(75.29 / 95.06) - Teoría de Algoritmos - FIUBA

Informe Trabajo Práctico 1

Grupo : 3 + 1

Integrantes

- Matias Onorato (93179)
- Juan Cruz Opizzi (99807)
- Francisco Strambini (92135)
- Alexis Daciuk (97630)

Introducción: Algoritmo Gale-Shapley

El problema

En 1962 Gale y Shapley estudiaron el problema de emparejar dos grupos, entendiendo como emparejamiento la asignación de un individuo de un grupo con el otro, y viceversa.

En este problema tenemos dos grupos, ***A*** y ***B***, de *n* individuos cada uno, siendo *n* la dimensión del problema. Además, cada individuo del grupo ***A*** y ***B*** tiene una lista de preferencia donde ordena de forma discriminada, estricta y de forma decreciente a cada individuo del del grupo al cual no pertenece, según su orden de preferencia para formar pareja.

El objetivo del problema consiste es crear un emparejamiento en el que cada pareja sea satisfactoria para los individuos que la crean en base a las preferencias de cada uno. es decir crear un emparejamiento estable.

Para el problema planteado, Gale y shapley lograron implementar un algoritmo que logra este emparejamiento mencionado.

```
Inicialmente todos los proponentes estan sin emparejar

MIENTRAS (exista un proponente sin pareja)
  -propone a los que no haya propuesto segun su
  orden de preferencia.
  SI (el propuesto esta libre)
    se acepta propuesta
  SI NO
    SI (el propuesto prefiere a su actual pareja)
      no se arma la pareja
    SI NO
      se arma la pareja
    FIN SI
  FIN SI
FIN MIENTRAS
```

para este algoritmo, Gale y shapley demostraron que:

- El algoritmo termina en un número finito de pasos y termina, a la sumo, en *n^2* pasos.
- El algoritmo produce un emparejamiento estable, es decir que no existe ningun par bloqueante entre un individuo de ***A*** y otro de ***B*** que quiera romper un emparejamiento constituido.
- El algoritmo Gale-Shapley genera un emparejamiento optimo para los proponentes y pesimo para los propuestos.

Parte 1 - Resolución del problema del Club “PICA-PICA”

1.1 Variante de Gale-Shapley con Indiferencias

Encontramos que podemos ejecutar una variante conocida del algoritmo de Gale -Shapley si tomamos un correcto criterio de desempate en caso de ser necesario.

Si a diferencia del algoritmo tradicional, en nuestra variante permitimos más de un jugador en las lista de preferencias con un mismo numero de preferencia, ordenada de forma decreciente, es decir dado $a \in$ ***A***, *x,y,z* $\in$ ***B*** jugadores de los Grupos ***A*** y ***B***, un diccionario `{p:preferencia, j:jugador}` formando la lista `a → {p:1, j:x} →{p:2, j:y} →{p:2, j:z}` como lista de referencia de ***A***, el algoritmo de pareo con indiferencias sigue ofreciendo un correcto funcionamiento, entregando una pareja para cada jugador en un pareo conocido como debilmente estable en los casos donde exista indiferencia. Este pareo sigue ofreciendo un pareo estable por definición, ya que no altera el funcionamiento del algoritmo de Gale-shapley al momento de su comprobación de estabilidad.

La variante es representada por el algoritmo:

```
Inicialmente todos los proponentes estan sin emparejar

MIENTRAS (exista un proponente sin pareja)
  -propone a los que no haya propuesto segun su
  orden de preferencia.
  SI (el propuesto esta libre)
    se acepta propuesta
  SI NO
    SI (el propuesto prefiere debilmente a su actual pareja)
      no se arma la pareja
    SI NO
      se arma la pareja
    FIN SI
  FIN SI
FIN MIENTRAS
```

1.2 - Estabilidad de la Variante y criterios de desempate

Por lo visto en la introducción, el algoritmo de Gale-Shelley es por naturaleza estable, por lo cual analizamos nuestra variación para ver si su nueva implementación produce algún cambio que afecte su estabilidad a la hora de encontrar un posible bloqueo luego de formadas las parejas.

Primer criterio desempate

Si definimos como criterio de desempate en el pareo (y por ende al consultar la preferencia) que la preferencia del proponente tiene que ser estrictamente mayor a la referencia de la pareja del propuesto, por más que esta sea la misma en más de un caso, el algoritmo de Gale-Shapley no encuentra alteraciones en su estabilidad. Esto se debe a que con el criterio mencionado, ante cada intento de formar una pareja entre los jugadores de ambos grupos, al tener que ser esta preferencia estrictamente mayor, ningún jugador podrá cambiar su pareja para ninguno de los casos en los que se repita ese mismo valor de preferencia, manteniendo la comprobación del pareo debilmente estable.

Segundo criterio desempate (solicitado corrección)

Otro ejemplo de desempate quedaría determinado por la comparación de los nombres de los participantes, comparando letra a letra para determinar cuál es estrictamente mayor (en termino de strings). Este criterio no genera redundancias, por lo tanto no afectaría la estabilidad del algoritmo. Frente a cada caso de igual ranking, la comparación de nombres pasa a ser otra relación de comparación "estrictamente mayor", al igual q el metodo del ranking. Solo fallaria si ambos participantes poseen el mismo nombre.

1.3 - Complejidad de la Variante

La ejecución del algoritmo de Gale- Shappley se puede pensar como el recorrido de una matriz formada por las listas de preferencias de los integrantes de un grupo, por ejemplo $a,b,c \in \boldsymbol{A}$ - $x,y,z \in \boldsymbol{B}$ representaría la matriz *3x3*

a	b	c
{p: 1, j: x}	{p:1, j:x}	{p:1, j:z}
{p: 2, j: y}	{p:1, j:z}	{p:1, j:y}
{p: 2, j: z}	{p:3, j:y}	{p:2, j:x}

para la cual se empieza a recorrer cada una de las preferencias hasta que todos los elementos de $\boldsymbol{A}$ forman pareja. En cada caso se ejecuta una simple comparación de complejidad $O(1)$ para formar pareja, y en el peor de los casos hay que recorrer toda la matriz para que todas las parejas queden formadas.

por lo tanto el algoritmo de Gale-Shapley presenta una complejidad $O(n^2)$, siendo n el número de integrantes de un grupo.

1.4 - ¿Es posible un desempate mediante un tiro de moneda?

Esta opción no es posible.

Al igual que el caso considerado al final del punto 1.2, no podemos permitir que ante un empate se generen casos donde el criterio de selección de pareja no sea arbitrario. Si permitimos que el azar resuelva un empate en una lista de preferencias, en lugar de constatar ante cada caso de cambio de pareja el pareo debilmente estable antes mencionado, se podrían generar casos de bloqueo que romperían la estabilidad del pareo.

