1. (Algoritmos voraces) Se conoce el valor actual $v_1^0, \ldots, v_n^0$ de las acciones de n de una "bola de cristal" que permite conocer el valor que tendrán las acc Es decir, los valores $v_1^1, \ldots, v_1^m$ que tendrán las acciones de la empresa 1 respectivamente; los valores $v_2^1, \ldots, v_2^m$ que tendrán las acciones de la empr días respectivamente, etcétera. En general, v_i^j es el valor que tendrá una Estos datos vienen dados en una matriz $V[0..n, 0..m]$.

Dar un algoritmo voraz que calcule el máximo dinero posible a obtener al ca acciones, a partir de una suma inicial de dinero D.

Se asume que siempre habrá suficientes acciones para comprar y que no se venta. También se asume que puede comprar una fracción de acción si no este caso la ganancia es proporcional a la fracción que se compró. Recordar su valor.

Se pide lo siguiente:

(a) Indicar de manera simple y concreta, cuál es el criterio de selección vo
(b) Indicar qué estructuras de datos utilizarás para resolver el problema.
(c) Explicar en palabras cómo resolverá el problema el algoritmo.
(d) Implementar el algoritmo en el lenguaje de la materia de manera preci

Enunciado:

- Se conoce el valor de las acciones de n empresas
- Se conoce el valor de las acciones durante los siguientes m días
- En general v_i_j es el valor que tendrá la empresa i dentro de j
- Estos datos estan acumulados en V[0..n, 0..m]
- Dar algoritmo que calcule el máximo dinero posible a obtener al cabo de los m días comprando y vendiendo acciones, a partir de una suma inicial de dinero D

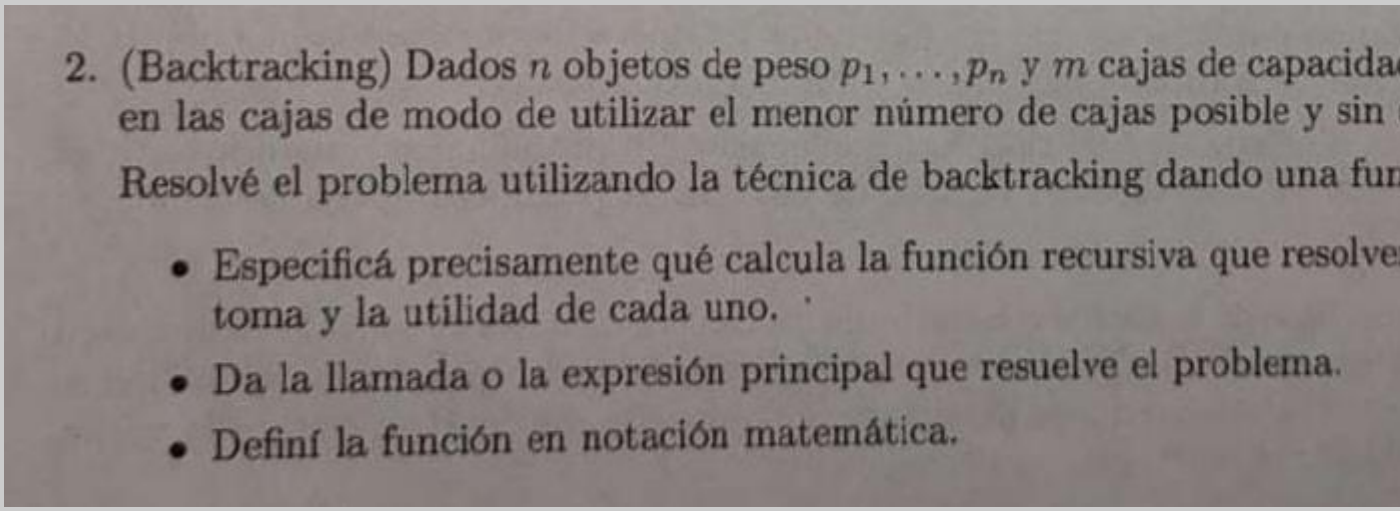
a) Si mañana x acciones aumentan su valor, se compran y se venden en un orden tq se compra primero la que más beneficios da. Si en este orden alguna no alcanza, se compra y vende una fracción de las restantes. Esto para cada día siempre y cuando se pueda comprar.

