) (0.75 pt.) Consider the problem of COLORING: given an undirected graph G and a natural number $c > 0$, to determine whether one can paint the vertices of G with c colors in such a way that any edge has the endpoints painted with different colors.

Suppose that we represent the inputs for the problem of COLORING with the type *inp_COLORING* defined as follows:

```
struct inp_COLORING {
  vector<vector<int>> G; // graph represented with adjacency lists
  int c;
};
```

Implement a polynomial reduction from COLORING to DISEQUALITIES:

```
inp_DISEQUALITIES reduction(const inp_COLORING& ec);
```

(d) (0.5 pt.) Do you see feasible to find a polynomial algorithm for DISEQUALITIES? Why?

Final EDA Exam **Length: 3 hours** **12/01/2017**

Problem 1 **(2 pts.)**

Fill the following gaps as precisely as possible:

(a) (0.25 pts.) A graph with n vertices has $O(\boxed{})$ edges.

(b) (0.25 pts.) A connected graph with n vertices has $\Omega(\boxed{})$ edges.

(c) (0.25 pts.) A complete graph with n vertices has $\Omega(\boxed{})$ edges.

(d) (0.25 pts.) A min-heap with n vertices has $\Theta(\boxed{})$ leaves.

(e) (0.25 pts.) A binary search tree with n vertices has height $\Omega(\boxed{})$.

(f) (0.25 pts.) A binary search tree with n vertices has height $O(\boxed{})$.

(g) (0.25 pts.) An AVL tree with n vertices has height $\Omega(\boxed{})$.

(h) (0.25 pts.) An AVL tree with n vertices has height $O(\boxed{})$.

Problem 2 **(3 pts.)**

Let $G = (V, E)$ be a directed graph with weights $\omega : E \to \mathbb{R}$. We want to solve the problem of, given a vertex $s \in V$, to compute the distance from s to all vertices of the graph.

Recall that:

- a *path* is a sequence of vertices that are connected consecutively by arcs; that is to say, $(u_0, u_1, \ldots, u_k)$ such that for all $1 \leq i \leq k$, we have $(u_{i-1}, u_i) \in E$.
- the *weight* of a path is the sum of the weights of its arcs:

$$\omega(u_0, u_1, \ldots, u_k) = \sum_{i=1}^{k} \omega(u_{i-1}, u_i).$$

- the *distance* of a vertex u to a vertex v is the weight of the path (called *minimum path*) with minimum weight among those leaving from u and arriving at v, if it exists.

(a) (0.2 pts.) Let us assume that for any edge $e \in E$, we have $\omega(e) = 1$; that is to say, all weights are 1. Which algorithm can we use to solve efficiently the problem of the distances?

(b) (0.2 pts.) Let us assume that for any edge $e \in E$, we have $\omega(e) \geq 0$; that is to say, all weights are non-negative. Which algorithm can we use to solve efficiently the problem of the distances?

(c) (0.2 pts.) Which algorithm can we use to solve efficiently the problem of the distances, if some weights can be negative?

(d) (0.4 pts.) Which condition over the cycles of the graph must be satisfied so that, for all pairs of vertices $u, v \in V$, there exists a minimum path from u to v?

(e) (1 pt.) A function $\pi : V \to \mathbb{R}$ is a *potential* of the graph if it satisfies that, for any edge $(u, v) \in E$, we have $\pi(u) - \pi(v) \leq \omega(u, v)$. Moreover, the *reduced weights* ω_π are defined as $\omega_\pi(u, v) = \omega(u, v) - \pi(u) + \pi(v)$ for any $(u, v) \in E$.

Prove that if c is a path from u to v then $\omega_\pi(c) = \omega(c) - \pi(u) + \pi(v)$.

(f) (1 pt.) Suppose that the graph has a potential π. Then, it can be proved that for all pairs of vertices $u, v \in V$ there is a minimum path with weights ω from u to v. Assuming this fact, explain how to use the potential π to compute the distance from a given vertex s to all vertices of the graph with an alternative algorithm to that of part (c) when the weights can be negative.

Problem 3 **(2 pts.)**

We define a type *matrix* to represent square matrices of real numbers. Consider the following program:

```
typedef vector<vector<double>> matrix;

matrix aux(const matrix& A, const matrix& B) {
  int n = A.size ();
  matrix C(n, vector<double>(n, 0));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = 0; k < n; ++k)
        C[i][j] += A[i][k] * B[k][j];
  return C;
}

matrix mystery(const matrix& M) {
  int n = M.size ();
  matrix P(M), Q(M);
  for (int i = 1; i < n; ++i) {
    P = aux(P, M);
    Q = aux(Q, P);
  }
  return Q;
}
```

(a) (0.5 pts.) What does the function *matrix mystery*(**const** *matrix*& *M*) compute in terms of the matrix *M*?

(b) (0.5 pts.) If *M* is an $n \times n$ matrix, what is the cost in the worst case of the function *matrix mystery*(**const** *matrix*& *M*) as a function of *n*? Justify your answer.

(c) (1 pt.) Implement in C++ a function that computes the same as the function *matrix mystery*(**const** *matrix*& and which takes $\Theta(n^3 \log n)$ time in the worst case. Also implement the auxiliary functions you use, except for *aux* and the functions of the standard library of `C++`. Justify that the cost of your function is as required.

Problem 4 **(3 pts.)**

The problem of HAMILTONIAN GRAPH consists in, given an (undirected) graph, to decide whether it is Hamiltonian, that is to say, whether there exists a cycle that visits all its vertices exactly once. It is well-known that HAMILTONIAN GRAPH is an NP-complete problem.

It is also known that, from the proof that a problem belongs to class NP, one can derive a brute force algorithm that solves it. The following function **bool** *ham*(**const vector**<**vector**<**int**>>& G)

which, given a graph G represented with adjacency lists, returns if G is Hamiltonian, implements this algorithm in the case of HAMILTONIAN GRAPH.

```
1  bool ham_rec(const vector<vector<int>>& G, int k, vector<int>& p) {
2    int n = G.size ();
3    if (k == n) {
4      vector<bool> mkd(n, false);
5      for (int u : p) {
6        if (mkd[u]) return false;
7        mkd[u] = true;
8      }
9      for (int k = 0; k < n; ++k) {
10       int u = p[k];
11       int v;
12       if (k < n−1) v = p[k+1];
13       else         v = p [0];
14       if (find (G[u].begin (), G[u].end (), v) == G[u].end ()) return false;
15     }
16     return true;
17   }
18   for (int v = 0; v < n; ++v) {
19     p[k] = v;
20     if (ham_rec(G, k+1, p)) return true;
21   }
22   return false;
23 }
24
25 bool ham(const vector<vector<int>>& G) {
26   int n = G.size ();
27   vector<int> p(n);
28   return ham_rec(G, 0, p);
29 }
```

Note: The function of the STL library

```
Iterator find ( Iterator first , Iterator last , int val );
```

returns an iterator to the first element in the range $[first, last)$ that compares equal to *val*. If no such element exists, the function returns last.

(a) (0.5 pts.) Identify the variable in the previous program which represents the witness when the function *ham* returns **true**.

(b) (0.5 pts.) Identify the code corresponding to the verifier in the previous program. To that end, use the line numbers on the left margin.

(c) (1 pt.) Fill the following gaps so that the function *ham2* computes the same as the function *ham*, but more efficiently.

```
bool ham2_rec(const vector<vector<int>>& G, int k, int u, vector<int>& next) {
  int n = G.size ();
  if ( $\boxed{\quad}$ == n)
    return find (G[u].begin (), G[u].end (), $\boxed{\quad}$ ) $\neq$ G[u].end();

  for (int v : G[u])
    if (next[v] == $\boxed{\quad}$ ) {
      next[u] = $\boxed{\quad}$ ;
      if (ham2_rec(G, k+1, v, next)) return true;
      next[u] = −1;
    }
  return false;
}

bool ham2(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> next(n, −1);
  return ham2_rec(G, $\boxed{\quad}$ , 0, next);
}
```

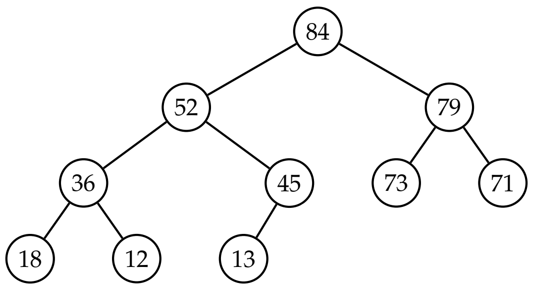
(d) (0.5 pts.) Suppose that the adjacency lists of the representation of *G* are sorted (for example, increasingly). Explain how to use this to make the function of part (c) more efficient.

(e) (0.5 pts.) Suppose that *G* is a **disconnected** graph. Explain how to use this to make the function of part (c) more efficient.

Final EDA Exam **Length: 3 hours** **09/06/2017**

Problem 1 **(2 pts.)**

(a) (0.5 pts.) Given the following max-heap:



draw the max-heap resulting from adding 55 and deleting the maximum element (in this order). You do not need to justify anything.

(b) (0.5 pts.) Using that P ⊆ NP, prove that P ⊆ co-NP.

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

```
int f(const vector<int>& v, int i, int j) {
  // Precondition: 0 ≤ i ≤ j < v.size()
  if (i == j) return v[i];
  int p = (j − i + 1)/3;
  int m1 = i+p;
```

```
    int m2 = m1+p;
    return f(v, i, m1) + f(v, m1+1, m2) + f(v, m2+1, j);
  }
```

Compute the asymptotic cost in time of f as a function of $n = j - i + 1$.

(d) (0.5 pts.) Consider the following statement:

The function n^n grows asymptotically faster than any other function.

Is it true? If so, justify it. Otherwise give a counterexample.

Problem 2 **(2 pts.)**

Given a vector A with n different integers and another vector B with m different integers, we want to compute a vector (of different integers) which is the intersection of the two (that is, that contains the common elements).

For example, if $A = (3,1,6,0)$ and $B = (4,6,1,2,7)$, then any of the two vectors $(1,6)$, $(6,1)$ would be a valid answer.

Let us assume that A and B **are not necessarily** sorted. Give a high-level description of how you would implement a function

vector<**int**> *intersection* (**const vector**<**int**>& A, **const vector**<**int**>& B);

that returns the intersection of A and B with a cost in time of $O(n+m)$ **in the average case**. Justify the cost.

Problem 3 **(3 pts.)**

Given a directed acyclic graph (DAG) G, the *level* of its vertices is defined inductively as follows:

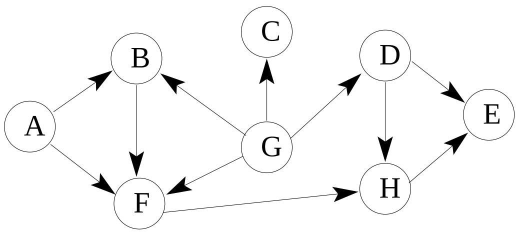
- if v is a root of G (a vertex without predecessors) then $\text{level}(v) = 0$
- otherwise,

$$\text{level}(v) = 1 + \max\{\text{level}(u) \mid u \text{ is a predecessor of } v\}$$

Moreover, the *depth* of G is the largest level of any vertex:

$$\text{depth}(G) = \max\{\text{level}(v) \mid v \text{ vertex of } G\}$$

(a) (0.9 pts.) Fill out the following table pointing out, for each vertex of the given DAG, its level. What is the depth of the DAG? You do not need to justify anything.



level :

A	B	C	D	E	F	G	H

depth : $\boxed{}$

(b) (0.4 pts.) For each of the following statements, mark with an X the corresponding cell depending on whether it is true or false. You do not need to justify anything.

Note: Every right answer will add 0.1 points; every wrong answer will subtract 0.1 points, except when there are more wrong answers than right ones, in which case the grade of the exercise will be 0.

(1) For any vertex u of a DAG G, if u is a leaf (vertex without successors) then $\text{level}(u) = \text{depth}(G)$.

(2) For any vertex u of a DAG G, if $\text{level}(u) = \text{depth}(G)$ then u is a leaf.

(3) The depth of a DAG with n vertices is $O(n)$.

(4) The depth of a DAG with n vertices is $\Omega(\log n)$.

	(1)	(2)	(3)	(4)
TRUE				
FALSE				

(c) (1.7 pts.) Here it is assumed that graphs are represented with adjacency lists, and that vertices are identified with consecutive natural numbers 0, 1, etc.

Fill the gaps of the following function:

vector<**int**> *levels*(**const vector**<**vector**<**int**>>& G);

which, given a DAG $G = (V, E)$, returns a vector that, for each vertex $u \in V$, contains the value $\text{level}(u)$ in position u. Give and justify the cost in time in the worst case in terms of $n = |V|$ and $m = |E|$.

```
vector<int> levels(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> lvl(n, −1), pred(n, 0);

  for (int u = 0; u < n; ++u)
    for (int v : G[u])
      ++pred[v];

  queue<int> Q;
  for (int u = 0; u < n; ++u)
    if (pred[u] == 0) {
      Q.push(u);
      
    }

  while (not Q.empty()) {
    int u = Q.front (); Q.pop();
    for (int v : G[u]) {
      −−pred[v];
      
      if (pred[v] == 0) Q.push(v);
  } }
  return lvl ;
}
```

Cost and justification:

Problem 4 **(3 pts.)**

Given $n > 0$, a *derangement* (of size n) is a permutation of $\{0,\ldots,n-1\}$ without fixed points; that is, π is a derangement if and only if $\pi(i) \neq i$ for all i, $0 \leq i < n$. For example $\pi = (2,0,3,1)$ is a derangement, but $\pi' = (1,3,2,0)$ is not (since $\pi'(2) = 2$).

Fill out the following C++ code for generating all derangements of size n. You do not need to justify anything.

```
void write(const vector<int>& p, int n) {
  for (int k = 0; k < n; ++k) cout << " " << p[k];
  cout << endl;
}

void generate(int k, int n, vector<int>& p, vector<bool>& used) {
      if (                                        ) write(p, n);
      else {
        for (int i = 0; i < n; ++i)
          if (                                        ) {

 
           generate(k+1, n, p, used);

        }
      }
};

void generate_all (int n) {
  vector<int> p(n);
  vector<bool> used(n, false);
  generate (                                        , n, p, used);
}
```

Final EDA Exam Length: 3 hours 19/01/2018

Problem 1 (2.5 pts.)

In this exercise, by *heaps* we mean *min-heaps* of integer numbers. We will also assume their usual implementation, in which a complete binary tree with n elements is represented with a vector of size $n + 1$ whose first component (the one with index 0) is not used.

(a) (1.25 pts.) Let v be a vector of size $n + 1$ that represents a complete binary tree with n integers. Moreover, except possibly for a certain node with index i (where $1 \leq i \leq n$), it meets the *min-heap* property: if a node has a value x and one of its descendants has value y, then $x \leq y$.

Implement in C++ the function

void *shift_down*(**vector**<**int**>& v, **int** i);

which sinks the node with index i until the min-heap property is restored for all nodes.

(b) (1.25 pts.) Implement in C++ a function

void *update* (**vector**<**int**>& v, **int** i, **int** x);

which given:

- a vector v that represents a heap with n integers,
- an index i such that $1 \leq i \leq n$, and
- an integer x,

assigns the value x to the node with index i (discarding the previous value in the node) and updates v so that it still represents a heap.

You can use the function *shift_down* of the previous exercise, as well as the following function *shift_up*:

```
void shift_up (vector<int>& v, int i) {
    if (i ≠ 1 and v[i/2] > v[i]) {
        swap(v[i], v[i/2]);
        shift_up(v, i/2);
}   }
```

Problem 2 **(2 pts.)**

Given a permutation π of $\{1,\ldots,n\}$, the value $\pi(i)$ is a *left maximum* if $\pi(i) > \pi(j)$ for all j, $1 \le j < i$. The first element $\pi(1)$ is always a left maximum.

For example $\pi = (3,1,5,4,6,2)$ has 3 left maxima (namely, $\pi(1) = 3$, $\pi(3) = 5$ and $\pi(5) = 6$).

Fill the gaps in the following C++ code that writes the permutations of $\{1,\ldots,n\}$ that have exactly k left maxima, where $1 \le k \le n$.

```
int n, k;
vector<int> perm;
vector<bool> used;

void write() {
  cout << perm[1];
  for (int i = 2; i ≤ n; ++i) cout << ' ' << perm[i];
```

```
    cout << endl;
  }

  void generate(int i, int lm, int mx) {
    if ( ____________ > k or ____________ < k ) return;
    if ( ______________________ ) write();
    else {
      for (int j = 1; j ≤ n; ++j)
        if ( ______________________ ) {
          ______________________
          perm[i] = ______________ ;
          if ( ______________________ ) generate(i+1, lm+1, j);
          else generate(i+1, lm, mx);
          ______________________
        }
    }
  }

  int main() {
    cin >> n >> k;
    perm = vector<int>(n+1);
    used = vector<bool>(n+1, false);
    generate (1, 0, 0);
  }
```

Problem 3 **(3 pts.)**

(a) (1 pt.) Consider the following program:

```
int x = ...;   // initialized to some value
int mystery(const string& s) {
  int h = 0;
  for (int k = s.size() − 1; k ≥ 0; −−k)
    h = x*h + int(s[k]);
  return h;
}
```

What is function *mystery* computing?

If one wanted to use function *mystery* as a hash function for the type **string**, would 0 be an adequate value for x? Why?

(b) (1 pt.) For each of the following statements, mark with an X its cell depending on whether it is true or false. You do not need to justify anything.

Note: Each right answer will add 0.25 points; each wrong answer will subtract 0.25 points, except for the case when there are more wrong answers than right ones, in which the grade of the exercise will be 0.

Recall: SAT is the problem of, given a propositional formula, to decide whether it is satisfiable or not; and HAM is the problem of, given a graph, to decide whether it is hamiltonian.

(1) If there exists an NP problem that belongs to P then P = NP.

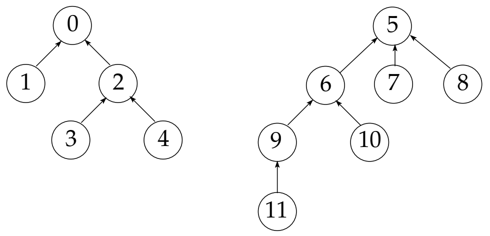
(2) If there is a polynomial reduction from a problem X to SAT then there is a polynomial reduction from X a HAM.

(3) There are problems of the class NP for which there is no known algorithm that solves them.

(4) If a problem X can be reduced polynomially to a problem $Y \in$ P then $X \in$ P.

	(1)	(2)	(3)	(4)
TRUE				
FALSE				

(c) (0.5 pts.) Consider the following forest of rooted trees, which represents a collection of disjoint sets:



The value returned by *find*(2) is $\boxed{}$.

The value returned by *find*(9) is $\boxed{}$.

(d) (0.5 pts.) Consider the following variant of binary search, which given:

- an element x,

- a vector v sorted in non-decreasing order, and
- two positions l and r,

returns a position between l and r where x appears in v (or -1 if there is none):

```
int posicio (double x, const vector<double>& v, int l, int r) {
  if (l > r) return −1;
  int p = l + int((r−l)/3.0);
  if (x < v[p]) return posicio (x, v,   l, p−1);
  if (x > v[p]) return posicio (x, v, p+1, r  );
  return p;
}
```

Analyze the asymptotic cost in the worst case.

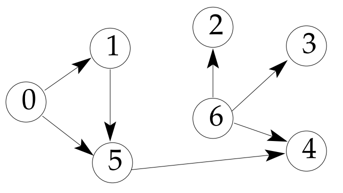
Problem 4 **(2.5 pts.)**

Given two digraphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ defined over the same set of vertices V, the *composition* of G_1 and G_2 is the digraph denoted by $G_1 \circ G_2$ whose vertices are V, and whose edges are the following set:

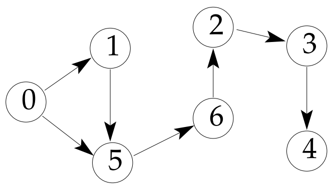
$$E = \{(u,w) \mid \exists v \text{ such that} (u,v) \in E_1 \wedge (v,w) \in E_2\}$$

Note: In this exercise it is allowed that digraphs have *auto-cycles*, that is, edges of the form (u,u).

(a) (0.5 pts.) Consider the following digraphs G_1 and G_2:



G_1



G_2

Draw $G_1 \circ G_2$.

(b) (1 pt.) Assume that G_1 and G_2 are represented with adjacency matrices. Implement in C++ a function

```
typedef vector<vector<int>> matrix;
matrix comp(const matrix& G1, const matrix& G2);
```

which, given the adjacency matrices of G_1 and of G_2, computes the adjacency matrix of $G_1 \circ G_2$ in time $\Theta(|V|^3)$ in the worst case. Justify that the cost is as required.

(c) (1 pt.) Explain how to implement function *comp* in time better than $\Theta(|V|^3)$.

Final EDA Exam **Length: 3 hours** **20/06/2018**

Problem 1 **(2 pts.)**

(a) (0.5 pts.) Class *stack*<*T*> of STL has two member functions *top*:

T& *top*();
const *T*& *top*() **const**;

On the other hand, class *priority_queue* <*T*> has one function *top* only:

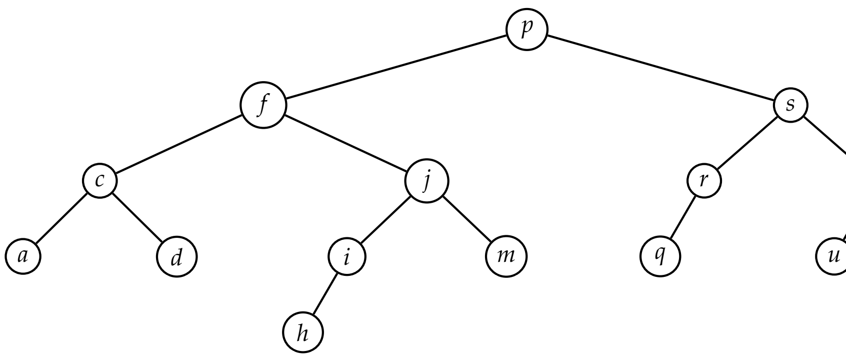
const *T*& *top*() **const**;

What is the problem with class *priority_queue* <*T*> having a function *T*& *top*()?

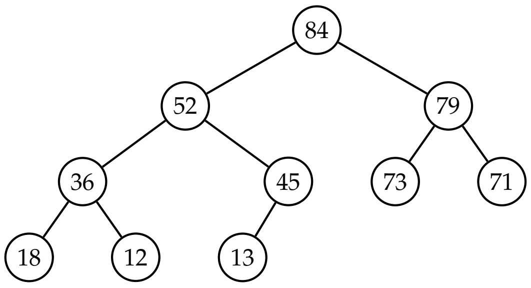
(b) (0.5 pts.) For the alphabet $\{A, B, C, D\}$ and the following table of absolute frequencies for 100 characters, give a Huffman code and draw its tree. You do not need to justify anything.

A	70
B	10
C	15
D	5

(c) (0.5 pts.) Draw the AVL tree resulting from inserting the three keys x, y, z in this order to the AVL tree of the figure. Note that the keys are ordered in alphabetic order. You do not need to justify anything.



(d) (0.5 pts.) Given the following max-heap:



draw the max-heap resulting from adding 55 and deleting the maximum element (in this order). You do not need to justify anything.

Problem 2 **(3 points)**

(a) (1 pt.) Consider a recurrence of the form

$$T(n) = T(n/b) + \Theta(\log^k n)$$

where $b > 1$ and $k \geq 0$. Show that its solution is $T(n) = \Theta(\log^{k+1} n)$.

Justification:

Note: $\log^k n$ is short for $(\log(n))^k$.

Hint: do a change of variable.

(b) (1 pt.) Given an $n \geq 1$, 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 $t = 3$).

Fill the gaps in the following code so that the function

bool *search* (**const vector**<**int**>& *a*, **int** *x*),

given a non-empty vector *a* that contains a unimodal sequence and an integer *x*, returns whether *x* appears in the sequence or not. No justification is needed.

```
bool search (const vector<int>& a, int x, int l, int r) {
  if ( $\boxed{\qquad\qquad}$ ) return x == a[l];
  int m = (l+r)/2;
  auto beg = a.begin ();
  if (a[m] < $\boxed{\qquad\qquad}$ )
    return search (a, x, m+1, r) or binary_search (beg + l, beg + $\boxed{\qquad\qquad}$ , x);
  else
    return search (a, x, l, m) or binary_search (beg + $\boxed{\qquad\qquad}$ , beg + r + 1, x);
}

bool search (const vector<int>& a, int x) { return search (a, x, 0, $\boxed{\qquad\qquad}$ ); }
```

Note: Given an element *x* and iterators *first*, *last* such that the interval [*first*, *last*) is sorted (increasingly or decreasingly), the function *binary_search*(*first*, *last*, *x*) returns **true** if *x* appears in the interval [*first*, *last*) and **false** otherwise, in logarithmic time in the size of the interval in the worst case.

(c) (1 pt.) Analyse the cost in time in the worst case of a call *search* (*a*, *x*), where *a* is a vector of size $n > 0$ that contains a unimodal sequence and *x* is an integer. Describe a situation in which this worst case can take place.

Problem 3 **(3 pts.)**

A student of EDA is preparing for the lab exam and gets stuck in a problem of the list "Graph Algorithms" of the Judge. After thinking for a while, he realizes that he must, essentially, given a graph (undirected, represented with adjacency lists), determine whether it is 2-colorable; that is, whether it is possible to assign a color *red* or *blue* to each vertex of the graph, so that adjacent vertices are not painted with the same color.

Unfortunately, the Judge returns a Time Limit Exceeded verdict to all his submissions. At this point the student decides to look in the Internet for a more efficient code, and finds the following one in a forum:

```
bool two_col_aux(const vector<vector<int>>& g, int u, vector<bool>& col,
                 vector<bool>& marked, bool is_red) {
  col[u] = is_red;
  marked[u] = true;
  for (int v : g[u]) {
    if (not marked[v]) {
      if (not two_col_aux(g, v, col, marked, not is_red)) return false;
    }
  }
  return true;
}

bool two_colourable (const vector<vector<int>>& g) {
  int n = g. size ();
  vector<bool> marked(n, false);
  vector<bool> col(n);
  for (int u = 0; u < n; ++u) {
    if (not marked[u]) {
      if (not two_col_aux(g, u, col, marked, false)) return false;
    }
  }
  return true;
}
```

Now he expects that this code will be efficient enough, since after a bit of analysis he concludes that it takes time $O(|V| + |E|)$. He submits it to the Judge, and it turns out that... Wrong Answer!

(a) (2 pts.) Write a correct version of *two_col_aux* so that *two_colorable* determines whether a graph is 2-colorable in time $O(|V| + |E|)$.

```
bool two_col_aux(const vector<vector<int>>& g, int u, vector<bool>& col,
                 vector<bool>& marked, bool is_red)
```

(b) (1 pt.) Finally the student finds a way to fix the code and the Judge accepts it. Uneasy, he carries on reading the forum. The author of the wrong code claims that his program can be easily extended to the problem of 3-colorability with the same asymptotic cost. Do you believe it? Why?

Problem 4 **(2 pts.)**

Consider the following problem: given n positive integer numbers a_0, a_1, ..., a_{n-1} and two other integers l and u such that $0 \le l \le u \le \sum_{i=0}^{n-1} a_i$, we have to find all possibles solutions $(x_0, x_1, \ldots, x_{n-1}) \in \{0,1\}^n$ to the double inequation:

$$l \le a_0x_0 + a_1x_1 + \cdots + a_{n-1}x_{n-1} \le u$$

(a) (1 pt.) Complete the following program to solve the previous problem:

```
int n, l, u;
vector<int> a;
vector<bool> x;

void sols (int k, int sum_chosen, int sum_rest) {
  if (k == n) {
    for (int i = 0; i < n; ++i) cout << x[i];
    cout << endl;
  } else {
    if (                                ) {
      x[k] = 0; sols (k+1, sum_chosen, sum_rest − a[k]);
    }
    if (                                ) {
      x[k] = 1; sols (k+1, sum_chosen + a[k], sum_rest − a[k]);
    } } }

int main() {
  cin >> n >> l >> u;
  a = vector<int>(n);
  for (int i = 0; i < n; ++i) cin >> a[i];

  int s = 0;
  for (int i = 0; i < n; ++i) s += a[i];

  x = vector<bool>(n);
  sols (        ,         ,         );
}
```

(b) (1 pt.) The program in the previous exercise writes the solutions in **increasing** lexicographic ordering. For example, for the input $n = 3$, $l = 3$, $u = 5$ and $a_i = i + 1$ $(0 \leq i \leq 2)$, the output is:

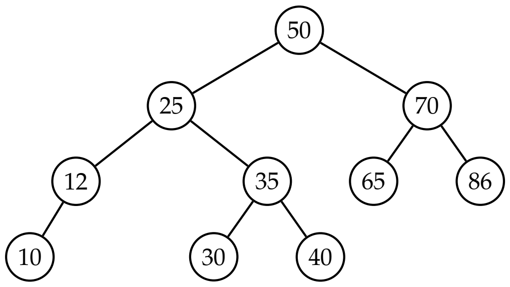
```
001
011
101
110
```

Explain how to change function *sols* to write all solutions in **decreasing** lexicographic ordering, that is, in the reverse order of the previous example, without using additional space. In particular, the algorithm consisting in storing all the output in a vector, and at the end sort it and write it will not be considered as a valid answer.

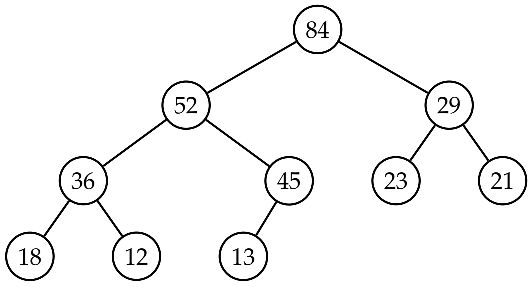
Final EDA Exam **Length: 3 hours** **14/01/2019**

Problem 1 **(3.5 pts.)**

(a) (0.75 pts.) Draw the AVL tree resulting from adding 31 to the AVL tree below:



(b) (0.75 pts.) Draw the heap resulting from removing the maximum from the following max-heap:

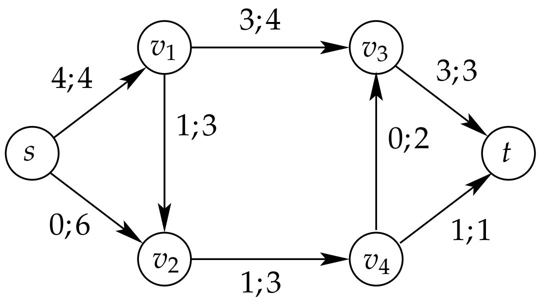


(c) (0.5 pts.) The solution to the recurrence $T(n) = 3T(n/9) + \Theta(n)$ is

(d) (0.5 pts.) The solution to the recurrence $T(n) = 3T(n/9) + \Theta(\sqrt{n})$ is $\boxed{}$

(e) (0.5 pts.) The solution to the recurrence $T(n) = 3T(n/9) + \Theta(1)$ is $\boxed{}$

(f) (0.5 pts.) Consider the following flow network with origin s and destination t, and the indicated flow:



For each arc, we indicate the flow along the arc and its capacity, separated by a semi colon. E.g., the arc from v_1 to v_3 carries 3 units of flow and has capacity 4.

Draw the residual network of this network with respect to the given flow.

Problem 2 **(4 pts.)**

In this exercise we will work with a type *polynomial* for representing polynomials in a variable x with integer coefficients.

(a) (0.5 pts.) Suppose that we implement polynomials with vectors of integers (that is, *polynomial* is **vector<int>**), so that for $k = 0, 1, \ldots$ the k-th value of the vector is the coefficient of x^k. Hence, for example the vector $\{1, 0, -3\}$ represents the polynomial $1 - 3x^2$.

Given a polynomial p and integers $c \neq 0$ and $k \geq 0$, consider the following function:

```
polynomial mystery(const polynomial& p, int c, int k) {
  int n = p.size ();
  polynomial q(n + k, 0);
  for (int i = 0; i < n; ++i)
    q[i + k] = c*p[i];
  return q;
}
```

Answer:
the function *mystery* computes
and its cost in time in terms of $n = p.size$ () and k is $\Theta($ $)$.

(b) (1.5 pts.) Suppose that now we implement polynomials with a dictionary with integer keys and integer values, so that a pair of the dictionary with key k and value v (with $v \neq 0$) represents the monomial $v \cdot x^k$. So for example the dictionary with pairs key-value $\{(0,1),(2,-3)\}$ represents the polynomial $1 - 3x^2$.

More specifically, let us assume that *polynomial* is *unordered_map*<**int**,**int**>.

Which of the data structures seen at class could be used for implementing *unordered_map*<**int**,**int**>?

Fill the gaps in the following reimplementation of the function *mystery* that uses this new representation of polynomials.

```
polynomial mystery(const polynomial& p, int c, int k) {
  polynomial q;
  for (auto mon : p)
    q[          ] =           ;
  return q;
}
```

Give and justify an accurate upper bound of the cost in time in the average case.

(c) (0.25 pts.) From this section onwards we will suppose again that we implement polynomials with vectors of integers (that is, *polynomial* is **vector**<**int**>).

