Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

**IIC1103 – Introducción a la Programación**
**1 - 2016**

# Examen

**Instrucciones generales:**

- El examen consta de cuatro preguntas con igual ponderación (1/4 del total cada una).
- El examen dura 3 horas, y solo se reciben consultas sobre el enunciado en los primeros 30 minutos.
- Las últimas páginas son un recordatorio; no está permitido utilizar material de apoyo adicional.
- Si el enunciado de la pregunta no lo indica explícitamente, no es necesario validar los ingresos del usuario. Puedes asumir que no hay errores por ingreso de datos.
- Puedes definir e implementar todas las funciones o métodos auxiliares que necesites.
- Entrega hojas de respuesta para todas las preguntas, incluso si no contestas.
- Responde cada pregunta en una hoja separada, marcando tu RUT y el número de pregunta en todas las hojas prestando atención a la dirección del trazo, tal como se muestra en el siguiente ejemplo:


# Pregunta 1

Una *secuenciación* de un genoma es una secuencia de caracteres `A, C, G` o `T` (en mayúsculas) que no están separados por un espacio y donde el orden de los caracteres es relevante. Estas secuencias se usan para representar el ADN de organismos.

## Parte a) (20 puntos)

Escribe una función que reciba un string y verifique si ese string es efectivamente la secuenciación de un genoma. Por ejemplo, los strings `'ACGACGAC'` y `'ACGACT'` son secuenciaciones de genomas, pero `'BAC'`, `'AC AC'` o `'acA'` no lo son.

## Parte b) (40 puntos)

Escribe una función que, dado un número `n` y un string `s` y retorne una lista con todos los substrings de `s` de largo `n` que aparezcan solo una vez en `s`. Retorna `False` en caso que no existan substrings que cumplan estas condiciones. Por ejemplo, para `s = 'ACGACG'` los substrings de largo 3 que no se repiten son `'CGA'` y `'GAC'`. En cambio, para `s = 'ACGACGAC'` no existe ningún substring de largo 3 que no se repita.

# Pregunta 2

En Cataluña es tradición jugar a los *castells*, los cuales son torres o castillos humanos de varios pisos que tratan de alcanzar la mayor altura posible (ver Figura 1). Se arman por pisos y cada piso está conformado por una cierta cantidad de personas en un círculo (tomados de los brazos). El objetivo es llegar lo más alto sin caerse en el intento.

El alto del castillo se mide en pisos, siendo la base el piso 0. La probabilidad de que se caiga un castillo se calcula así: Cada piso agrega una probabilidad de caída de $P * N_p$, donde $P$ es el número del piso y $N_p$ el número de personas en ese piso. Por ejemplo, un castillo de 7 pisos (del 0 al 7) con 4 personas en los pisos del 1 al 4 y dos personas en los pisos del 5 al 7, tiene una probabilidad de caída de: $1 * 4 + 2 * 4 + 3 * 4 + 4 * 4 + 5 * 2 + 6 * 2 + 7 * 2 = 76\,\%$.

Debes desarrollar un programa Python para representar los castells incluyendo:

## Parte a) (10 puntos

Una clase `Castell` con los métodos necesarios para poder crear objetos con atributos: nombre del castillo, número de pisos (de 0 a $N$) y cantidad de personas en cada piso.

## Parte b) (15 puntos)

Un método de la clase `Castell` que calcule la probabilidad de caída del *castell* descrita en el enunciado. El método debe retornar dicha probabilidad, la cual es un número entero entre 0 y 100 (ambos inclusive).

## Parte c) (35 puntos)

Un programa principal donde se lea el archivo `castillos.txt`, que contiene en cada línea la descripción de un castillo. Para cada línea se debe crear un objeto de clase `Castell`, calcular la probabilidad de caída y guardar el resultado en un archivo `resultados.txt` solo cuando la probabilidad de caída sea menor o igual a $50\,\%$.

El formato del archivo `castillos.txt` es:

$nombre; n_{pisos}; personas_1; personas_2; personas_3; \ldots; personas_n$

Donde: *nombre* es un string, $n_{pisos}$ es el número de pisos (de 0 a $N$) y $personas_n$ es el número de personas en el piso $n$. El número de personas en la base no es relevante.

El formato del archivo de salida `resultados.txt` es:

$nombre; pisos; total\_personas; probabilidad\_caida$

Donde: Los dos primeros parámetros son los mismos del archivo de entrada, *total_personas* es el número total de personas sin considerar la base y *probabilidad_caida* es la probabilidad que tiene el *castell* de caerse.