1.5 - Desarrolle un algoritmo que dado una pareo y las preferencias determina si el mismo es matching estable. ¿Qué complejidad algorítmica tiene?

ver 1.6 para ejecutar el algoritmo

Análisis de Complejidad

analizamos por partes:

parte inicial en el main.py

```
    if punto == "1.5":
        print('Parejas por archivo de pareo:')
        nombre_archivo_pareo = sys.argv[3]
        grupoA, grupoB = armarParejas(jugadores, nombre_archivo_pareo) #
O(armarParejas(n,a))

    if grupoA != None:
        for jugador in grupoA: # O(n)
            print(jugador.nombre + ", " + jugador.pareja.nombre)
        print ('Es estable?:', parejasEstables(grupoA,grupoB)) #
O(parejasEstables(n,n))
```

O(algoritmo 1.5) = O(armarParejas(n,a)) + O(n) + O(parejasEstables(n,n))

carga de parejas por archivo

```
def armarParejas(jugadores, nombre_archivo_pareo):
    grupoA = jugadores[slice(0,len(jugadores)//2)] # O(1)
    grupoB = jugadores[slice(len(jugadores)//2,len(jugadores))] # O(1)

    archivo_pareo = open(Path("../../assets/txt/" + nombre_archivo_pareo),"r") #
O(1)
    lineas_pareo = archivo_pareo.read().splitlines() # O(n)

    for pareo in lineas_pareo: # O(n)
        jugador1 = next(filter(lambda x: x.nombre == pareo.split(',')
[0].strip(),jugadores)) # O(n)
        jugador2 = next(filter(lambda x: x.nombre == pareo.split(',')
[1].strip(),jugadores)) # O(n)
        jugador1.formarPareja(jugador2) # O(1)

    return grupoA, grupoB # O(1)
```

O(armarParejas(n,a)) = O(n^2)

```
def parejasEstables(grupoA, grupoB):
    for jugadorA in grupoA: # O(n)
        for jugadorB in grupoB: # O(n)
            if jugadorA.pareja != jugadorB and jugadorA.prefiere(jugadorB) and
jugadorB.prefiere(jugadorA): # O(1)
```

```
                return False
    return True
```

O(parejasEstables(n,n)) = O(n^2)

por lo tanto

O(algoritmo 1.5) = O(n^2) + O(n) + O(n^2) O(algoritmo 1.5) = O(n^2)

carga de estructuras iniciales (solicitado corrección)

```
def cargarJugadores(nombre_archivo_jugadores):
    archivoJugadores = open(Path("../../assets/txt/" +
nombre_archivo_jugadores),"r")#O(1)
    lineasJugadores = archivoJugadores.read().splitlines()#O(n)

    jugadores = [Jugador(x.split(',')[0],x.split(',')[1],x.split(',')[2]) for x in
lineasJugadores] #O(n)
    jugadores.sort(key= lambda x : x.ranking) #O(n.log(n))

    archivoJugadores.close()#O(1)

    for j in  jugadores: #O(n)
        archivoPrefs = open(Path("../../assets/txt/" + j.archivoPrefs),"r") #O(1)
        prefs = archivoPrefs.read().splitlines() #O(n)
        j.asignarPreferencias([{"jugador":next(filter(lambda y: y.nombre ==
x.split(',')[0],jugadores)),  "nPref":int(x.split(',')[1])} for x in prefs])
#O(nlog(n)) - O(for x in prefs).O(filter)

    archivoPrefs.close()

    return jugadores
```

O(cargarJugadores(n)) = #O(n) + O(n.log(n)) + O(n).(O(n)+O(nlog(n))) O(cargarJugadores(n)) = #O((n^2)log(n))

1.6 - Programación de los algoritmos

instrucciones ejecución del TP:

Primero, generar los archivos de preferencias y ranking general

```
cd TP1/src/tools
python player_generator.py
```

el mismo guarda un *archivo_jugadores.rank* en la carpeta *TP1/assets/txt* con el nombre segun indica el tp, por ejemplo *20_jugadores.rank* y un archivo de preferencia por jugador *jugador_ranking.pref*

punto 1.1

luego, para ejecutar el punto 1.1 realizar los siguientes pasos:

```
cd TP1/src/Primera_Parte
python main.py 1.1 archivo_jugadores.rank
```

se imprime en pantalla las parejas que forman un pareo estable y se comprueba algoritmicamente dicha estabilidad.

punto 1.5

para ejecutar el punto 1.5 realizar los siguientes pasos: Colocar un archivo *parejas_alternativas.txt* en el directorio *TP1/assets/txt* con las parejas propuestas, junto a los archivos de jugadores y preferencias.

luego ejecutar los siguientes comandos:

```
cd TP1/src/Priemra_Parte
python main.py 1.5 archivo_jugadores.rank parejas_alternativas.txt
```

La salida del programa imprimira las parejas cargadas y si las mismas son estables o no segun sus archivos de preferencias.

1.7 - ¿Tiene su programa la misma complejidad algorítmica que la teórica? (revisión para correción)

Analizo complejidad del algoritmo de Gale-Shapley implementado

```
def armarParejasEstables(jugadores):
    grupoA = jugadores[slice(0,len(jugadores)//2)] # O(1)
    grupoB = jugadores[slice(len(jugadores)//2,len(jugadores))] # O(1)

    proponentesLibres = len(grupoA)

    while proponentesLibres>0 : # O(n)
        for proponente in grupoA: # O(n)
            if proponente.pareja is None: # O(1)
                propuesto = proponente.proximoCandidato() # O(1)
                if propuesto.pareja is None: # O(1)
                    proponente.formarPareja(propuesto) # O(1)
                    proponentesLibres -= 1  # O(1)
                elif propuesto.prefiere(proponente): # O(log(n)) (prefiere(a)
busqueda binaria de vector ordenado)
                    proponente.formarPareja(propuesto) # O(1)
    return grupoA, grupoB
```

O(armarParejasEstables(n)) = O((n^2).log(n))

Esta complejidad difiere de la complejidad teórica de gale shapley tradicional O(n^2), pero mejora frente a la complejidad de la primer entrega que marcaba O(n^3).

Parte 2 - Funciones matemáticas / estadísticas

2.1 - Proponga algoritmos para cada una de las resoluciones

los algoritmos estan propuestos en */TP1/src/Segunda_Parte*

en *implementación_lista.py* estan implementadas todas las funciones para la lista y el array. En python encontramos la particularidad que los array y las listas funcionan de la misma forma, ambas quedan determinadas por el comando []

en *implementación_lista_ordenada.py* estan implementadas todas las funciones para el vector ordenado

en *implementación_abb.py* estan implementadas todas las funciones para nuestra implementación elegida, un arbol de busqueda binario.

2.2 - Analisis de complejidad algorítmica

Antes que nada aclaramos que para el calculo de complejidad "n" siempre es la cantidad de elementos de la estructura a menos que se indique lo contrario

Lista y vector en python:

Máximo

Recorremos todo el vector / lista quedandonos con el mayor elemento visitado, esto claramente es O(n) en tiempo y O(1) en espacio porque usamos una variable para guardar el maximo.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
maximo = vector[0]

Para cada elemento del vector:
   si vector[i] > maximo:
     maximo = vector[i]


Devuelvo maximo
```

Media

Como es necesario recorrer todos los elementos de la lista para obtener la sumatoria, esto es O(n), esto se guarda en una variable por lo que es O(1) en espacio.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
  suma = 0

  Para cada elemento del vector:
    suma + = vector[i]

  media = suma / largo(vector)

  Devuelvo media
```

Moda

Como recorremos toda la lista, esto es O(n) y como usamos una lista para guardar los elementos mas frecuentes, en espacio es O(n) en el peor caso, que seria que todos los elementos tienen igual frecuencia, en ese caso devolvemos la lista entera.