Consider the following function, which defines the operator $+$ for polynomials for writing in C++ the sum of two polynomials p and q as $p + q$:

```
polynomial operator+(const polynomial& p, const polynomial& q) {
  polynomial s(max(p.size (), q. size ()));
  for (int i = 0; i < p.size (); ++i)
    s[i] += p[i];
  for (int i = 0; i < q.size (); ++i)
    s[i] += q[i];
  return s;
}
```

Let $n = \max(p.size(), q.size())$.

Answer:
the cost of the function in terms of n is $\Theta($ $\boxed{}$ $)$.

(d) (1.75 pts.) Fill the gaps in the following function, which defines the operator $*$ for polynomials for writing in C++ the product of two polynomials p and q as $p * q$. You can use the functions that appear in the previous sections. Suppose that $p.size() = q.size() = n$, where $n \geq 1$ is a power of 2. The cost of the function must be **strictly better** than $\Theta(n^2)$.

```
polynomial operator*(const polynomial& p, const polynomial& q) {
  int n = p. size ();
  if (n == 1) return [                              ] ;
  int n2 = n/2;

  polynomial p0(n2), p1(n2);
  for (int k = 0; k < n2; ++k) p0[k] = p[k];
  for (int k = n2; k < n; ++k) p1[k − n2] = p[k];

  polynomial q0(n2), q1(n2);
  for (int k = 0; k < n2; ++k) q0[k] = q[k];
  for (int k = n2; k < n; ++k) q1[k − n2] = q[k];

  polynomial p0_q0 = p0 * q0;
  polynomial p1_q1 = p1 * q1;
```

}

Analyse the cost of the function in terms of n.

Problem 3 **(2.5 pts.)**

Given a natural $N \geq 1$, we consider the set of the first N naturals $\{0, \ldots, N-1\}$. A *set cover* of $\{0, \ldots, N-1\}$ is a collection of subsets S_0, ..., S_{M-1} of $\{0, \ldots, N-1\}$ such that each of the first N naturals is included in at least one of the subsets, that is, $\bigcup_{i=0}^{M-1} S_i = \{0, \ldots, N-1\}$.

The problem of SET-COVER consists in, given:

- two naturals K and N,
- subsets S_0, ..., S_{M-1} of $\{0, \ldots, N-1\}$ that cover $\{0, \ldots, N-1\}$,

to determine if, among the given subsets, one can choose k subsets S_{i_0}, S_{i_1}, ..., $S_{i_{k-1}}$ with $k \leq K$ which also form a set cover of $\{0, \ldots, N-1\}$.

For example, suppose that $N = 4$, $S_1 = \{0,1,3\}$, $S_2 = \{0,3\}$ and $S_3 = \{0,2,3\}$. Then if $K = 2$ the answer is TRUE (since $S_1 \cup S_3 = \{0,1,2,3\}$), but if $K = 1$ the answer is FALSE (since there is no single S_i that covers the set $\{0,1,2,3\}$).

(a) (0.75 pts.) Consider the following implementation of the function

```
bool set_cover (int K, int N, const vector<set<int>>& S);
```

for solving the problem of SET-COVER:

```
bool set_cover_rec (int K, int N, const vector<set<int>>& S,
                    vector<int>& w, int c) {
  if (c == N) return true;
  if (K == 0) return false;
  for (int i = 0; i < S.size (); ++i) {
    for (int x : S[i]) {
      if (w[x] == 0) ++c;
      ++w[x];
    }
    if ( set_cover_rec (K−1, N, S, w, c)) return true;
    for (int x : S[i]) {
      −−w[x];
      if (w[x] == 0) −−c;
  } }
  return false;
}

bool set_cover (int K, int N, const vector<set<int>>& S) {
  vector<int> w(N, 0);
  return set_cover_rec (K, N, S, w, 0);
}
```

This implementation has a serious efficiency bug. Explain what is this error and rewrite the previous code with the required changes to fix it.

(b) (1 pt.) Given a graph $G = (V, E)$, a *vertex cover* is a subset of vertices $W \subseteq V$ such that every edge $\{u, v\} \in E$ has at least one of the two endpoints in W, that is, $u \in W$ or $v \in W$.

The problem of VERTEX-COVER consists in, given a natural K and a graph $G = (V, E)$, to determine whether there exists a subset $W \subseteq V$ with at most K vertices that is a vertex cover of G.

Of the two functions defined next, one is a reduction of VERTEX-COVER to SET-COVER and the other is not. Prove that the right one is indeed correct and give a counterexample for the wrong one.

(1) Given $G = (V, E)$, with $V = \{u_0, \ldots, u_{N-1}\}$,

$$f(K, G) = (K, N, S),$$

where $S = \{S_e \,|\, e \in E\}$ and $S_e = \{i \,|\, u_i \in e\}$.

(2) Given $G = (V, E)$, with $E = \{e_0, \ldots, e_{N-1}\}$,

$$g(K, G) = (K, N, S),$$

where $S = \{S_u \,|\, u \in V\}$ and $S_u = \{i \,|\, u \in e_i\}$.

Correct reduction and proof:

Incorrect reduction and counterexample:

(c) (0.75 pts.) Suppose we have been able to reduce VERTEX-COVER to SET-COVER, and that we know that VERTEX-COVER is NP-complete. Can we deduce only from this that SET-COVER is an NP-complete problem? If so, justify it. Otherwise show that SET-COVER is NP-complete (assuming that VERTEX-COVER reduces to SET-COVER and that VERTEX-COVER is NP-complete).

Final EDA Exam **Length: 3 hours** **17/06/2019**

Problem 1 **(3 pts.)**

(a) (1.25 pts.) Let us denote with $\wedge$ the logical AND, with $\vee$ the logical OR and with $\neg$ the logical NOT. Given Boolean variables $x_1, ..., x_n$, we define the formula $\text{xor}(x_1, ..., x_n)$ recursively as follows:

- If $n = 1$ then $\text{xor}(x_1) = x_1$;
- Otherwise:

$$\begin{aligned}\text{xor}(x_1,...,x_n) \quad = \quad & \big((\text{xor}(x_1,...,x_{\lceil\frac{n}{2}\rceil}) \quad \wedge \quad \neg\text{xor}(x_{\lceil\frac{n}{2}\rceil+1},...,x_n)) \vee \\ & (\neg\text{xor}(x_1,...,x_{\lceil\frac{n}{2}\rceil}) \quad \wedge \quad \text{xor}(x_{\lceil\frac{n}{2}\rceil+1},...,x_n))\big).\end{aligned}$$

For example, $\text{xor}(x_1, x_2) = \big((x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)\big)$, and

$$\text{xor}(x_1,x_2,x_3) = \Big(\Big(\big((x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)\big) \wedge \neg x_3\Big) \vee \Big(\neg\big((x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)\big) \wedge x_3\Big)\Big)$$

Give a reasoned expression in asymptotic notation Θ of the size of the formula $\text{xor}(x_1, ..., x_n)$ as a function of n. Assume that every Boolean variable, as well as every symbol (parentheses, $\wedge$, $\vee$, $\neg$) occupy 1 byte.

(b) (1.25 pts.) For a fixed natural $k \geq 2$, the problem of k-COLOR consists in, given an undirected graph $G = (V, E)$, to determine if there exists a k-coloring of the graph; that is, a function $C : V \rightarrow \{1, \ldots, k\}$ such that for all $\{u, v\} \in E$ we have $C(u) \neq C(v)$.

We want to justify that k-COLOR reduces to $k + 1$-COLOR. Consider the following proposal of reduction:

"Given a graph G, we define its image via the reduction as the graph itself. It is clear that it is an algorithm that takes polynomial time in the size of the input. And if G admits a k-coloring, then the same assignment of colors is a $(k+1)$-coloring."

Is this reduction correct? If it is not, why?

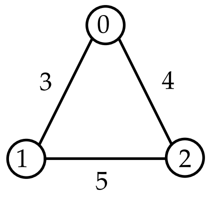
(c) (0.5 pts.) In the database of a marriage agency there is a set H of men and a set D of women. The agency also has the list L of pairs (h,d), with $h \in H$ and $d \in D$, such that h and d would be willing to marry each other. Each h and each d can marry at most once. Let us consider the problem of finding the largest number of marriages that can be formed. Indicate which well-known problem of graph theory this problem translates into.

Problem 2 **(3 pts.)**

In this problem we consider undirected graphs with positive integer costs on the edges. Vertices are identified with consecutive numbers 0, 1, 2, etc. We represent graphs with adjacency lists using vectors of vectors of pairs of **int**s, where the first component is the cost of the edge and the second, the adjacent vertex:

```
typedef pair<int,int>                CostVertex;
typedef vector<vector<CostVertex>> Graph;
```

(a) (0.75 pts.) Consider the graph shown next:



Complete the following code so that variable g takes as value the given graph.

```
Graph g = { [                                        ] };
```

(b) (0.75 pts.) Consider the following fragment of code:

```
vector<CostVertex> mystery(const Graph& g) {
  int n = g. size ();
  vector<CostVertex> t(n, {−1, −1});
  vector<int> a(n, INT_MAX);
  a[0] = 0;
  vector<bool> marked(n, false);
  priority_queue <CostVertex, vector<CostVertex>, greater<CostVertex> > pq;
  pq.push({0, 0});
  while (not pq.empty()) {
    int u = pq.top (). second;
    pq.pop ();
    if (not marked[u]) {
      marked[u] = true;
      for (CostVertex cv : g[u]) {
        int c = cv. first ;
        int v = cv.second;
        if (a[v] > a[u] + c) {
          a[v] = a[u] + c;
          t[v] = {c, u};
          pq.push({a[v], v});
    } } } }
  return t;
}
```

At the end of the execution of function *mystery*, what is the meaning of the value of $a[u]$ for a given vertex u? No justification is needed.

(c) (1.5 pts.) Consider the code of exercise (b). Assume that g is connected.

If t is the vector returned by *mystery*(g), let T be the set of edges $\{u,v,c\}$ such that $0 \leq u < n$, $t[u] = \{c,v\}$ and $v \neq -1$. It can be proved (you do not have to do it) that T induces a subgraph of g that is connected and without cycles, that is, a tree.

Justify that T is a **spanning** tree.

Is it necessarily a **minimum** spanning tree? If the answer is yes, justify it. Otherwise, give a reasoned counterexample.

Problem 3 **(2 pts.)**
Write a function *pred_succ* which, given a binary search tree T and a value x, returns two pointers, one to the node with the greatest key amongst the nodes with key less than or equal to x, and another pointing to the node with the least key amongst the nodes with key greater than or equal to x. If there is not any value in T that is less than or equal to x, then the first returned pointer should be *NULL*; similarly, if there is not any value in T that is greater than or equal to x, then the second returned pointer should be *NULL*.

Use the following definitions:

```
struct node {
  int    key;
  node*  left ;
  node*  right ;
};

// A binary search tree is implemented with a node*

pair<node*, node*> pred_succ(const node*& T, int x);
```

For example, if T is an empty binary search tree, the result of *pred_succ* (T,x) will be <*NULL*, *NULL*> independently of x. If x is in T, then the result will be <p, p>, where p points to the node that contains x. If x does not appear in T and is strictly less than any key in T, and T is not empty, then the result will be <*NULL*, p>, where p points to the node with the least key of T, etc.

Your function must have cost in time $O(h(T))$ for any T and x, where $h(T)$ is the height of T. You do not need to justify the cost.

If you use auxiliary functions, implement them too.

Problem 4 **(2 pts.)**

In the travelling salesman problem, a salesman must visit the clients of n different cities. The distance between city i and city j is a positive number $D[i][j]$. The salesman wants to leave his own city, visit once and only once each other city, and return to the starting point. His goal is to do that and minimize the total distance of the journey.

(a) (1.5 pts.) Fill the gaps in the following program so that it solves the travelling salesman problem. Cities are identified with consecutive numbers 0, 1, 2, etc. Assume that the city of the salesman is city 0.

```
struct TSP {

  vector<vector<double>> D;
  int n;
  vector<int> next, best_sol ;
  double best_tot_dist ;

  void recursive (int v, int t, double c) {
    if (t == n) {
      c +=                        ;
      if (c < best_tot_dist ) {
        best_tot_dist =                        ;
        best_sol =                        ;
        best_sol [v] = 0;
      }
    }
    else for (int u = 0; u < n; ++u)
        if (u ≠ v and next[u] == −1) {
          next[v] = u;
          recursive (u, t+1,                        );
          next[v] =                        ;
        }
  }

  TSP(const vector<vector<double>>& D) {
    this−>D = D;
    n = D.size ();
    next = vector<int>(n, −1);
    best_tot_dist = DBL_MAX;
    recursive (0, 1, 0);
  }
};
```

(*main* on the following page)

```
int main () {
  int n;
  cin >> n;

  vector<vector<double>> D(n, vector<double>(n));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      cin >> D[i][j ];

  TSP tsp(D);
  cout << "Best total distance: " << tsp. best_tot_dist << endl;
  vector<int> b = tsp. best_sol ;
  cout << "Sequence of cities: 0";
  int c = b [0];
  while (c ≠ 0) {
    cout << ' ' << c;
    c = b[c ];
  }
  cout << ' ' << 0 << endl;
}
```

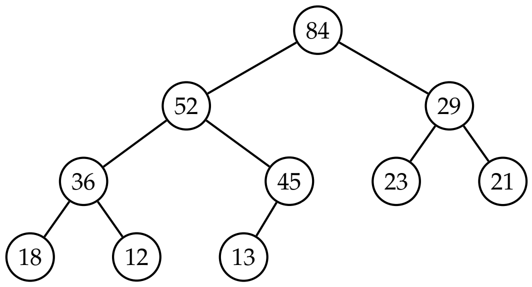
(b) (0.5 pts.) Add code to the first line of function *recursive* to make the program more efficient:

```
  void recursive (int v, int t, double c) {
    [                                          ]
    if (t == n) {
```

Final EDA exam **Length: 3h** **13/01/2020**

Problem 1 **(1.5 pts.)**

(a) (0.5 pts.) Write the heap that is obtained after adding 64 to the following max-heap. You do not have to justify your answer.



(b) (0.5 pts.) Write the **recurrence** that expresses the cost of Strassen algorithm to multiply $n \times n$ matrices. You do not have to justify your answer.

$T(n) =$

(c) (0.5 pts.) Consider the following code:

```
int g(int p);
int f(int n, int p) {
      if (n == 0) return p;
      else        return 1 + f(n/2, g(p));
}
```

If we know that the cost of function g is quadratic, which is the asymptotic cost in time of f as a function of n?

Problem 2 **(3 pts.)**

Given a directed acyclic graph (DAG) G, the *level* of its vertices is defined inductively as follows:

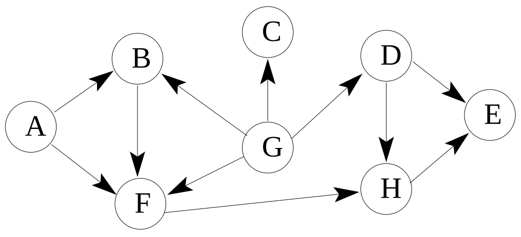
- if v is a root of G (a vertex without predecessors) then $\text{level}(v) = 0$
- otherwise,

$$\text{level}(v) = 1 + \max\{\text{level}(u) \mid u \text{ is a predecessor of } v\}$$

Moreover, the *depth* of G is the largest level of any vertex:

$$\text{depth}(G) = \max\{\text{level}(v) \mid v \text{ vertex of } G\}$$

(a) (0.5 pts.) Fill out the following table pointing out, for each vertex of the given DAG, its level. What is the depth of the DAG? You do not need to justify anything.



level :

A	B	C	D	E	F	G	H

depth : $\boxed{}$

(b) (0.8 pts.) For each of the following statements, mark with an X the corresponding cell depending on whether it is true or false. You do not need to justify anything.

Note: Every right answer will add 0.2 points; every wrong answer will subtract 0.2 points, except when there are more wrong answers than right ones, in which case the grade of the exercise will be 0.

(1) For any vertex u of a DAG G, if u is a leaf (vertex without successors) then $\text{level}(u) = \text{depth}(G)$.

(2) For any vertex u of a DAG G, if $\text{level}(u) = \text{depth}(G)$ then u is a leaf.

(3) The depth of a DAG with n vertices is $O(n)$.

(4) The depth of a DAG with n vertices is $\Omega(\log n)$.

	(1)	(2)	(3)	(4)
TRUE				
FALSE				

(c) (1.7 pts.) Here it is assumed that graphs are represented with adjacency lists, and that vertices are identified with consecutive natural numbers 0, 1, etc.

Fill the gaps of the following function:

```
vector<int> levels(const vector<vector<int>>& G);
```

which, given a DAG $G = (V, E)$, returns a vector that, for each vertex $u \in V$, contains the value $\text{level}(u)$ in position u. Give and justify the cost in time in the worst case in terms of $n = |V|$ and $m = |E|$.

```
vector<int> levels(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> lvl(n, −1), pred(n, 0);

  for (int u = 0; u < n; ++u)
    for (int v : G[u])
      [                                    ]

  queue<int> Q;
  for (int u = 0; u < n; ++u)
    if (pred[u] == 0) {
      Q.push(u);
      [                                    ]
    }

  while (not Q.empty()) {
    int u = Q.front (); Q.pop();
    for (int v : G[u]) {
      −−pred[v];
      [                                    ]
      if (pred[v] == 0) Q.push(v);
  } }
  return lvl;
}
```

Cost and justification:

Problem 3 **(2.25 pts.)**
The president of an oil company contacts us in order to decide in which cities of a remote country one should install gas stations. We know the cost of installing a gas station in each city, and we also have the list of all roads of the country, identified by pairs $\{c_1, c_2\}$, where c_1 and c_2 are the two cities that this road connects. Our goal is to decide in which cities we should install a gas station so that the installation cost is minimum while satisfying that, for each road $\{i, j\}$, there is a gas station in i, in j, or in both cities.

a) (2 pts) Fill in the gaps in the following program in order to solve the aforementioned problem. Cities are identified by consecutive numbers 0, 1, 2, etc.

```
struct GasStations {
  int n;
  vector<int> cost;
  vector<vector<int>> roads;
  vector<bool> best_solution;
  int  best_cost ;
  void rec(int i, vector<bool>& partial_sol, int partial_cost ) {
    if (i == n) {
      best_cost = partial_cost ;
      best_solution = partial_sol ;
    }
    else {
      if ( ____________________ < best_cost) {
        partial_sol [i] = ________ ;
        rec( ____ ,partial_sol, ____________________ );
      }

      bool needed = false;
      for (auto& c : roads[i])
        if ( ______ and ____________ ) needed = true;

      if (not needed) {
        partial_sol [i] = ______ ;
        rec( ____ ,partial_sol, _________ );
      } } } // closing rec

  GasStations (const vector<int>& c, vector<vector<int>>& r) {
    n = c.size (); best_cost = INT_MAX;
    this−>cost = c; this−>roads = r;
    vector<bool> partial_sol(n,false );
    rec (0, partial_sol ,0);
  } };

int main ( ){
  int n; // number of cities
  cin >> n;
  vector<int> c(n); // cost of each city
  for (int i = 0; i < n; ++i) cin >> c[i];
  vector<vector<int>> r(n); // roads (graph with adjacency lists)
  int c1, c2;
  while (cin >> c1 >> c2) { // Road between c1 and c2
    r[c1].push_back(c2);
    r[c2].push_back(c1);
  }
  GasStations gas(c,r);
  cout << "Best cost: " << gas.best_cost << endl;
  cout << "Chosen cities:";
  for (int i = 0; i < n; ++i)
    if (gas. best_solution [i]) cout << " " << i;
  cout << endl;
}
```

b) (0.25 pts) Let us now assume that we have to solve a slightly different problem: we have to maximize the total installation cost. What would you change in the previous code? If you consider that you have to change too many things, you can explain at a high level which algorithm you would implement.

Problem 4 **(3.25 pts.)**

Let us remember the **3-SAT** problem: a *boolean variable* can only take values 0 (false) and 1 (true). A *literal* is a boolean variable x or its negation $\neg x$. A *clause* is a disjunction of literals. A *3-CNF* is a conjunction of clauses that contain exactly 3 literals. An assignemt I of values to the boolean variables satisfies a literal x iff $I(x) = 1$, and satisfies a literal $\neg x$ iff $I(x) = 0$. The **3-SAT** problem consists of, given a 3-CNF F, determining whether there exists an assignment that satisfies **at least** one literal in each clause of F. So far, nothing new.

We now introduce the problem **ONE-IN-THREE-SAT** that consists of, given a 3-CNF F, determining whether there exists an assignment satisfying **exactly** one literal in each clause of F.

(a) (0.75 pts.) Consider the 3 clauses:

$$\begin{array}{ccccc} x_1 & \vee & \neg x_2 & \vee & x_3 \\ x_1 & \vee & x_4 & \vee & x_5 \\ \neg x_2 & \vee & x_4 & \vee & x_5 \end{array}$$

Write an assignment that satisfies exactly one literal per clause.

$I(x_1) = \boxed{}$, $I(x_2) = \boxed{}$, $I(x_3) = \boxed{}$, $I(x_4) = \boxed{}$, $I(x_5) = \boxed{}$.

Is there any assignemt I satisfying exactly one literal in each clause and such that $I(x_1) = 1$? Justify your answer.

Write a **small** set of 3-literal-clauses that is a positive input for **3-SAT** but not for **ONE-IN-THREE-SAT**. We will accept sets with at most 4 clauses. However, 3 clauses should be enough. Justify your answer.

(b) (1.5 pts.) Given a 3-literal clause $C = l_1 \lor l_2 \lor l_3$, we define the set of clauses $R(C) = \{C_1, C_2, C_3\}$, where

$$\begin{array}{rcccccc} C_1 = & \neg l_1 & \lor & a & \lor & b \\ C_2 = & l_2 & \lor & b & \lor & c \\ C_3 = & \neg l_3 & \lor & c & \lor & d \end{array}$$

and a, b, c, d are fresh boolean variables. *Note:* if x is a variable and literal l is $\neg x$, we will understand that $\neg l$ corresponds to x.

Prove that if I is an assignment that satisfies exactly one literal in each of clause of $R(C)$, then I satisfies at least one literal in C.

Prove that, given an assignment I satisfying at least one literal in C, we can build an assignment I' satisfying exactly one literal in each clause of $R(C)$ and that coincides with I over literals l_1, l_2 and l_3.

(c) (0.5 pts.) Prove that **ONE-IN-THREE-SAT** is NP-hard.

(d) (0.5 pts.) Prove that **ONE-IN-THREE-SAT** is NP-complete.

Final EDA Exam Length: 2 h 45min 09/06/2020

Each student is randomly assigned either option A or B for each problem.

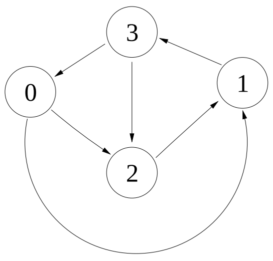
Problem 1 (2 pts.)

Quizz `https://jutge.org/problems/X53101_en` to be solved via Jutge.org. You have 30 minutes to submit your answers.

Problem 2A (2 pts.)

Given a directed graph G, we want to determine whether G is Hamiltonian: that is, whether there exists some cycle that visits each vertex exactly once. You can see below an example of an input file and the graph it represents. The input file starts with a line of the form $n\ m$, where n is the number of vertices (that will be represented by numbers between 0 and $n-1$) and m is the number of arcs. After that, there are m lines of the form $u\ v$ corresponding to arcs $u \to v$.

```
4 6
0 2
0 1
2 1
3 2
1 3
3 0
```



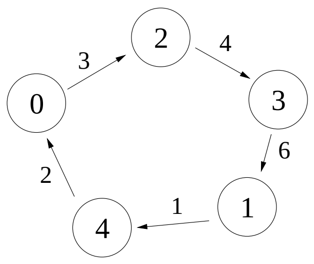
In this case, the graph is Hamiltonian, since $2 \to 1 \to 3 \to 0 \to 2$ is a cycle with the desired properties. Fill in the gaps in the following code with an instruction or expression in order to determine whether a directed graph is Hamiltonian.

```
class HamiltonianGraph {
  vector<vector<int>> G;
  int n;
  vector<int> s;        // partial solution being built
  vector<bool> used;
  bool found;           // whether we have found some Hamiltonian cycle
  vector<int> sol;      // if found, sol stores the Hamiltonian cycle

  void recursive ( ){
    int u = s.back ();
    if (s.size () == n) {
      for (int v : G[u]) {
        if ( [                    ] ){
          found = true;
          sol = s;
      } } }
    else {
      for (int v : G[u]) {
        if ( [                       ] ){
          s.push_back(v);
          used[v] = [              ] ;
          recursive ();
          ____________________ ;
          ____________________ ;
          if (found) return; } } } }

public:
  HamiltonianGraph (vector<vector<int>> &G) {
    this->G = G;
    n = G.size ();
    used = vector<bool>(n,false);
    s = vector<int>(1,0);
    used[0] = true;
    found = false;
    recursive ();
  }

  bool has_solution ( ){return found;}
  vector<int> solution( ) {return sol;}
};

int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> G(n);
  for (int i = 0; i < m; ++i){
    int u, v;
    cin >> u >> v;
    G[u].push_back(v);
  }
  HamiltonianGraph ham(G);
  cout << ham.has_solution () << endl;
}
```

Problem 2B **(2 pts.)**

We want to solve the well-known Travelling Salesman Problem: given a set of cities and the distances among them we have to find, among all cycles that visit all cities exactly once, one with minimum distance. You can see below an example of an input file and a cycle with minimum distance for it. The input file starts with an integer n, that corresponds to the number of cities, followed by an $n \times n$ symmetric matrix of natural numbers with all distances.

```
5
0  7  3  9  2
7  0  6  6  1
3  6  0  4  2
9  6  4  0  3
2  1  2  3  0
```



In this case the cycle with minimum distance has distance 16. Fill in the gaps in the following code with an instruction or expression in order to find a cycle with minimum distance.

```
const int infinite = numeric_limits<int>::max();
class TSP {
  int n;
  vector<vector<int>> M;
  vector<int> s;           // partial solution being built
  vector<bool> used;
  vector<int> best_sol;    // best cycle found so far
  int best_cost;           // cost of the best cycle found so far

  void recursive(int c) {
    int u = s.back();
    if (s.size() == n) {
      c += M[u][0];
      if (c < best_cost) {
        best_cost = c;
        best_sol = s;
      }
    }
    else {
      for (int v = 0; v < n; ++v)
        if (                    ) {
          if (c + M[u][v] < best_cost) {
            s.push_back(v);
            used[v] =                ;
            recursive(                );
                            ;
                            ; } } } } }

public:
  TSP(vector<vector<int>>& M) {
    this->M = M;
    n = M.size();
    s = vector<int>(1,0);
    used = vector<bool>(n,false);
    used[0] = true;
    best_cost = infinite;
    recursive(0);
  }

  vector<int> solution() {return best_sol;}
  int cost() {return best_cost;}
};


int main(){
  int n;
  cin >> n;
  vector<vector<int>> M(n, vector<int>(n));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      cin >> M[i][j];
  TSP tsp(M);
  cout << tsp.cost() << endl;
}
```

Problem 3A **(3 pts.)**

The head of a steel factory wants us to help him in deciding the order in which the metal pieces they need to manufacture have to be produced. Fortunately, the problem is easy to define, since the only existing constraint is that there are pairs of pieces (p_1, p_2) such that p_1 needs to be produced before p_2. An input file has the following format:

```
5 4
0 1   1 3   2 3   3 4
```

The first line indicates that we have n pieces (numbers between 0 and $n-1$) and m pairs of pieces with an ordering constraint. In the example above, we have 5 pieces and 4 pairs. Then we find m pairs $p_1 p_2$ indicating that piece p_1 has to be produced before piece p_2. A possible solution to this concrete problem is the order: $(0,2,1,3,4)$.

Knowing that any input file will have at least a possible order, consider the solution to the problem given in the file `ex-packages.cc`. You will notice that, among others, the file contains an implementation of a priority queue with some additional operations. When analyzing costs, you can ignore *assert* instructions and their cost.

(a) (1 pt.) Analyze the worst-case asymptotic cost in time of function *write_packages* as a function of n and m.

(b) (1 pt.) Explain in words how you would implement a program that improves the worst-case asymptotic cost in time of the solution given in file `ex-packages.cc`. Justify the cost of your solution.

(c) (1 pt.) It is easy to see that this problem often admits multiple solutions. For example, $(0,1,2,3,4)$ is also a correct order for the previous instance. Among all solutions, we want the lexicographically smallest one.

We remind you that a solution $p = (p_1, p_2, \ldots, p_n)$ is lexicographically smaller than another solution $q = (q_1, q_2, \ldots, q_n)$ if there exists an index k with $1 \leq k \leq n$ such that for all j with $1 \leq j < k$ we have $p_j = q_j$ but $p_k < q_k$. That is, if, if we consider the solutions as words, p would appear before q in the dictionary.

Modify the given code so that it computes the lexicographically smallest solution. Write only the parts you have modified, that should not be too many.

File `ex-packages.cc`:

```
#include<iostream>
#include<vector>
#include<assert.h>

using namespace std;

template <typename Elem, typename Key>
class PriorityQueue {
private:
  vector<pair<Elem,Key>> v; // Table for the heap (position 0 is not used)
public:
  //Constructor. Creates an empty priority queue.
  PriorityQueue () {
    v.push_back({Elem(),Key()});
  }

  //Inserts a new element.
  //PRE: x.first does not belong to the priority queue
  void insert (const pair<Elem,Key> & x) {
    assert (not contains (x. first ));
    v.push_back(x);
    shift_up ( size ());
  }

  //Removes and returns the minimum element.
  pair<Elem,Key> remove_min () {
    if (empty()) throw "Priority queue is empty";
    pair<Elem,Key> x = v[1];
    v[1] = v.back ();
    v.pop_back ();
    shift_down (1);
    return x;
  }

  //Returns the minimum element
  pair<Elem,Key> minimum () {
    if (empty()) throw "Priority queue is empty";
    return v [1];
  }

  //Returns the size of the priority queue.
  int size () {
    return v. size () − 1;
  }

  //Indicates if the priority queue is empty.
  bool empty () {
    return size () == 0;
  }

  // Pre: x is an element in the priority queue
  // the key of x is ¿ newKey
  // Post: the key of x is updated to newKey
  void decrease_key (const Elem& x, const Key& newKey) {
    assert (contains (x ));
    int idx = find(x);
    v[idx ]. second = newKey;
    shift_up (idx );
  }

  // Returns whether x belongs to the priority queue
  bool contains(const Elem& x) {
    int idx = find(x);
    return idx ≠ −1;
  }

  // PRE: x belongs to the priority queue
  // Returns the key of the element x
  Key key(const Elem&x ){
    assert (contains (x ));
    return v[find (x )]. second;
  }

private:

  // Returns the idx in v of the element x
  // Returns -1 if not present
  int find (const Elem& x) {
    for (uint i = 1; i < v.size (); ++i) if (v[i ]. first == x) return i;
    return −1;
  }

  //Shifts a node up in the tree, as long as needed.
  void shift_up (int i) {
    if (i ≠ 1 and v[i/2].second > v[i ]. second) {
      swap(v[i ], v[i /2]);
      shift_up (i /2);
    } }

  //Shifts a node down in the tree, as long as needed.
  void shift_down (int i) {
    int n = size ();
    int c = 2*i;
    if (c ≤ n) {
      if (c+1 ≤ n and v[c+1].second < v[c ]. second) c++;
      if (v[i ]. second > v[c ]. second) {
        swap(v[i ],v[c ]);
```

```
            shift_down (c );
        } } }

};


void write_packages  (const vector<vector<int>>& G) {

  int n = G.size ();

  vector<int> indegree(n ,0);
  for (int u = 0; u < n; ++u)
    for (int v:G[u]) ++indegree[v ];

  PriorityQueue<int,int> Q;
  for (int u = 0; u < n; ++u) Q.insert({u, indegree [u ]});

  while (not Q.empty()) {
    pair<int,int> p = Q.remove_min();
    assert (p.second == 0);
    cout << p. first  << " ";
    for (int v : G[p. first ]) {
      int d = Q.key(v );
      Q. decrease_key (v ,d−1);
    }
  }
  cout << endl;

}

int main(){
  int n, m;
  cin >> n >> m;
  vector<vector<int>> G(n);

  for (int i = 0; i < m; ++i) {
    int x, y;
    cin >> x >> y;
    G[x ].push_back(y );
  }
  write_packages (G);
}
```

Problem 3B **(3 pts.)**

You are surely familiar with Dijkstra's algorithm. In this problem, we consider a weighted directed graph and we want to know the distance between two nodes. The input is given in the following format:

```
3 3
0 2 99   0 1 40   1 2 60
0 2
```

The first line indicates that we have n nodes (numbers between 0 and $n-1$) and m arcs. After that we find m arcs represented by triples u v c meaning that there is an arc from u to v with weight c. Finally we find a pair of vertices x and y, indicating that we want to know the distance from x to y.

You do not have to solve this problem, but rather analyze and improve the solution we provide in file `ex-dijkstra.cc`. When analyzing costs, you can ignore *assert* instructions and their cost.

(a) (1 pt.) Answer the following questions. Answers must only depend on n and m and, except for the first question, you must always justify your answer.

Which is the cost of function *find*? $O(\boxed{})$

How many times can, at most, a given vertex be inserted into the priority queue Q?

How many times can, at most, a given vertex be removed from the priority queue Q?

How many times, at most, is function *decrease_key* called?