Figura 1: Un *castell* de los Castellers de Sants

# Pregunta 3

Para bajar el estrés luego de los exámenes, una buena idea es invitar a tus compañeros de curso a jugar naipes. Piensan en juntarse luego de esta prueba, y decides empezar a ejercitar tus habilidades con las cartas al mismo tiempo que practicas programación.

## Parte a) (30 puntos)

Escribe una función `ordenar_cartas` que reciba una lista de cartas y retorne una lista con las cartas ordenadas según pinta y número. La lista de cartas está en el formato $[[n_1, p_1], [n_2, p_2], ...]$

donde $n_i$ es el número de la carta $i$ y $p_i$ es su pinta.

La pinta $p$ puede ser $C$ (corazones), $P$ (picas), $D$ (diamantes) o $T$ (tréboles). Considera que el orden de las pintas, de izquierda a derecha es corazones, picas, diamantes y tréboles. Un ejemplo de mano ordenada de cartas se muestra en la Figura 2. Los números de las cartas van del 1 al 13. (El As se representa por 1 y las cartas literales (J,Q,R) por 11, 12 y 13.)

## Parte b) (30 puntos)

Una compañera encontró una caja con cartas ordenadas en el centro de alumnos, además de otras cartas sueltas. Implementa una función que reciba la baraja y una carta, inserte la carta en la baraja, y retorne una lista con la baraja ordenada incluyendo la nueva carta. Tanto la baraja como la carta se encuentran en el formato señalado anteriormente.

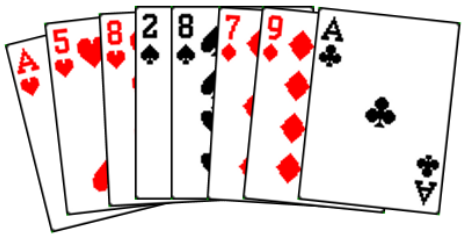

Figura 2: Conjunto de cartas ordenado según número y pinta, de izquierda a derecha

# Pregunta 4

Gary está de vacaciones en Chile. Su club le ha dado `D` días para visitar el país, y quiere saber como podría utilizar esos días.

Gary tiene una lista de ciudades que le gustaría visitar. Cada elemento de esa lista es una lista de dos elementos: el nombre de la ciudad ($n_i$) y la cantidad de días ($d_i$) que necesita para visitar esa ciudad.

Escribe un programa en Python que en forma **RECURSIVA** obtenga todas las combinaciones de ciudades que Gary podría visitar en los `D` días, para una lista de ciudades y días. Cada combinación debe cumplir con que la suma de los días sea <= `D`.

Por ejemplo, si las lista de ciudades es:

```
ciudades = [ [‘‘Arica’’,6], [‘‘La Serena’’, 3], [‘‘Valdivia’’,7] ]
```

y $D = 12$, entonces su salida debe ser:

```
 6 dias: Arica
 9 dias: Arica, La Serena
 3 dias: La Serena
 9 dias: La Serena, Arica
10 dias: La Serena, Valdivia
 7 dias: Valdivia
10 dias: Valdivia, La Serena
```

En el ejemplo se muestran como alternativas diferentes algunas que cambian el orden las ciudades, por ej. `(Arica, La Serena)` y `(La Serena, Arica)`. En tu solución puedes considerar todas estas combinaciones o puedes indicar solo una de ellas. Hazlo como te resulte más fácil. Por otro lado, ten presente que podría haber alguna situación en que no alcance el tiempo para visitar ni una sola ciudad.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC1103- Introducción a la Programación

# Recordatorio contenidos examen - Primer semestre 2016

En los ejemplos, lo marcado como `<texto>` se interpreta como lo que debes rellenar con tu código según cada caso.