- Temporal: *O(n)*
- Espacial: *O(n)*

```
  inicializo la lista de mas frecuentes y mi actual con el primer elemento del
vector
  frecuencia = 0
  frecuencia_actual = 0
  recorro el vector:

    si el numero que estoy parado no es el actual:
      mi actual pasa a ser el nuevo numero y seteo su frecuencia en 0

    sumo 1 a la frecuencia del actual

    si por ahora mi actual es el unico elemento de maxima frecuencia:
      mi lista de mas frecuentes es solo el actual
      y la frecuencia maxima es la frecuencia del actual

    si el actual tiene igual frecuencia que los demas elementos de mayor
frecuencia:
      agrego el actual a la lista de mas frecuentes

    Devuelvo lista de los mas frecuentes
```

Mediana

Lo que hacemos primero es ordenar la lista, lo cual es O(n log n) en tiempo. Luego, si la lista tiene una cantidad par de elementos, devolvemos el promedio de los 2 elementos medios de la lista, de lo contrario la mediana es el valor en el medio.

- Temporal: *O(n log n)*
- Espacial: *O(1)*

```
Ordeno el vector de menor a mayor

Si largo(vector) es impar :
  La mediana es el elemento vector[(( largo -1 ) / 2)]
Sino:
  La mediana es el promedio de los 2 elementos medios del vector
```

Desviación estándar

Calcular la media es O(n) en tiempo y O(1) en espacio, despues se recorre toda la lista calculando la suma de las distancias medias siendo la distancia media de cada elemento, el elemento menos la media al cuadrado, lo cual es O(n) en tiempo y O(1) en espacio, almacenar le media de sumas es O(1) en espacio y O(n) en tiempo (porque calculamos el largo de la lista, esto se podria haber calculado en el ciclo anterior pero O(n) + O(n) = O(n)). Finalmente, calcular la raiz cuadrada es O(log n) en tiempo, y O(1) en espacio (es inplace). Finalmente, en complejidad temporal esta funcion es O(n) + O(n) + O(n) + O(log n) = O(n) + O(log n) = O(n) y en complejidad espacial es O(1).

- Temporal: *O(n)*
- Espacial: *O(1)*

```
media = media(vector)

suma_distancias = 0

Por cada elemento del vector:
  distancia_media = (vector[i] - media ) ^ 2

  suma_distancias + = distancia_media

suma_media = suma_distancias / largo(vector)

desviacion_estandar = raiz_cuadrada(suma_media)

devuelvo desviacion_estandar
```

Permutaciones del conjunto

Como este algoritmo tiene que generar todas las permutaciones posibles y a la vez tiene que devolver dichas combinaciones (guardadas en una lista de listas) este algoritmo es tanto O(n!) en espacio como en tiempo.

- Temporal: *O(n!)*
- Espacial: *O(n|)*

```
permutaciones = []

generar_permutaciones(k, lista):
  si k == 1:
```

```
      agrego la lista a la lista de permutaciones
      Retorno
    para i entre 0 y k-1:
      generar_permutaciones(k-1, lista)
      si i es par:
        swap(i, k-1)
      sino:
        swap(0, k-1)

  generar_permutaciones(largo(lista), lista)
  Devuelvo la lista de permutaciones
```

Variaciones del conjunto tomados de r elementos (r«n)

Este algoritmo es una version modificada de una implementacion de la libreria standard de Python, la cual es "itertools.combinations(iterable, r)": Devuelve subsecuencias de r elementos de la lista permitiendo que los elementos individuales se repitan más de una vez. Las combinaciones se emiten en ordenamiento lexicográfico. Entonces si la lista está ordenada, las tuplas combinadas se producirán en forma ordenada. Los elementos se tratan como únicos en función de su posición, no de su valor. Entonces, si los elementos de la lista son únicos, las combinaciones generadas también serán únicas. La complejidad de este algoritmo es O(n+r-1)! en tiempo, pero dado que el 1 se desprecia y r siempre es menor a n se deduce:

O(n+r-1)! = O(n+r)! = O(n+n)! = O(2*n)! = O(n)!

En complejidad espacial tambien es O(n)! ya que guardamos una lista de todas las combinaciones posibles de longitud r.

- Temporal: *O(n!)*
- Espacial: *O(n!)*

```
  n = largo(lista)
  variaciones = []
  indices = lista de rango r
  variaciones.append(list(lista[i] for i in indices))
  Cilo:
      para i en reversa hasta r:
        si indices[i] != i + n - r:
          return
        sino:
          return
        indices[i] ++
        para j entre i+1 y r:
          indices[j] = indices[j - 1] + 1
          variaciones.append(list(number_list[i] for i in indices))
  Devuelvo la lista de variaciones
```

Variaciones con repetición del conjunto de r elementos (r«n)

Este algoritmo, al igual que el anterior, es una version modificada de una implementacion de la libreria standard de Python, exactamente esta "itertools.combinations_with_replacement(iterable, r)": La unica diferencia con las varicaiones sin elementos repetidos es esta: Los elementos se tratan como únicos en función de su posición, no de su valor. Entonces, si los elementos de la lista son únicos, las combinaciones generadas también serán únicas. La complejidad tanto espacial y temporal es la misma que la de variaciones de r elementons no repetidos, ya que en esencia son la misma idea con un poco de variacion.

- Temporal: *O(n!)*
- Espacial: *O(n!)*

```
n = largo(lista)
variaciones = []
indices = [0] * r
variaciones.append(list(lista[i] for i in indices))
Ciclo:
  desde i en reversa hasta r:
  si indices[i] != n - 1:
    return
  else:
    return
  indices[i:] = [indices[i] + 1] * (r - i)
  variaciones.append(list(lista[i] for i in indices))
  Devuelvo la lista de variaciones
```

Vector ordenado:

Máximo

En un vector ordenado descendente el primer elemento siempre tiene el maximo valor. Por ello esto solo requiere visitar un solo elemento y guardarlo en una sola variable.

- Temporal: *O(1)*
- Espacial: *O(1)*

```
Devuelvo el primer elemento del vector
```

Media

Es el mismo algoritmo que el usado para la lista y el vector, ya que en ambas estructuras no cambia el como se obtiene la media.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
recorro el vector:
  sumatoria = sumar todos los elementos
```

```
    Devuelvo la sumatoria dividido el largo de la lista
```

Moda

El pseudo codigo es exactamente el mismo algoritmo usado en 'lista y vector'.

- Temporal: *O(1)*
- Espacial: *O(n)*

Mediana

Como sabemos que posiciones del vector hay que visitar y solo son un par de posiciones, esto es O(1), tanto en tiempo como en espacio.

- Temporal: *O(1)*
- Espacial: *O(1)*