Which is the asymptotic cost in time, in the worst case, of function *dijkstra*?

(b) (1.5 pts.) Modify the solution we have provided so that function $find$ has, in the worst case, asymptotic cost $\Theta(1)$. You can assume in the following that searching in an *unordered_map* always has constant cost. Hint: you can modify other functions, apart from $find$.

Write only the parts that have been modified.

(c) (0.5 pts.) Assuming that $find$ has constant time, which is the cost in time of the new solution? If we know that $m = \Theta(n^2)$, how does the cost of $dijkstra$ in the new solution compare asymptotically with the cost in the initial solution we have provided?

File `ex-dijkstra.cc`:

```
#include<iostream>
#include<vector>
#include<unordered_map>
#include<limits>
#include<assert.h>

using namespace std;

const int infinite = numeric_limits<int>::max();

template <typename Elem, typename Key>
class PriorityQueue {
private:
  vector<pair<Elem,Key>> v; // Table for the heap (position 0 is not used)
public:
  //Constructor. Creates an empty priority queue.
  PriorityQueue () {
    v.push_back({Elem(),Key()});
  }

  //Inserts a new element.
  //PRE: x.first does not belong to the priority queue
  void insert (const pair<Elem,Key> & x) {
    assert (not contains(x.first ));
    v.push_back(x);
    shift_up (size ());
  }

  //Removes and returns the minimum element.
  pair<Elem,Key> remove_min () {
    if (empty()) throw "Priority queue is empty";
    pair<Elem,Key> x = v[1];
    v[1] = v.back ();
    v.pop_back ();
    shift_down (1);
    return x;
  }

  //Returns the minimum element.
  pair<Elem,Key> minimum () {
    if (empty()) throw "Priority queue is empty";
    return v[1];
  }

  //Returns the size of the priority queue.
  int size () {
    return v.size () − 1;
  }
```

```
    //Indicates if the priority queue is empty.
    bool empty () {
      return size () == 0;
    }

    // Pre: x is an element in the priority queue
    // the key of x is ¿ newKey
    // Post: the key of x is updated to newKey
    void decrease_key (const Elem& x, const Key& newKey) {
      assert (contains (x ));
      int idx = find (x );
      v[idx ]. second = newKey;
      shift_up (idx );
    }

    // Returns whether x belongs to the priority queue
    bool contains (const Elem& x) {
      int idx = find (x );
      return idx ≠ −1;
    }

    // PRE: x belongs to the priority queue
    // Returns the key of the element x
    Key key(const Elem&x ){
      assert (contains (x ));
      return v[find (x )]. second;
    }

  private:

    // Returns the idx in v of the element x
    // Returns -1 if not present
    int find (const Elem& x) {
      for (uint i = 1; i < v. size (); ++i) if (v[i ]. first == x) return i ;
      return −1;
    }

    //Shifts a node up in the tree, as long as needed.
    void shift_up (int i) {
      if (i ≠ 1 and v[i/2].second > v[i ]. second) {
        swap(v[i ], v[i /2]);
        shift_up (i /2);
      } }

    //Shifts a node down in the tree, as long as needed.
    void shift_down (int i) {
      int n = size ();
      int c = 2*i ;
      if (c ≤ n) {
        if (c+1 ≤ n and v[c+1].second < v[c].second) c++;
        if (v[i].second > v[c].second) {
          swap(v[i], v[c]);
          shift_down(c);
        } } }

};

  int dijkstra (const vector<vector<pair<int,int>>>& G, int x, int y) {
    int n = G.size();
    vector<bool> removed(n,false);
    PriorityQueue<int,int> Q;
    Q.insert({x,0});

    while (not Q.empty()) {
      pair<int,int> m = Q.remove_min();
      int u = m.first;
      int d_u = m.second;
      assert (not removed[u]);
      removed[u] = true;
      if (u == y) return d_u;
      for (const pair<int,int>& p : G[u]) {
        int v = p.first;
        int c_v = p.second;
        if (not removed[v]){
        if      (not Q.contains(v))     Q.insert({v, d_u + c_v});
        else if (Q.key(v) > d_u + c_v)  Q.decrease_key(v, d_u + c_v);
        }
      }
    }
    return −1;
  }

  int main(){
    int n, m;
    int c = 0;
    while (cin >> n >> m) {
      ++c;
      vector<vector<pair<int,int>>> G(n); //pairs are (vertex,weight)
      for (int i = 0; i < m; ++i) {
        int u, v, c;
        cin >> u >> v >> c;
        G[u].push_back({v,c});
      }
      int x, y;
      cin >> x >> y;

      int d = dijkstra(G,x,y);
```

```
        if  (d == −1) cout ≪ "no path from " ≪ x ≪ " to " ≪ y ≪ endl;
        else  cout ≪ d ≪ endl;
    }
}
```

Problem 4A **(3 pts.)**

Consider the following two decision problems:

VILLAGES: given

- a set of villages P,
- a set of paths between them $C = \{c_1, c_2, \ldots, c_m\}$, where each path c_i is a pair of different villages,
- and a natural number k,

we want to know whether there exists a subset $S \subseteq P$ of size k such that for all $u, v \in S$ with $u \neq v$ we have $\{u, v\} \in C$. That is, we want to know whether there exists a subset of k villages such that they all are pairwise connected by a direct path.

ENEMIES: given

- a set of workers T,
- a set of enmities among them $E = \{e_1, e_2, \ldots, e_q\}$, where each enmity e_i is a pair of different workers that hate each other,
- and a natural number r,

we want to know whether the exists a subset $M \subseteq T$ of size r such that for all $u, v \in M$ with $u \neq v$ we have that $\{u, v\} \notin E$. That is, we want to know whether there exists a set of r workers such that no worker has an enmity towards another one.

(a) (0.25 pts.) Consider the following input for problem VILLAGES:

- $P = \{1, 2, 3, 4, 5, 6\}$
- $C = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{3, 6\}\}$
- $k = 4$

Is it a positive input? Justify your answer.

(b) (0.25 pts.) Consider the following input for the problem ENEMIES:

- $T = \{1, 2, 3, 4, 5\}$
- $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$

- $r = 3$

Is it a positive input? Justify your answer.

(c) (0.75 pts.) Prove that ENEMIES $\in$ NP.

(d) (1.25 pts.) Write a polynomial reduction from VILLAGES to ENEMIES and prove that it is correct.

(e) (0.5 pts.) If we know that VILLAGES is an NP-complete problem, can we claim that ENEMIES is NP-complete? Justify your answer.

Problem 4B **(3 pts.)**
Consider the following two decision problems:

TEAM: given

- a set of workers T,
- a set of wishes to belong to the same team $D = \{d_1, d_2, \ldots, d_m\}$, where each element d_i is a pair of workers that want to work together,
- and a natural number k,

we want to know whether there exists a subset $E \subseteq T$ of size k such that for any pair $u, v \in E$ with $u \neq v$ we have that $\{u, v\} \in D$. That is, we want to know whether there exists a subset of k workers such they all have expressed the desire to work with all other members of the team.

RESEARCHERS: given

- a set of research teams I,
- a set of meetings between pairs of teams $M = \{m_1, m_2, \ldots, m_q\}$, where each meeting m_i is pair of teams,
- and a natural number r,

we want to know whether there exists a subset $S \subseteq I$ of size r such that for any pair $\{u, v\} \in M$ we have that either $u \in S$, or $v \in S$ or both u and v belong to S. That is, we want to know whether we can choose r teams in such a way that in any meeting there is at least one of the chosen teams.

(a) (0.25 pts.) Consider the following input for problem TEAM:

- $T = \{1, 2, 3, 4, 5, 6\}$
- $D = \{\{1,3\}, \{1,4\}, \{1,5\}, \{1,6\}, \{2,3\}, \{2,5\}, \{4,5\}, \{4,6\}, \{5,6\}\}$,
- $k = 4$

Is it a positive input? Justify your answer.

(b) (0.25 pts.) Consider the following input for problem RESEARCHERS:

- $I = \{1,2,3,4,5\}$
- $M = \{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\}\}$,
- $r = 2$

Is it a positive input? Justify your answer.

(c) (0.75 pts.) Prove that RESEARCHERS $\in$ NP.

(d) (1.25 pts.) Write a polynomial reduction from TEAM to RESEARCHERS and prove it is correct.

(e) (0.5 pts.) If we know that TEAM is an NP-complete problem, can we claim that RESEARCHERS is NP-complete? Justify your answer.

Final EDA Exam **Length: 3h** **08/01/2021**

Problem 1 **(2.5 pts.)**

Given a vector v of n natural numbers, we want to determine whether there exists a *dominant* element, that is, an element that appears more than $n/2$ times. For example:

- If $v = \{5, 2, 5, 2, 8, 2, 2\}$, then 2 is the dominant element because it appears $4 > 7/2$ times.
- If $v = \{3, 2, 3, 3, 2, 3\}$, then 3 is the dominant element because it appears $4 > 6/2$ times.
- If $v = \{6, 1, 6, 1, 6, 9\}$, there is no dominant element because no element appears more than $6/2$ times.

We want to write a C++ function that receives a vector v and returns the dominant element of v, or the number -1 if such an element does not exist.

(a) (1.25 pts.) A former EDA student reads the problem and realizes that, if one uses dictionaries, it is possible to obtain a very clean and quite efficient solution:

```
int majority_map (const vector<int>& v) {
  map<int,int> M;
  int n = v.size ();
  for (auto& x : v) {
    ++M[x];
    if (M[x] > n/2) return x;
  }
  return −1;
}
```

If we assume that *map* is implemented as an AVL tree, analyze the code worst-case cost as a function of n.

If we now assume that there is no dominant element, that instead of a *map* we use an *unordered_map*, and that it is implemented as a hash table, what is the average cost of the code as a function of n?

(b) (1.25 pts.) A brilliant student provides us with a solution based on divide and conquer:

```
int times (const vector<int>& v, int l, int r, int x) {
  if (l > r) return 0;
  return (v[l] == x) + times(v,l+1,r,x); }

int majority_pairs (const vector<int>& v, int tie_breaker ) {
  if (v.size () == 0) return tie_breaker ;
  else {
    int n = v.size ();
    if (n % 2 == 1) tie_breaker = v.back ();
    vector<int> aux;
    for (int i = 0; i < n − 1; i+=2)
      if (v[i] == v[i+1]) aux.push_back(v[i ]);
    int cand = majority_pairs (aux, tie_breaker );
    if (cand == −1) return −1;
    int n_times = times(v,0,n−1,cand);
    if (n_times > n/2 or (2*n_times == n and cand == tie_breaker )) return cand;
    else return −1;
  } }

int majority_pairs (const vector<int>& v) {
  return majority_pairs (v,−1);
}
```

Analyze the worst-case cost of a call to *majority_pairs* as a function of n.

Problem 2 **(2 pts.)**
As if we were in a movie, the death of a distant aunt gives us a huge inheritance of M million euros. Immediately, we decide to spend all our money as quickly as possible. To do this, we visit the web page of a real estate agent and make a list of all the apartments we are interested in, with their corresponding price in millions of euros.

Finally, before quitting our job as a computer scientist, we decide to write a program that computes all ways to buy a subset of the apartments we are interested in for **exactly** M million euros.

For example, if $M = 10$ and we store all apartment prices in a vector $p = [4,2,6,2,3]$, then the program should write $\{0,2\}$ i $\{1,2,3\}$. That is, solutions contain the indices in vector p of the apartments we have to buy.

(a) (1 pt.) Complete the following code in order to solve this problem:

```
vector<int> p; // prices of the properties
int money;     // total money we have to spend

void write_choices (vector<int>& partial_sol, int partial_sum, int idx) {
  if ( __________ > __________ ) return;
  if ( ______________________________ ) {
    if (partial_sum == money) {
      cout <<"{";
      for (int i = 0; i < partial_sol.size(); ++i)
        cout << (i == 0 ? "" : ",") << partial_sol[i];
      cout <<"}" << endl;
  } }
  else {
    ____________________________________
    write_choices ( partial_sol , __________ , __________ );
    ____________________________________
    write_choices ( partial_sol , __________ , __________ );
} }

int main() {
  int n;
  cin >> money >> n;
  p = vector<int>(n);
  for (auto& x : p) cin >> x;
  vector<int> partial_sol;
  write_choices ( partial_sol , 0, 0);
}
```

(b) (1 pt.) After running the program on average-sized apartment lists, we realize that the program is too slow. Explain how you would implement some additional pruning mechanism in order to improve the behavior of the program. You do not need to write code, but you have to give enough details so that from your description one can easily implement the pruning mechanism.

Problem 3 **(2.5 pts.)**
Consider the following two decision problems:

COLORABILITY: given

- a graph G with vertices V and edges E,
- and a natural number k

we want to know whether there exists a function $c : V \to \{1,2,\cdots,k\}$ so that for any edge $\{u,v\} \in E$ it holds that $c(u) \neq c(v)$.

DISTINCT-ONES: given

- a set N of natural numbers (maybe with repeated elements),
- and a natural number p

we want to know whether it is possible to distribute the numbers of N in p sets (meaning each number should go to exactly one set), so that if two numbers go to the same set, they cannot both have a 1 in a same position of their binary representation. For example: $8 = 1000_2$ and $5 = 0101_2$ can go to the same set, but $3 = 011_2$ and $6 = 110_2$ cannot because the second bit of both of them is 1.

(a) (0.5 pts.) Consider the following instance of DISTINCT-ONES:

- $N = \{3,6,8,20,22\}$
- $p = 3$

Is it a positive instance? Justify your answer.

(b) (1 pt.) Prove that DISTINCT-ONES $\in$ NP.

(c) (1 pt.) Prove that the following reduction from COLORABILITY to DISTINCT-ONES is a correct polynomial reduction:

Given a instance (G,k) of COLORABILITY, where G has vertices $V = \{v_1, v_2, \cdots, v_n\}$ and edges $E = \{e_0, e_1, \cdots, e_{m-1}\}$, we build the following instance (N, p) of DISTINCT-ONES:

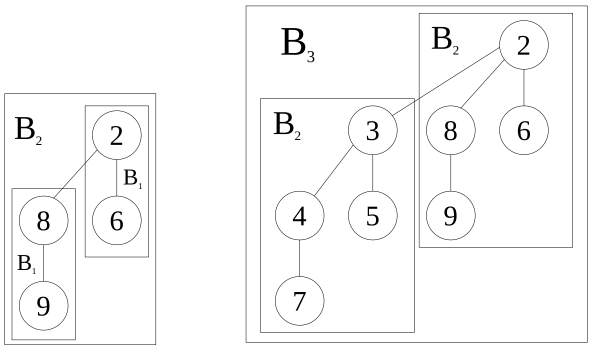
- $N = \{x_1, x_2, \cdots, x_n\}$, where all x_i have m bits and the j-th bit of x_i is 1 if and only if $v_i \in e_j$ for all $0 \leq j < m$. Hence, each x_i corresponds to a vertex v_i of G.
- $p = k$.

Problem 4 **(3 pts.)**

Let us define B_k, a *binomial tree of order k*, as follows:

- B_0 is a tree made up of a single node.
- For all $k > 0$, the tree B_k results from considering a tree B_{k-1} with root r and adding, as the leftmost child of r, another tree B_{k-1}.

In the following figure you can see how trees B_2 and B_3 are formed.

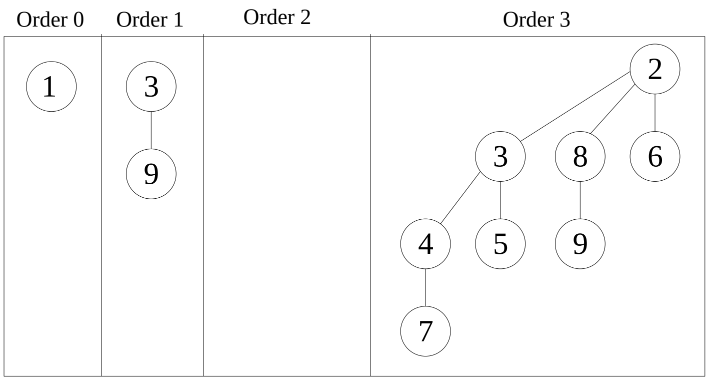


(a) (0.5 pts.) How many nodes has a tree B_k? Prove it formally.

A *binomial heap* is a set of binomial trees that satisfies the following properties:

- Each binomial tree satisfies the min-heap property: the key of a node is greater than or equal to the key of its parent.
- For every $k \geq 0$, there is at most one binomial tree of order k.

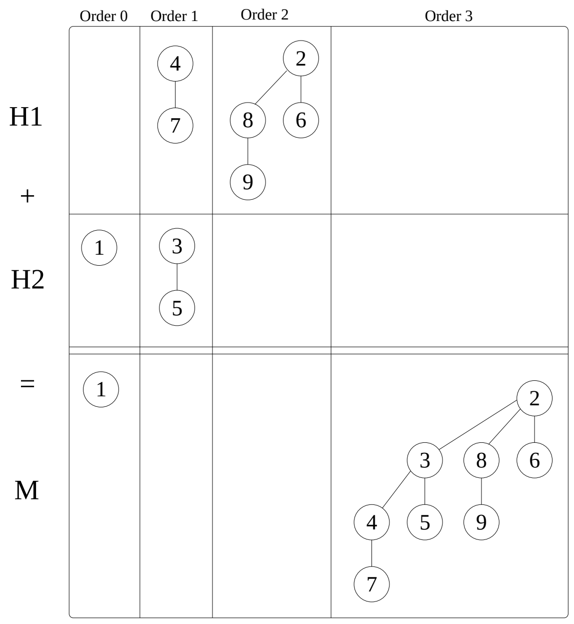
In the following figure, you can see a binomial heap with 11 nodes:



(b) (0.75 pts.) We want to build a binomial heap with n nodes. How many binomial trees of each order will it have? Example: if $n = 11$, it will have one binomial tree of order 0, one of order 1 and one of order 3 (see figure above).

(c) (0.75 pts.) Given two binomial trees A and B of order k that satisfy the min-heap property, how can we combine them in constant time in order to build a binomial tree of order $k + 1$ that satisfies the min-heap property and contains all nodes of A and B?

(d) (1 pt.) One good property of binomial heaps is that we can merge two binomial heaps that have $\Theta(n)$ nodes in time $\Theta(\log n)$. The algorithm is very similar to the addition of binary numbers: we will process the trees from smaller to larger order. For each k, we will merge the trees of order k, considering that we can have a "carry" of order k from the previous sum. You have an example in the following figure:



Complete the following code to merge two binomials heaps with integer keys:

```
class BinomialHeap {
  class Node {
  public:
    int key;
    vector<Node*> children;
    Node(int k, vector<Node*> c) : key(k), children (c){}
  };
  typedef Node* Tree;
  vector<Tree> roots; // roots[k] is the binomial tree of order k, NULL if none
  BinomialHeap(vector<Tree>& r) : roots(r) {}

  // Given t1, t2 binomial trees of order k that satisfy the min-heap property,
  // returns a binomial tree of order k+1 satisfying the min-heap property that
  // contains all elements of t1 and t2. You can use this function in your code.
  Tree mergeTreesEqualOrder(Tree t1, Tree t2);
  void merge(BinomialHeap& h);
public:
  BinomialHeap(){}
  void push(int k);
  void pop();
  int top();  };

void BinomialHeap::merge ( BinomialHeap& h ){
  // Make sure vector roots has same size in both heaps (makes code simpler)
  while (h.roots.size() < roots.size()) h.roots.push_back(NULL);
  while (h.roots.size() > roots.size())  roots.push_back(NULL);
  vector<Tree> newRoots(roots.size());
  Tree carry = NULL;
  for (int k = 0; k < roots.size(); ++k) {
    if (roots[k] == NULL and h.roots[k] == NULL) {
      newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
      carry = $\boxed{\phantom{xxxxxxxx}}$ ; }
    else if (roots[k] == NULL) {
      if (carry == NULL) newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
      else {
        newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
        carry = mergeTreesEqualOrder( $\boxed{\phantom{xxxxxxxx}}$ , $\boxed{\phantom{xxxxxxxx}}$ ); }
    }
    else if (h.roots[k] == NULL) {
      if (carry == NULL) newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
      else {
        newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
        carry = mergeTreesEqualOrder( $\boxed{\phantom{xxxxxxxx}}$ , $\boxed{\phantom{xxxxxxxx}}$ ); }
    }
    else {
      newRoots[k] = $\boxed{\phantom{xxxxxxxx}}$ ;
      carry = mergeTreesEqualOrder( $\boxed{\phantom{xxxxxxxx}}$ , $\boxed{\phantom{xxxxxxxx}}$ ); }
  }

  if (carry $\neq$ NULL) newRoots.push_back( $\boxed{\phantom{xxxxxxxx}}$ );
  roots = newRoots;
}
```

Explain why, if both heaps have $\Theta(n)$ elements, this function takes time $\Theta(\log n)$.

4

Solutions to Mid Term Exams

Solution Midterm Exam EDA 31/03/2016

Proposed solution to problem 1

(a) $\Theta(n^{\log 7})$

(b) The master theorem for subtractive recurrences states that given a recurrence $T(n) = aT(n-c) + \Theta(n^k)$ with $a, c > 0$ and $k \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < 1, \\ \Theta(n^{k+1}) & \text{if } a = 1, \\ \Theta(a^{\frac{n}{c}}) & \text{if } a > 1. \end{cases}$$

(c) $\Theta(n)$

Proposed solution to problem 2

(a) The cost of the program is determined by the sum of the costs of two groups of instructions:

A. The instructions that are executed at each iteration of the loop: the evaluation of the condition $i \leq n$, the call $f(i)$, the evaluation of the condition $i == p$ and the increment i++.

B. The instructions that are executed when the condition of the **if** is true: the call $g(n)$ and the increment p *= 2.

We count separately the contribution to the total cost of the two groups of instructions:

A. If the cost of $f(m)$ is $\Theta(m)$, then at the i-th iteration the cost of **A** is $\Theta(i)$. So in total the contribution to the cost is $\sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$.

B. If the cost of $g(m)$ is $\Theta(m)$, then every time that the condition of the **if** is true the cost of **B** is $\Theta(n)$. Since this happens $\Theta(\log n)$ times, the contribution to the total cost is $\Theta(n \log n)$.

Hence in total the cost of $h(n)$ is $\Theta(n^2) + \Theta(n \log n) = \Theta(n^2)$.

(b) Let us consider the same two groups of instructions of the previous exercise, and again we count separately their contribution to the total cost:

A. If the cost of $f(m)$ is $\Theta(1)$, then the cost of **A** at each iteration is $\Theta(1)$. Since $\Theta(n)$ are performed, the cost in total is $\Theta(n)$.

B. If the cost of $g(m)$ is $\Theta(m^2)$, then every time that the condition of the **if** is true the cost of **B** is $\Theta(n^2)$. Since this happens $\Theta(\log n)$ times, the cost is $\Theta(n^2 \log n)$.

Hence in total the cost of $h(n)$ is $\Theta(n) + \Theta(n^2 \log n) = \Theta(n^2 \log n)$.

Proposed solution to problem 3

(a) The i-th row uses $i+1$ cells ($0 \le i < n$). In total, the consumed space is

$$\sum_{i=0}^{n-1}(i+1) = \sum_{j=1}^{n} j = \Theta(n^2).$$

(b) A possible solution:

```
int p(int x) { return x*(x+1)/2;}

int mystery(int k, int l, int r) {
  if (l+1 == r) return l;
  int m = (l+r)/2;
  if (p(m) ≤ k) return mystery(k, m, r);
  else          return mystery(k, l, m);
}

pair<int,int> row_column(int n, int k) {
  if (k < 0 or k ≥ p(n)) return {−1,−1};
  int i = mystery(k, 0, n);
  return {i, k − p(i)};
}
```

(c) Let $C(N)$ be the cost in the worst case of a call to the function *mystery*$(k,\ l,\ r)$, where $N = r - l + 1$. The recurrence that describes $C(N)$ is

$$C(N) = C(N/2) + \Theta(1),$$

since one makes one recursive call over an interval of half the size, and operations that take constant time. By using the master theorem of divisive recurrences, the solution to the recurrence is $C(N) = \Theta(\log N)$.

Therefore, the cost in the worst case of a call to function *row_column*(n,k) is $\Theta(\log n)$.

(d) The index i of the searched row is the natural number $0 \le i < n$ such that $p(i) \le k < p(i+1)$. We can compute it by solving the second degree equation $p(x) = k$ and taking $i = \lfloor x \rfloor$:

```
int p(int x) { return x*(x+1)/2;}

pair<int,int> row_column(int n, int k) {
  if (k < 0 or k ≥ p(n)) return {−1,−1};
  int i(floor ((sqrt(1.+8*k) − 1)/2));
  return {i, k − p(i)};
}
```

Proposed solution to problem 4

(a) Let us show it by contradiction. Let us assume that i is such that $0 \le i < n-1$ i $f_A(i+1) < f_A(i)$. Let us take $k = i+1$, $j = f_A(i+1)$ and $l = f_A(i)$, so that $0 \le i < k < n$ and $0 \le j < l < n$. Since $j = f_A(k)$, we have that $A_{k,j} \le A_{k,l}$. And as $l = f_A(i)$, we have that $A_{i,l} \leq A_{i,j}$; in fact, since $j < l$, it must be $A_{i,l} < A_{i,j}$. Thus, by adding we have that $A_{i,l} + A_{k,j} < A_{i,j} + A_{k,l}$. This contradicts that A is a Monge matrix.

(b) For each even i, one can compute the column where the leftmost minimum of the i-th row of A appears as follows. From the recursive call over B_1, we have the column j_1 where the leftmost minimum of the i-th row of A appears between columns 0 and $\frac{n}{2} - 1$. Similarly, from the recursive call over B_2, we have the column j_2 where the leftmost minimum of the i-th row of A appears between columns $\frac{n}{2}$ and $n - 1$. Hence, $f_A(i) = j_1$ if $A_{i,j_1} \leq A_{i,j_2}$, and $f_A(i) = j_2$ otherwise.

For each odd i, one determines $f_A(i)$ by examining the columns between $f_A(i-1)$ and $f_A(i+1)$ (defining $f_A(n) = n - 1$ for notational convenience), and choosing the index where the leftmost minimum appears. This has cost $\Theta(f_A(i+1) - f_A(i-1) + 1)$. In total:

$$\sum_{i=1, i \text{ odd}}^{n-1} \Theta(f_A(i+1) - f_A(i-1) + 1) = \Theta((n-1) - f_A(0) + \frac{n}{2}) = \Theta(n)$$

since $0 \leq f_A(0) < n$.

(c) Let $C(n)$ be the cost of the proposed algorithm for computing the function f_A for all rows of a Monge matrix A of size $n \times n$.

Apart from the two recursive calls in step (2) over matrices of size $\frac{n}{2} \times \frac{n}{2}$, steps (1) and (3) require time $\Theta(n)$ in total. So the recurrence that describes the cost is

$$C(n) = 2C(n/2) + \Theta(n).$$

By using the master theorem of divisive recurrences, the solution to the recurrence is $C(n) = \Theta(n \log n)$.

Solution Midterm Exam EDA 07/11/2016

Proposed solution to problem 1

(a)

	Best case	*Average case*	*Worst case*
Quicksort (with Hoare's partition)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion	$\Theta(n)$	**Do not fill**	$\Theta(n^2)$

(b) $\Theta(\sqrt{n})$

(c) $\Theta(\sqrt{n} \log n)$

(d) $\Theta(n)$

(e) Karatsuba's algorithm computes the product of two natural numbers of n bits in time $\Theta(n^{\log_2 3})$.

(f) Strassen's algorithm computes the product of two matrices of size $n \times n$ in time $\Theta(n^{\log_2 7})$.

Proposed solution to problem 2

(a) A possible solution:

```
#include <vector>

using namespace std;

bool dic_search (const vector<int>& a, int l, int r, int x) {
  if (l > r) return false;
  int m = (l+r)/2;
  if (a[m] < x) return dic_search (a, m+1, r, x);
  if (a[m] > x) return dic_search (a, l, m−1, x);
  return true;
}

bool search (const vector<int>& a, int l, int r, int x) {
  if (l+1 == r) return a[l] == x or a[r] == x;
  int m = (l+r)/2;
  if (a[m] ≥ a[l]) {
    if (a[l] ≤ x and x ≤ a[m]) return dic_search (a, l, m, x);
    else                       return search (a, m, r, x);
  }
  else {
    if (a[m] ≤ x and x ≤ a[r]) return dic_search (a, m, r, x);
    else                       return search (a, l, m, x);
} }

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

(b) Let $C(n)$ be the cost of dealing with a vector of size n (be it with function *search* or with function *dic_search*) in the worst case (for example, when the element x does not appear in the sequence). Apart from operations of constant cost (arithmetic computations, comparisons and assignments between integers, vector accesses), exactly one call is made over a vector of size half of that of the input. Therefore, the cost is determined by the recurrence:

$$C(n) = C(n/2) + \Theta(1),$$

which, by the master theorem of divisive recurrences, has solution $C(n) = \Theta(\log(n))$.

Proposed solution to problem 3

(a) It computes m^n.

(b) The cost of the function is determined by the cost of the loop. Each iteration requires time $\Theta(1)$, since only arithmetic operations and integer assignments are performed. Therefore, the cost is proportional to the number of iterations. We observe that if y is even, then y is reduced to half its value. And if y is odd with $y > 1$, then at the next iteration the value $y - 1$ is considered, which is even, and then it is reduced to half its value. Hence if $n \geq 1$ the number of iterations is between $1 + \lfloor \log(n) \rfloor$ and $1 + 2\lfloor \log(n) \rfloor$. Altogether, the cost is $\Theta(\log(n))$.

Proposed solution to problem 4

(a) We have that $\phi - 1 = \frac{\sqrt{5}+1}{2} - 1 = \frac{\sqrt{5}-1}{2}$, and then

$$\phi \cdot (\phi - 1) = \frac{\sqrt{5}+1}{2} \cdot \frac{\sqrt{5}-1}{2} = \frac{(\sqrt{5}+1)(\sqrt{5}-1)}{4} = \frac{5-1}{4} = 1$$

Hence $\phi^{-1} = \phi - 1$.

(b) By induction over n:

- **Base case $n = 0$:** we have $F(0) = 0$, and

$$F(n) = \left. \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} \right|_{n=0} = \frac{1-1}{\sqrt{5}} = 0.$$

- **Base case $n = 1$:** we have $F(1) = 1$, and

$$F(n) = \left. \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} \right|_{n=1} = \frac{\phi + \phi^{-1}}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1.$$

- **Inductive case $n > 1$:** by induction hypothesis,

$$F(n) = F(n-2) + F(n-1) = \frac{\phi^{n-2} - (-\phi)^{-(n-2)}}{\sqrt{5}} + \frac{\phi^{n-1} - (-\phi)^{-(n-1)}}{\sqrt{5}} =$$

$$= \frac{\phi^{n-1}(\phi^{-1}+1) - (-\phi)^{-(n-1)}(-\phi+1)}{\sqrt{5}} = \frac{\phi^{n-1} \cdot \phi - (-\phi)^{-(n-1)}(-\phi^{-1})}{\sqrt{5}} =$$

$$= \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

(c) We have that

$$\lim_{n\to+\infty}\frac{F(n)}{\phi^n}=\lim_{n\to+\infty}\frac{\frac{\phi^n-(-\phi)^{-n}}{\sqrt{5}}}{\phi^n}=\lim_{n\to+\infty}\frac{1-(\frac{-1}{\phi^2})^n}{\sqrt{5}}=\frac{1}{\sqrt{5}}$$

since $\phi>1$, $\phi^2>1$ and $\lim_{n\to+\infty}(\frac{-1}{\phi^2})^n=0$. So $F(n)=\Theta(\phi^n)$.

Solution Midterm Exam EDA 20/04/2017

Proposed solution to problem 1

(a) Insertion sort has cost $\Theta(n)$ when the vector is already sorted, and cost $\Theta(n^2)$ when it is sorted backwards. So the average cost is:

$$(1-\frac{\log n}{n})\Theta(n)+\frac{\log n}{n}\Theta(n^2)=\Theta(n-\log n+n\log n)=\Theta(n\log n)$$

(b) The function returns whether n is a prime number. The cost is determined by the loop, which makes $O(\sqrt{n})$ iterations, each of which has constant cost. Thus the cost is $O(\sqrt{n})$.

(c) Each of the numbers that are multiplied in $n!$ are less than or equal to n. Since we multiply n of them, we have $n! \leq n^n$ for all $n \geq 1$. So $n! = O(n^n)$, taking $n_0 = 1$ and $C = 1$.

(d) Except for 1, all numbers that are multiplied in $n!$ are greater than or equal to 2. Since we multiply $n-1$ of them, we have $n! \geq 2^{n-1} = \frac{1}{2}\cdot 2^n$ for all $n \geq 1$. So $n! = \Omega(2^n)$, taking $n_0 = 1$ and $C = \frac{1}{2}$.

Proposed solution to problem 2

(a) A possible solution:

```
int top_rec (const vector<int>& a, int l, int r) {
  if (l + 1 ≥ r) {
    if (a[l] < a[r]) return r;
    else             return l;
  }
  int m = (l+r)/2;
  if (a[m−1] > a[m]) return top_rec(a, l, m−1);
  if (a[m+1] > a[m]) return top_rec (a, m+1, r);
  return m;
}

int top(const vector<int>& a) {
  return top_rec (a, 0, a.size()−1);
}
```

Let $C(n)$ be the cost in time in the worst case of function *top_rec* on a vector of size n. In the worst case (for example, when the top is in one of the ends of the vector) a recursive call will be made on a subvector of size $n/2$, in addition to operations of constant cost (arithmetic computations, comparisons and assignments of integers, vector accesses). So we have the recurrence $C(n) = C(n/2) + \Theta(1)$, which by the master theorem of divisive recurrences has solution $C(n) = \Theta(\log n)$.

(b) A possible solution that uses the function **int** *top*(**const vector**<**int**>& *a*) of the previous exercise and the function *binary_search* of STL:

```
bool search(const vector<int>& a, int x) {
  int t = top(a);
  int n = a.size();
  if ( binary_search(a. begin(), a. begin() + t,     x)) return true;
  if ( binary_search(a.rbegin(), a.rbegin() + n − t, x)) return true;
  return false;
}
```

In another possible solution, the previous *search* function can be replaced by:

```
bool bin_search_inc(const vector<int>& a, int l, int r, int x) {
  if (l > r) return false;
  int m = (l+r)/2;
  if (a[m] < x) return bin_search_inc(a, m+1, r,   x);
  if (a[m] > x) return bin_search_inc(a,   l, m−1, x);
  return true;
}

bool bin_search_dec(const vector<int>& a, int l, int r, int x) {
  if (l > r) return false;
  int m = (l+r)/2;
  if (a[m] < x) return bin_search_dec(a,   l, m−1, x);
  if (a[m] > x) return bin_search_dec(a, m+1, r,   x);
  return true;
}

bool search(const vector<int>& a, int x) {
  int t = top(a);
  int n = a.size();
  if ( bin_search_inc(a, 0, t−1, x)) return true;
  if ( bin_search_dec(a, t, n−1, x)) return true;
  return false;
}
```

When function *search* searches in a vector of size n, in addition to calling function *top*, one or two binary searches are made, each of which has cost $O(\log n)$, and also operations of constant cost. So the cost of *search* is $O(\log n) + O(\log n) + O(1) = O(\log n)$. Moreover, if for example the top of the sequence is in one of the ends of the vector, then the cost is $\Theta(\log n) + O(\log n) + O(1) = \Theta(\log n)$. So the cost in the worst case is $\Theta(\log n)$.

Proposed solution to problem 3

(a) After m calls to function *reserve* on an initially empty vector, the capacity of the vector is of $C(m) = Am$ elements. So the number of calls to *reserve* after n calls to *push_back* is the least m such that $Am \geq n$, that is, $\lceil \frac{n}{A} \rceil$. Since A is constant, m is $\Theta(n)$.

(b) By induction.

- **Base case:** $m = 0$. We have that $\frac{BA^m - B}{A-1}|_{m=0} = \frac{B-B}{A-1} = 0 = C(0)$.

- **Inductive case:** assuming that it is true for m, let us show it is also true for $m+1$. By induction hypothesis, $C(m) = \frac{BA^m - B}{A-1}$. Then:

$$C(m+1) = AC(m) + B = A\frac{BA^m - B}{A-1} + B = \frac{BA^{m+1} - AB + AB - B}{A-1} = \frac{BA^{m+1} - B}{A-1}$$

(c) After m calls to function *reserve* on an initially empty vector, the capacity of the vector is of $C(m) = \frac{BA^m - B}{A-1}$ elements. So the number of calls to *reserve* after n calls to *push_back* is the least m such that $\frac{BA^m - B}{A-1} \geq n$, that is, $\lceil \log_A(\frac{n(A-1)+B}{B}) \rceil$. Since A and B are constants, m is $\Theta(\log n)$.

Proposed solution to problem 4

(a) A possible solution:

```
int stable_partition (int x, vector<int>& a) {
  int n = a.size();
  vector<int> w(n);
  int i = 0;
  for (int y : a)
    if (y ≤ x) {
      w[i] = y;
      ++i;
    }
  int r = i−1;
  for (int y : a)
    if (y > x) {
      w[i] = y;
      ++i;
    }
  for (int k = 0; k < n; ++k)
    a[k] = w[k];

  return r;
}
```

The cost in time is $\Theta(n)$, since the vector is traversed 3 times, and each iteration of these traversals only requires constant time. The cost in space of the auxiliary memory is dominated by vector w, which has size n. Hence the cost in space is $\Theta(n)$.

(b) The function transposes the two subvectors of a from l to p and from $p+1$ to r: if before the call $a[l..r]$ is $A_l, \ldots, A_p, A_{p+1}, \ldots, A_r$, then after the call $a[l..r]$ is $A_{p+1}, \ldots, A_r, A_l, \ldots, A_p$.

(c) Solution:

```
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 l−1;
  }
  int m = (l+r)/2;
  int p = stable_partition_rec (x, a, l, m);
  int q = stable_partition_rec (x, a, m+1, r);
  mystery(a, p+1, m, q);
  return p+q−m;
}
```

Let $C(n)$ be the cost of function *stable_partition_rec* on a vector of size $n = r - l + 1$ in the worst case. On the one hand two recursive calls are made on vectors of size $n/2$. On the other, the cost of the non-recursive work is dominated by function *mystery*, which takes time which is linear in the size of the vector that is being transposed. Using the hypothesis of the statement (which happens, for example, in a vector in which elements smaller and bigger than x alternate successively), this vector has size $\Theta(n)$. So we have the recurrence $C(n) = 2C(n/2) + \Theta(n)$, which by the master theorem of divisive recurrences has solution $\Theta(n \log n)$. In conclusion, the cost of function *stable_partition* on a vector of size n in the worst case is $\Theta(n \log n)$.

Solution Midterm Exam EDA 06/11/2017

Proposed solution to problem 1

(a) The cost is $\Theta(n\sqrt{n})$.

(b) The cost in the worst case and in the average case is $\Theta(n)$.

(c) $\Theta(\sqrt{n})$.

(d) $\Theta(n^2 \log n)$.

(e) $\Theta(2^n)$.

Proposed solution to problem 2

(a) The value $b[y]$ counts the number of elements of the sequence a that are less than or equal to y.

(b) A possible solution:

```
int median(int n, const vector<int>& b) {
  int l = 0;
  int r = b.size() − 1;
  while (l ≠ r−1) {
    int q = (l+r)/2;
    if   (2*b[q] < n) l = q;
    else              r = q;
  }
  return r;
}
```

(c) At each iteration of the loop, the value $r - l$ is reduced by half. As initially $l = 0$ and $r = m$, we have that $\Theta(\log m)$ iterations are performed. And since each iteration has a constant cost (arithmetic operations with integers, vector accesses, etc.), we conclude that the cost of the function is $\Theta(\log m)$.

Proposed solution to problem 3

(a) A possible solution:

```
complex operator*(const complex& a, const complex& b) {
  return {a.real * b.real − a.imag * b.imag, a.real * b.imag + a.imag * b.real};
}

complex exp(complex z, int n) {
  if (n == 0) return {1, 0};
  complex x = exp(z, n/2);
  complex y = x*x;
  if (n % 2 == 1) y = y*z;
  return y;
}
```

(b) The cost $C(n)$ of the recursive function *exp* as a function of n is determined by the recurrence $C(n) = C(n/2) + \Theta(1)$: a single recursive call is made over a problem of size $n/2$, and the rest of the operations have constant cost. By applying the master theorem of divisive recurrences, we have that the solution is $C(n) = \Theta(\log n)$ as required.

Proposed solution to problem 4

(a) We have that

$$U(m) = T(2^m) = T(2^m/2) + \log(2^m) = T(2^{m-1}) + \log(2^m) = U(m-1) + m$$

(b) By applying the master theorem of subtractive recurrences for solving the recurrence $U(m) = U(m-1) + m$, we have that $U(m) = \Theta(m^2)$.

(c) As $U(m) = \Theta(m^2)$ i $U(m) = T(2^m)$, we have that $T(n) = T(2^{\log n}) = U(\log n) = \Theta((\log n)^2)$.

Solution Midterm Exam EDA 19/04/2018

Proposed solution to problem 1

(a) Function g grows faster:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n^{n^2}}{2^{2^n}} = \lim_{n\to\infty} 2^{\log(\frac{n^{n^2}}{2^{2^n}})} = 2^{\lim_{n\to\infty}\log(\frac{n^{n^2}}{2^{2^n}})} = 0$$

since

$$\lim_{n\to\infty}\log(\frac{n^{n^2}}{2^{2^n}}) = \lim_{n\to\infty}(\log(n^{n^2}) - \log(2^{2^n})) = \lim_{n\to\infty}(n^2\log n - 2^n) = -\infty$$

(b) The cost of the algorithm in the average case is $\Theta(n^2)$:

$$\frac{1}{n^3}\cdot\Theta(n^4) + \frac{1}{n}\cdot\Theta(n^3) + (1 - \frac{1}{n^3} - \frac{1}{n})\cdot\Theta(n) = \Theta(n) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$$

(c) By using the Master Theorem of divisive recurrences we have that $a = 2$, $b = 4$, $\alpha = \log_4 2 = \frac{1}{2}$ and $k = \frac{1}{2}$. So $\alpha = k$, and therefore $T(n) = \Theta(n^\alpha \cdot \log n) = \Theta(\sqrt{n}\cdot\log n)$.

Proposed solution to problem 2

(a) The base case for $k = 0$ is true: $A^0 \cdot x_0 = x_0 = x(0)$. For the inductive case, when $k > 0$, we have by induction hypothesis that $x(k-1) = A^{k-1}\cdot x_0$. So $x(k) = A\cdot x(k-1) = A\cdot(A^{k-1}\cdot x_0) = (A\cdot A^{k-1})\cdot x_0 = A^k\cdot x_0$.

(b) In the first place A^k is computed using the algorithm of fast exponentiation in time $\Theta(\log k)$ (since n is constant). Then the result of multiplying A^k times *x0* is returned, which can be done in time $\Theta(1)$ (again, as n is constant). The cost in time of the algorithm is then $\Theta(\log k)$.

Proposed solution to problem 3

(a) $x = x_2\cdot 3^{2n/3} + x_1\cdot 3^{n/3} + x_0$.

(b) $x\cdot y = x_2y_2\cdot 3^{4n/3} + (x_1y_2 + x_2y_1)\cdot 3^{3n/3} + (x_0y_2 + x_1y_1 + x_2y_0)\cdot 3^{2n/3} + (x_1y_0 + x_0y_1)\cdot 3^{n/3} + x_0y_0$.

(c) $x\cdot y =$

$$\begin{aligned}
&x_2y_2\cdot 3^{4n/3}\\
&+(x_1y_2 + x_2y_1)\cdot 3^{3n/3}\\
&+((x_0+x_1+x_2)\cdot(y_0+y_1+y_2) - x_2y_2 - (x_1y_2+x_2y_1) - (x_1y_0+x_0y_1) - x_0y_0)\cdot 3^{2n/3}\\
&+(x_1y_0 + x_0y_1)\cdot 3^{n/3}\\
&+x_0y_0
\end{aligned}$$

We use 7 products.

(d) To compute the product of x and y of n digits, the algorithm computes recursively $(x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2)$, x_2y_2, $x_1y_2 + x_2y_1$, $x_1y_0 + x_0y_1$ and x_0y_0, which are products of numbers of $n/3$ digits. Then $x \cdot y$ is computed using the equation of the previous section. As numbers are represented in base 3, to multiply by a power of 3 consists in adding zeroes to the right and can be done in time $\Theta(n)$. The involved additions can also be done in time $\Theta(n)$. Therefore the cost $T(n)$ satisfies the recurrence $T(n) = 7T(n/3) + \Theta(n)$, which has solution $\Theta(n^{\log_3 7})$.

Proposed solution to problem 4

(a) We define function $U(m) = T(b^m)$. Then $T(n) = U(\log_b(n))$. Moreover, we have:

$$U(m) = T(b^m) = T(b^m/b) + \Theta(\log^k(b^m)) = T(b^{m-1}) + \Theta(m^k \log^k(b)) =$$

$$= T(b^{m-1}) + \Theta(m^k) = U(m-1) + \Theta(m^k)$$

The Master Theorem of subtractive recurrences claims that if we have a recurrence of the form $U(m) = U(m-c) + \Theta(m^k)$ with $c > 0$ and $k \geq 0$, then $U(m) = \Theta(m^{k+1})$. So the solution to the recurrence of the statement is $T(n) = \Theta((\log_b(n))^{k+1}) = \Theta(\log^{k+1} n)$.

(b) A possible solution:

```
bool search(const vector<int>& a, int x, int l, int r) {
  if (l == r) return x == a[l];
  int m = (l+r)/2;
  auto beg = a.begin();
  if (a[m] < a[m+1])
    return search(a, x, m+1, r) or binary_search(beg + l,    beg + m + 1, x);
  else
    return search(a, x, l,    m) or binary_search(beg + m+1, beg + r + 1, x);
}

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

(c) The worst case takes place for instance when x does not appear in a. In this situation the cost $T(n)$ is described by the recurrence $T(n) = T(n/2) + \Theta(\log n)$, as we make one recursive call over a vector of size $\frac{n}{2}$, and the cost of the non-recursive work is dominated by the binary search, which has cost $\Theta(\log(\frac{n}{2})) = \Theta(\log(n))$. By applying the first section we have that the solution is $T(n) = \Theta(\log^2(n))$.

Solution Midterm Exam EDA 05/11/2018

Proposed solution to problem 1

(a) The answers:

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
TRUE	X		X	X		X			X	X
FALSE		X			X		X	X		

Proposed solution to problem 2

(a) The cost in the worst case is $\Theta(n^3)$. The worst case always happens.

(b) The cost in the best case is $\Theta(n^2)$. The best case happens, for example, when the two matrices only have *true* in their coefficients.

The cost in the worst case is $\Theta(n^3)$. The worst case happens, for example, when the two matrices only have *false* in their coefficients.

(c) The function considers the Boolean matrices as matrices of integers, in which *false* is interpreted as 0 and *true* as 1. Then we apply Strassen's algorithm for the product of matrices. Let M be the resulting matrix. The coefficient of the i-th row and j-th column of M is

$$m_{ij} = \sum_{k=0}^{n-1} (a_{ik} \cdot b_{kj}).$$

As the coefficients of the input matrices are 0 or 1, the product as integer numbers is the same as the logical $\wedge$ operation. So m_{ij} counts the number of pairs (a_{ik}, b_{kj}) where both a_{ik} and b_{kj} are true at the same time. Therefore, to obtain the logical product we only have to define p_{ij} as 1 if $m_{ij} > 0$, and 0 otherwise. The cost is dominated by Strassen's algorithm, which has cost $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.8})$.

Proposed solution to problem 3

(a) Given an integer $n \geq 0$, function *mystery* computes $\lfloor \sqrt{n} \rfloor$.

(b) First of all we find the cost $C(N)$ of the function *mystery_rec* in terms of the size $N = r - l + 1$ of the interval $[l, r]$. This cost follows the recurrence $C(N) = C(N/2) + \Theta(1)$, as at each call a recursive call is made on an interval with half the elements, and there is additional non-recursive work with constant cost. According to the master theorem of divisive recurrences, the solution to this recurrence is $\Theta(\log N)$. As *mystery*(n) consists in calling *mystery_rec*$(n,\ 0,\ n{+}1)$, we can conclude that the cost of *mystery*(n) is $\Theta(\log n)$.

Proposed solution to problem 4

(a) $\Theta(n)$

(b) $\Theta(n^2)$

(c) $\Theta(n \log n)$

(d) $\Theta(n \log n)$

(e) A possible solution:

```
void my_sort(vector<int>& v) {
  int n = v.size ();
  double lim = n * log(n);
  int c = 0;
  for (int i = 1; i < n; ++i) {
    int x = v[i];
    int j;
    for (j = i; j > 0 and v[j − 1] > x; −−j) {
      v[j] = v[j − 1];
      ++c;
    }
    v[j] = x;
    if (c > lim) {
      merge_sort(v);
      return;
    }
  }
}
```

(f) First of all we observe that, as the outermost **for** loop makes $n-1$ iterations, each of which has cost $\Omega(1)$, the cost of the overall execution is $\Omega(n)$.

If for example the vector is already sorted in increasing order, then *my_sort* behaves as insertion sort: the innermost **for** loop is never entered, c is always 0, and *merge_sort* is not called. As each iteration of the outermost **for** loop has constant cost, the cost in this case is $\Theta(n)$. So the cost of *my_sort* in the best case is $\Theta(n)$.

To see the cost in the worst case, we distinguish two cases:

- Suppose that *merge_sort* is not called. Then the cost is proportional to the final value of variable c. As *merge_sort* is not called, we have that $c \leq n \ln n$, and the cost is $O(n \ln n) = O(n \log n)$.
- Suppose that *merge_sort* is eventually called. If it is called at the end of the first iteration of the outermost **for** loop, then the cost is $O(n)$ from the innermost **for** loop plus $\Theta(n \log n)$ of the *merge_sort*. Altogether, the cost is $\Theta(n \log n)$.

  If *merge_sort* is called at the end of the second, or third, etc. iteration of the outermost **for** loop, then *merge_sort* was not called at the previous iteration of the outermost **for** loop. Moreover, in the last iteration c may have increased in i at most, so when *merge_sort* is called we have that $n \ln n < c \leq i + n \ln n \leq n + n \ln n \leq n \ln n + n \ln n = 2n \ln n$ (for n big enough), and hence $c = \Theta(n \log n)$. Since the cost of *merge_sort* is $\Theta(n \log n)$, altogether the cost is $\Theta(n \log n)$.

The worst case happens for example when the vector is sorted backwards, that is, in decreasing order. As in this case insertion sort has cost $\Theta(n^2)$, at some point of the execution of *my_sort* the subprocedure *merge_sort* will be called, and by the previous reasoning the cost will be $\Theta(n \log n)$.

Solution Midterm Exam EDA 25/04/2019

Proposed solution to problem 1

(a) The answers:

(1) $f = O(g)$: if $\alpha > 1$.

(2) $f = \Omega(g)$: if $\alpha \leq 1$.

(3) $g = O(f)$: if $\alpha \leq 1$.

(4) $g = \Omega(f)$: if $\alpha > 1$.

(b) $T(n) = \Theta(2^{\frac{n}{2}}) = \Theta(\sqrt{2}^n)$

(c) $T(n) = \Theta(n \log n)$

(d) $\Theta(n \log n)$

(e) $\Theta(n \log n)$

Proposed solution to problem 2

(a) A call *mystery*(v, 0, $v.size()-1$, m) permutes the elements of v so that (at least) the m first positions of v contain the m smallest elements, sorted in increasing order.

(b) When one makes a call *mystery*(v, 0, $n-1$, n), first **int** q = *partition* (v, l, r) is executed with $l = 0$ and $r = n-1$, with cost $\Theta(n)$. Moreover, as the vector contains different integers sorted in increasing order and the function *partition* takes as a pivot the left-most element, we have $q = 0$. Hence we have that the call *mystery*(v, l, q, m) is made with $q = l = 0$ and so takes constant time. Besides, $p = 1$. If $n > 1$ finally a recursive call *mystery*(v, $q+1$, r, $m-p$) is made, where $q+1 = 1$, $r = n-1$ and $m-p = n-1$.

In turn, when executing this call the cost of **int** q = *partition* (v, l, r) is $\Theta(n-1)$, again the cost of *mystery*(v, l, q, m) is $\Theta(1)$, and if $n-1 > 1$ again a recursive call *mystery*(v, $q+1$, r, $m-p$), is made, where $q+1 = 2$, $r = n-1$ and $m-p = n-2$.

Repeating the argument, we see that the accumulated cost is

$$\Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2).$$

Proposed solution to problem 3

(a) A possible solution:

```
vector<bool> prod(const vector<bool>& x, const vector<bool>& y) {

  if (x.size() == 0 or y.size() == 0) return vector<bool>();

  vector<bool> z = twice(twice(prod(half(x), half(y))));
  vector<bool> one = vector<bool>(1, 1);
```

```
    if      ($x.back()$ == 0 and $y.back()$ == 0) return $z$;
    else if ($x.back()$ == 1 and $y.back()$ == 0) return $sum(z, y)$;
    else if ($y.back()$ == 1 and $x.back()$ == 0) return $sum(z, x)$;
    else {
      vector<bool> $x2$ = $twice(half(x))$;
      vector<bool> $y2$ = $twice(half(y))$;
      return $sum(sum(sum(z, x2), y2), one)$;
    }
  }
```

Let $T(n)$ be the cost of *prod*(x,y) if $n = x.size() = y.size()$. Only one recursive call is made, over vectors of size $n-1$. The non-recursive work has cost $\Theta(n)$. So the cost follows the recurrence

$$T(n) = T(n-1) + \Theta(n).$$

According to the Master Theorem of Subtractive Recurrences, the solution of this recurrence behaves asymptotically as $\Theta(n^2)$. Hence, the cost is $\Theta(n^2)$.

(b) Karatsuba algorithm, which has cost $\Theta(n^{\log 3})$.

Proposed solution to problem 4

(a) A possible way of completing the code:

```
    bool search1(int $x$, const vector<vector<int>>& $A$, int $i$, int $j$, int $n$) {
    if ($n$ == 1) return $A[i][j]$ == $x$;
    int $mi$ = $i + n/2 - 1$;
    int $mj$ = $j + n/2 - 1$;
    if ($A[mi][mj] < x$) return
        search1($x$, $A$, $mi$+1,     $j$, $n/2$) or
        search1($x$, $A$, $mi$+1, $mj$+1, $n/2$) or
        search1($x$, $A$,      $i$, $mj$+1, $n/2$);
    if ($A[mi][mj] > x$) return
        search1($x$, $A$, $mi$+1,      $j$, $n/2$) or
        search1($x$, $A$,      $i$,      $j$, $n/2$) or
        search1($x$, $A$,      $i$, $mj$+1, $n/2$);
    return true;
  }
```

We note that the result of calling *search1* $(x,\ A,\ 0,\ 0,\ N)$ is **true** if and only if x occurs in A.

To analyse the cost of this call, first of all we study the general case of calling *search1* $(x,\ A,\ i,\ j,\ n)$ as a function of n. Let $T(n)$ be the cost in the worst case of this call. As in the worst case 3 calls are made with last parameter $n/2$ and the cost of the non-recursive work is constant, we have the recurrence:

$$T(n) = 3T(n/2) + \Theta(1)$$

Applying the Master Theorem of Divisive Recurrences, we have $T(n) = \Theta(n^{\log_2 3})$.

Hence, the cost of calling *search1* $(x,\ A,\ 0,\ 0,\ N)$ is $\Theta(N^{\log_2 3})$.

(b) It is correct. Let us see that the loop maintains the following invariant: if x occurs in A, then the occurrence is between rows 0 and i (included), and between columns j and $N-1$ (included). At the beginning of the loop, the invariant holds. And at each iteration it is preserved:

- If $A[i][j] > x$ then x cannot occur at row i: by the invariant we have that if x occurs then it must be in a column from j to $N-1$. But if $j \leq k < N$ then $A[i][k] \geq A[i][j] > x$.
- If $A[i][j] < x$ then x cannot occur at column j: by the invariant we have that if x occurs then it must be in a row from 0 to i. But if $0 \leq k \leq i$ then $A[k][j] \leq A[i][j] < x$.

Finally, if the program returns **true** with the **return** of inside the loop, the answer is correct as $A[i][j] = x$. If it returns **false** with the **return** of outside the loop, then $i < 0$ or $j \geq N$. In any case, by the invariant we deduce that x does not appear in A.

(c) At each iteration either i is decremented by 1, or j is incremented by 1, or we return. Moreover, initially i has value $N-1$, and at least has value 0 before exiting the loop. Similarly, at the beginning j has value 0 and at most has value $N-1$ before exiting the loop. As in the worst case we can make $2N-1$ iterations and each of them has cost $\Theta(1)$, the cost is $\Theta(N)$.

Solution Midterm Exam EDA 11/11/2019

Proposed solution to problem 1

(a) $\Theta(\sqrt{n}\log n)$

(b) We compute:

$$\lim_{n\to\infty}\frac{\log(\log n^2)}{\log n} = \lim_{n\to\infty}\frac{\log(2\log n)}{\log n} = \lim_{n\to\infty}\frac{\log 2 + \log(\log n)}{\log n} =$$

$$= \lim_{n\to\infty}\frac{\log 2}{\log n} + \lim_{n\to\infty}\frac{\log(\log n)}{\log n} = \lim_{n\to\infty}\frac{\log(\log n)}{\log n}$$

With the variable renaming $n = 2^m$ we have that the previous limit is equal to

$$\lim_{m\to\infty}\frac{\log(\log 2^m)}{\log 2^m} = \lim_{m\to\infty}\frac{\log m}{m} = 0$$

Hence it only holds that $\log(n) \in \Omega(\log(\log(n^2)))$

Proposed solution to problem 2

(a) It returns $f \circ g$, the function composition of f with g. The cost of *mystery* is the cost of the auxiliary function *mystery_aux*, that is given by the recurrence $T(n) = T(n-1) + \Theta(1)$, which has asymptotic solution $T(n) \in \Theta(n)$.

(b) It returns f^k. That is, a function such that $f^k(x) = \underbrace{f(f(\ldots(f}_{k}(x))))$. As a function of k, its cost is given by the recurrence $T(k) = T(k-1) + \Theta(1)$, with solution $\Theta(k)$.

(c)

```
vector<int> mystery_2_quick(const vector<int>& f, int k) {
   if (k == 0) {
      vector<int> r(f.size ());
      for (int i = 0; i < f.size (); ++i) r[i] = i;
      return r;
   }
  else if (k%2 == 0) {
    vector<int> aux = mystery_2_quick(f,k/2);
    return mystery(aux,aux);
  }
  else {
    vector<int> aux = mystery_2_quick(f,k/2);
    return mystery(f,mystery(aux,aux));
  }
}
```

The recurrence that describes the cost of this function is $T(k) = T(k/2) + \Theta(1)$, which has solution $\Theta(\log k)$.

Proposed solution to problem 3

(a) It is not difficult to see that function *max_sum* essentially implements selection-sort, that we know has worst-case cost of $\Theta(m^2)$. The only difference is in the line where we update *sum*, that takes constant time and is executed m times. Hence, the total cost is $\Theta(m^2) + \Theta(m) = \Theta(m^2)$.

(b) If we understand the code, we can realize that it sorts the elements in S from larger to smaller and then multiplies them pairwise following this order. In order to improve the efficiency, we only have to sort the vector with *merge sort*, so that the cost is $\Theta(m \log m)$, and multiply the elements pairwise in the resulting order. The cost will be $\Theta(m \log m)$.

(c) Assume that x_0 i x_1 are the two largest elements in S and consider an expression that contains the products $x_0 * y$ and $x_1 * z$, for some $y,z \in S$. What we will do is to replace these two products by $x_0 * x_1$ and $y * z$. We know observe that $(x_0 * x_1 + y * z) - (x_0 * y + x_1 * z) = x_0(x_1 - y) + (y - x_1)z = x_0(x_1 - y) - (x_1 - y)z = (x_0 - z)(x_1 - y) > 0$. The last step is due to the fact that $x_0 > z$ and $x_1 > y$ since x_0 and x_1 are the largest elements in S, and they are all different. Hence the original expression was not maximum since the resulting expression after the replacement is larger.

Let us now prove the correctness of *max_sum* by induction on m:

- *Base case* ($m = 0$). The algorithm is correct since it returns 0, the maximum possible sum of products.
- *Induction step.* Let $m > 0$ and let us assume the induction hypothesis: the maximum expression for a set with $< m$ elements can be obtained by sorting the elements from larger to smaller and pairing them consecutively in this order. If we sort the m elements $x_0 > x_1 > x_2 > x_3 > \cdots > x_{m-1}$, we know, by the previous property, that the maximum expression contains the product $x_0 * x_1$ followed by an expression formed by the numbers $\{x_2, x_3, \ldots, x_{m-1}\}$. Obviously, this expression will be the largest we can form with $\{x_2, x_3, \ldots, x_{m-1}\}$ and by applying the induction hypothesis we know it will be $x_2 * x_3 + \cdots + x_{m-2} * x_{m-1}$. Hence, the maximum expression is $x_0 * x_1 + x_2 * x_3 + \cdots + x_{m-2} * x_{m-1}$, as we wanted to prove.

Proposed solution to problem 4

(a)

```
int f(const vector<int>& p, int l,  int r){
  if (l + 1 ≥ r) return (p[l] ≤ p[r] ? l : r);
  else {
    int m = (l+r)/2;
    if (p[m] > p[m+1]) return f(p,m+1,r);
    else if (p[m−1] < p[m]) return f(p,l,m−1);
    else return m;
  }
}

pair<int,int> max_profit (const vector<int>& p) {
  return {f(p,0,p.size()−1), p.size()−1};
}
```

The cost of *max_profit* will be the cost of f, whose cost is given by the recurrence $T(n) = T(n/2) + \Theta(1)$, from which we obtain the cost $\Theta(\log n)$.

(b)

```
int max_profit (const vector<int>& p, int k) {
  int m = p[k];
  for (int i = k − 1; i ≥ 0; −−i)
    m = min(m,p[i]);

  int M = p[k];
  for (int i = k + 1; i < p.size (); ++i)
    M = max(M,p[i]);

  return M − m;
}
```

(c) We can use a divide-and-conquer approach. Given a vector p we consider its middle point m and we split the vector into two equal-sized parts. Recursively, we compute the maximum profit if we buy and sell in the left part of the vector, and then, also recursively, the maximum profit if we buy and sell in the right part of the vector. Finally, using the function in b) we compute the maximum profit of a period that includes the middle point m (that is, we buy in the left part of the vector and we sell in the right). The final result is the maximum of the three computed profits.

We have designed a divide-and-conquer algorithm where there are two recursive calls where we half the size, and we then perform a linear amount of work in order to compute the maxim profit that includes point m. Hence, the recurrence that determines the cost is $T(n) = 2T(n/2) + \Theta(n)$, which has asymptotic solution $\Theta(n \log n)$.

Remark: there are more efficient solutions not necessarily based on divide and conquer.