## 1. Tipos de datos y operadores

| Tipo de dato | Clase | Ejemplo |
|---|---|---|
| Números enteros | `int` | `2` |
| Números reales | `float` | `2.5` |
| Números complejos | `complex` | `2 + 3j` |
| Valores booleanos | `bool` | `True/False` |
| Cadenas de texto | `str` | `"hola"` |

| Operación | Descripción | Ejemplo |
|---|---|---|
| `+` | Suma | `2.3+5.4` |
| `-` | Resta | `45.45-10.02` |
| `-` | Negación | `-5.4` |
| `*` | Multiplicación | `(2.3+4.2j)*3` |
| `**` | Potenciación | `2**8` |
| `/` | División | `100/99` |
| `//` | División entera | `100//99` |
| `%` | Módulo | `10%3` |

Prioridad (de mayor a menor): `()`; `**`; `*`, `/`, `//` o `%`; `+` o `-` .

| Operación | Descripción | Ejemplo |
|---|---|---|
| `== (!=)` | Igual (distinto) a | `2==2` |
| `<(<=)` | Menor (o igual) | `1<1.1` |
| `>(>=)` | Mayor (o igual) | `3>=1` |
| `and` | Ambos `True` | `2>1 and 2<3` |
| `or` | Algún `True` | `2!=2 or 2==2` |
| `not` | Negación | `not True` |

Prioridad (de mayor a menor): `()`; `or`; `and`; `not`; comparadores.

## 2. Funciones predefinidas

- `int(arg)` convierte `arg` a entero.
- `float(arg)` convierte `arg` a número real.
- `str(arg)` convierte `arg` a cadena de texto (string).
- `list(arg)` genera una lista con elementos según `arg`, que debe ser *iterable* (strings, listas, tuplas, `range`).

## 3. Función `print`

- Un argumento:
  `print(arg)`
- Dos o más argumentos:
  `print(arg1, arg2,arg3)`
- Uso de parámetros, por ejemplo, para eliminar salto de línea y separar con guión:
  `print(arg, sep='-', end='')`

## 4. Función `input`

- `ret = input(texto)` guarda en `ret` un `str` ingresado.
- `ret = int(input(texto))` guarda en `ret` un `int` ingresado.
- `ret = float(input(texto))` guarda en `ret` un `float` ingresado.

## 5. `if/elif/else`

```
if <cond 1> :
    <codigo si se cumple cond 1>
    if <cond 1.1> :
        <codigo si se cumple 1.1>
    else :
        <codigo si no se cumple 1.1>
elif <cond 2> :
    <codigo si se cumple cond 2 pero no cond 1>
else :
    <codigo si no se cumple cond 1 ni cond 2>
```

## 6. `while`

```
while <condicion> :
    <codigo que se ejecuta repetidas
      veces mientras se cumpla
      condicion>
```

## 7. Funciones propias

```
def funcion(<argumentos>):
    <codigo de funcion>
    return <valor de retorno>
```

Variables y parámetros definidos dentro de funciones no son visibles fuera de la función (*scope* local).

# 8. Programación orientada a objetos

```
class <NombreClase>:
    def __init__(self, <parametros>):
        self.<atributos> = <algun parametro>
    def __str__(self):
        <codigo sobrecarga de funcion str()>
        return <string que representa
                        los atributos>
    def <metodo propio>(self, <parametros>):
        <codigo de modulo propio>
```

# 9. Strings, clase `str`

Acceso a caracteres particulares con operador `[]`, partiendo con índice cero. Porción de string con *slice*, por ejemplo si `string='Hola'`, `string[1:3]` es `'ol'`. Algunos métodos y funciones de strings:

- Operador `+`: une (concatena) dos strings.
- Operador `in`: cuando `a in b` retorna `True`, entonces el string `a` está contenido en el string `b`.
- `string.find(a)`: determina si `a` está contenido en `string`. Retorna la posición (índice) dentro de `string` donde comienza la primera aparición del sub-string `a`. Si no está, retorna `-1`.
- `string.upper()`, `string.lower()`: retorna `string` convertido a mayúsculas y minúsculas, respectivamente.
- `string.strip()`: retorna un nuevo string en que se eliminan los espacios en blanco iniciales y finales de `string`.
- `string.split(a)`: retorna una lista con los elementos del `string` que están separados por el string `a`. Si se omite `a`, asume que el separador es uno o más espacios en blanco o el salto de línea.
- `p.join(lista)`: suponiendo que `p` es un string, retorna un nuevo string conteniendo los elementos de la lista "unidos" por el string `p`.
- Función `len(string)`: entrega el número de caracteres de `string`.