```
    si el vector es impar:
      Devuelvo el elemento del medio
    sino:
      Devuelvo el promedio de los 2 elementos del medio
```

Desviación estándar

El pseudo codigo es exactamente el mismo algoritmo usado en 'lista y vector'.

- Temporal: *O(n)*
- Espacial: *O(1)*

Permutaciones del conjunto

El pseudo codigo es exactamente el mismo algoritmo usado en 'lista y vector'.

- Temporal: *O(n!)*
- Espacial: *O(n|)*

Variaciones del conjunto tomados de r elementos (r«n)

El pseudo codigo es exactamente el mismo algoritmo usado en 'lista y vector'.

- Temporal: *O(n!)*
- Espacial: *O(n|)*

Variaciones con repetición del conjunto de r elementos (r«n)

El pseudo codigo es exactamente el mismo algoritmo usado en 'lista y vector'.

- Temporal: *O(n!)*

- Espacial: *O(n|)*

Arboy de Busquda Binaria:

Máximo

- Bajamos todo a la derecha, donde va a estar el maximo, tarda O(log N) (caso normal) Tarda O(N) en el peor caso que seria que el arbol sea una lista enlazada. El mejor caso es O(1) (la raiz es el maximo)
- Temporal: *O(log n)*
- Espacial: *O(1)*

```
mientras el nodo a la derecho no sea nulo:
    mi actual es el nodo derecho
Devuelvo el actual que va a ser el maximo
```

Media

Recorremos inorder el ABB contando y sumando todos los nodos, lo cual es O(n) en tiempo. Y usamos un par de variables para guardar los resultados, O(1) en espacio.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
suma = sumo todos los nodos
cantidad = cuento todos los nodos
calculo el promedio como suma / cantidad
Devuelvo el promedio
```

Moda

Recorremos de forma inorder calculando la frecuencia de cada elemento, y usamos un contador y variable maximo para guardar el/los elemento/s de mayor frecuencia. Es O(n) en espacio porque en el peor de los casos todos los elementos tienen igual frecuencia y tenemos que devolver todos los elementos. Es O(n) en tiempo porque hacemos simplemente recorridos inorder.

- Temporal: *O(n)*
- Espacial: *O(n)*

```
val = 0
contador = 0
contador_max = 0
moda = []
Recorro el arbol inorder "visitando" cada nodo
Cada vez que visito un nodo:
  si el siguiente elemento es el mismo:
    sumo uno al contador
```

```
    sino:
      seteo el contador y mi valor actual es el nuevo elemento
    luego si la frecuencia de ese valor es mayor a mi maximo anterior:
      lo guardo en la moda
  Devuelvo la moda
```

Mediana

Lo que hacemos es recorrer Inorder el arbol, teniendo un contador del nodo actual, chequeando si el nodo actual es la mediana. Obviamente en caso de encontrarse la media se chequea si hay que devolver el valor medio o el promedio de los valores medios. La mediana se guarda en una variable y al ser un recorrido inorder (recorrido lineal de los nodos) es O(n) en tiempo.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
    si el arbol esta vacio:
      Devuelvo 0
    contador = cuento los nodos del arbol
    contadorActual = 0
    actual = nodo_raiz
    mientras no me caiga del arbol:
      si no hay hijo izquierdo:
        contadorActual ++
        si el nodo actual es la mediana):
          Devuelvo el dato
        de no serlo:
          Devuelvo el promedio de este nodo con el anterior
        avanzo por el lado derecho
      sino:
          Busco el predecesor inorder del actual

          Convierto al actual como hijo derecho del predecesor inorder
          si no puedo avanzar por el lado derecho:
            avanzo por el lado izquierdo
          sino:
            contadorActual ++
            si el actual es la mediana:
              Devuelvo el dato
            si la moda es el promedio del actual con el anterior:
              Devuelvo el promedio de este nodo con el anterior
            avanzo por el lado derecho
```

Desviación estándar

Lo que se hace es contar la cantidad de nodos, sumar todos los nodos, con esto obtenemos la media. Luego para todos los nodos de forma inorder, sumamos todas las distancias. La desviacion estandar es la raiz cuadrada de la media de la sumatoria de distancias. Sumar, contar y calcular la suma de distancias de todos los nodos son O(n), porque las 3 se hacen de forma inorder, calcular la raiz cuadrada es O(log(n)). Entonces la

complejidad temporal de esta funcion es O(n) y su complejidad espacial es O(1), solo necesitamos algunas variables para guardar los resultados.

- Temporal: *O(n)*
- Espacial: *O(1)*

```
suma = sumo todos los nodos
cont = cuento todos los nodos
media = suma / cont
suma_distancias = 0
suma_distancias = aux_desviacion(actual, media)
media_de_suma = suma_distancias / cont
Devuelvo la raiz cuadrada de media_de_suma