Solution Midterm Exam EDA 23/04/2020

Proposed solution to problem 2

Cruises, X35804:

```
#include <iostream>
#include <map>
using namespace std;

struct Info {
  string code;
  int price;
};

int main() {
  map<int, Info> M;
  char c;
  while (cin >> c) {
    if (c == 'n') {
      cout << "num: " << M.size() << endl;
    }
    else if (c == 'u') {
      string code;
      int length, price;
      cin >> code >> length >> price;
      Info inf; inf.price = price; inf.code = code;
      M[length]={code,price};
    }
    else if (c == 'q') {
      int length;
      cin >> length;
      if (M.count(length) == 1) cout << M[length].price << endl;
      else cout << −1 << endl;
    }
    else if (c == 'p') {
      cout << string(10, '−') << endl;
      for (auto& p : M)
        cout << p.second.code << " " << p.first << " " << p.second.price << endl;
      cout << string(10, '*') << endl;
    }
    else {
      if (M.size() < 2) cout << "no" << endl;
      else {
        auto it = M.begin(); ++it;
        cout << it−>second.code << " " << it−>first << " " << it−>second.price << endl;
      }
    }
  }
}
```

Donations, X22314:

```
#include <iostream>
#include <map>
using namespace std;


int main() {
  map<string, int> M;
  char c;
  while (cin ≫ c) {
    if (c == 'N') {
      cout ≪ "number: " ≪ M.size() ≪ endl;
    }
    else if (c == 'D') {
      string nif;
      int money;
      cin ≫ nif ≫ money;
      M[nif] += money;
    }
    else if (c == 'Q') {
      string nif;
      cin ≫ nif;
      if (M.count(nif) ≠ 0) cout ≪ M[nif] ≪ endl;
      else cout ≪ −1 ≪ endl;
    }
    else if (c == 'P') {
      bool primer = true;
      for (auto& p : M) {
        if ((p. first [p. first .length()−2] − '0')%2 == 0) {
          cout ≪ (primer?"":" ") ≪ p.first ;
          if (primer) primer = false;
        }
      }
      cout ≪ endl;
    }
    else { // c == 'L'
      if (M.size() == 0) cout ≪ "NO LAST NIF" ≪ endl;
      else {
        auto it = M.end(); −−it;
        cout ≪ it−>first ≪ " " ≪ it−>second ≪ endl;
      }
    }
  }
}
```

Anniversaries, X79163:

```
#include <iostream>
#include <map>
using namespace std;


struct Data {
  string event;
  int relevance ;
};


int main() {
  int maximum_relevance = 0;
  map<string, Data> M;
  char c;
  while (cin >> c) {
    if (c == 'n') {
      cout << "number events: " << M.size() << endl;
    }
    else if (c == 's') {
      string date, event;
      int relevance;
      cin >> date >> event >> relevance;
      if (M.count(date)) cout << "ERROR: repeated date" << endl;
      else {
        M[date] = {event, relevance};
        if (relevance > maximum_relevance) maximum_relevance = relevance;
      }
    }
    else if (c == 'a') {
      string date;
      cin >> date;
      cout << M[date].event << endl;
    }
    else if (c == 'm') {
      cout << "maximum relevance: " << maximum_relevance << endl;
    }
    else { // c == 'e'
      if (M.size() < 2) cout << "ERROR: at least two events needed" << endl;
      else {
        int total = 0;
        auto it = M.end();
        --it;
        total += it->second.relevance;
        it = M.begin();
        total += it->second.relevance;
        cout << total << endl;
      }
    }
  }
}
```

Proposed solution to problem 3

Cool vector, X30043:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int position (const vector<int>& v, int e, int d) {
  if (e+1 == d) return e;
  int m = (e+d)/2;
  if (v[e] ≥ v[m]) return position (v, m, d);
  else             return position (v, e, m);
}

int search (int x, const vector<int>& v, int e, int d) {
  if (e > d) return −1;
  if (e == d) return (v[e] == x ? e : −1);
  int m = (e + d)/2;
  if (x < v[m]) return search (x, v, m + 1, d);
  else          return search (x, v, e,     m);
}

int search (int x, const vector<int>& v) {
  int n = v.size ();
  int j = position (v, 0, n−1);
  int p = search (x, v, 0, j);
  if (p ≠ −1) return p;
  return search (x, v, j+1, n−1);
}
```

Cool vector, X74873:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int position (const vector<int>& v, int e, int d) {
  if (e+1 == d) return e;
  int m = (e+d)/2;
  if (v[e] ≥ v[m]) return position (v, m, d);
  else             return position (v, e, m);
}

int search (int x, const vector<int>& v, int e, int d) {
  if (e > d) return −1;
  if (e == d) return (v[e] == x ? e : −1);
  int m = (e + d + 1)/2;
  if (x > v[m]) return search(x, v, e, m − 1);
  else          return search(x, v, m, d);
}

int search(int x, const vector<int>& v) {
  int n = v.size();
  int j = position(v, 0, n−1);
  int p = search(x, v, 0, j);
  if (p ≠ −1) return p;
  return search(x, v, j+1, n−1);
}
```

Cool vector, X83303:

```
#include <iostream>
#include <vector>
using namespace std;

int position(const vector<int>& v, int e, int d) {
  if (e+1 == d) return e;
  int m = (e+d)/2;
  if (v[e] ≤ v[m]) return position(v, m, d);
  else             return position(v, e, m);
}

int search(int x, const vector<int>& v, int e, int d) {
  if (e > d) return −1;
  if (e == d) return (v[e] == x ? e : −1);
  int m = (e + d)/2;
  if (x > v[m]) return search(x, v, m + 1, d);
  else          return search(x, v, e,     m);
}

int search(int x, const vector<int>& v) {
  int n = v.size();
  int j = position(v, 0, n−1);
  int p = search(x, v, 0, j);
  if (p ≠ −1) return p;
  return search(x, v, j+1, n−1);
}
```

Cool vector, X90362:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int position(const vector<int>& v, int e, int d) {
  if (e+1 == d) return e;
  int m = (e+d)/2;
  if (v[e] ≤ v[m]) return position (v, m, d);
  else             return position (v, e, m);
}

int search (int x, const vector<int>& v, int e, int d) {
  if (e > d) return −1;
  if (e == d) return (v[e] == x ? e : −1);
  int m = (e + d + 1)/2;
  if (x < v[m]) return search (x, v, e, m − 1);
  else          return search (x, v, m, d);
}

int search (int x, const vector<int>& v) {
  int n = v. size ();
  int j = position (v, 0, n−1);
  int p = search (x, v, 0, j );
  if (p ≠ −1) return p;
  return search (x, v, j+1, n−1);
}
```

Solution Midterm Exam EDA 06/11/2020

Proposed solution to problem 1

(a)
```
bool tri_search (const vector<int>& v, int l, int r, int x) {
  if (l > r) return false;
  else {
    int n_elems = (r−l+1);
    int f = l + n_elems/3;
    int s = r − n_elems/3;
    if (v[f] == x or v[s] == x) return true;
    if (x < v[f]) return tri_search (v,l,f−1,x);
    if (x < v[s]) return tri_search (v,f+1,s−1,x);
    else return tri_search (v,s+1,r,x);
  }
}
```

In the worst case, all recursive calls are made until $l > r$. The recurrence that expresses the cost of the program in this case is:

$T(n) = T(n/3) + \Theta(1)$

that has solution $T(n) \in \Theta(\log n)$.

(b) Let $f = n(\log n)^{1/2}$ and $g = n(\log n)^{1/3}$.

In order to see that they are $\Omega(n)$ and not $\Theta(n)$ we only have to see that the following limits are infinite:

$$\lim_{x\to\infty} \frac{n(\log n)^{1/2}}{n} = \lim_{x\to\infty} (\log n)^{1/2} = \infty$$

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n} = \lim_{x\to\infty} (\log n)^{1/3} = \infty$$

In order to see that they are $O(n \log n)$ and not $\Theta(n \log n)$ we only have to see that the following limits are zero:

$$\lim_{x\to\infty} \frac{n(\log n)^{1/2}}{n \log n} = \lim_{x\to\infty} \frac{(\log n)^{1/2}}{\log n} = \lim_{x\to\infty} \frac{1}{(\log n)^{1/2}} = 0$$

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n \log n} = \lim_{x\to\infty} \frac{(\log n)^{1/3}}{\log n} = \lim_{x\to\infty} \frac{1}{(\log n)^{2/3}} = 0$$

Finally, $f \notin \Theta(g)$ can be proved by seeing that the following limit is not a strictly positive constant:

$$\lim_{x\to\infty} \frac{n(\log n)^{1/3}}{n(\log n)^{1/2}} = \lim_{x\to\infty} \frac{(\log n)^{1/3}}{(\log n)^{1/2}} = \lim_{x\to\infty} \frac{1}{(\log n)^{1/6}} = 0$$

Proposed solution to problem 2

(a) The idea of this algorithm is that, whenever we find a natural number, we count its posterior occurrences in the vector and mark them with -1 in order not to count them again. Hence, when we visit an element marked with -1 we can avoid the internal loop.

If we build a vector where all numbers are different, then this optimitzation is useless. Moreover, if all elements are different, there is no dominant element (unless $n = 1$) and the two loops are executed the maximum number of times. Hence, we are in the worst-case scenario. The body of the inner loop is constant, and hence we only need to count how many times it is executed. Given an i, the inner loop is executed $n - i$ times. Since i ranges from 0 to $n - 1$, the total cost is $n + (n - 1) + (n - 2) + \cdots + 1 = \Theta(n^2)$.

The best case occurs, for exemple, in a vector with one single element repeated n times. In this case, when $i = 0$ we visit and mark all elements with -1. For all other i, the inner loop will not be executed. Hence, the best-case cost is $\Theta(n)$.

If we know that v has at most 100 diferent numbers, then the inner loop will be executed at most 100 times. That is, there will be at most 100 i's for which the inner loop will be executed. These i will contribute in the worst case a cost of $\Theta(100n) = \Theta(n)$. For the remaining i's (we have at most n), the inner loop will not be executed and hence, they will contribute a cost of at most $\Theta(n)$. Hence, the worst-case has changed and is now $\Theta(n)$.

(b) For this exercise we first remind that insertion sort has worst-case cost $\Theta(n^2)$ and best-case cost $\Theta(n)$. For quicksort, worst-case cost is $\Theta(n^2)$ and best-case cost is $\Theta(n \log n)$.

In order to analyze the cost of *dominant_sort*, let us first ignore the call to *own_sort*. The rest of the loop visits each element at most once, doing there a constant amount of work. We want to remark that sometimes not all elements are visited, since the code stops when the dominant element is found. The worst case takes place when all elements are visited and no dominant element is found (this takes $\Theta(n)$ time). The best case happens when the first visited element is the dominant one but we can see that, in order to detect that this is the case, it has to visit at least $n/2$ elements. Hence, the best-case cost is also $\Theta(n)$. Hence, if we ignore the call to *own_sort*, the code always takes $\Theta(n)$ time.

If *own_sort* is insertion sort, best-case time is $\Theta(n) + \Theta(n) = \Theta(n)$, and worst-case time is $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.

If *own_sort* is quicksort, best-case time is $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$, and worst-case time is $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.

(c) The completed code is:

```
int dominant_divide (const vector<int>& v, int l, int r) {
  if (l == r) return v[l];
  int n_elems = (r−l+1);
  int m = (l+r)/2;
  int maj_left  = dominant_divide(v, l, m);
  if ( maj_left  ≠ −1 and times(v, l, r, maj_left ) > n_elems/2) return maj_left ;
  int maj_right = dominant_divide(v, m+1, r);
  if (maj_right ≠ −1 and times(v, l, r, maj_right) > n_elems/2) return maj_right;
  return −1; }
```

In order to analyze its cost, we realize the in the worst case two recursive calls are made, each of them with half the size. Also, two calls to *times* are made. The rest of the code takes constant time. If we pay attention to function *times*, we know that its cost can be described by $T(n) = T(n-1) + \Theta(1)$, that has solution $T(n) \in \Theta(n)$. Hence, the recurrence that describes the cost in worst case of this program is

$$T(n) = 2T(n/2) + \Theta(n)$$

that has solution $T(n) \in \Theta(n \log n)$.

5

Solutions to Lab Exams

Solution to Lab Exam EDA - shift 1 13/12/2010

Shift 1

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <queue>
#include <cassert>

using namespace std;

const int N_DIRS = 4;
const int di[N_DIRS] = { 1,  0, −1,  0};
const int dj[N_DIRS] = { 0,  1,  0, −1};

struct Pos {
  int i, j;
  Pos(int ii = −1, int jj = −1) : i(ii), j(jj) { }
};

bool ok(int n, int m, int i, int j) {
  return
    0 ≤ i and i < n and
    0 ≤ j and j < m;
}

int search(const vector < vector<char> >& map, int n, int m, int i0, int j0) {
  const int MINFTY = −1;
  vector< vector<int> > dist(n, vector<int> (m, MINFTY));
  queue<Pos> q;
  int max_dist = MINFTY;
  q.push(Pos(i0,j0));
  dist[i0][j0] = 0;
  while (not q.empty()) {
    Pos p = q.front();
    q.pop();
    int i = p.i;
    int j = p.j;
    for(int k = 0; k < N_DIRS; ++k) {
      int ii = i + di[k];
      int jj = j + dj[k];
      if (ok(n,m,ii,jj) and map[ii][jj] ≠ 'X' and dist[ii][jj] == MINFTY) {
        q.push(Pos(ii,jj));
        dist[ii][jj] = 1 + dist[i][j];
        if (map[ii][jj] == 't') max_dist = dist[ii][jj];
      }
    }
  }
  return max_dist;
}

int main(void) {
  int n, m;
  cin >> n >> m;
  vector < vector<char> > map(n, vector<char>(m));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      cin >> map[i][j];
  int f, c;
  cin >> f >> c;
  --f;
  --c;
  int dist = search(map, n, m, f, c);
  if (dist >= 0) cout << "distancia maxima: " << dist << endl;
  else           cout << "no es pot arribar a cap tresor" << endl;
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<bool> Vec;
typedef vector<Vec > Mat;

void next(int i, int j, int n, int& ni, int& nj) {
  if (j < n-1) {
    ni = i;
    nj = j+1;
  }
  else {
    ni = i+1;
    nj = 0;
  }
}

const int N_DIRS = 8;
const int DI[N_DIRS] = { -1, -1,  0,  1,  1,  1,  0, -1};
const int DJ[N_DIRS] = {  0, -1, -1, -1,  0,  1,  1,  1};

bool ok(int i, int j, int n) {
  return
    0 <= i and i < n and
    0 <= j and j < n;
}
```

```
bool safe (int i, int j, int n, Mat& m) {
  for (int k = 0; k < 8; ++k) {
    int ii = i + DI[k];
    int jj = j + DJ[k];
    if (ok(ii, jj, n) and m[ii][jj]) return false;
  }
  return true;
}

void escriu (int n, const Mat& m) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      if (m[i][j]) cout << 'K';
      else         cout << '.';
    cout << endl;
  }
  cout << "----------" << endl;
}

void search (int i, int j, int n, int r, int s, Mat& m) {
  if (s == r) escriu (n, m);
  else if (i != n) {
    int ni, nj;
    next(i, j, n, ni, nj);
    if (safe (i, j, n, m)) {
      m[i][j] = true;
      search (ni, nj, n, r, s+1, m);
    }
    m[i][j] = false;
    search (ni, nj, n, r, s, m);
  }
}

int main(void) {
  int n, r;
  cin >> n >> r;
  Mat m(n, Vec(n, false));
  search (0, 0, n, r, 0, m);
}
```

Shift 2

Solution to problem 1

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

typedef map<string, int>::iterator  Iterator ;

int main(void) {
  map<string, int> bagmul;
  string command;
  while (cin ≫ command) {
    if (command == "store") {
      string word;
      cin ≫ word;
      pair<Iterator , bool> res = bagmul.insert (pair<string,int>(word, 1));
      if (not res .second)  ++bagmul[word];
    }
    else if (command == "delete") {
      string word;
      cin ≫ word;
       Iterator  i = bagmul.find(word);
      if (i ≠ bagmul.end()) {
        if (i−>second == 1)   bagmul.erase(i );
        else                  −−bagmul[word];
      }
    }
    else if (command == "minimum?") {
      if (bagmul.empty())  cout ≪ "indefinite minimum" ≪ endl;
      else {
         Iterator  i = bagmul.begin ();
        cout ≪ "minimum: "
             ≪ i−> first  ≪ ", "
             ≪ i−>second ≪ " time(s)" ≪ endl;
      }
    }
    else {
      if (bagmul.empty())  cout ≪ "indefinite maximum" ≪ endl;
      else {
         Iterator  i = bagmul.end();
        −−i;
        cout ≪ "maximum: "
             ≪ i−>first  ≪ ", "
             ≪ i−>second ≪ " time(s)" ≪ endl;
      }
    }
    }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<char> CVec;
typedef vector<CVec> CMat;
typedef vector<bool> BVec;
typedef vector<BVec> BMat;

const int N_DIRS = 4;
const int DR[N_DIRS] = { 0,−1, 0, 1};
const int DC[N_DIRS] = { 1, 0,−1, 0};

struct Point {
  int r, c;
  Point(int rr, int cc) : r(rr), c(cc) { }
};

bool ok(int r, int c, int n, int m) {
  return
    0 ≤ r and r < n and
    0 ≤ c and c < m;
}

void search(int re, int ce, int n, int m, const CMat& t, vector<Point>& v, BMat& seen) {
  Point p = v.back();
  if (p.r == re and p.c == ce) {
    for (int i = 0; i < v.size(); ++i)
      cout ≪ t[v[i].r][v[i].c];
    cout ≪ endl;
  }
  else {
    for (int k = 0; k < N_DIRS; ++k) {
      int rr = p.r + DR[k];
      int cc = p.c + DC[k];
      if (ok(rr, cc, n, m) and not seen[rr][cc]) {
        seen[rr][cc] = true;
        v.push_back(Point(rr,cc));
        search(re, ce, n, m, t, v, seen);
        v.pop_back();
        seen[rr][cc] = false;
      }
    }
  }
}

int main(void) {
  int n, m;
  cin >> n >> m;
  CMat  t(n, CVec(m));
  BMat seen(n, BVec(m, false ));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      cin >> t[i][j];
  int ri, ci, re, ce;
  cin >> ri >> ci >> re >> ce;
  seen[ri][ci] = true;
  vector<Point> v(1, Point(ri, ci));
  search(re, ce, n, m, t, v, seen);
}
```

Solution to Lab Exam EDA 19/5/2011

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

struct Point {
  int r, c;
  Point(int rr, int cc) : r(rr), c(cc) {}
};

bool ok(int n, int m, const Point& p) {
  return
    0 ≤ p.r  and p.r < n and
    0 ≤ p.c  and p.c < m;
}

const int N_DIRS_KNIGHT = 8;
const int DR_KNIGHT[8] = {−2, −1, 2, 1, −2, −1, 2,  1};
const int DC_KNIGHT[8] = {−1, −2, −1, −2, 1, 2, 1,  2};

const int N_DIRS_BISHOP = 4;
const int DR_BISHOP[4] = { 1, −1, 1, −1};
const int DC_BISHOP[4] = { 1, 1, −1, −1};

int dfs(int n, int m, const Point& p,
        vector< vector<char> >& map, vector< vector<bool> >& marked,
        const int N_DIRS, const int DR[], const int DC[]) {
  int s = 0;
  marked[p.r][p.c] = true;
  for (int i = 0; i < N_DIRS; ++i) {
    Point q(p.r + DR[i], p.c + DC[i]);
    if (ok(n,m,q) and map[q.r][q.c] ≠ 'T' and not marked[q.r][q.c])
      s += dfs(n, m, q, map, marked, N_DIRS, DR, DC);
  }
  if ('0' ≤ map[p.r][p.c]  and map[p.r][p.c] ≤ '9') {
    s += map[p.r][p.c] − '0';
    map[p.r][p.c] = '.';
  }
  return s;
}

int main(void) {
  int n, m;
  while (cin >> n >> m) {
    vector<Point> knights, bishops;
    vector< vector<char> > map(n, vector<char>(m));
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j) {
        cin >> map[i][j];
        switch(map[i][j]) {
        case 'K': knights.push_back(Point(i,j)); break;
        case 'B': bishops.push_back(Point(i,j)); break;
        };
      }
    int s = 0;
    vector< vector<bool> > marked_knight(n, vector<bool>(m, false));
    for (int k = 0; k < knights.size(); ++k) {
      Point p = knights[k];
      if (not marked_knight[p.r][p.c])
        s += dfs(n, m, p, map, marked_knight, N_DIRS_KNIGHT, DR_KNIGHT, DC_KNIGHT);
    }
    vector< vector<bool> > marked_bishop(n, vector<bool>(m, false));
    for (int k = 0; k < bishops.size(); ++k) {
      Point p = bishops[k];
      if (not marked_bishop[p.r][p.c])
        s += dfs(n, m, p, map, marked_bishop, N_DIRS_BISHOP, DR_BISHOP, DC_BISHOP);
    }
    cout << s << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <set>

using namespace std;

typedef long long int lint;

int main(void) {

  lint suma = 0;
  set<lint> selec;
  set<lint> resta;

  int n;
  cin >> n;
  string op;
  lint val;
  while (cin >> op >> val) {
    if (op == "deixar") {
      if (selec.size() < n) {
        selec.insert(val);
```

```
          suma += val;
        }
        else {
          lint  min = *( selec .begin ());
          if  (min < val) {
            selec . insert (val );
            selec . erase ( selec .begin ());
            suma = suma + val − min;
            resta . insert (min);
          }
          else  resta . insert (val );
        }
      }
      else {
        if  (*( selec .begin ())  ≤ val) {
          selec . erase (val );
          suma −= val;
          if  ( resta . size ()  > 0) {
            set<lint>:: iterator   it  =  resta .end ();
            −−it;
            selec . insert (* it );
            suma += *it ;
            resta . erase ( it );
          }
        }
        else  resta . erase (val );
      }
      cout ≪ suma ≪ endl;
    }
}
```

Solution to Lab Exam EDA 13/12/2011

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<char> VC;
typedef vector<VC>   MC;
typedef vector<bool> VB;
typedef vector<VB>   MB;

const int N_DIRS = 4;
const int DI[N_DIRS] = { 1,  0, −1,  0};
const int DJ[N_DIRS] = { 0,  1,  0, −1};

bool ok(int n, int m, int i, int j) {
  return
    0 ≤ i and i < n and
    0 ≤ j and j < m;
}

int search(const MC& map, int n, int m, int i, int j, MB& marked) {
  int t = 0;
  if (map[i][j] == 't') ++t;
  marked[i][j] = true;
  for (int k = 0; k < N_DIRS; ++k) {
    int ii = i + DI[k];
    int jj = j + DJ[k];
    if (ok(n, m, ii, jj) and map[ii][jj] ≠ 'X' and not marked[ii][jj])
      t += search(map, n, m, ii, jj, marked);
  }
  return t;
}

int main(void) {
  int n, m;
  cin ≫ n ≫ m;
  MC map(n, VC(m));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      cin ≫ map[i][j];
  int f, c;
  cin ≫ f ≫ c;

  MB marked(n, VB(m, false));
  cout ≪ search(map, n, m, f−1, c−1, marked) ≪ endl;
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

void b(int k, int n, string& s) {
  if (k == n) cout << s << endl;
  else {
    s[k] = 'A'; b(k+1, n, s);
    s[k] = 'C'; b(k+1, n, s);
    s[k] = 'G'; b(k+1, n, s);
    s[k] = 'T'; b(k+1, n, s);
  }
}

int main() {
  int n;
  cin >> n;
  string s(n, 'X');
  b(0, n, s);
}
```

Solution to Lab Exam EDA 14/05/2012

Solution to problem 1

```
#include<iostream>
#include<string>
#include<map>

using namespace std;

typedef map<string,string>::iterator ite;

int main() {
    string s;
    map<string,string> liats;
    while(cin >> s) {
        if (s == "info") {
            cout << "PARELLES:" << endl;
            for (ite it = liats.begin(); it != liats.end(); ++it)
                if (it->second != "" and it->first < it->second)
                    cout << it->first << " " << it->second << endl;
            cout << "SOLS:" << endl;
            for (ite it = liats.begin(); it != liats.end(); ++it)
                if (it->second == "")
                    cout << it->first << endl;
            cout << "----------" << endl;
        }
        else {
            string x, y;
            cin >> x >> y;
            if (liats[x] != "") liats[liats[x]] = "";
            if (liats[y] != "") liats[liats[y]] = "";
            liats[x] = y;
            liats[y] = x;
        }
    }
}
```

Solution to problem 2

```
#include<iostream>
#include<vector>
#include<queue>
#include<algorithm>

using namespace std;

const int xf[8] = {-1,-1,-1,0,0,1,1,1};
const int yf[8] = {−1,0,1,−1,1,−1,0,1};
const int x[4] = {0,0,1,−1};
const int y[4] = {1,−1,0,0};

bool cerca_bolet (vector<vector<char> >& M, pair<int,int> p) {
    queue<pair<int, int> > Q;
    if (M[p.first ][p.second] == 'X') return false;
    M[p.first ][p.second] = 'X'; Q.push(p);
    while (not Q.empty()) {
        pair<int,int> q = Q.front (); Q.pop();
        for (int i = 0; i < 4; ++i) {
            int u = q.first + x[i ];
            int v = q.second + y[i ];
            if (M[u][v] == 'B') return true;
            if (M[u][v] ≠ 'X') {
                M[u][v] = 'X';
                Q.push(make_pair(u,v ));
            }
        }
    }
    return false;
}

int main() {
    int f, c;
    while (cin ≫ f ≫ c) {
        vector<vector<char> > M(f, vector<char> (c));
        pair<int, int> p;
        queue<pair<int,int> > F;
        for (int i = 0; i < f; ++i) {
            for (int j = 0; j < c; ++j) {
                cin ≫ M[i][j];
                if (M[i][j] == 'P') {p.first = i; p.second = j;}
                if (M[i][j] == 'F') F.push(make_pair(i,j ));
            }
        }
        while (not F.empty()) {
            int i = (F.front ()). first ;
            int j = (F.front ()). second;
            F.pop ();
            for (int k = 0; k < 8; ++k) M[i+xf[k]][j+yf[k]] = 'X';
        }
        if ( cerca_bolet (M,p)) cout ≪ "si" ≪ endl;
        else cout ≪ "no" ≪ endl;
    }
}
```

Solution to Lab Exam EDA 29/11/2012

Solution to problem 1

```
#include <iostream>
#include <sstream>
#include <vector>
#include <queue>

using namespace std;

typedef vector< queue<string> > VQS;

void read_queues(int n, VQS& v) {
    v = VQS(n);
    for (int k = 0; k < n; ++k) {
        string line;
        getline(cin, line);
        istringstream in(line);
        string name;
        while (in >> name) v[k].push(name);
    }
}

void process_exits (VQS& v) {
    cout << "SORTIDES" << endl;
    cout << "--------" << endl;

    string op;
    while (cin >> op) {
        if (op == "SURT") {
            int idx;
            cin >> idx;
            --idx;
            if (0 ≤ idx and idx < v.size() and not v[idx].empty()) {
                cout << v[idx].front() << endl;
                v[idx].pop();
            }
        }
        else {
            string name;
            int idx;
            cin >> name >> idx;
            --idx;
            if (0 ≤ idx and idx < v.size())
                v[idx].push(name);
            }
        }
    cout << endl;
}

void write_final_contents (VQS& v) {
    cout << "CONTINGUTS FINALS" << endl;
    cout << "-----------------" << endl;
    for (int k = 0; k < v.size (); ++k) {
        cout << "cua " << k+1 << ":";
        while (not v[k].empty()) {
            cout << ' ' << v[k].front ();
            v[k].pop ();
        }
        cout << endl;
    }
}

int main() {
    int n;
    cin >> n;
    string line;
    getline (cin, line); // Read empty line.

    VQS v;
    read_queues(n, v);
    process_exits (v);
    write_final_contents (v);
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

void compute_topological_ordering (const VVI& g, VI& indeg, VI& ord) {
    int n = g.size ();
    priority_queue <int, vector<int>, greater<int> > pq;
    for (int u = 0; u < n; ++u)
        if (indeg[u] == 0) pq.push(u);

    ord = VI(n);
    int cnt = 0;
    while (not pq.empty()) {
        int u = pq.top ();
        pq.pop ();
        ord[cnt] = u;
        ++cnt;
        for (int k = 0; k < g[u].size(); ++k) {
            int v = g[u][k];
            --indeg[v];
            if (indeg[v] == 0) pq.push(v);
        }
    }
}

void write(const VI& v) {
    cout << v[0];
    for (int k = 1; k < v.size(); ++k)
        cout << ' ' << v[k];
    cout << endl;
}

int main() {
    int n, m;
    while (cin >> n >> m) {
        VVI g(n);
        VI indeg(n, 0);
        for (int k = 0; k < m; ++k) {
            int u, v;
            cin >> u >> v;
            g[u].push_back(v);
            ++indeg[v];
        }
        VI ord;
        compute_topological_ordering(g, indeg, ord);
        write(ord);
    }
}
```

Solution to Lab Exam EDA 22/5/2013

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> VI;

int n, m;
VI div;

bool ok(int y) {
  for (int i = 0; i < m; ++i)
    if (y % div[i] == 0)
      return false;
  return true;
}

void backtracking(int k, int x) {
  if (k == n) cout << x << endl;
  else {
    for (int v = 0; v ≤ 9; ++v) {
      int y = 10*x + v;
      if (ok(y)) backtracking(k+1, y);
    }
  }
}

int main() {
  while (cin >> n >> m) {
    div = VI(m);
    for (int i = 0; i < m; ++i)
      cin >> div[i];
    backtracking(0, 0);
    cout << "----------" << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;
```

```
typedef vector<char> VC;  typedef vector< int> VI;
typedef vector< VC > VVC; typedef vector< VI > VVI;

typedef pair<int,int> P;

const int DF[] = {1, 1,−1,−1, 2, 2,−2,−2};
const int DC[] = {2,−2, 2,−2, 1,−1, 1,−1};

const int oo = 200 ∗ 200;

bool ok(int i, int j, int n, int m, const VVC& t) {
  return i ≥ 0 and i < n and j ≥ 0 and j < m and t[i][j] ≠ 'X';
}

int distance(int f0, int c0, const VVC& t) {
  int n = t    .size ();
  int m = t [0]. size ();
  VVI d(n, VI(m, +oo));
  queue<P> q;
  q.push(P(f0, c0));
  d[f0][c0] = 0;
  while (not q.empty()) {
    P p = q.front ();
    q.pop ();
    int f = p. first ;
    int c = p.second;
    if (t[f][c] == 'p') return d[f][c];
    for (int k = 0; k < 8; ++k) {
      int i = f + DF[k];
      int j = c + DC[k];
      if (ok(i, j, n, m, t) and d[i][j] == +oo) {
        q.push(P(i, j));
        d[i][j] = d[f][c] + 1;
      }
    }
  }
  return +oo;
}

int main() {
  int n, m;
  while (cin ≫ n ≫ m) {
    VVC t(n, VC(m));

    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        cin ≫ t[i][j];
    int f0, c0;
    cin >> f0 >> c0;
    int dist = distance(f0-1, c0-1, t);
    if (dist == +oo) cout << "no" << endl;
    else             cout << dist << endl;
  }
}
```

Solution to Lab Exam EDA 4/12/2013

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int UNDEF = −1;
const int N_DIRS = 4;
const int DI[N_DIRS] = { 1,  0, −1,  0};
const int DJ[N_DIRS] = { 0,  1,  0, −1};

struct Pos {
  int i, j;
  Pos(int ii, int jj) : i(ii), j(jj) { }
};

bool ok(int n, int m, int i, int j) {
  return
    0 ≤ i and i < n and
    0 ≤ j and j < m;
}

int bfs(const vector < vector<char> >& map, int i0, int j0) {
  int n = map   .size ();
  int m = map[0].size ();
  queue<Pos> q;
  q.push( Pos(i0,j0) );
  vector< vector<int> > dist(n, vector<int>(m, UNDEF));
  dist[i0][j0] = 0;
  while (not q.empty()) {
    Pos p = q.front ();
    q.pop ();
    int i = p.i;
    int j = p.j;
    if (map[i][j] == 't') return dist[i][j];
    else {
      for(int k = 0; k < N_DIRS; ++k) {
        int ii = i + DI[k];
        int jj = j + DJ[k];
        if (ok(n, m, ii, jj) and map[ii][jj] ≠ 'X' and dist[ii][jj] == UNDEF) {
          q.push( Pos(ii,jj) );
          dist[ii][jj] = 1 + dist[i][j];
        }
      }
    }
  }
  return UNDEF;
}


int main(void) {

  int n, m;
  cin >> n >> m;

  vector < vector<char> > map(n, vector<char>(m));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      cin >> map[i][j];

  int f, c;
  cin >> f >> c;

  int dist = bfs(map, f-1, c-1);
  if (dist > 0) cout << "distancia minima: " << dist << endl;
  else          cout << "no es pot arribar a cap tresor" << endl;
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <map>
#include <assert.h>

using namespace std;

typedef map<string,int> MSI;
typedef vector<string>  VS;
typedef vector<bool>    VB;
typedef vector<int>     VI;
typedef vector<VI>      VVI;

int n_outputs, n_inputs;
VVI gdir, ginv;
MSI s2v;
VS v2s;

int string2vertex(const string& s) {
  auto i = s2v.find(s);
  if (i != s2v.end())
    return i->second;
  else {
    int v = v2s.size();
```

```
      v2s.push_back(s);
      s2v.insert(make_pair(s, v));
      return v;
    }
  }

  int main() {

    n_outputs = n_inputs = 0;

    string token;

    cin >> token;
    assert(token == "OUTPUT");
    while (cin >> token and token != "END") {
      ++n_outputs;
      string2vertex(token);
    }

    cin >> token;
    assert(token == "INPUT");
    while (cin >> token and token != "END") {
      ++n_inputs;
      string2vertex(token);
    }

    while (cin >> token and token != "END") {

      string s;
      cin >> s;
      int ov = string2vertex(s);
      if (ov+1 > gdir.size()) gdir.resize(ov+1);

      cin >> s;
      int iv1 = string2vertex(s);
      if (iv1 + 1 > ginv.size()) ginv.resize(iv1 + 1);

      if (token == "NOT") {
        gdir[ov ].push_back(iv1);
        ginv[iv1].push_back(ov );
      }
      else {
        cin >> s;
        int iv2 = string2vertex(s);
        if (iv2 + 1 > ginv.size()) ginv.resize(iv2 + 1);
        if (token == "AND") {
          gdir[ov].push_back( min(iv1,iv2) );
          gdir[ov].push_back( max(iv1,iv2) );
        }
        else {
          gdir[ov].push_back( max(iv1,iv2) );
          gdir[ov].push_back( min(iv1,iv2) );
        }
        ginv[iv1].push_back(ov);
        ginv[iv2].push_back(ov);
      }
    }

    int n = gdir.size();
    VI ddir(n, 0);
    for (int v = 0; v < n; ++v)
      ddir[v] = gdir[v].size();

    VI bag;
    for (int v = n_outputs; v < n_inputs + n_outputs; ++v)
      bag.push_back(v);

    VI ord;
    while (not bag.empty()) {
      int v = bag.back();
      ord.push_back(v);
      bag.pop_back();
      for (auto w : ginv[v]) {
        --ddir[w];
        if (ddir[w] == 0)
          bag.push_back(w);
      }
    }