b)
```
type Acciones = tuple
                    id_empresa: nat
                    valor_compra: float
                    valor_venta: float
                end tuple
```

c) El algoritmo recorre día tras día los valores de las acciones, se toma aquella acción de la empresa que más crecimiento tenga porcentualmente al siguiente día, luego se compra tanto de esa acción como se pueda para vender al otro día y repetir el proceso para los siguientes días

```
fun acciones(v: array[0..n, 0..m] of float, D: float) ret S: float
      var set_acciones: Set of Acciones
      var day: int
      var acción: Acciones
      var fracción: float
      var ganancia: float
      set_acciones := empty_set()
      S := D
      day := 0
      while(day < m - 1) do
            fracción := 0.0
            ganancia := 0.0
            acción := elegir_max_ganancia_porcentual(V, day)
            while(S != 0) do
                  if(accion.valor_compra ≤ S and S != 0)then
                        S := S - acción.valor compra
                        ganancia := ganancia + accion.valor_venta
                  else if (accion.valor_compra > S and S != 0) then
                        fracción := S / acción.valor_compra
                        S := 0
                        ganancia := ganancia + fracción *
                  accion.valor_venta
                  fi
            od
                  S := S + ganancia
                  day := day + 1
      od
      destroy_set(set_acciones)
end fun

fun elegir_max_ganancia_porcentual(v: array[0..n, 0..m] of float,
day: nat) ret res: Acciones
      var max_porcentaje: float
      var max_aux: float
      max_porcentaje := -inf
      for i := 0 to n do
            if V[i, day] < V[i, day + 1] then
                  max_aux :=  V[i, day + 1] - V[i, day] * 100) / V[i,
day]
                  if max_aux > max_porcentaje then
                        max_porcentaje := max_aux
                        res.id_empresa := i
                        res.valor_compra := V[i, day]
```

```
                        res.valor_venta := V[i, day+1]
                    fi
                fi
            od
end fun
```



2. (Backtracking) Dados n objetos de peso $p_1, \ldots, p_n$ y m cajas de capacida
en las cajas de modo de utilizar el menor número de cajas posible y sin

Resolvé el problema utilizando la técnica de backtracking dando una fu

- Especificá precisamente qué calcula la función recursiva que resolve toma y la utilidad de cada uno.
- Da la llamada o la expresión principal que resuelve el problema.
- Definí la función en notación matemática.

Enunciado:

- n objetos de peso $p_1,...,p_n$
- m cajas con capacidad $q_1,...,q_m$
- se desea almacenar los objetos en cajas utilizando la menor cantidad posible sin exceder la capacidad de cada una

Función recursiva:

c(i, j, k) “Número de cajas a utilizar para almacenar i objetos entre {1,..,j} cajas con la capacidad actual k”