aux_desviacion(raiz, media):
  si el arbol esta vacio:
    Devuelvo 0
  distancia_media = actual - media
  suma_dist += dist_media^2
  Recorro inorder el arbol calculando suma_dist para cada nodo
  Devuelvo la suma de todos los suma_dist
```

Permutaciones del conjunto

Pseudo codigo: Decidimos re utilizar la funcion de permutaciones de la lista para la implementacion del arbol, sabemos que no es este el espiritu de la materia, pero lo hacemos por los siguientes motivos:

1. La implementacion de esta solucion no tiene un orden menor a O(n!). Al menos no de una forma trivial, podria haber una solucion extremadamente compleja que sea mas optima que esta, pero al menos nosotros no encontramos una mejor.
2. Guardar todos los elementos del arbol con un recorrido inorder y calcular las variaciones con dicha lista tiene la misma complejidad en temporal y espacial que hacerlo directamente sobre la lista, por lo que no solo es una solucion practica sino que no agrega complejidad al algoritmo.

Por los mismos motivos decidimos hacer lo mismo para ambas variaciones.

- Temporal: *O(n!)*
- Espacial: *O(n!)*

Variaciones del conjunto tomados de r elementos (r«n)

- Temporal: *O(n!)*
- Espacial: *O(n!)*

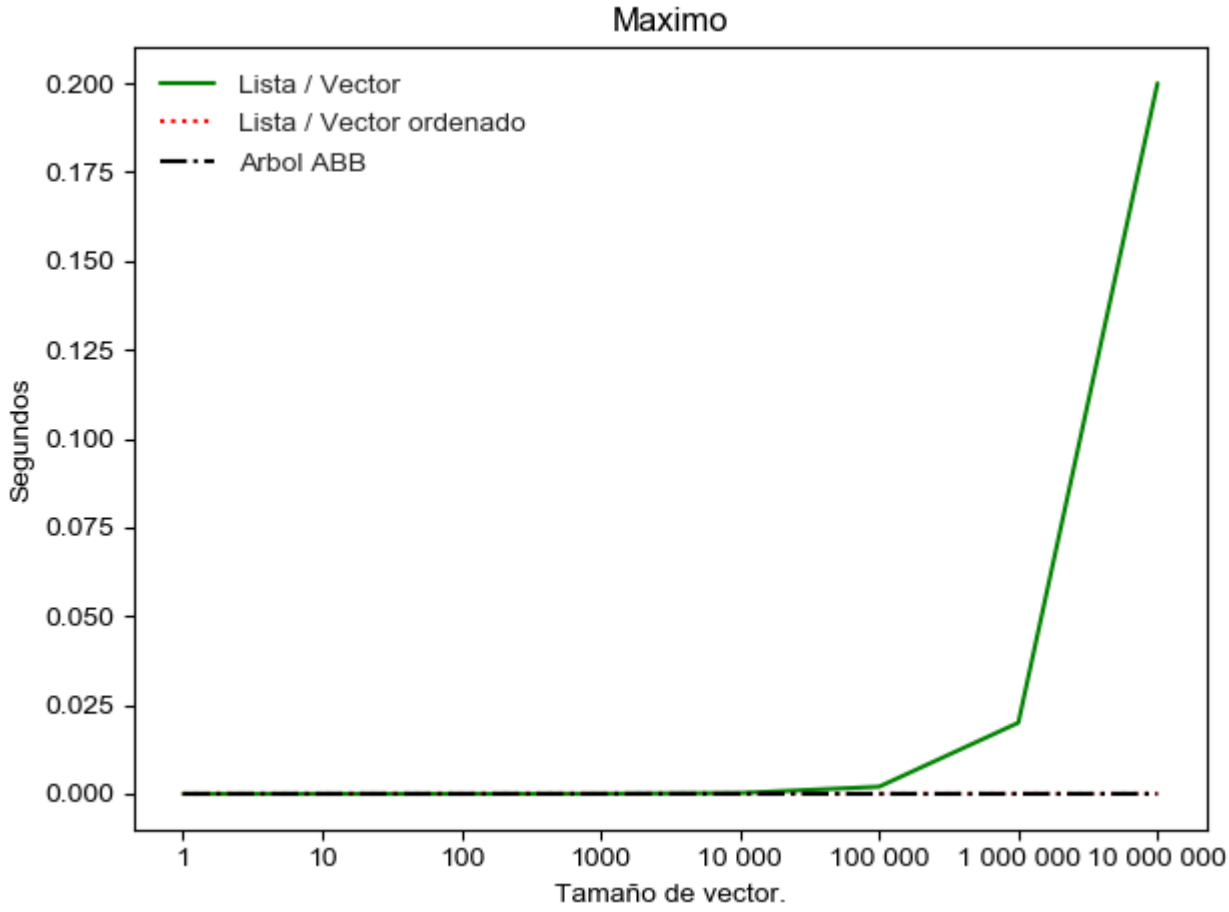
Variaciones con repetición del conjunto de r elementos (r«n)

- Temporal: *O(n!)*
- Espacial: *O(n!)*

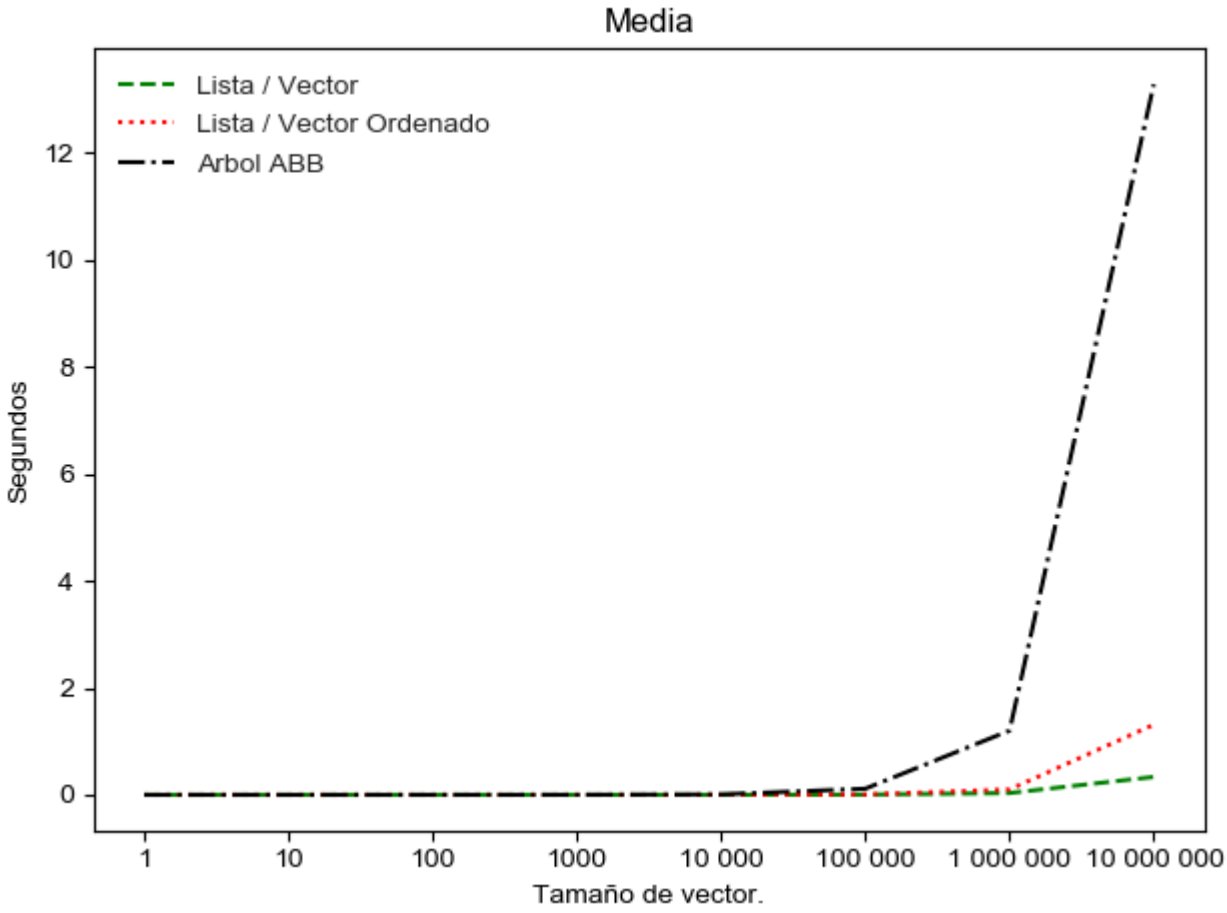
2.3 - Gráficos de Complejidad

Para armar los graficos se uso la libreria **time** de Python

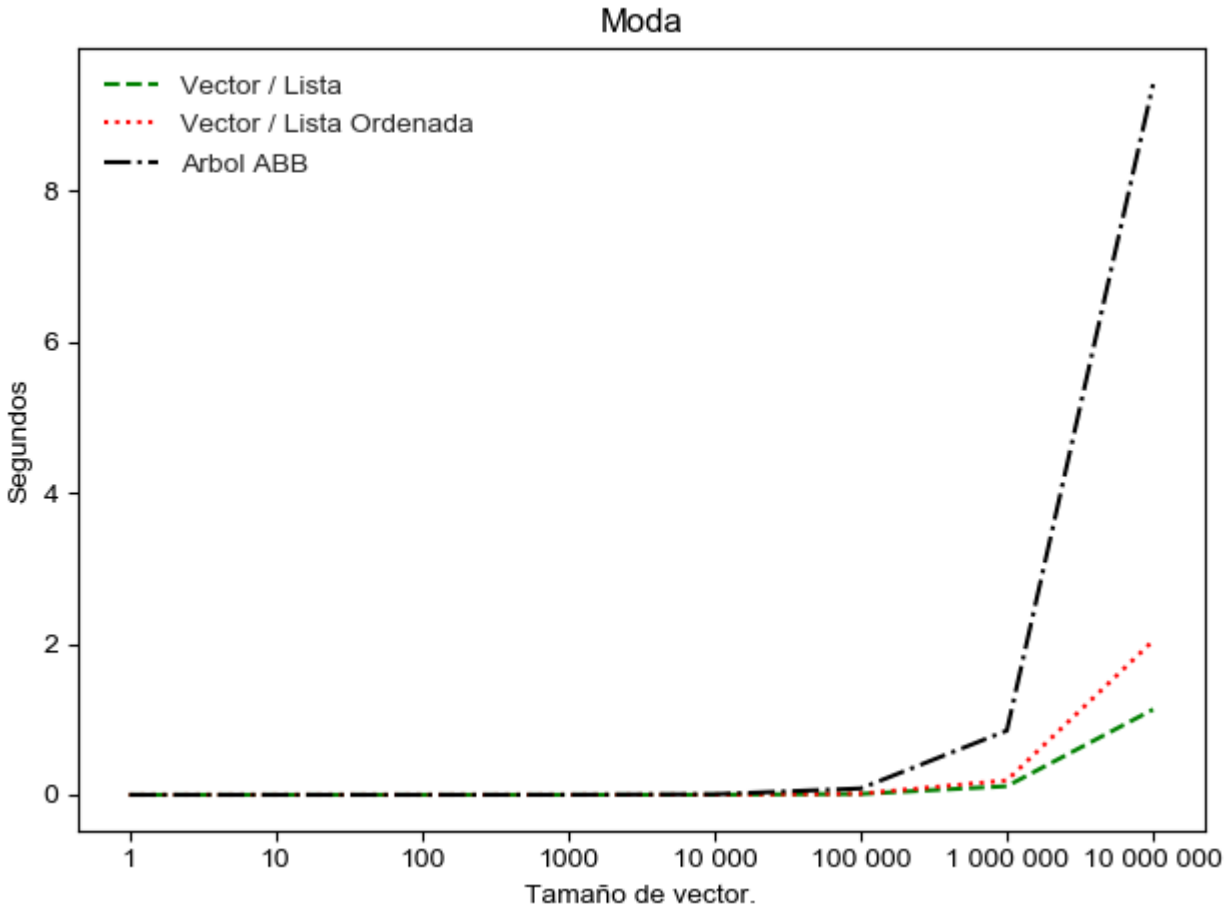