    VB val(n);
    while (cin >> token) {
      val[ n_outputs ] = (token == "T");
      for (int v = n_outputs+1; v < n_inputs + n_outputs; ++v) {
        cin >> token;
        val[v] = (token == "T");
      }

      for (auto v : ord) {
        if      (gdir[v].size() == 1) {
          val[v] = not val[gdir[v][0]];
        }
        else if (gdir[v].size() == 2) {
          int iv1 = gdir[v][0];
          int iv2 = gdir[v][1];
          if (iv1 < iv2) val[v] = val[iv1] and val[iv2];
          else           val[v] = val[iv1]  or val[iv2];
        }
      }
      cout << (val[0] ? 'T' : 'F');
      for (int v = 1; v < n_outputs; ++v)
        cout << ' ' << (val[v] ? 'T' : 'F');
      cout << endl;
    }
  }
```

Solution to Lab Exam EDA 19/5/2014

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<bool>  VB;
typedef vector<int>   VI;
typedef vector<VI>    VVI;

void path(int xi, int xf, const VI& p) {
  if (xf == xi) cout << xi;
  else {
    path(xi, p[xf], p);
    cout << " " << xf;
  }
}

void bfs(const VVI& g, int xi, int xf) {
  int n = g.size();
  VI   p(n, −1);
  VB mkd(n, false);
  VI cur, pos;

  cur.push_back(xi);
  mkd[xi] = true;
  while (not cur.empty()) {
    for (int x : cur) {
      if (x == xf) {
        path(xi, xf, p);
        cout << endl;
        return;
      }
      for (int y : g[x])
        if (not mkd[y]) {
          pos.push_back(y);
          mkd[y] = true;
          p[y] = x;
        }
    }
    swap(pos, cur);
  }
}

int main() {
  int n, m;
  while (cin >> n >> m) {
    VVI g(n);
    for (int k = 0; k < m; ++k) {
      int x, y;
      cin >> x >> y;
      g[x].push_back(y);
    }
    for (int x = 0; x < n; ++x)
      sort (g[x].begin (), g[x].end ());

    bfs (g, 0, n−1);
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<bool> VB;

void bt(int k, int n, string& w, VB& mkd) {
  if (k == n) cout << w << endl;
  else {
    for (int i = 0; i < n; ++i)
      if (not mkd[i] and (k == 0 or 'a' + i ≠ w[k−1] + 1)) {
        mkd[i] = true;
        w[k] = 'a' + i;
        bt(k+1, n, w, mkd);
        mkd[i] = false;
      }
  }
}

int main() {
  int n;
  cin >> n;
  string w(n, 'a');
  VB mkd(n, false);
  bt (0, n, w, mkd);
}
```

Solution to Lab Exam EDA 22/12/2014

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<bool> VB;
typedef vector<int>  VI;
typedef vector<VI>   VVI;

const int UNDEF = −1;

bool cyclic (int x, const VVI& g, VB& mkd, VI& par) {
  if (mkd[x]) return true;
  mkd[x] = true;
  for (int y : g[x])
    if (par[x] ≠ y) {
      par[y] = x;
      if ( cyclic (y, g, mkd, par)) return true;
    }
  return false;
}

int nombre_arbres(const VVI& g) {
  int n = g. size ();
  VB mkd(n, false);
  VI par(n, UNDEF);
  int n_arb = 0;
  for (int x = 0; x < n; ++x) {
    if (not mkd[x]) {
      if ( cyclic (x, g, mkd, par)) return UNDEF;
      else ++n_arb;
    }
  }
  return n_arb;
}

int main() {
  int n, m;
  while (cin >> n >> m) {
    VVI g(n);
    for (int k = 0; k < m; ++k) {
      int x, y;
      cin >> x >> y;
      g[x ].push_back(y);
      g[y ].push_back(x);
    }
    int n_arb = nombre_arbres(g);
    if (n_arb == UNDEF) cout << "no" << endl;
    else                 cout << n_arb << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> VI;

int bt(int k, const VI& m, int x, int sum_par, int max_und) {
  if (sum_par > x or sum_par + max_und < x) return 0;
  if (k == m.size()) return 1;
  int cnt = 0;
  for (int v = 0; v <= 2; ++v)
    cnt += bt(k+1, m, x, sum_par + v*m[k], max_und − 2*m[k]);
  return cnt;
}

int main() {
  int x, n;
  while (cin >> x >> n) {
    VI m(n);
    int s = 0;
    for (int k = 0; k < n; ++k) {
      cin >> m[k];
      s += m[k];
    }
    cout << bt(0, m, x, 0, 2*s) << endl;
  }
}
```

Solution to Lab Exam EDA 25/05/2015

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<char> VC;
typedef vector<VC> VVC;

int bt(int i, int j, VVC& sol, int curr) {
  int n = sol    .size();
  int m = sol[0].size();
  if (i == n) return curr;
  int next_i, next_j;
  if (j == m−1) {
    next_i = i+1;
    next_j = 0;
  }
  else {
    next_i = i;
    next_j = j+1;
  }

  sol[i][j] = 'L';
  int new_lols = 0;
  if (i ≥ 2              and sol[i−1][j  ] == 'O' and sol[i−2][j  ] == 'L') ++new_lols;
  if (          j ≥ 2    and sol[ i ][j−1] == 'O' and sol[ i ][j−2] == 'L') ++new_lols;
  if (i ≥ 2 and j ≥ 2    and sol[i−1][j−1] == 'O' and sol[i−2][j−2] == 'L') ++new_lols;
  if (i ≥ 2 and j+2 < m and sol[i−1][j+1] == 'O' and sol[i−2][j+2] == 'L') ++new_lols;
  int nl = bt(next_i, next_j, sol, curr + new_lols);

  sol[i][j] = 'O';
  int no = bt(next_i, next_j, sol, curr);

  return max(nl, no);
}

int main() {
  int n, m;
  while (cin >> n >> m) {
    VVC sol(n, VC(m));
    cout << bt(0, 0, sol, 0) << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<char>  VC;
typedef vector<VC>   VVC;
typedef vector<bool> VB;
typedef vector<VB>   VVB;

bool ok(int i, int j, const VVC& t) {
  int n = t    .size();
  int m = t[0].size();
  return i ≥ 0 and i < n and j ≥ 0 and j < m and t[i][j] ≠ 'X';
}

const int di[] = {0, 1,  0, −1};
const int dj[] = {1, 0, −1,  0};

bool possible(int i_ini, int j_ini, int i_fin, int j_fin, const VVC& t, VVB& mkd) {
  if (mkd[i_ini][j_ini]) return false;
  mkd[i_ini][j_ini] = true;
  if (i_ini == i_fin and j_ini == j_fin) return true;
  for (int k = 0; k < 4; ++k) {
    int i = i_ini + di[k];
    int j = j_ini + dj[k];
    if (ok(i, j, t) and possible(i, j, i_fin, j_fin, t, mkd)) return true;
  }
  return false;
}

int main() {
  int n, m;
  while (cin ≫ n ≫ m) {
    VVC t(n, VC(m));
    int i_ini, j_ini, i_fin, j_fin;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        cin ≫ t[i][j];

    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        if (t[i][j] == 'I') {
          i_ini = i;
          j_ini = j;
        }
        else if (t[i][j] == 'F') {
          i_fin = i;
          j_fin = j;
```

```
            }
            else if (t[i][j] == 'M') {
               t[i][j] = 'X';
               if (i−1 ≥ 0 and t[i−1][j] ≠ 'M') t[i−1][j] = 'X';
               if (i+1 < n and t[i+1][j] ≠ 'M') t[i+1][j] = 'X';
               if (j−1 ≥ 0 and t[i][j−1] ≠ 'M') t[i][j−1] = 'X';
               if (j+1 < m and t[i][j+1] ≠ 'M') t[i][j+1] = 'X';
            }

      VVB mkd(n, VB(m, false));
      if (possible(i_ini, j_ini, i_fin, j_fin, t, mkd)) cout << "SI" << endl;
      else                                              cout << "NO" << endl;
   }
}
```

Solution to Lab Exam EDA 22/12/2015

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const vector<pair<int, int>> DIRS = { {1, 0}, {−1, 0}, {0, 1}, {0, −1} };
const int UNDEF = −1;

int dist_2on_tresor_mes_llunya (int i0, int j0, vector<vector<char>>& mapa) {
  int max_dist  = UNDEF;
  int max_dist2 = UNDEF;
  queue<pair< pair<int, int>, int>> q;
  q.push({{i0, j0}, 0});
  mapa[i0][j0] = 'X';
  while (not q.empty()) {
    int i = q.front (). first . first ;
    int j = q.front (). first .second;
    int d = q.front (). second;
    q.pop ();
    for (auto dir : DIRS) {
      int ii = i + dir . first ;
      int jj = j + dir .second;
      if (mapa[ii ][ jj ] ≠ 'X') {
        if (mapa[ii ][ jj ] == 't' ) {
          max_dist2 = max_dist;
          max_dist  = d+1;
        }
        q.push({{ii , jj }, d+1});
        mapa[ii ][ jj ] = 'X';
      }
    }
  }
  return max_dist2;
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<char>> mapa(n+2, vector<char>(m+2, 'X'));
  for (int i = 1; i ≤ n; ++i)
    for (int j = 1; j ≤ m; ++j)
      cin >> mapa[i][j ];

  int f, c;
  cin >> f >> c;

  int d2 = dist_2on_tresor_mes_llunya (f, c, mapa);
  if (d2 == UNDEF) cout << "no es pot arribar a dos o mes tresors" << endl;
  else             cout << "segona distancia maxima: " << d2 << endl;
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

void escriu (int f, int c, const vector<int>& col) {
  for (int i = 0; i < f; ++i) {
    for (int j = 0; j < c; ++j)
      if (col[i] == j) cout << "R";
      else             cout << ".";
    cout << endl;
  }
  cout << endl;
}

void torres (int f, int c, int i, vector<int>& col, vector<bool>& marked) {
  if (i == f) escriu (f, c, col );
  else
    for (int j = 0; j < c; ++j)
      if (not marked[j]) {
        col[i] = j;
        marked[j] = true;
        torres (f, c, i+1, col , marked);
        marked[j] = false;
      }
}

int main() {
  int f, c;
  cin >> f >> c;
  vector<int> col(f);
  vector<bool> marked(c, false);
  torres (f, c, 0, col , marked);
}
```

Solution to Lab Exam EDA 19/05/2016

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef pair<int,int> P;

int main(void) {
  int n, m;
  while (cin >> n >> m) {
    vector<vector<P>> g(n);
    for (int k = 0; k < m; ++k) {
      int x, y, c;
      cin >> x >> y >> c;
      --x;
      --y;
      g[x].push_back(P(c, y));
      g[y].push_back(P(c, x));
    }
    vector<bool> mkd(n, false);
    mkd[0] = true;
    priority_queue <P, vector<P>, greater<P> > pq;
    for (P x : g[0]) pq.push(x);
    int sz = 1;
    int sum = 0;
    while (sz < n) {
      int c = pq.top().first ;
      int x = pq.top().second;
      pq.pop();
      if (not mkd[x]) {
        mkd[x] = true;
        for (P y : g[x]) pq.push(y);
        sum += c;
        ++sz;
      }
    }
    cout << sum << endl;
  }
}
```

Solution to problem 2

```
#include <vector>

using namespace std;

// Pre: l ≤ r, x < v[r].
// Return the smallest i s.t. l ≤ i ≤ r and x < v[i].
int rightmost(double x, const vector<double>& v, int l, int r) {
  if (l == r) return l;
  int m = (l+r)/2;
  if (x < v[m]) return rightmost(x, v,   l, m);
  else          return rightmost(x, v, m+1, r);
}

int rightmost(double x, const vector<double>& v) {
  return rightmost(x, v, 0, v.size());
}
```

Solution to Lab Exam EDA 15/12/2016

Shift 1

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> su;
vector<vector<bool>> r_mkd, c_mkd;
vector<vector<vector<bool>> > s_mkd;

void write() {
  cout ≪ endl;
  for (int i = 0; i < 9; ++i) {
    cout ≪ su[i][0] + 1;
    for (int j = 1; j < 9; ++j)
      cout ≪ ' ' ≪ su[i][j] + 1;
    cout ≪ endl;
  }
}

bool fill (int i, int j) {
  if (i == 9)          return true;
  if (j == 9)          return fill (i+1, 0);
  if (su[i][j] ≠ −1) return fill (i, j+1);
  for (int v = 0; v < 9; ++v)
    if (not r_mkd[i][v] and not c_mkd[j][v] and not s_mkd[i/3][j/3][v]) {
      r_mkd[i][v] = c_mkd[j][v] = s_mkd[i/3][j/3][v] = true;
      su[i][j] = v;
      if ( fill (i, j+1)) return true;
      r_mkd[i][v] = c_mkd[j][v] = s_mkd[i/3][j/3][v] = false;
    }
  su[i][j] = −1;
  return false;
}

int main() {
  su = vector<vector<int>>(9, vector<int>(9));
  int n;
  cin ≫ n;
  cout ≪ n ≪ endl;
  while (n−−) {
    r_mkd = c_mkd = vector<vector<bool>>(9, vector<bool>(9, false));
    s_mkd = vector<vector<vector<bool>> >(3, vector<vector<bool>>(3, vector<bool>(9, false)));
    for (int i = 0; i < 9; ++i)
      for (int j = 0; j < 9; ++j) {
        char c;
        cin >> c;
        if (c == '.') su[i][j] = -1;
        else {
          int v = c - '1';
          r_mkd[i][v] = c_mkd[j][v] = s_mkd[i/3][j/3][v] = true;
          su[i][j] = v;
        }
      }
    if ( fill (0, 0)) write ();
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool search(const vector<int>& a, int l, int r, int x) {
  if (l+1 == r) return a[l] == x or a[r] == x;
  int m = (l+r)/2;
  if (a[m] ≥ a[l]) {
    if (a[l] ≤ x and x ≤ a[m]) {
      return binary_search (a.begin()+l, a.begin()+m+1, x);
    }
    else return search(a, m, r, x);
  }
  else {
    if (a[m] ≤ x and x ≤ a[r]) {
      return binary_search (a.begin()+m, a.begin()+r+1, x);
    }
    else return search(a, l, m, x);
  }
}

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

Shift 2

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

int m, n;
vector<int> l;

void gen(int i, int pos) {
  if (pos >= -m/2 and pos <= m/2) {
    if (i == n) cout << pos << endl;
    else {
      gen(i+1, pos+l[i]);
      gen(i+1, pos-l[i]);
    }
  }
}

int main() {
  cin >> m >> n;
  l = vector<int>(n);
  for (int& x : l) cin >> x;
  gen(0, 0);
}
```

Solution to problem 2

```
#include <iostream>

using namespace std;

typedef int Matrix[2][2];

void product(int m, const Matrix& a, const Matrix& b, Matrix& c) {
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j) {
      c[i][j] = 0;
      for (int k = 0; k < 2; ++k)
        c[i][j] = (c[i][j] + a[i][k] * b[k][j]) % m;
    }
}

void identity (Matrix& b) {
  b[0][0] = b[1][1] = 1;
  b[0][1] = b[1][0] = 0;
}

void power(int n, int m, const Matrix& a, Matrix& b) {

  if (n == 0)
    identity (b);
  else {
    if (n%2 == 0) {
      Matrix c;
      power(n/2, m, a, c);
      product(m, c, c, b);
    }
    else {
      Matrix c, d;
      power(n/2, m, a, c);
      product(m, c, c, d);
      product(m, a, d, b);
    }
  }
}

int main() {
  Matrix a;
  a[0][0] = 1;      a[0][1] = 1;
  a[1][0] = 1;      a[1][1] = 0;
  int n, m;
  while (cin >> n >> m) {
    Matrix b;
    power(n, m, a, b);
    cout << b[1][0] << endl;
  }
}
```

Solution to Lab Exam EDA 25/05/2017

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

enum Letter {CON = 0, VOC};

void g(int k, int n, vector<string>& let, string& sol, vector<vector<bool>>& mkd) {
  if (k == 2*n) cout << sol << endl;
  else {
    int k2 = k%2;
    for (int i = 0; i < n; ++i)
      if (not mkd[k2][i]) {
        sol[k] = let[k2][i];
        mkd[k2][i] = true;
        g(k+1, n, let, sol, mkd);
        mkd[k2][i] = false;
      }
  }
}

int main() {
  int n;
  cin >> n;
  vector<vector<bool>> mkd(2, vector<bool>(n, false));
  vector<string> let(2);
  cin >> let[CON] >> let[VOC];
  string sol(2*n, ' '); // Need a char for filling.
  g(0, n, let, sol, mkd);
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <queue>

using namespace std;

const int UND = −1;

int main() {
  int n, t;
  while (cin >> n >> t) {
      vector<unordered_set<int>> g(n);
      while (--t >= 0) {
        int s;
        cin >> s;
        vector<int> pub(s);
        for (int i = 0; i < s; ++i)
          cin >> pub[i];

        for (int i = 0; i < s; ++i)
          for (int j = i+1; j < s; ++j) {
            int u = pub[i];
            int v = pub[j];
            g[u].insert(v);
            g[v].insert(u);
          }
      }
      vector<int> dst(n, UND);
      queue<int> q;
      dst[0] = 0;
      q.push(0);
      while (not q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : g[u])
          if (dst[v] == UND) {
            dst[v] = dst[u] + 1;
            q.push(v);
          }
      }
      for (int u = 0; u < n; ++u) {
        cout << u << ": ";
        if (dst[u] == UND) cout << "no";
        else               cout << dst[u];
        cout << endl;
      }
      for (int k = 0; k < 10; ++k) cout << "-";
      cout << endl;
    }
  }
```

Solution to Lab Exam EDA 25/05/2017

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

void write(int n, int p, const vector<string>& s, vector<int>& sol) {
  for (int j = 0; j < p; ++j) {
    cout << "subset " << j+1 << ": {";
    string aux = "";
    for (int i = 0; i < n; ++i)
      if (sol[i] == j) {
        cout << aux << s[i];
        aux = ",";
      }
    cout << "}" << endl;
  }
  cout << endl;
}

void g(int k, int n, int p, const vector<string>& s, vector<int>& sol) {
  if (k == n) write(n, p, s, sol);
  else
    for (int i = 0; i < p; ++i) {
      sol[k] = i;
      g(k+1, n, p, s, sol);
    }
}

int main() {
  int n, p;
  cin >> n;
  vector<string> s(n);
  for (auto& x : s) cin >> x;
  cin >> p;
  vector<int> sol(n);
  g(0, n, p, s, sol);
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;
```

```
typedef pair<int,int> P;

int main() {
  int n, m;
  while (cin >> n >> m) {
    vector< vector<P> > g(n);
    int tot = 0;
    for (int k = 0; k < m; ++k) {
      int x, y, c;
      cin >> x >> y >> c;
      g[x].push_back({c, y});
      g[y].push_back({c, x});
      tot += c;
    }
    vector<bool> mkd(n, false);
    mkd[0] = true;
    priority_queue <P, vector<P>, greater<P> > pq;
    for (P p : g[0])
      pq.push(p);
    int sz = 1;
    int sum = 0;
    while (sz < n) {
      int c = pq.top().first;
      int x = pq.top().second;
      pq.pop();
      if (not mkd[x]) {
        mkd[x] = true;
        for (P p : g[x])
          pq.push(p);
        sum += c;
        ++sz;
      }
    }
    cout << tot - sum << endl;
  }
}
```

Solution to Lab Exam EDA 28/05/2018

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int>  VI;
typedef vector<VI >  VVI;

int n, m;
VVI g;
VI  c;

const int UNDEF = −1;

// Returns true if no conflict is found.
bool propagate(int x, int col) {
  if (c[x] == UNDEF) {
    c[x] = col;
    for (int y: g[x])
      if (not propagate(y, 1−col)) return false;
    return true;
  }
  else return c[x] == col;
}

bool two_colorable (int x) {
  if (x == n) return true;
  if (c[x] == UNDEF) return propagate(x, 0) and two_colorable (x+1);
  else               return                     two_colorable (x+1);
}

int main() {
  while (cin >> n >> m) {
    g = VVI(n);
    for (int k = 0; k < m; ++k) {
      int x, y;
      cin >> x >> y;
      g[x].push_back(y);
      g[y].push_back(x);
    }
    c = VI(n, UNDEF);
    if ( two_colorable (0)) cout << "yes" << endl;
    else                    cout << "no"  << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Permutation {

  vector<int> v;

public:

  Permutation(int n)                    : v( n ) { }
  Permutation(const Permutation& s) : v(s.v) { }

  Permutation& operator = (const Permutation& s) {
    v = s.v;
    return *this;
  }

  int& operator [](int i)         { return v[i]; }
  int   operator []( int i) const { return v[i]; }

  Permutation& operator *= (const Permutation& s) {
    for (int i = 0; i < v.size (); ++i)
      v[i] = s[v[i]];
    return *this;
  }

  void pow(int k) {
    if (k == 0)
      for (int i = 0; i < v.size (); ++i)
        v[i] = i;
    else {
      Permutation s = *this;
      s.pow(k/2);
      if (k % 2 == 0) {
        *this = s;
        *this *= s;
      }
      else {
        *this *= s;
        *this *= s;
      }
    }
  }
};
```

```
int main() {
  int n;
  while (cin >> n) {
    Permutation p(n);
    for (int i = 0; i < n; ++i) {
      cin >> p[i];
    }
    int k;
    cin >> k;
    p.pow(k);

    for (int x = 0; x < n; ++x)
      cout << (x == 0 ? "" : " ") << p[x];
    cout << endl;
  }
}
```

Solution to Lab Exam EDA 03/12/2018

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int top(const vector<int>& v, int l, int r) {
  if (l + 1 ≥ r) {
    if (v[l] < v[r]) return r;
    else             return l;
  }
  int m = (l+r)/2;
  if (v[m−1] > v[m]) return top(v, l, m−1);
  if (v[m+1] > v[m]) return top(v, m+1, r);
  return m;
}

bool bin_search(bool inc, const vector<int>& v, int l, int r, int x) {
  if (l == r) return v[l] == x;
  int m = (l+r)/2;
  bool cond;
  if (inc) cond = (x ≤ v[m]);
  else     cond = (x ≥ v[m]);
  if (cond) return bin_search(inc, v, l, m, x);
  else      return bin_search(inc, v, m+1, r, x);
}

bool search(int x, const vector<int>& v) {
  int n = v.size();
  int t = top(v, 0, n−1);
  return bin_search(true, v, 0, t, x) or bin_search(false, v, t, n−1, x);
}
```

Solution to problem 2

```
#include <iostream>
#include <queue>
#include <climits>

using namespace std;
using P = pair<int,int>;

const int oo = INT_MAX;

int cost(const vector<vector<P>>& g, const vector<int>& z, int a, int b) {
  int n = g.size();
  vector<int> dist(n, +oo);
  priority_queue<P, vector<P>, greater<P>> pq;
  dist[a] = 0;
  pq.push({0, a});
  while (not pq.empty()) {
    auto t = pq.top();
    pq.pop();
    int d = t.first;
    int u = t.second;
    if (d == dist[u]) {
      if (u == b) return dist[b];
      for (auto e : g[u]) {
        int v = e.first;
        int w = e.second;
        int dv = dist[u] + w + (v == b ? 0 : z[v]);
        if (dist[v] > dv) {
          dist[v] = dv;
          pq.push({dv, v});
        }
      }
    }
  }
  return +oo;
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<int> z(n);
  for (int& x : z) cin >> x;
  vector<vector<P>> g(n);
  while (m--) {
    int u, v, w;
    cin >> u >> v >> w;
    g[u].push_back({v, w});
    g[v].push_back({u, w});
  }
  int a, b;
  while (cin >> a >> b) {
    int c = cost(g, z, a, b);
    cout << "c(" << a << "," << b << ") = ";
    if (c == +oo) cout << "+oo" << endl;
    else          cout <<   c   << endl;
  }
}
```

Solution to Lab Exam EDA 20/05/2019

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool is_cyclic (const VVI& g) {
  int n = g.size ();
  VI indeg(n, 0);
  for (int u = 0; u < n; ++u)
    for (int v : g[u])
      ++indeg[v];

  vector<int> cands;
  for (int u = 0; u < n; ++u)
    if (indeg[u] == 0)
      cands.push_back(u);

  while (not cands.empty()) {
    int u = cands.back ();
    cands.pop_back ();
    --n;
    for (int v : g[u]) {
      --indeg[v];
      if (indeg[v] == 0)
        cands.push_back(v);
    }
  }
  return n > 0;
}


int main() {
  int n;
  while (cin >> n) {
    VVI g(n);
    int m;
    cin >> m;
    for (int k = 0; k < m; ++k) {
      int x, y;
      cin >> x >> y;
      g[x].push_back(y);
    }
    if ( is_cyclic (g)) cout << "yes" << endl;
    else                cout << "no" << endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

void write(const vector<int>& x, const vector<bool>& sol) {
  cout << '{';
  bool first = true;
  for (int i = 0; i < int(x.size()); ++i)
    if (sol[i]) {
      if (first) first = false;
      else cout << ',';
      cout << x[i];
    }
  cout << '}' << endl;
}

void bt(int s, int k, const vector<int>& x, int sp, int sn, vector<bool>& sol) {
  if (sp > s or sp + sn < s) return;
  if (k == int(x.size())) write(x, sol);
  sol[k] = false;  bt(s, k+1, x,    sp,       sn − x[k], sol);
  sol[k] = true;   bt(s, k+1, x, sp + x[k], sn − x[k], sol);
}

int main() {
  int s, n;
  cin >> s >> n;
  vector<int> x(n);
  int sum = 0;
  for (int i = 0; i < n; ++i) {
    cin >> x[i];
    sum += x[i];
  }
  vector<bool> sol(n);
  bt(s, 0, x, 0, sum, sol);
}
```

Solution to Lab Exam EDA 02/12/2019

Solution to problem 1

```
#include <iostream>
#include <map>

using namespace std;

void addGame(const string& game, map<string,int>& m) {
  ++m[game];
}

bool canBuy(const string& game, map<string,int>& m, int totalGames){
  return (m[game] + 1 ≤ totalGames − m[game]);
}

int main(){
  int n;
  while (cin ≫ n) {

    map<string,int> m;
    int totalGames = 0;
    for (int i = 0; i < n; ++i) {
      string game; cin ≫ game;
      addGame(game,m);
      ++totalGames;
    }

    int g; cin ≫ g;
    for (int i = 0; i < g; ++i) {
      string game; cin ≫ game;
      if (canBuy(game,m,totalGames)){
        addGame(game,m);
        ++totalGames;
      }
    }

    for (auto& g:m) cout ≪ g.first ≪ " " ≪ g.second ≪ endl;
    cout ≪ string(20, '−') ≪ endl;
  }
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <climits>
#include <queue>
using namespace std;

typedef vector<int> VI;
typedef vector<char> VC;
typedef vector<vector<char>> VVC;
typedef vector<vector<int>> VVI;
typedef pair<int,int> PII;

vector<PII> dirs = {{0,1},{0,−1},{1,0},{−1,0}};

bool pos_ok(const VVC& T, uint i, uint j){
  return i ≥ 0 and i < T.size () and j ≥ 0 and j < T[0].size () and T[i][j] ≠ '#';
}

PII search(const VVC& T) { // returns (distance,people)
  int n = T.size (), m = T[0].size ();
  VVI dist(n,VI(m,INT_MAX)), pers(n,VI(m,−1));
  queue<PII> Q;

  Q.push({0,0});
  dist [0][0] = 0;
  pers [0][0] = (T[0][0] == 'P' );

  while (not Q.empty()) {
    int i = Q.front (). first , j = Q.front (). second;
    Q.pop();
    if (T[i][j] == 'T') return {dist [i][j], pers [i][j]};
    for (auto& d:dirs) {
      int ni = i + d. first , nj = j + d.second;
      int nd = dist [i][j] + 1;
      int np = pers [i][j] + int(T[ni][nj] == 'P' );
      if (pos_ok(T,ni,nj)) {
        if (dist [ni][nj] == INT_MAX){
          Q.push({ni,nj});
          dist [ni][nj] = nd;
          pers [ni][nj] = np;
        }
        else if (dist [ni][nj] == nd and pers[ni][nj] < np)
          pers [ni][nj] = np;
      }
    }
  }
  return {−1,0};
}
```

```
int main(){
  int n, m;
  while (cin >> n >> m){
    VVC T(n,VC(m));
    bool found = false;
    for (int i = 0; i < n; ++i){
      for (int j = 0; j < m; ++j) {
        cin >> T[i][j];
        if (T[i][j] == 'T') found = true;
      }
    }
    if (not found) cout << "The telecos ran away." << endl;
    else {
      PII res = search(T);
      if (res.first == −1) cout << "The telecos is hidden." << endl;
      else cout << res.first << " " << res.second << endl;
    }
  }
}
```

Solution to Lab Exam EDA 10/06/2020

Solution to problem 1

Buying shares, X39187:

```
#include <iostream>
#include <vector>

using namespace std;

void rec(vector<char>& parcial, int idx, int monedes, int accions) {
  if (idx == int(parcial.size())) {
    for (auto& x : parcial) cout << x;
    cout << endl;
  }
  else {
    if (monedes > 0) {
      parcial[idx] = 'b';
      rec(parcial, idx+1, monedes−1, accions+1);
    }

    if (accions > 0) {
      parcial[idx] = 's';
      rec(parcial, idx+1, monedes+1, accions−1);
    }
  }
}

int main(){
  int n, c;
  cin >> n >> c;
  vector<char> parcial(n);
  rec(parcial,0,c,0);
}
```

Compensated words, X46137:

```
#include <iostream>
#include <vector>

using namespace std;

void rec(vector<char>& parcial, int idx, int a, int b) {
    if (idx == int(parcial.size())) {
    for (auto& x : parcial) cout << x;
    cout << endl;
  }
  else {
    if (abs((a+1)−b) ≤ 2) {
      parcial[idx] = 'a';
      rec(parcial,idx+1,a+1,b);
    }

    if (abs(a−(b+1)) ≤ 2) {
      parcial[idx] = 'b';
      rec(parcial,idx+1,a,b+1);
    }
  }
}

int main() {
  int n;
  cin >> n;
  vector<char> parcial(n);
  rec(parcial,0,0,0);
}
```

Up and down, X57029:

```
#include <iostream>
#include <vector>

using namespace std;

void rec(vector<char>& parcial, int idx, int alcada_actual) {
  if (idx == int(parcial.size())) {
    for (auto& x : parcial) cout << x;
    cout << endl;
  }
  else {
    if (alcada_actual ≠ 0) {
      parcial[idx] = 'd';
      rec(parcial,idx+1,alcada_actual − 1);
    }

    parcial[idx] = 'h';
    rec(parcial,idx+1,alcada_actual);

    parcial[idx] = 'u';
    rec(parcial,idx+1,alcada_actual + 1);
  }
}

int main() {
  int n;
  cin >> n;
  vector<char> parcial(n);
  rec(parcial,0,0);
}
```

Solution to problem 2

Film searcher, X14417:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

vector<pair<int,int>> dirs = {{1,0},{−1,0},{0,1},{0,−1}};

bool ok(const vector<vector<string>>& M, int i, int j) {
  return i ≥ 0 and i < int(M.size()) and j ≥ 0 and j < int(M[0].size()) and M[i][j] ≠ "*";
}

int bfs(const vector<vector<string>>& M, pair<int,int>& orig, pair<int,int>& dest) {
  int n = M.size();
  int m = M[0].size();
  vector<vector<int>> dist(n,vector<int>(m,−1));
  queue<pair<int,int>> Q;

  Q.push(orig);
  dist[orig.first][orig.second] = 0;

  while (not Q.empty()) {
    pair<int,int> u = Q.front();
    Q.pop();
    if (u == dest) return dist[u.first][u.second];
    for (auto& d : dirs) {
      pair<int,int> v = u;
      v.first += d.first;
      v.second += d.second;
      if (ok(M,v.first,v.second) and dist[v.first][v.second] == −1) {
        Q.push(v);
        dist[v.first][v.second] = dist[u.first][u.second] + 1;
      }
    }
  }
  return −1;
}


pair<int,int> position(const vector<vector<string>>& M, const string& s){
  for (uint i = 0; i < M.size(); ++i){
    for (uint j = 0; j < M[0].size(); ++j){
      if (M[i][j] == s) return {i,j};
    }
  }
  return {−1,−1};
}

int main() {
  int n, m;
  while (cin >> n >> m) {
    vector<vector<string>> M(n,vector<string>(m));
    for (int i = 0; i < n; ++i){
      for (int j = 0; j < m; ++j){
        cin >> M[i][j];
      }
    }
    int p;
    cin >> p;
    vector<string> paraules(p);
    for (int i = 0; i < p; ++i) cin >> paraules[i];

    vector<pair<int,int>> positions(p+1);
    positions [0] = {0,0};
    for (int i = 0; i < p; ++i)
      positions [i+1] = position (M,paraules[i]);

    int total = 0;
    bool ok = true;
    for (uint i = 0; ok and i < positions.size() - 1; ++i) {
      int d = bfs(M,positions[i], positions [i+1]);
      if (d == -1) ok = false;
      else total += d;
    }
    if (ok) cout << (total + p) << endl;
    else cout << "impossible" << endl;
  }
}
```

Knight's game, X39759:

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>

using namespace std;

const int infinite = numeric_limits<int>::max();

vector<pair<int,int>> dirs = {{1,2},{1,-2},{-1,2},{-1,-2},
                              {2,1},{-2,1},{2,-1},{-2,-1}};


bool inside (int n, int m, const pair<int,int>& x){
  return x.first >= 0 and x.first < n and x.second >= 0 and x.second < m;
```

```
}

int bfs (int n, int m, const pair<int,int>& ini, const pair<int,int>& fi) {
  vector<vector<int>> dist(n,vector<int>(m,infinite));

  queue<pair<int,int>> Q;
  Q.push(ini);
  dist[ini.first][ini.second] = 0;

  while (not Q.empty()){
    pair<int,int> x = Q.front();
    Q.pop();
    if (x == fi) return dist[x.first][x.second];
    for (auto& d : dirs) {
      pair<int,int> y = x;
      y.first += d.first;
      y.second += d.second;
      if (inside(n,m,y) and dist[y.first][y.second] == infinite) {
        dist[y.first][y.second] = dist[x.first][x.second] + 1;
        Q.push(y);
      }
    }
  }
  return −1;
}

int main() {
  int n, m, W, L, p;
  while (cin >> n >> m >> W >> L >> p) {
    vector<pair<int,int>> objs(p+1);
    objs[0] = {0,0};
    for (int i = 1; i ≤ p; ++i) {
      int f, c;
      cin >> f >> c;
      objs[i] = {f,c};
    }

    int points = 0;
    int best_points = 0;
    bool stop = false;
    for (uint i = 0; not stop and i < objs.size() − 1; ++i) {
      int dist = bfs(n,m,objs[i], objs[i+1]);
      if (dist == −1) stop = true;
      else {
        points += W;
        points −= L*dist;
        if (points > best_points) best_points = points;
      }
    }
    cout << best_points << endl;
  }
}
```