Una forma de iterar sobre los caracteres de `string`:

```
for char in string:
    <operaciones con el caracter char>
```

# 10. Listas

Secuencias de elementos-objetos. Se definen como `lista = [<elem1>,<elem2>,...,<elemN>]`. Los elementos pueden o no ser del mismo tipo. Para acceder al elemento `i`, se usa `lista[i]`. La sublista `lista[i:j]` incluye los elementos desde la posición `i` hasta `j-1`. Algunos métodos y funciones de listas:

- Operador `+`. concatena dos listas.
- Operador `in`: `a in b` retorna `True` cuando el elemento `a` está contenido en la lista `b`. Si no está contenido, retorna `False`.
- `lista.append(a)`: agrega `a` al final de la lista.
- `lista.insert(i,a)`: inserta el elemento `a` en la posición `i`, desplazando los elementos después de `i`.
- `lista.pop(i)`: retorna el elemento de la lista en la posición `i`, y lo elimina de `lista`.
- `lista.remove(elem)`: elimina la primera aparición de `elem` en la lista.
- Función `len(lista)`: entrega el número de elementos de `lista`.

Para iterar sobre los elementos de `lista`:

```
for elem in lista:
    <lo que quieran hacer con elem>
```

# 11. Archivos

Abrir un archivo:
`archivo = open(<nombre archivo>,<modo>)`,
p. ej. `archivo = open('archivo.txt','r')`. `<modo>` puede ser `'w'` para escribir un archivo nuevo, `'r'` para leer un archivo (predeterminado), y `'a'` para escribir en un archivo ya existente, agregando datos al final del archivo.

Algunos métodos del objeto que retorna la función `open`:

- `archivo.readline()`: retorna un string con la línea siguiente del archivo, comenzando al inicio del archivo.
- `archivo.write(string)`: escribe en el archivo el string `string`.
- `archivo.close()`: cierra el archivo.

Para leer un archivo entero puedes usar `for`, que iterará línea por línea del archivo:

```
archivo = open('archivo.txt','r')
for linea in archivo:
    <lo que quieran hacer con linea>
```

# 12. Búsqueda y ordenamiento

**Búsqueda secuencial o lineal** Busca secuencialmente un elemento dentro de una lista de tamaño $n$ hasta encontrarlo. Peor caso: elemento no está en lista, $n$ comparaciones. Uso: lista desordenada.

**Búsqueda binaria** Supone lista ordenada. Divide la lista en sublistas dependiendo del valor del elemento buscado: si el elemento es mayor que el elemento medio de la lista, se sigue por la sublista derecha; si no, por la lista izquierda. Peor caso: para una lista de $n$ elementos, se realizan $log_2(n)$ comparaciones.

**Ordenamiento por selección** Busca el índice del elemento más pequeño de la lista, e intercambia los valores entre el índice encontrado y el primer elemento; luego, encuentra el segundo elemento más pequeño, y lo intercambia con el elemento de la segunda posición, realizando la misma operación para el tercero, cuarto, etc..

**Ordenamiento por inserción** Itera por los elementos de la lista, partiendo por el primer elemento, y generando una lista ordenada con los elementos mientras itera. Es similar a cómo una persona ordena una lista de cartas.

**Métodos de Python para ordenamiento**

- `lista.sort()`: ordena `lista` en forma ascendente.

# 13. Recursión y backtracking

Elementos de una función recursiva:

- Caso base: para terminar la recursión.
- Llamada recursiva: llamada a la misma función dentrl de la función, con parámetros distintos que hacen disminuir el problema original.

Para programar una función recursiva, descomponer la función en elementos que puedan ser llamados con subconjuntos de datos. Ejemplo: suma de los primeros $N$ números

```
def suma(N):
    if N == 1: # caso base
        return 1:
    else: #descomponer en N y restantes N-1
        return N + suma(N-1)
```

Backtracking: cuando hay muchas formas o caminos de buscar una solución. Se puede formar un aŕbol con las soluciones, y el algoritmo comienza en el nodo raíz. Cada camino dentro del árbol es un código recursivo, y dentro del código se exploran las ramas del árbol. Ejemplo: encontrar la suma de todas las formas de combinar una lista de números:

```
def suma_conm_lista(lista, sublista=[], suma=0):

    # Caso base: sublista tiene N elementos,
    # o la lista tiene 0 elementos
    if len(lista) == 0:
        print sublista, suma
        return
    else:
        # Quito un elemento de la lista y lo
        # agrego a la sublista para llamar
        # a la función en forma recursiva,
        # y aumento la suma con el
        # elemento que saqué
        for i in range(len(lista)):
            suma_conm_lista( lista[:i]+lista[i+1:],\
                sublista+[lista[i]], suma+lista[i])
```