Maximo



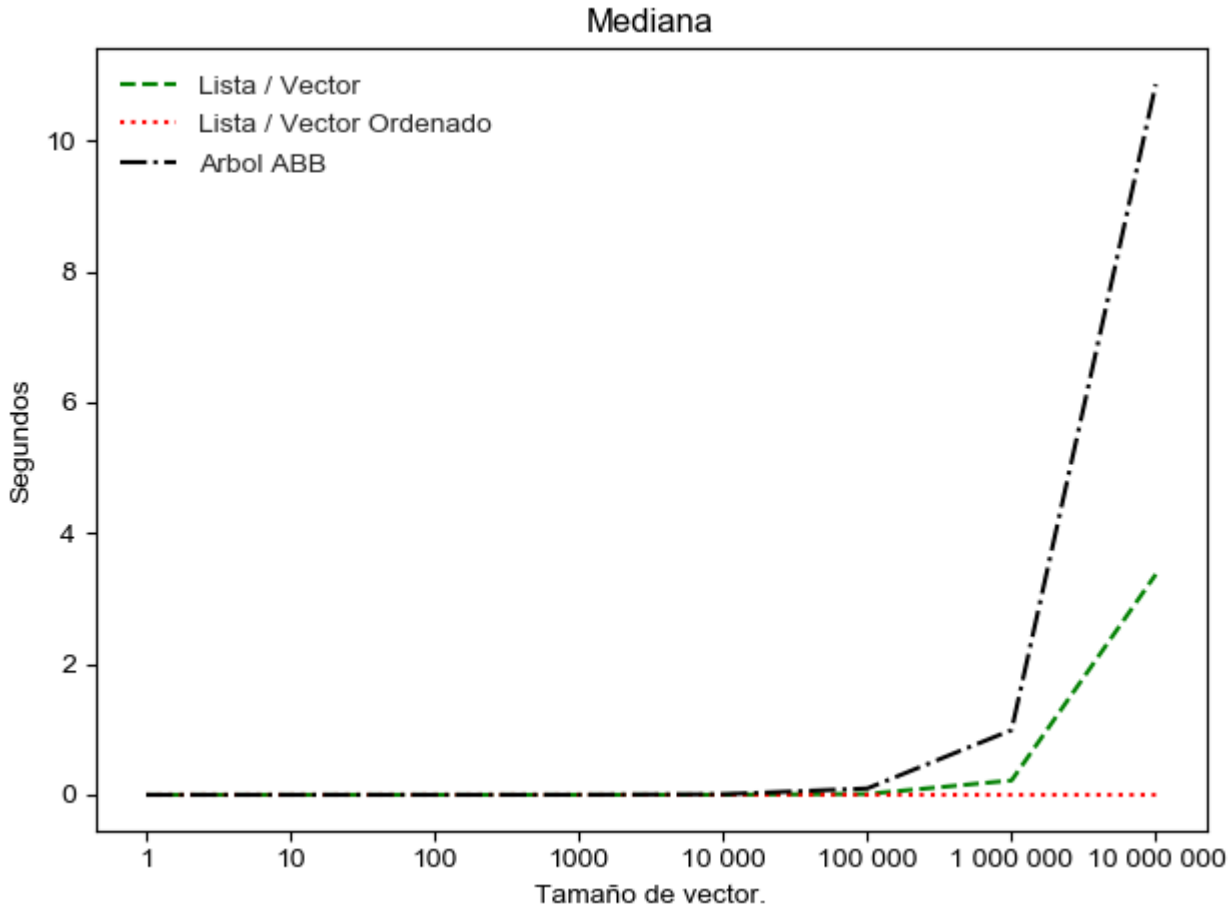
Media


Media
Lista / Vector
Lista / Vector Ordenado
Arbol ABB
Segundos
0
2
4
6
8
10
12
1
10
100
1000
10 000
100 000
1 000 000
10 000 000
Tamaño de vector.

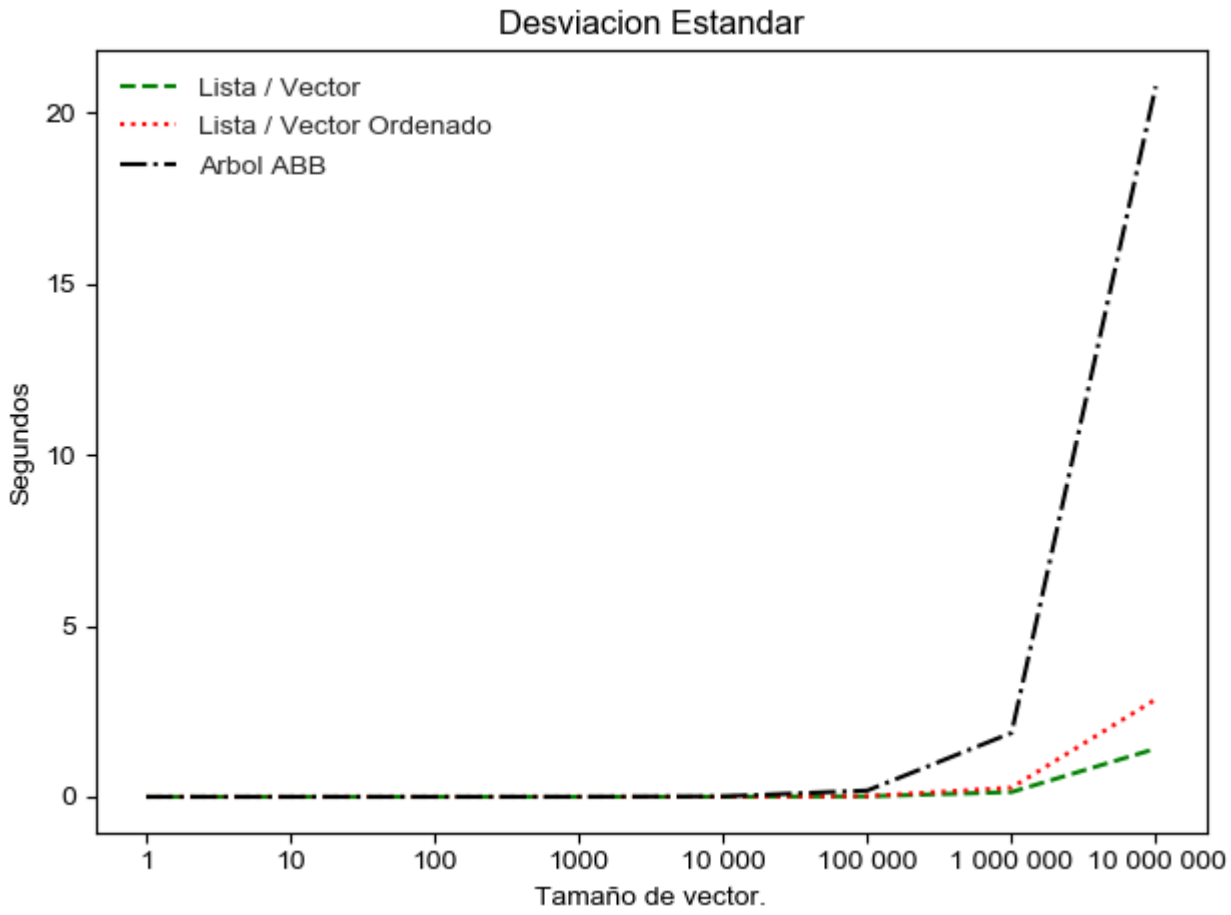
Moda


Moda
Vector / Lista
Vector / Lista Ordenada
Arbol ABB
Segundos
0
2
4
6
8
1
10
100
1000
10 000
100 000
1 000 000
10 000 000
Tamaño de vector.

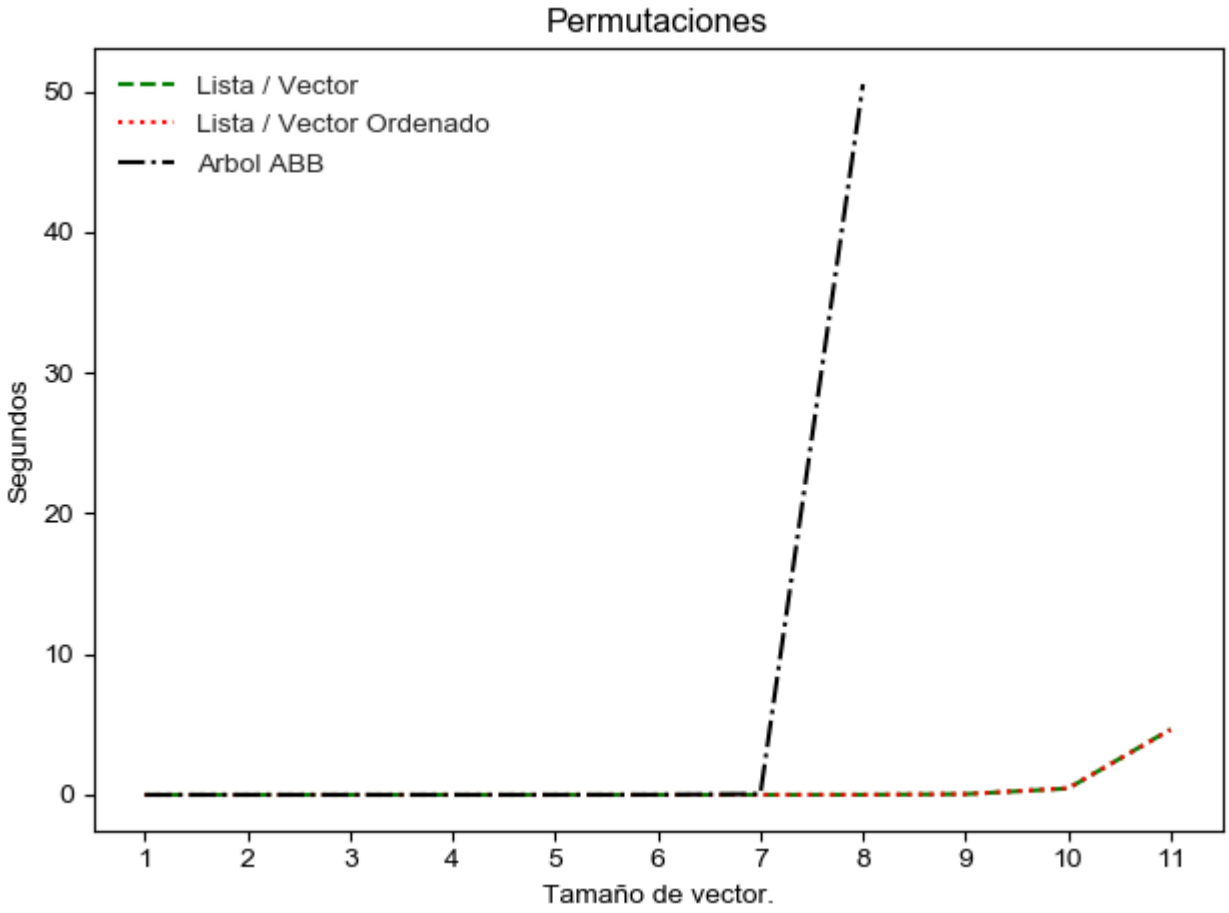
Mediana



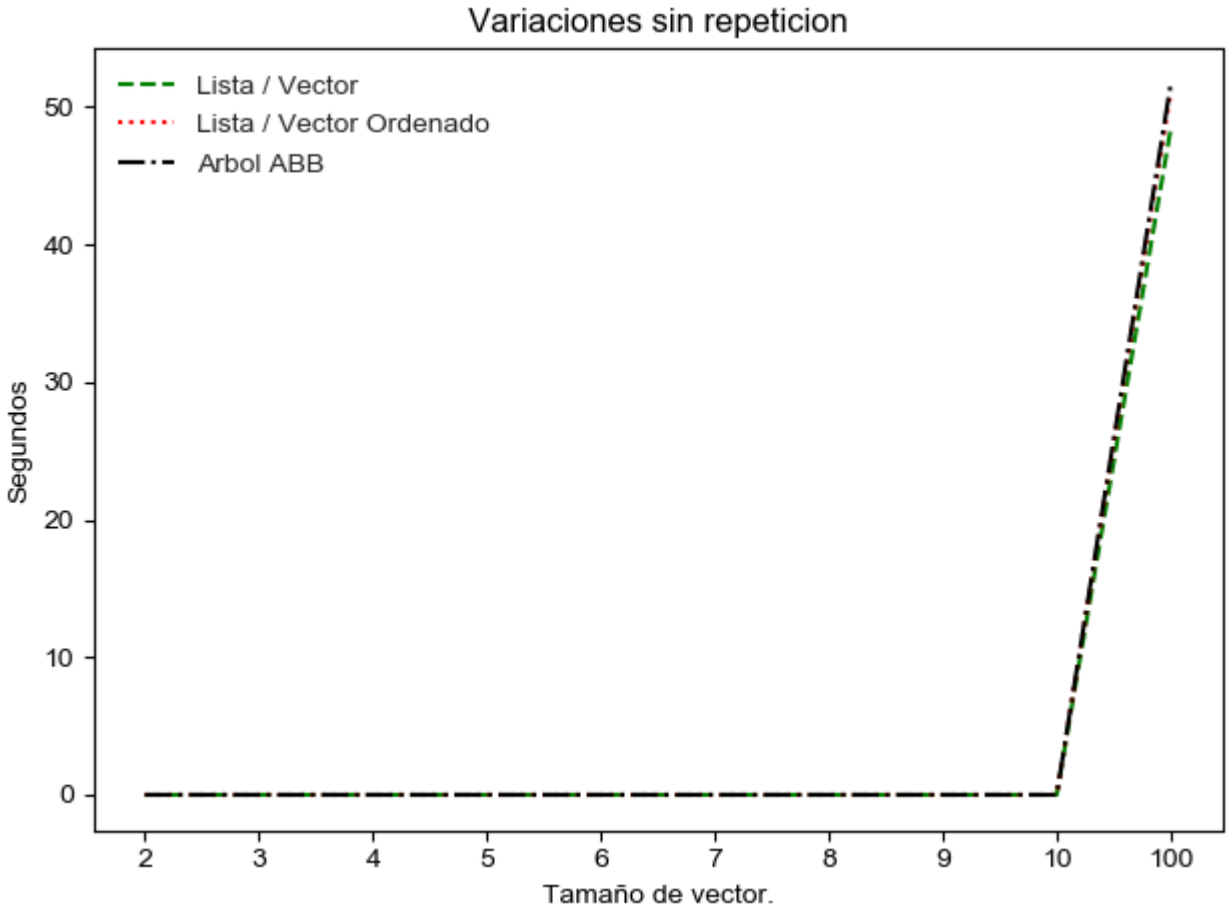
Desviacion Estandar



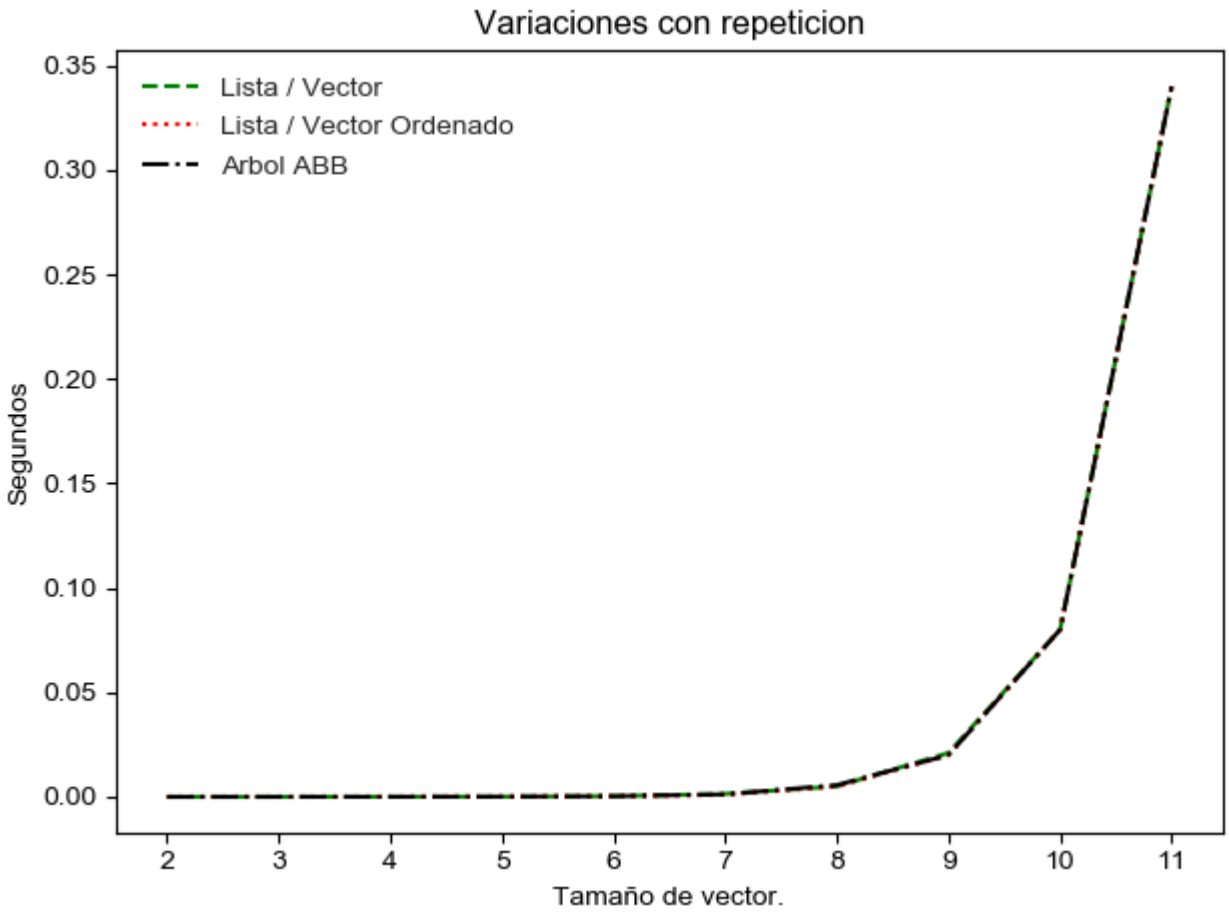
Permutaciones



Variaciones sin repeticion



Variaciones con repeticion



2.4 - Programación Algoritmos

Como ejecutar las distintas implementaciones

1. Generamos el archivo txt con la cantidad de numeros que queremos, siendo **n** la cantidad de numeros

```
cd TP1/src/Tools
python number_generator.py n
```

2. Ejecutar la implementacion deseada y funcion deseada conjunto

```
cd TP1/src/Segunda_Parte/
implentacion_lista.py ruta_archivo funcion r
```

Siendo **r** un argumento necesario solo para las variaciones

Las funciones son las siguientes:

- Maximo : **maximo**
- Media : **media**
- Moda : **moda**
- Mediana : **mediana**

- Desviacion Estandar : **desviacion_estandar**
- Permutaciones : **permutaciones**
- Variaciones de r en r : **variaciones**
- Variaciones de r en r con repeticion : **variaciones_con_repeticion** La ruta del archivo, si fue generado con el script en **tools** esta en **../assets/txt/numbers.txt**