Solució to Lab Exam EDA 11/01/2021

Shift 1

Solution to problem 1

```
#include <iostream>
#include <vector>
#include <stdlib.h>


using namespace std;

bool compatible(int left , int right , int d) {
  return abs( left −right) ≤ d;
}

void write_balanced_permutations (int n, int d, vector<int>& partial_sol , vector<bool>& used) {
  if (int( partial_sol .size ()) == n) {
    cout ≪ "(";
    for (int i = 0; i < n; ++i) cout ≪ (i == 0 ? "" : ",") ≪ partial_sol [i];
    cout ≪ ")" ≪ endl;
  }
  else {
    for (int k = 1; k ≤ n; ++k) {
      if (not used[k]) {
        if ( partial_sol .size () == 0 or compatible( partial_sol .back (), k,d)){
          partial_sol .push_back(k);
          used[k] = true;
          write_balanced_permutations (n,d, partial_sol ,used);
          used[k] = false;
          partial_sol .pop_back ();
        }
      }
    }
  }
}

void write_balanced_permutations (int n, int d) {
  vector<bool> used(n+1,false);
  vector<int> partial_sol ;
  write_balanced_permutations (n,d, partial_sol ,used);
}

int main(){
  int n, d;
  cin ≫ n ≫ d;
  write_balanced_permutations (n,d);
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>

using namespace std;

void dfs(int u, const vector<vector<int>>& g, vector<int>& vis) {
  if (vis[u]) return;
  vis[u] = true;
  for (int v : g[u])
    dfs(v, g, vis);
}

int main() {
  int n, u, v, m;
  while (cin >> n >> u >> v >> m) {
    vector<vector<int>> g(n);
    vector<vector<int>> i(n); // Inverted graph (flipped edges)
    while (m--) {
      int x, y;
      cin >> x >> y;
      g[x].push_back(y);
      i[y].push_back(x);
    }
    vector<int> fwd(n, false);
    dfs(u, g, fwd);
    if (not fwd[v]) cout << 0 << endl;
    else {
      vector<int> bwd(n, false);
      dfs(v, i, bwd);
      int sum = 0;
      for (int x = 0; x < n; ++x) {
        if (fwd[x] and bwd[x]) cout << x << endl;
        sum += (fwd[x] and bwd[x]);
      }
      cout << sum - 2 << endl;
    }
  }
}
```

Shift 2

Solution to problem 1

```
#include <iostream>
#include <vector>

using namespace std;


bool compatible(int left, int mid, int right, int n) {
  return left + right ≤ 2∗mid;
}

void write_no_well_permutations(int n, vector<int>& partial_sol, vector<bool>& used) {
  if (int(partial_sol.size()) == n) {
    cout ≪ "(";
    for (int i = 0; i < n; ++i) cout ≪ (i == 0 ? "" : ",") ≪ partial_sol[i];
    cout ≪ ")" ≪ endl;
  }
  else {
    for (int k = 1; k ≤ n; ++k) {
      if (not used[k]) {
        if (partial_sol.size() ≤ 1 or
              compatible(partial_sol[partial_sol.size()−2], partial_sol.back(), k,n)){
          partial_sol.push_back(k);
          used[k] = true;
          write_no_well_permutations(n, partial_sol, used);
          used[k] = false;
          partial_sol.pop_back();
        }
      }
    }
  }
}

void write_no_well_permutations(int n) {
  vector<bool> used(n+1,false);
  vector<int> partial_sol;
  write_no_well_permutations(n, partial_sol, used);
}

int main(){
  int n;
  cin ≫ n;
  write_no_well_permutations(n);
}
```

Solution to problem 2

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits.h>

using namespace std;
using P = pair<int,int>;

const int oo = INT_MAX;

int dijkstra (const vector<vector<P>>& g, int x, int y) {
  int n = g.size ();
  vector<int> dist(n, +oo);
  priority_queue <P, vector<P>, greater<P>> q;
  dist [x] = 0;
  q.push({0, x});
  while (not q.empty()) {
    auto t = q.top ();
    q.pop ();
    int u = t.second;
    int d = t. first ;
    if (u == y) return dist [y];
    if (d == dist[u]) {
      for (auto p : g[u]) {
        int v = p.second;
        int l = p. first ;
        int d2 = max(dist[u], l );
        if (d2 < dist[v]) {
          dist [v] = d2;
          q.push({d2, v});
        }
      }
    }
  }
  return +oo;
}

int main() {
  int n, m;
  while (cin >> n >> m) {
    vector<vector<P>> g(n);
    while (m−−) {
      int x, y, l;
      cin >> x >> y >> l;
      g[x].push_back({l, y});
    }
    cout << dijkstra (g, 0, 1) << endl; } }
```

6

Solutions to Final Exams

Solution of the Final EDA Exam 08/06/2016

Proposed solution to problem 1

(a) The master theorem for dividing recurrences claims that if we have a recurrence of the form $T(n) = aT(n/b) + \Theta(n^k)$ with $a > 0$, $b > 1$ and $k \geq 0$, then, letting $\alpha = \log_b a$,

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } \alpha < k, \\ \Theta(n^k \log n) & \text{if } \alpha = k, \\ \Theta(n^\alpha) & \text{if } \alpha > k. \end{cases}$$

(b) 2

(c) *priority_queue* <*T*>'s are implemented with max-heaps. If there existed a function

```
T& top();
```

then a user of the class could modify the root of the heap and break the invariant that each node is greater than or equal to its children.

Proposed solution to problem 2

(a) Yes

(b) No (leaves must be black)

(c) No (the children of a red node must be black)

(d) Yes

(e) No (it must be a binary search tree)

(f) No (the number of black nodes in a path from the root to a leaf must be constant)

Proposed solution to problem 3

(a) Let $C(n)$ be the cost of the algorithm in the worst case as a function of n. Then:

- If $s \geq \frac{1}{2}$: $C(n) = C(sn) + \Theta(1)$
- If $s \leq \frac{1}{2}$: $C(n) = C((1-s)n) + \Theta(1)$

(b) The worst case takes place, for instance:

- if $s \geq \frac{1}{2}$, when $x < v[l]$.
- if $s \leq \frac{1}{2}$, when $x > v[r]$.

(c) We can solve the recurrences separately, or notice that

$$C(n) = C(\max(s, 1-s)n) + \Theta(1) = C(\frac{n}{\frac{1}{\max(s,1-s)}}) + \Theta(1)$$

As $0 < s < 1$, we have $0 < \max(s, 1-s) < 1$, and so $\frac{1}{\max(s,1-s)} > 1$. By applying the master theorem for dividing recurrences we have that $\alpha = k = 0$, and therefore the cost is $\Theta(\log n)$ (independently of s).

Proposed solution to problem 4

(a) A possible solution:

```
#include <limits>

const int oo = numeric_limits<int>::max();

int minimum_cost(const vector<vector<pair<int,int>>>& G, int x, int y) {
  vector<int> cost(G.size (), +oo);
  cost [x] = 0;
  deque<int> dq;
  dq.push_back(x);
  while (not dq.empty()) {
    int u = dq.front ();
    dq.pop_front ();
    if (u == y) return cost [y];
    for (auto p : G[u]) {
      int v = p. first ;
      int w = p.second;
      if ( cost [v] > cost[u] + w) {
        cost [v] = cost [u] + w;
        if (w == 0) dq.push_front (v);
        else        dq.push_back(v);
      } } }
  return −1;
}
```

(b) There is a cost that is proportional to the number of vertices (initialization of the vector of costs). But the cost of the loop is proportional to the number of edges that are visited (being thus the worst case when all edges are visited). In total the cost is $\Theta(|V| + |E|)$.

(c) $\Theta((|V| + |E|) \log |V|)$

Proposed solution to problem 4

(a) A possible solution:

```
bool ok(const vector<int>& s, const set<pair<int,int>>& D) {
  for (auto d : D)
    if (s[d. first ] == s[d.second])
      return false;
  return true;
}

bool has_solution (int k, vector<int>& s, const inp_DISEQUALITIES& e) {
  if (k == e.n) return ok(s, e.D);
  for (int v = e.l; v ≤ e.u; ++v) {
    s[k] = v;
    if ( has_solution (k+1, s, e)) return true;
  }
  return false;
}
```

```
bool has_solution (const inp_DISEQUALITIES& e) {
  vector<int> s(e.n);
  return has_solution (0, s, e);
}
```

(b) When there is no solution, the algorithm considers each of the $(u - l + 1)^n$ possible assignments of the n variables to the values in $[l, u]$. For each of these assignments, work is performed with cost $\Omega(1)$. Hence, the cost in the worst case is $\Omega((u - l + 1)^n)$.

(c) A possible solution:

```
inp_DISEQUALITIES reduction(const inp_COLORING& ec) {
  inp_DISEQUALITIES ed;
  ed.l = 1;
  ed.u = ec.c;
  ed.n = ec.G.size ();
  for (int u = 0; u < ec.G.size (); ++u)
    for (int v : ec.G[u])
      if (u < v)
        ed.D.insert ({u, v});
  return ed;
}
```

(d) No. By the previous exercise we have that, as COLORING is NP-hard, then so is DISEQUALITIES. Thus, if there were a polynomial algorithm for solving DISEQUALITIES, we would have $P = NP$, and we would have solved a long-standing open problem in theoretical computer science.

Solution of the Final EDA Exam 12/01/2017

Proposed solution to problem 1

(a) (0.25 pts.) A graph with n vertices has $O(n^2)$ edges.

(b) (0.25 pts.) A connected graph with n vertices has $\Omega(n)$ edges.

(c) (0.25 pts.) A complete graph with n vertices has $\Omega(n^2)$ edges.

(d) (0.25 pts.) A min-heap with n vertices has $\Theta(n)$ leaves.

(e) (0.25 pts.) A binary search tree with n vertices has height $\Omega(\log n)$.

(f) (0.25 pts.) A binary search tree with n vertices has height $O(n)$.

(g) (0.25 pts.) An AVL tree with n vertices has height $\Omega(\log n)$.

(h) (0.25 pts.) An AVL tree with n vertices has height $O(\log n)$.

Proposed solution to problem 2

(a) Breadth-first search.

(b) Dijkstra's algorithm.

(c) Bellman-Ford's algorithm.

(d) There cannot be any cycle with negative weight in the graph.

(e) By induction over the number of arcs of the path.

- **Base case:** If the path has not arc, the source vertex u is the same as the target vertex v. So in this case we have that $\omega_\pi(c) = \omega(c) = 0$. Since $\omega(c) - \pi(u) + \pi(v) = 0$, what we wanted to prove holds.
- **Inductive case:** Assume that the path has k arcs, that is, is of the form $(u_0, u_1, \ldots, u_k)$, where $u_0 = u$ and $u_k = v$. As $u_1, \ldots, u_k$ is a path from u_1 to u_k with $k-1$ arcs, we can apply the induction hypothesis. So $\omega_\pi(u_1, \ldots, u_k) = \omega(u_1, \ldots, u_k) - \pi(u_1) + \pi(u_k)$. But

$$\begin{aligned}
\omega_\pi(u_0,\ldots,u_k) &= \omega_\pi(u_0,u_1) + \omega_\pi(u_1,\ldots,u_k)\\
&= \omega_\pi(u_0,u_1) + \omega(u_1,\ldots,u_k) - \pi(u_1) + \pi(u_k)\\
&= \omega(u_0,u_1) - \pi(u_0) + \pi(u_1) + \omega(u_1,\ldots,u_k) - \pi(u_1) + \pi(u_k)\\
&= \omega(u_0,u_1) - \pi(u_0) + \omega(u_1,\ldots,u_k) + \pi(u_k)\\
&= \omega(u_0,\ldots,u_k) - \pi(u_0) + \pi(u_k)
\end{aligned}$$

(f) If π is a potential, then the reduced weights ω_π are non-negative. So we can apply Dijkstra's algorithm to compute the distances with weights ω_π from s to all vertices. Then we can compute the distances with weights ω using the following observation: if $u, v \in V$ i c is the minimum path with weights ω from u to v (which exists by hypothesis), then c is the minimum path with weights ω_π from u to v and $\omega(c) = \omega_\pi(c) + \pi(u) - \pi(v)$.

Proposed solution to problem 3

(a) If M is an $n \times n$ matrix, function *matrix mystery*(**const** *matrix*& M) computes $M^{\sum_{i=1}^{n} i}$, or equivalently, $M^{\frac{n(n+1)}{2}}$.

(b) The product of two matrices $n \times n$, which is computed by function *aux*, takes $\Theta(n^3)$ time. Since $\Theta(n)$ iterations are performed, and in each of them two matrix products are computed with cost $\Theta(n^3)$, in total the cost is $\Theta(n^4)$.

(c) A possible solution:

```
matrix exp(const matrix& M, int k) {
  if (k == 1) return M;
  matrix P = exp(M, k/2);
  if (k % 2 == 0) return aux(P, P);
  else            return aux(aux(P, P), M);
}

matrix mystery(const matrix& M) {
  int n = M.size ();
  return exp(M, n*(n+1)/2);
}
```

Fast exponentiation makes $\Theta(\log(n(n+1)/2)) = \Theta(\log(n))$ matrix products, each of which takes $\Theta(n^3)$ time. In total, the cost is $\Theta(n^3 \log n)$.

Proposed solution to problem 4

(a) The witness is p.

(b) The code of the verifier is between lines 4 and 16.

(c) A possible solution:

```
bool ham2_rec(const vector<vector<int>>& G, int k, int u, vector<int>& next) {
  int n = G.size ();
  if (k == n)
    return find (G[u].begin (), G[u].end (), 0) ≠ G[u].end ();

  for (int v : G[u])
    if (next[v] == −1) {
      next[u] = v;
      if (ham2_rec(G, k+1, v, next)) return true;
      next[u] = −1;
    }
  return false;
}

bool ham2(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> next(n, −1);
  return ham2_rec(G, 1, 0, next);
}
```

(d) One can replace the call

```
return find(G[u].begin(), G[u].end(), 0) ≠ G[u].end();
```

by

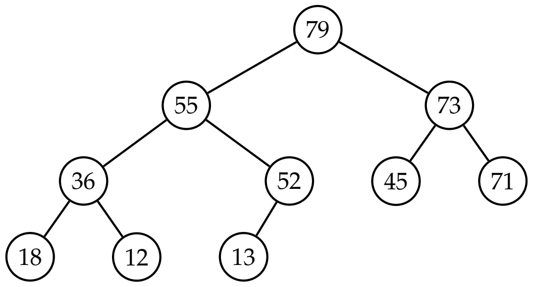
```
return not G[u].empty() and G[u][0] == 0;
```

(e) If G is not connected then it cannot be Hamiltonian. In this case the function only needs to return **false**.

Solution of the Final EDA Exam 09/06/2017

Proposed solution to problem 1

(a) The resulting max-heap is:



(b) To prove that P $\subseteq$ co-NP we only need to see that if L is a problem from class P, then its complement $\overline{L}$ belongs to NP. But if L is from P then there is a deterministic polynomial algorithm A that decides L. Let us consider now the algorithm $\overline{A}$ that does the same as A, but returns 1 when A returns 0, and returns 1 when A returns 0. Then $\overline{A}$ decides $\overline{L}$, and since $\overline{A}$ takes polynomial time, we have $\overline{L} \in$ P $\subseteq$ NP.

(c) The cost $C(n)$ of f in function of n follows the recurrence

$$C(n) = 3C(n/3) + \Theta(1)$$

as there are 3 recursive calls on subvectors of size $n/3$ and additionally operations of constant cost are performed. By the Master Theorem of Divisive Recurrences, the solution to the recurrence is $C(n) = \Theta(n)$.

(d) It is not true. For example, $n^{2n} = (n^n)^2$ grows asymptotically faster than n^n.

Proposed solution to problem 2

We must use a dictionary with integer keys implemented with a hash table, and a vector which is initially empty that will contain the intersection.

First of all we pass over A and add all its elements to the dictionary. We make n insertions to the dictionary, each of which takes time $O(1)$ on average. So the first pass costs $O(n)$ on average.

In the second place we pass over B and, for each of its elements, we check if it already belongs to the dictionary. If so, the element is added to the vector of the intersection with a *push_back*. Otherwise nothing is done. Hence we make m lookups to the dictionary, each of which takes time $O(1)$ on average. Since each *push_back* takes constant time, the cost of the second pass is $O(m)$ on average.

In total, the cost is $O(n+m)$ on average.

Proposed solution to problem 3

(a) The filled table is:

level :

A	B	C	D	E	F	G	H
0	1	1	1	4	2	0	3

depth : $\boxed{4}$

(b) The filled table is:

	(1)	(2)	(3)	(4)
TRUE		X	X	
FALSE	X			X

(c) A possible solution:

```
vector<int> levels(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> lvl(n, −1), pred(n, 0);

  for (int u = 0; u < n; ++u)
    for (int v : G[u])
      ++pred[v];

  queue<int> Q;
  for (int u = 0; u < n; ++u)
    if (pred[u] == 0) {
      Q.push(u);
      lvl [u] = 0;
    }

  while (not Q.empty()) {
    int u = Q.front (); Q.pop();
    for (int v : G[u]) {
      −−pred[v];
      lvl [v] = max(lvl[v], lvl [u]+1);
      if (pred[v] == 0) Q.push(v);
  } }
  return lvl ;
}
```

The construction of the vectors has cost $\Theta(n)$. The first loop has cost $\Theta(n+m)$. The second loop has cost $\Theta(n)$. The third loop has cost $\Theta(n+m)$. In total the cost is $\Theta(n+m)$.

Proposed solution to problem 4

```
void write(const vector<int>& p, int n) {
  for (int k = 0; k < n; ++k) cout << " " << p[k];
  cout << endl;
}

void generate(int k, int n, vector<int>& p, vector<bool>& used) {
      if (k == n) write(p, n);
      else {
        for (int i = 0; i < n; ++i)
          if (not used[i] and k ≠ i) {
            used[i] = true;
            p[k] = i;
            generate(k+1, n, p, used);
            used[i] = false;
        }
        }
};

void generate_all (int n) {
  vector<int> p(n);
  vector<bool> used(n, false);
  generate (0, n, p, used);
}
```

Solution of the Final EDA Exam 19/01/2018

Proposed solution to problem 1

(a) A possible solution:

```
void shift_down (vector<int>& v, int i) {
    int n = v.size()−1;
    int c = 2∗i;
    if (c ≤ n) {
        if (c+1 ≤ n and v[c+1] < v[c]) c++;
        if (v[i] > v[c]) {
            swap(v[i],v[c]);
            shift_down(v, c);
} } }
```

(b) A possible solution:

```
void update (vector<int>& v, int i, int x) {
  int y = v[i];
  v[i] = x;
  if (y < x) shift_down(v, i);
  else       shift_up ( v, i);
}
```

Proposed solution to problem 2

```
int n, k;
vector<int> perm;
vector<bool> used;

void write() {
  cout ≪ perm[1];
  for (int i = 2; i ≤ n; ++i) cout ≪ ' ' ≪ perm[i];
  cout ≪ endl;
}

void generate(int i, int lm, int mx) {
  if (lm > k or n − i + 1 + lm < k) return;
  if (i == n+1) write();
  else {
    for (int j = 1; j ≤ n; ++j)
      if (not used[j]) {
        used[j] = true;
        perm[i] = j;
        if (j > mx) generate(i+1, lm+1, j);
        else        generate(i+1, lm, mx);
        used[j] = false;
      }
  }
}
```

```
int main() {
  cin >> n >> k;
  perm = vector< int>(n+1);
  used = vector<bool>(n+1, false);
  generate (1, 0, 0);
}
```

Proposed solution to problem 3

(a) Given a **string** s of size n, function *mystery* computes the evaluation of s as a polynomial in x: $\sum_{i=0}^{n-1} s[i] \cdot x^i$.

The value 0 would not be adequate for x if one wanted to use *mystery* as a hash function because the function would always return $s[0]$; in other words, all strings that start with the same character would have the same hash value.

(b) The answers:

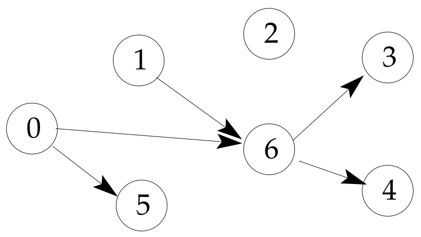
	(1)	(2)	(3)	(4)
TRUE		X		X
FALSE	X		X	

(c) The value returned by *find*(2) is 0. The value returned by *find*(9) is 5.

(d) In the worst case, the searched value is strictly greater than all values in the vector. Hence at each recursive call the algorithm chooses the larger of the two parts, which is $\frac{2}{3}$ of the size of the interval $[l, \cdots, r]$. Moreover, apart from the recursive call only operations of constant cost are performed. So, if $n = r - l + 1$, the cost $T(n)$ in the worst case is determined by the recurrence $T(n) = T(\frac{2}{3}n) + \Theta(1)$. By applying the master theorem of divisive recurrences, we have that the solution to the recurrence is $\Theta(\log n)$.

Proposed solution to problem 4

(a) The composition is:



(b) A possible solution:

```
matrix comp(const matrix& G1, const matrix& G2) {
  int n = G1.size ();
  matrix C(n, vector<int>(n, false ));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = 0; k < n and not C[i][j]; ++k)
        C[i][j] = G1[i][k] and G2[k][j];
  return C;
}
```

In the worst case (for example, when the graphs G_1 and G_2 are empty) the three loops go over the values from 0 to $|V| - 1$. As each iteration of the innermost loop has constant cost, in total the cost in the worst case is $\Theta(|V|^3)$.

(c) An alternative way of obtaining the adjacency matrix C of the composition would be to compute the product P of the adjacency matrices of G_1 and G_2 as matrices of integer numbers, and then take $C[i][j] = 1$ if and only if $P[i][j] > 0$.

As the product of matrices can be efficiently computed by using Strassen's algorithm in time $\Theta(|V|^{\log_2 7})$, this would be the cost of the resulting algorithm.

Solution of the Final EDA Exam **20/06/2018**

Proposed solution to problem 1

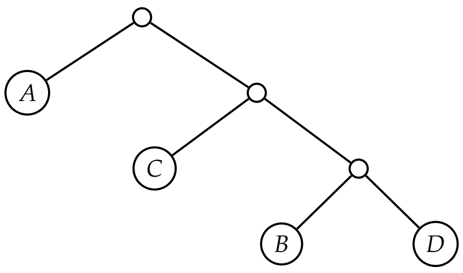
(a) *priority_queue* <*T*>'s are implemented with max-heaps. If there existed a function

T& *top*();

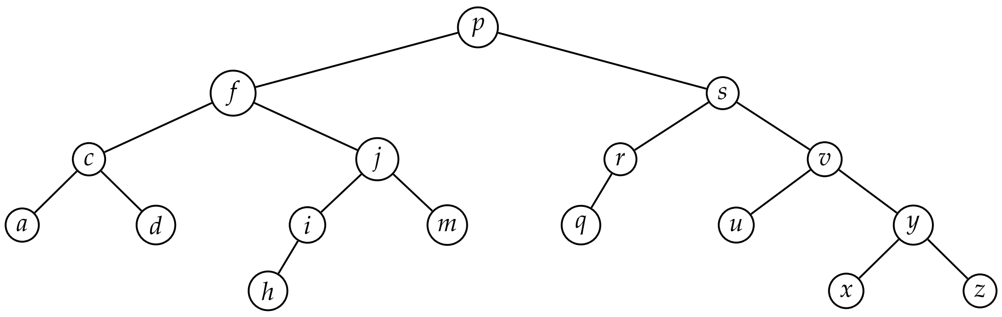
then a user of the class could modify the root of the heap and break the invariant that each node is greater than or equal to its children.

(b)

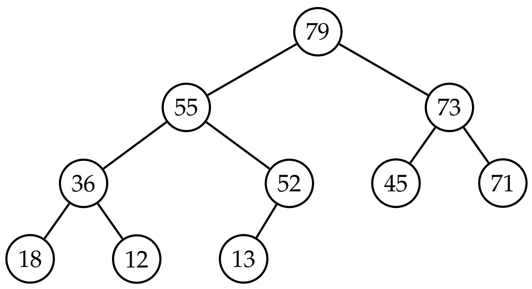
A	0
B	110
C	10
D	111



(c)



(d)



Proposed solution to problem 2

(a) We define function $U(m) = T(b^m)$. Then $T(n) = U(\log_b(n))$. Moreover, we have:

$$U(m) = T(b^m) = T(b^m/b) + \Theta(\log^k(b^m)) = T(b^{m-1}) + \Theta(m^k \log^k(b)) =$$

$$= T(b^{m-1}) + \Theta(m^k) = U(m-1) + \Theta(m^k)$$

The Master Theorem of subtractive recurrences claims that if we have a recurrence of the form $U(m) = U(m-c) + \Theta(m^k)$ with $c > 0$ and $k \geq 0$, then $U(m) = \Theta(m^{k+1})$. So the solution to the recurrence of the statement is $T(n) = \Theta((\log_b(n))^{k+1}) = \Theta(\log^{k+1} n)$.

(b) A possible solution:

```
bool search(const vector<int>& a, int x, int l, int r) {
  if (l == r) return x == a[l];
  int m = (l+r)/2;
  auto beg = a.begin();
  if (a[m] < a[m+1])
    return search(a, x, m+1, r) or binary_search(beg + l,    beg + m + 1, x);
  else
    return search(a, x, l,    m) or binary_search(beg + m+1, beg + r + 1, x);
}

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

(c) The worst case takes place for instance when x does not appear in a. In this situation the cost $T(n)$ is described by the recurrence $T(n) = T(n/2) + \Theta(\log n)$, as we make one recursive call over a vector of size $\frac{n}{2}$, and the cost of the non-recursive work is dominated by the binary search, which has cost $\Theta(\log(\frac{n}{2})) = \Theta(\log(n))$. By applying the first section we have that the solution is $T(n) = \Theta(\log^2(n))$.

Proposed solution to problem 3

(a) A possible solution is:

```
bool two_col_aux(const vector<vector<int>>& g, int u, vector<bool>& col,
                 vector<bool>& marked, bool is_red) {
  col[u] = is_red;
  marked[u] = true;
  for (int v : g[u]) {
    if (not marked[v]) {
      if (not two_col_aux(g, v, col, marked, not is_red)) return false;
    }
    else if (col[v] == col[u]) return false;
  }
  return true;
}
```

(b) The problem of 3-colorability is NP-complete. Therefore, if the claim in the forum were right, we would have a polynomial-time algorithm that solved an NP-complete problem, and so P = NP. Since up to now the problem of whether P = NP is still open (and, moreover, the dominant conjecture is that it is false), what is said in the forum is not plausible.

Proposed solution to problem 4

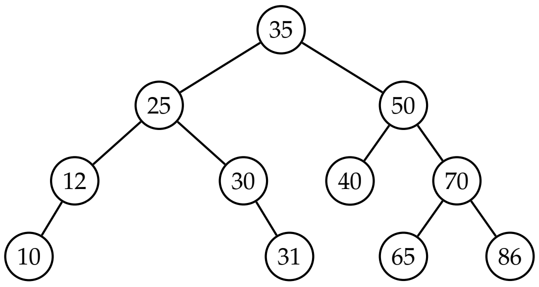
(a) $\boxed{\mathit{sum\_chosen} + \mathit{sum\_rest} \text{ - } a[k] >= l}$ $\boxed{\mathit{sum\_chosen} + a[k] <= u}$ $\boxed{\mathit{sols}(0, 0, s)}$

(b) It is only required to swap the recursive calls: that is, swap the two **if**s (together with their corresponding bodies) inside of the **else**.

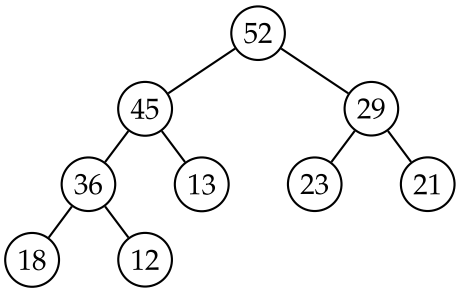
Solution of the Final EDA Exam 14/01/2019

Proposed solution to problem 1

(a) The resulting AVL tree:



(b) The resulting max-heap:

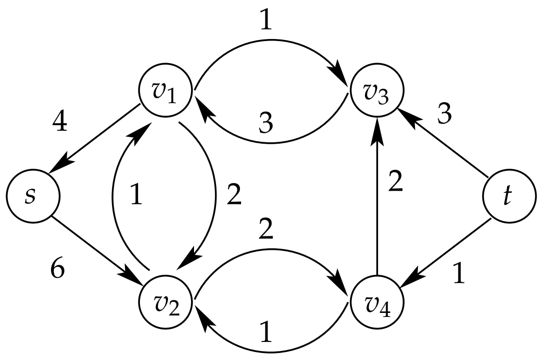


(c) $\Theta(n)$

(d) $\Theta(\sqrt{n}\log n)$

(e) $\Theta(\sqrt{n})$

(f) The residual network:



Proposed solution to problem 2

(a) The function *mystery* computes the product of the polynomial p by the monomial $c \cdot x^k$. Its cost in time is $\Theta(n+k)$.

(b) The type *unordered_map*<**int**,**int**> can be implemented with hash tables.

A way of completing the code:

```
polynomial mystery(const polynomial& p, int c, int k) {
  polynomial q;
  for (auto mon : p)
    q[mon.first + k] = c * mon.second;
  return q;
}
```

The cost of the function is dominated by the cost of the loop. Let $m = p.size()$. We do m iterations, and in each of them the most expensive operation is to access to a pair of the dictionary of q. Each access to this dictionary has cost $O(1)$ in the average case. Hence the cost in the average case is at most $O(m)$.

(c) The cost of the function is $\Theta(n)$.

(d) A possible solution (similar to Karatsuba's algorithm):

```
polynomial operator*(const polynomial& p, const polynomial& q) {
  int n = p.size ();
  if (n == 1) return polynomial(1, p[0]*q [0]);
  int n2 = n/2;

  polynomial p0(n2), p1(n2);
  for (int k = 0; k < n2; ++k) p0[k] = p[k];
  for (int k = n2; k < n; ++k) p1[k − n2] = p[k];

  polynomial q0(n2), q1(n2);
  for (int k = 0; k < n2; ++k) q0[k] = q[k];
  for (int k = n2; k < n; ++k) q1[k − n2] = q[k];

  polynomial p0_q0 = p0 * q0;
  polynomial p1_q1 = p1 * q1;
  polynomial p1_q0_plus_p0_q1 = ((p0 + p1) * (q0 + q1)) +
    mystery(p0_q0 + p1_q1, −1, 0);

  return p0_q0 +
    mystery(p1_q0_plus_p0_q1, 1, n2) +
    mystery(p1_q1, 1, n);
}
```

The function makes 3 recursive calls over polynomials of size $\Theta(n/2)$. Moreover, the non-recursive work (additions, calls to function *mystery*) has cost $\Theta(n)$. If we call $T(n)$ the cost of the function in terms of n, then we obtain the recurrence $T(n) = 3T(n/2) + \Theta(n)$. By applying the master theorem of divisive recurrences we obtain that the cost is $T(n) = \Theta(n^{\log 3})$, which is better than $\Theta(n^2)$.

Proposed solution to problem 3

(a) The efficiency bug consists is that every time we choose a new subset, we scan the whole vector S again, without considering the previously chosen subsets. This means that we can take the same subset several times, or consider different permutations of the same collection of subsets.

A possible way of fixing the code is to add a parameter that is the index of the first subset that we need to consider in the following choice:

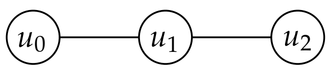
```
bool set_cover_rec (int f, int K, int N, const vector<set<int>>& S,
                    vector<int>& w, int c) {
  if (c == N) return true;
  if (K == 0) return false;
  for (int i = f; i < S.size (); ++i) {
    for (int x : S[i]) {
      if (w[x] == 0) ++c;
      ++w[x];
    }
    if ( set_cover_rec (i+1, K−1, N, S, w, c)) return true;
    for (int x : S[i]) {
      −−w[x];
      if (w[x] == 0) −−c;
    }
  }
  return false;
}

bool set_cover (int K, int N, const vector<set<int>>& S) {
  vector<int> w(N, 0);
  return set_cover_rec (0, K, N, S, w, 0);
}
```

(b) Let us see the function g defines a reduction from VERTEX-COVER to SET-COVER. Clearly it takes polynomial time in the size of the input of VERTEX-COVER. It remains to be seen that an input (K,G) of VERTEX-COVER is positive if and only if $g(K,G) = (K,N,S)$ is also positive for SET-COVER.

If (K,G) is positive for VERTEX-COVER then there is a vertex cover $W \subseteq V$ with $|W| \leq K$. We define $T = \{S_u \,|\, u \in W\}$. Then $|T| \leq K$. Now let i be such that $0 \leq i < N$. Then the edge $e_i = \{u,v\}$ satisfies that $i \in S_u$, $i \in S_v$. Moreover, as W is a cover, $u \in W$ or $v \in W$. If $u \in W$ then $S_u \in T$ and therefore we have a subset of T that includes i. The case $v \in W$ is analogous. Hence T is a subset cover of $\{0,\ldots,N-1\}$.

Conversely, if (K,N,S) is positive for SET-COVER then there is a subset cover $T \subseteq S$ of $\{0,\ldots,N-1\}$ with $|T| \leq K$. We define $W = \{u \in V \,|\, S_u \in T\}$. Then $|W| \leq K$. Now let $e_i \in E$ be an edge. As T is a cover, there is $S_u \in T$ such that $i \in S_u$. So u is an endpoint of e_i and moreover $u \in W$. In conclusion, W is a vertex cover of G.

On the other hand, the function f does not define a correct reduction. For example, suppose that $K = 1$ and G is the following graph:



Then the input (K,G) is positive for VERTEX-COVER, as vertex u_1 covers both edges. But $f(K,G) = (1,3,\{\{0,1\},\{1,2\}\})$ is not positive for SET-COVER, since there is no single subset that covers $\{0,1,2\}$.

(c) If VERTEX-COVER reduces polynomially to SET-COVER and moreover we have that VERTEX-COVER is NP-hard (given that it is NP-complete), then we can say that SET-COVER is NP-hard: given a problem X of class NP, we can compose the reduction from X to VERTEX-COVER with the reduction from VERTEX-COVER to SET-COVER, and in this way obtain a reduction from X to SET-COVER.

But with only that we cannot deduce that SET-COVER is NP-complete, as we need to justify that it belongs to NP. So let us see that indeed SET-COVER belongs to NP. In this case the witnesses are the size k and the k indices of the subsets that form the set cover. Clearly the witnesses have size not greater than the size of the input. And verifying that a choice of k subsets gives a set cover only requires to check that every number from 0 to $N - 1$ belongs to at least one of the subsets. This can be done in polynomial time in the size of the input as the subsets of the input form a set cover of $\{0, \ldots, N - 1\}$ and therefore the size of the input is $\geq N$.

Solution of the Final EDA Exam 17/06/2019

Proposed solution to problem 1

(a) Let $S(n)$ be the size of the formula $\text{xor}(x_1,...,x_n)$. From the definition we get the recurrence $S(n) = 4S(n/2) + \Theta(1)$. By applying the Master Theorem of Divisive Recurrences, we have that $S(n) = \Theta(n^2)$.

(b) Let R be the function of the statement. Let us see that R is not a reduction from k-COLOR to $k+1$-COLOR. It should satisfy that, given a graph G, if $R(G)$ is $k+1$-colorable then G is k-colorable. But this property is not met: for example, for $k = 2$, if G is a cycle of 3 vertices, we have that $R(G) = G$ is 3-colorable, but G is not 2-colorable.

(c) The problem of maximum bipartite matching.

Proposed solution to problem 2

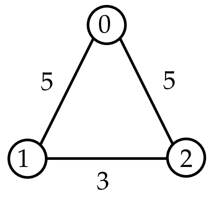
(a) A possible solution:

Graph g = { {{3, 1}, {4, 2}}, {{3, 0}, {5, 2}}, {{4, 0}, {5, 1}} };

(b) At the end of the execution of function *mystery*, the value of $a[u]$ is the distance from vertex 0 to vertex u.

(c) Function *mystery* implements Dijkstra's algorithm for finding the shortest paths from vertex 0 to all vertices of the graph g. Given that the graph is connected, every vertex can be reached from 0. Since T consists of the edges of the shortest paths from 0, it is a spanning tree.

But it is not necessarily a minimum spanning tree. For example, function *mystery* applied to this graph:



returns the edges {{0, 1}, {0, 2}}. The corresponding tree has cost 10. But the tree corresponding to edges {{0, 1}, {1, 2}} is also a spanning tree and has cost 8.

Proposed solution to problem 3

A possible solution:

```
node* pred(const node*& T, int x) {
  if       (T == NULL) return NULL;
  else if  (T−>key > x)  return pred(T−>left, x);
  else {
    node* U = pred(T−>right, x);
    if  (U == NULL) return T;
    else               return U;
  }
}
```

```
node* succ(const node*& T, int x) {
  if        (T ==  NULL) return NULL;
  else  if  (T−>key < x)  return succ(T−>right, x);
  else  {
    node* U = succ(T−>left, x);
    if  (U == NULL) return T;
    else                   return U;
  }
}

pair<node*, node*> pred_succ(const node*& T, int x) {
  return {pred(T, x), succ(T, x)};
}
```

Proposed solution to problem 4

(a) A possible solution:

```
struct TSP {

  vector<vector<double>> D;
  int n;
  vector<int> next, best_sol ;
  double best_tot_dist ;

  void recursive (int v, int t , double c) {
    if (t == n) {
      c += D[v][0];
      if (c < best_tot_dist ) {
        best_tot_dist = c;
        best_sol = next;
        best_sol [v] = 0;
      }
    }
    else for (int u = 0; u < n; ++u)
        if (u ≠ v and next[u] == −1) {
          next[v] = u;
          recursive (u, t+1, c+D[v][u]);
          next[v] = −1;
        }
  }

  TSP(const vector<vector<double>>& D) {
    this−>D = D;
    n = D.size ();
    next = vector<int>(n, −1);
    best_tot_dist = DBL_MAX;
    recursive (0, 1, 0);
  }
};
```

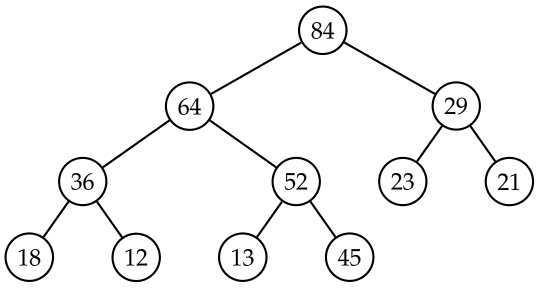
(b) A possible solution:

```
void recursive (int v, int t, double c) {
  if (c ≥ best_tot_dist ) return;
  if (t == n) {
```

Solution Final Exam EDA 13/01/2020

Proposed solution to problem 1

(a) The resulting heap is:



(b) $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$

(c) If we realize that the function call $g(p)$ is constant with respect to n, it is easy to see that the recurrence describing the cost of function f is $T(n) = T(n/2) + \Theta(1)$. Hence, the asymptotic cost is $\Theta(\log n)$.

Proposed solution to problem 2

(a) The filled table is:

level :

A	B	C	D	E	F	G	H
0	1	1	1	4	2	0	3

depth : $\boxed{4}$

(b) The filled table is:

	(1)	(2)	(3)	(4)
TRUE		X	X	
FALSE	X			X

(c) A possible solution:

```
vector<int> levels(const vector<vector<int>>& G) {
  int n = G.size ();
  vector<int> lvl(n, −1), pred(n, 0);

  for (int u = 0; u < n; ++u)
    for (int v : G[u])
      ++pred[v];

  queue<int> Q;
  for (int u = 0; u < n; ++u)
    if (pred[u] == 0) {
      Q.push(u);
      lvl [u] = 0;
    }

  while (not Q.empty()) {
    int u = Q.front (); Q.pop();
    for (int v : G[u]) {
      --pred[v];
      lvl[v] = max(lvl[v], lvl[u]+1);
      if (pred[v] == 0) Q.push(v);
   } }
  return lvl;
}
```

The construction of the vectors has cost $\Theta(n)$. The first loop has cost $\Theta(n+m)$. The second loop has cost $\Theta(n)$. The third loop has cost $\Theta(n+m)$. In total the cost is $\Theta(n+m)$.

Proposed solution to problem 3

(a)
```
struct GasStations {
  int n;
  vector<int> cost;
  vector<vector<int>> roads;
  vector<bool> best_solution;
  int best_cost ;

  void rec(int i, vector<bool>& partial_sol, int partial_cost ) {
    if (i == n) {
      best_cost = partial_cost ;
      best_solution = partial_sol ;
    }
    else {
      if ( partial_cost + cost[i] < best_cost ) {
        partial_sol [i] = true;
        rec(i+1, partial_sol , partial_cost + cost[i ]);
      }

      bool needed = false;
      for (auto& c : roads[i])
        if (c < i and not partial_sol [c]) needed = true;

      if (not needed) {
        partial_sol [i] = false;
        rec(i+1, partial_sol , partial_cost );
      } } } // tanquem rec
```

(b) Since we are asked to maximize installation cost, the optimal solution will place a gas station in each city and the cost will be the sum of the costs in array *c*. Hence, we would implement this idea from scratch.

Proposed solution to problem 4

(a) An assignment that satisfies exactly one literal per clause is

$I(x_1)=0,\ I(x_2)=1,\ I(x_3)=1,\ I(x_4)=1,\ I(x_5)=0$

There is yet another solution:

$I(x_1)=0,\ I(x_2)=1,\ I(x_3)=1,\ I(x_4)=0,\ I(x_5)=1$

There is no assignment satisfying exactly one literal per clause and such that $I(x_1)=1$. If we have $I(x_1)=1$ then, due to the first clause, we can infer that necessarily $I(x_2)=1$, $I(x_3)=0$. Due to the second clause, necessarily $I(x_4)=I(x_5)=0$. All in all, we have no possibility to satisfy the third clause.

A set of 3-literal clauses that is a positive input for **3-SAT** and negative for **ONE-IN-THREE-SAT** is:

$$\begin{array}{ccccc} x_1 & \vee & a & \vee & b \\ \neg x_1 & \vee & a & \vee & b \\ \neg x_1 & \vee & \neg a & \vee & \neg b \end{array}$$

The assignment $I(a)=1$, $I(b)=0$, $I(x_1)=0$ satisfies at least one literal per clause.

In order to see that we cannot satisfy exactly one literal per clause, let us realize that if we set $I(x_1)=1$, then necessarily $I(a)=I(b)=0$ due the first clause, but this already satisfies 2 literals in the last clause. If we set $I(x_1)=0$, a similar situation happens due to the second clause.

(b) Let I be an assignment that satisfies exactly one literal in each clause of $R(C)$. By reduction ad absurdum, let us assume that I does not satisfy at least one literal in C. Then, we have that $I(l_1)=I(l_2)=I(l_3)=0$.

Due to the first clause of $R(C)$, necessarily $I(a)=I(b)=0$ and due to the third clause, we have $I(c)=I(d)=0$. But then I does not satisfy the second clause of $R(C)$, which is a contradiction.

Let I be an assignment that satisfies at least one literal in C and let us try to build an assignment I' extending I and satisfying exactly one literal in each clause of $R(C)$.

If $I(l_2)=1$, then we take $I'(b)=I'(c)=0$, $I'(a)=I(l_1)$, $I'(d)=I(l_3)$.

Otherwise, $I(l_2)=0$.

- If $I(l_1)=1$, then we build $I'(a)=I'(c)=0, I'(b)=1, I'(d)=I(l_3)$.
- If $I(l_1)=0$, we know that $I(l_3)=1$ and we consider $I'(c)=1, I'(a)=I'(b)=I'(d)=0$.

(c) In order to see that **ONE-IN-THREE-SAT** is NP-hard, we will construct a reduction from **3-SAT**. Given a 3-CNF F with clauses $\{C_1,C_2,\dots,C_m\}$, the reductions builds a 3-CNF F' with clauses $\{R(C_1),R(C_2),\dots,R(C_m)\}$. The resulting 3-CNF F' has polynomial size wrt F because it has $3m$ clauses and we add $4m$ new variables. Obviously, it can be computed in polynomial time.

We have now to prove that F is a positive input for **3-SAT** iff F' is a positive input for **ONE-IN-THREE-SAT**. For this purpose we will use part b).

If $F \in$ **3-SAT** then we consider an assignment I satisfying it and extend it to an assignment I' satisfying exactly one literal in each clause of F'. Hence, $F' \in$ **ONE-IN-THREE-SAT**. Note that we can do this because each $R(C_i)$ uses disjoint fresh variables.

Si $F' \in$ **ONE-IN-THREE-SAT**, then any assignment that satisfies exactly one literal in each clause of F' satisfies at least a literal in each clause of F. Hence $F \in$ **3-SAT**.

(d) Since **ONE-IN-THREE-SAT** is NP-hard, we only have to prove that it is in NP. In this particular case, witness candidates are assignments, that have polynomial size wrt the formula. Checking whether a witness is correct is also polynomial, since we only have to traverse the formula and check that exactly one literal in each clause is satisfied. Finally, it is obvious that correct witnesses only exist for formulas where one can satisfy exactly one literal per clause.

Hence, since **ONE-IN-THREE-SAT** is NP and NP-hard, we can conclude that it is NP-complete.

Solution of the Final EDA Exam 09/06/2020

Proposed solution to problem 2A

```
void recursive ( ){
  int u = s.back ();
  if (s.size () == n) {
    for (int v : G[u]) {
      if (v == 0){
        found = true;
        sol = s;
      } } }
  else {
    for (int v : G[u]) {
      if (not used[v]) {
        s.push_back(v);
        used[v] = true;
        recursive ();
        s.pop_back ();
        used[v] = false;
        if (found) return; } } } }
```

Proposed solution to problem 2B

```
void recursive (int c) {
  int u = s.back ();
  if (s.size () == n) {
    c += M[u][0];
    if (c < best_cost ) {
      best_cost = c;
      best_sol = s;
    }
  }
  else {
    for (int v = 0; v < n; ++v)
      if (not used[v]) {
        if (c + M[u][v] < best_cost ) {
          s.push_back(v);
          used[v] = true;
          recursive (c + M[u][v]);
          s.pop_back ();
          used[v] = false; } } } }
```

Proposed solution to problem 3A

a) We can see that G represents a directed graph, and hence we can talk about vertices (pieces) and arcs (pairs with a concrete order).

The loop that computes *indegree* traverses the whole graph (vertices and arcs) and hence has cost $\Theta(n + m)$. Then, there are n insertions to the priority queue, each with cost $O(\log n)$.

Finally, the main loop is executed n times, since it starts with n elements in Q and at each iteration one element is removed until it is empty. Therefore, we have n calls to *remove_min*, each with cost $O(\log n)$. The internal loop that calls *decrease_key* is executed at most m times, one per arc. In order to analyze the cost of *decrease_key* we have to take into account the call to $find$, with cost $O(n)$ and the shift up with cost $O(\log n)$. Hence, the total cost of the m calls is in total $O(m \cdot n)$.

Summing up, we have a cost of $O(n + m + n \log n + m \cdot n) = O(n \log n + m \cdot n)$.

b) The key is to realize that the problem is asking for a topological sort of graph G. We know that this can be done in time $O(n + m)$. More concretely, we start by computing *in_degree* as in the given code. Then, we place into a container (stack, queue, list, etc), all vertices with indegree zero. While the container is not empty, we remove an element and decrement by 1 the *in_degree* of all adjacent vertices, adding to the container the vertices that have now *in_degree* zero.

c) We have to add a new method to the class (comparison function) and modify 3 lines:

```
// CHANGE (new function)
bool smaller (pair<Elem,Key>& p1, pair<Elem,Key>& p2) {
  return (p1.second < p2.second or
        (p1.second == p2.second and p1.first < p2.first ));
}

void shift_up (int i) {
  if (i ≠ 1 and smaller(v[i ],v[i /2])) { // CHANGE
    swap(v[i ], v[i /2]);
    shift_up (i /2);
} }

void shift_down (int i) {
 int n = size ();
 int c = 2*i;
 if (c ≤ n) {
   if (c+1 ≤ n and smaller(v[c+1],v[c ])) c++; // CHANGE
   if (smaller (v[c ],v[i ])) { // CHANGE
     swap(v[i ],v[c ]);
     shift_down (c );
   } } }
```

Proposed solution to problem 3B

a) The cost of function *find* is $O(n)$.

Insertions. Vertex x is inserted at the beginning of the function. In all other insertions we can see that whenever we insert a vertex into Q, it holds that Q does not contain it and the vertex has never been removed from Q previously (that is, the vertex has never been in Q). Hence, every vertex can only be inserted once.

Removals. Since every vertex is only added at most once, it is obvious that we can only remove it at most once.

Decrements. We can only call function *decrease_key* when we remove an element u from the queue and explore the arcs that get out of u. There can be, at most, one call for each outgoing arc from u. Hence, in total we can call *decrease_key* at most m times.

Cost. Function *decrease_key* first calls $find$, with cost $O(n)$ and then shifts an element up in the heap, with cost $O(\log n)$. Hence, its cost is $O(n)$.

The total cost is given by the n calls to *remove_min*, that imply time $O(n \log n)$, the n calls to *insert*, that imply time $O(n \log n)$ and the m calls to *decrease_key*, with time $O(m \cdot n)$. Therefore, the total time is $O(n \log n + m \cdot n)$.

b) We have to add an attribute to the class, reimplement $find$ and add 5 lines

```
unordered_map<Elem,int> elem2idx; // CHANGE: add attribute

void insert (const pair<Elem,Key> & x) {
  assert (not contains (x. first ));
  v.push_back(x);
  elem2idx[x. first ] = v. size () − 1; // CHANGE
  shift_up ( size ());
}

pair<Elem,Key> remove_min () {
  if (empty()) throw "Priority queue is empty";
  pair<Elem,Key> x = v[1];
  v[1] = v.back ();
  elem2idx[v [1]. first ] = 1; // CHANGE
  v.pop_back ();
  shift_down (1);
  elem2idx. erase (x. first ); // CHANGE
  return x;
}

// CHANGE: method reimplemented
int find (const Elem& x) {
  const auto& it = elem2idx.find (x);
  if ( it == elem2idx.end()) return −1;
  else return it −>second;
}

void shift_up (int i) {
  if (i ≠ 1 and v[i/2].second > v[i ].second) {
    swap(elem2idx[v[i ]. first ],elem2idx[v[ i /2]. first ]); // CHANGE
    swap(v[i ], v[ i /2]);
    shift_up ( i /2);
} }

void shift_down (int i) {
  int n = size ();
  int c = 2*i ;
  if (c ≤ n) {
    if (c+1 ≤ n and v[c+1].second < v[c ].second) c++;
    if (v[i ].second > v[c ].second) {
      swap(elem2idx[v[i ]. first ],elem2idx[v[c ]. first ]); // CHANGE
      swap(v[i ],v[c ]);
      shift_down (c );
} } }
```

c) If we repeat the analysis of a), the only thing that changes is that *decrease_key* now has cost $O(\log n)$. Hence, the total cost is $O((n+m)\log n)$.

   If $m = \Theta(n^2)$, the cost of the initial solution is $O(n \log n + n^2 \cdot n) = O(n^3)$, whereas with the new implementation of $find$ it has cost $O((n+n^2)\log n) = O(n^2 \log n)$. Therefore, there is indeed an asymptotic improvement.

Proposed solution to problem 4A

a) It is a positive input because {1,2,3,4} is a subset of size 4 that satisfies the requested properties.

b) It is a positive input because {2,3,5} is a subset of size 3 that satisfies the requested properties.

c) In order to show that ENEMIES ∈ NP, we need to find a non-deterministic polynomial algorithm.

   - Witnesses: the set of possible witnesses is made of all subsets of T of size r. It is easy to see that the size of a witness is smaller than the size of the input.
   - Verifier: the verifier will receive (T,E,r) and a witness W and will check, for each of the $r(r-1)/2$ pairs u,v of elements in W, that $\{u,v\} \notin E$. Each check has cost at most $O(q)$, and since $r \leq |T|$ it all has polynomial cost $O(q \cdot |T|^2)$.
   - Only positives inputs have verifiable witnesses: by definition, an input is positive if, and only if, there exists a subset $M \subseteq T$ of size r such that any pair of elements of M does not belong to E. This subset M corresponds to a witness, and hence verifiable witnesses only exist for positive instances.

d) Given an input for VILLAGES (P,C,k) we build an input for ENEMIES (T,E,r) as follows:

   - $T = P$
   - $E = pairs(P) \setminus C$, where $pairs(P) = \{\{u,v\} \mid u,v \in P \text{ i } u \neq v\}$.
   - $r = k$

   Building (T,E,r) from (P,C,k) can be done in polynomial time. The only necessary computation is to determine the set E, but this can be done by considering $pairs(P)$, that has quadratic size and for each pair perform a linear search in C.

   It is also easy to see that (T,E,r) has polynomial size wrt (P,C,k), since we only change E, that has size at most quadratic wrt the size of P.

Finally we have to prove that $(P,C,k) \in$ VILLAGES $\Longleftrightarrow (T,E,r) \in$ ENEMIES.

$\Longrightarrow$) If $(P,C,k) \in$ VILLAGES, this means that there exists $S \subseteq P$ of size k such that for all $u,v \in S$ with $u \neq v$ it holds that $\{u,v\} \in C$. Then, if we consider $M = S$, we realize that M is a subset of T with size r (because $r = k$), and that if we take any two different elements $u,v \in M$, it holds that $\{u,v\} \in C$ and hence $\{u,v\} \notin pairs(P) \setminus C = E$. Therefore, the existence of M proves that (T,E,r) is a positive input for ENEMIES.

$\Longleftarrow$) If $(T,E,r) \in$ ENEMIES, this means that there exists $M \subseteq T$ of size r such that for any pair $u,v \in M$ with $u \neq v$ it holds that $\{u,v\} \notin E$. Let us consider $S = M$. We know that S is a subset of P of size k (because $r = k$), and that if we take any pair $\{u,v\} \in S$ it holds that $\{u,v\} \notin E = pairs(P) \setminus C$, which implies $\{u,v\} \in C$. Hence, the existence of S proves that (P,C,k) is a positive input for VILLAGES.

e) In order to prove that it is NP-complete we have to see that it belongs to NP and is NP-hard. We can prove it as follows. In c) we have proved that ENEMIES $\in$ NP. In d) we have seen that VILLAGES can be reduced to ENEMIES. Since we now know that VILLAGES is NP-complete, and hence NP-hard, we can conclude that ENEMIES is NP-hard.

Proposed solution to problem 4B

a) It is a positive input because $\{1,4,5,6\}$ is a subset of size 4 that satisfies the requested properties.

b) It is a positive input because $\{1,2\}$ is a subset of size 2 that satisfies the requested properties.

c) In order to prove that RESEARCHERS $\in$ NP, we have to find a non-deterministic polynomial algorithm for it.

- Witnesses: the set of possible witnesses is made of all subsets of I of size r. It is easy to see that the size of a witness is smaller than the size of the input.
- Verifier: the verifier will receive (I,M,r) and a witness W and will check, for all meetings $\{u,v\} \in M$ (we have q meetings in total), that u or v belong to W. Each check can be done with cost at most $O(r)$ (size of W), and hence in total the verifier will take $O(q \cdot r)$ time. Since $r \leq |I|$, the latter is polynomial wrt the size of (I,C,r).
- Only positive inputs have verifiable witnesses: by definition, an input is positive if, and only if, there exists a subset $S \subseteq I$ of size r such that for any meeting $\{u,v\} \in M$, at least one of the two element belongs to S. This subset S corresponds to a witness, and hence verifiable witnesses only exist for positive instances.

d) Given an input for TEAM (T,D,k) we build an input for RESEARCHERS (I,M,r) as follows:

- $I = T$
- $M = pairs(T) \setminus D$, where $pairs(T) = \{\{u,v\} \mid u,v \in T \text{ i } u \neq v\}$.
- $r = |T| - k$

Building (I,M,r) from (T,D,k) can be done in polynomial time. The only necessary computation is to determine set M, but this can be done by traversing $pairs(T)$, that has quadratic size and for each element do a linear search in D.

It is also easy to see that (I,M,r) has polynomial size wrt (T,D,k), since we only change M, that has size at most quadratic wrt the size of T, and r, which also has polynomial size wrt (T,D,k).

Finally we have to prove that $(T,D,k) \in$ TEAM $\Longleftrightarrow (I,M,r) \in$ RESEARCHERS.

$\Longrightarrow$) If $(T,D,k) \in$ TEAM, this means that there exists $E \subseteq T$ of size k such that for any pair $u,v \in E$ with $u \neq v$ it holds that $\{u,v\} \in D$. Let us consider $S = T \setminus E$. We can see that S is a subset of I of size $r = |T| - k$, and that if we take a pair $\{u,v\} \in M$ then $u \in S$ o $v \in S$. This last point can be proved as follows: if it were not true, then $u \notin S$ and $v \notin S$, and since $S = T \setminus E$, we know that $u \in E$ and $v \in E$. Thanks to the properties of E, we know that $\{u,v\} \in D$, but this contradicts the fact that $\{u,v\} \in M = \mathit{pairs}(T) \setminus D$. Hence, the existence of S proves that (I,M,r) is a positive input for RESEARCHERS.

$\Longleftarrow$) If $(I,M,r) \in$ RESEARCHERS, this means that there exists $S \subseteq I$ of size $r = |T| - k$ such that for any pair $\{u,v\} \in M$ it holds that $u \in S$ or $v \in S$ (or both). Let us consider $E = T \setminus S$. We can see that E is a subset of T of size $|T| - (|T| - k) = k$ and that, moreover, if we consider two different vertices $u,v \in E$ we have $\{u,v\} \in D$. If it were not true, there should be two vertices $u,v \in E$ with $\{u,v\} \notin D$. Hence, $\{u,v\} \in M$. But if $u,v \in E = T \setminus S$ we know that $u,v \notin S$. This latter fact contradicts the initial property of S. Hence, the existence of E proves that (T,D,k) is a positive input for TEAM.

e) In order to prove that it is NP-complete we have to see that it belongs to NP and is NP-hard. We can prove it as follows. In c) we have proved that RESEARCHERS $\in$ NP. In d) we have seen that TEAM can be reduced to RESEARCHERS. Since we now know that TEAM is NP-complete, and hence NP-hard, we can conclude that RESEARCHERS is NP-hard.

Solution EDA Final Exam 08/01/2021

Proposed solution to problem 1

(a) We can see that the function visits each element of the vector once. For each element, the command `++M[x]` looks for x in the dictionary and increments its key. After that, there is another search for x in the dictionary. Hence the asymptotic cost is n times the cost of searching.

If we use an AVL tree, each search has cost $O(\log n)$ in the worst case, and hence the total cost is $O(n \log n)$.

If we use a hash table, each search has average cost $\Theta(1)$, and hence the total cost is $\Theta(n)$.

(b) We have to analyze the recursive function *majority_pairs*. Until the first loop, all instructions take constant time. The loop has cost $\Theta(n)$, and creates a vector of size at most $n/2$. Hence, the recursive call is done on a vector of size at most $n/2$. After that, a call is made to function *times* on a vector of size n. The cost of this function, since the vector is passed by reference, can be described by $C(n) = C(n-1) + \Theta(1)$, with solution $C(n) \in \Theta(n)$. All in all, the cost of function *majority_pairs* can be described by the recurrence $T(n) = T(n/2) + \Theta(n)$. Applying the master theorem, we can state that the worst-case cost is $\Theta(n)$.

Proposed solution to problem 2

(a) The resulting code is:

```
void write_choices (vector<int>& partial_sol, int partial_sum, int idx) {
  if (partial_sum > money) return;
  if (idx == p.size ()) {
    if (partial_sum == money) {
      cout <<"{";
      for (int i = 0; i < partial_sol.size (); ++i)
        cout << (i == 0 ? "" : ",") << partial_sol[i];
      cout <<"}" << endl;
    }
  }
  else {
     partial_sol.push_back(idx);
     write_choices ( partial_sol , partial_sum + p[idx], idx+1);
     partial_sol.pop_back ();
     write_choices ( partial_sol , partial_sum, idx+1);
  }
}
```

(b) In the previous code, we prune the solution when we have spent more money than we have.

Additionally, we can prune a partial solution when, even by considering that we choose all apartments yet to be processed, we cannot reach the amount of money we have to spend. That is, we prune the search when we have discarded too many apartments.

In order to implement this efficiently, we will add one parameter to the procedure *write_choices*. This parameter will contain the sum of the prices of all apartments yet to be processed. In the main procedure, we will initially sum all prices and this will be the value of the parameter in the first call to *write_choices*. In the two recursive calls, the parameter will be decremented by $p[idx]$. Finally, if we call *remaining_sum* this parameter, *write_choices* will start with the line:

if $(partial\_sum + remaining\_sum < money)$ **return**;

Proposed solution to problem 3

(a) Yes, it is a positive instance. For example, we can group them in 3 sets $\{3,8,20\}, \{6\}, \{22\}$. This distribution satisfies that numbers that have 1s in common positions are not in the same set. Note that:

$3 = 00011_2$, $6 = 00110_2$, $8 = 01000_2$, $20 = 10100_2$, $22 = 10110_2$

(b) We have to see three things:

- *Witnesses:* the witness candidates are all possible ways of distributing elements of N into p sets. The size of a candidate is polynomial in the size of the instance. In fact, it is linear.
- *Verifier:* it will receive a pair (N, p) and p sets $S_1, S_2, \cdots, S_p$ in which numbers in N have been distributed. For each set S_i, it will consider all pairs of numbers in S_i (at most a quadratic number of pairs in each S_i), and it will check that the binary representations of the pair have no 1 in a common position (this can be done in time linear in the number of bits of the largest number). All in all, this takes polynomial time.
- *Witenesses only for positive instances:* an instance is positive if and only if there is a way to distribute the numbers is sets in a correct way and this distribution is precisely a witness. Hence, positive instances have witnesses, whereas negative instances do not.

(c) The size of (N, p) is polynomial in the size of (G,k). On the one hand, $p = k$. On the other hand, N contains a number for each vertex of G, and the size of these numbers is the number of edges in G.

Let us see that $(G,k) \in$ COLORABILITY $\Leftrightarrow (N, p) \in$ DISTINCT-ONES:

$(G,k) \in$ COLORABILITY $\Rightarrow (N, p) \in$ DISTINCT-ONES:

Since (G,k) is a positive instance, there exists a function c that colors vertices with k colors in such a way that adjacent vertices have different color.

We can distribute the numbers x_i in p (that is equal to k) sets as follows: x_i goes to set S_j if and only if $c(v_i) = j$. We only need to check that two numbers with 1s in common positions never go to the same set. By reductio ad absurdum: let us assume that x_r and x_s are two numbers that go to a set S_j and have a 1 at position i. Since they both have a 1 at position i, vertices v_r and v_s are incident to edge e_i. Since they both go to set S_j, we have $c(v_r) = c(v_s) = j$. And this is not possible because adjacent vertices have different colors.

$(N, p) \in$ DISTINCT-ONES $\Rightarrow (G, k)$:

Since (N, p) is a positive instance, we can distribute the numbers in N in p sets $S_1, S_2, \cdots, S_p$ such that if two numbers have a 1 at a common position they go to different sets.

Let us consider the following coloring for vertices in G: $c(v_i) = j$ if and only if x_i belongs to set S_j. Since $p = k$, this coloring uses at most k colors. Moreover, if we take vertices v_r and v_s incident to an e_i, we know by definition that x_r and x_s will have the i-th bit set to 1, and hence will not go to the same set. Hence, c will assign them a different color.

Proposed solution to problem 4

(a) It is easy to see that the number of nodes $N(k)$ of a binomial tree B_k can be described by the following recurrence:

$$N(k) = \begin{cases} 1, & \text{if } k = 0 \\ 2 \cdot N(k-1), & \text{if } k > 0 \end{cases}$$

We can prove by induction on k that the solution to the recurrence is $N(k) = 2^k$. The base case is trivial, since $2^0 = 1$. Let us now assume the induction hypothesis $N(k-1) = 2^{k-1}$ and we can prove that $N(k) = 2 \cdot N(k-1) = 2 \cdot 2^{k-1} = 2^k$, as we wanted to prove.

(b) Since the number of nodes of a binomial tree of order k is 2^k, and there can be at most one binomial tree of order k for each k, we can see that in a binomial heap with n nodes there will be one (and only one) binomial tree of order k if and only if the k-th least significant bit of n in binary is 1.

(c) If the root of A is smaller than the root of B, we will add B as the leftmost child of the root of A. Otherwise, we will add A as the leftmost child of the root of B.

(d) The completed code is the following:

```
void BinomialHeap::merge(BinomialHeap& h){
// Make sure both have the same size (to make code simpler)
while (h.roots.size() < roots.size()) h.roots.push_back(NULL);
while (h.roots.size() > roots.size()) roots.push_back(NULL);
vector<Tree> newRoots(roots.size());
Tree carry = NULL;
for (int k = 0; k < roots.size(); ++k) {
  if (roots[k] == NULL and h.roots[k] == NULL) {
    newRoots[k] = carry;
    carry = NULL;}
  else if (roots[k] == NULL) {
    if (carry == NULL) newRoots[k] = h.roots[k];
    else {
      newRoots[k] = NULL;
      carry = mergeTreesEqualOrder(carry,h.roots[k]);}
  }
  else if (h.roots[k] == NULL) {
    if (carry == NULL) newRoots[k] = roots[k];
    else {
      newRoots[k] = NULL;
```

```
        $carry = mergeTreesEqualOrder(carry, roots[k]);$}
      }
      **else** {
        $newRoots[k] = carry;$
        $carry = mergeTreesEqualOrder(roots[k], h.roots[k]);$
      }
    }

    **if** $(carry \neq NULL)$ $newRoots.push\_back(carry);$
    $roots = newRoots;$
  }
```

We can check that all commands take constant time and hence we only need to count the number of iterations of the loop. This number is equal to the size of *roots*. It is not difficult to realize that this vector has as many positions as bits needed to represent n, and hence has $\Theta(\log n)$ positions. All in all, the cost is $\Theta(\log n)$.