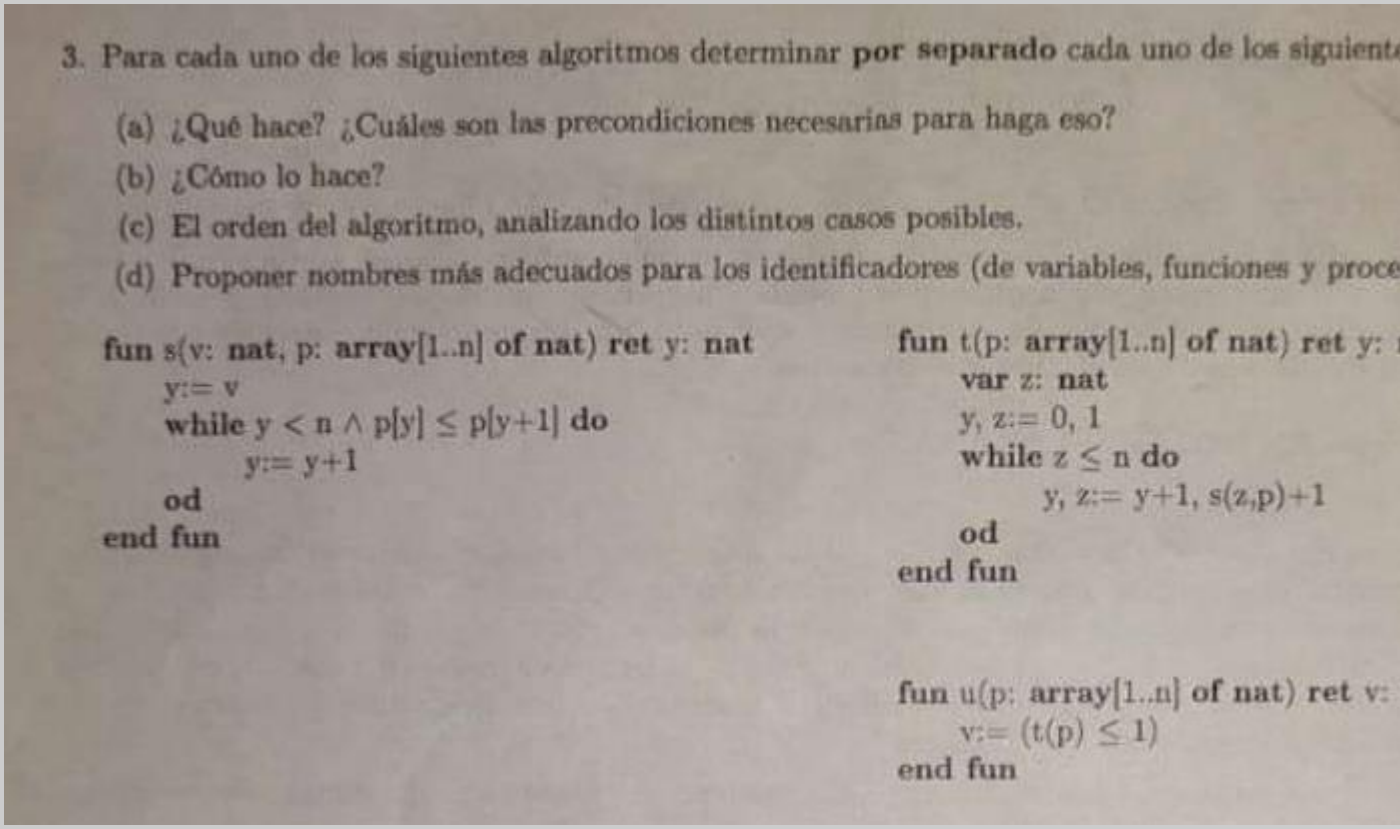
Llamada principal:

c(n, m, q_m)

Función matemática:

c(i, j, k)

0	, si i = 0
inf	, si i > 0 & j = 0
1 + c(i, j-1, q_{j-1})	, si k = 0
min(c(i,j-1,k), c(i-1, j, k - p_i))	, si i > 0 & j > 0 & k ≥ p_i
c(i , j-1, q_{j-1})	, si i > 0 & j > 0 & p_i ≥ k



3. Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguient

(a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para haga eso?

(b) ¿Cómo lo hace?

(c) El orden del algoritmo, analizando los distintos casos posibles.

(d) Proponer nombres más adecuados para los identificadores (de variables, funciones y proce

```
fun s(v: nat, p: array[1..n] of nat) ret y: nat
    y:= v
    while y < n ∧ p[y] ≤ p[y+1] do
        y:= y+1
    od
end fun
```

```
fun t(p: array[1..n] of nat) ret y:
    var z: nat
    y, z:= 0, 1
    while z ≤ n do
        y, z:= y+1, s(z,p)+1
    od
end fun
```

```
fun u(p: array[1..n] of nat) ret v:
    v:= (t(p) ≤ 1)
end fun
```

Algoritmo 1:

```
fun s(v: nat, p: array[1..n] of nat) ret y: nat
      y := v
      while y < n ^ p[y] ≤ p[y+1] do
            y := y + 1
      od
end fun
```

¿Qué hace?

Dado una secuencia y un índice de la misma, devuelve la posición del primer elemento mayor estricto que el siguiente.

¿Cuáles son las precondiciones necesarias para que haga eso?

{PRE: 1 ≤ v ≤ n}

¿Qué orden tiene?

En el peor caso (arreglo ordenado) el bucle while se ejecuta n - v, es decir es de orden o(n-v)

¿Cómo lo hace?

Toma la secuencia y un índice dentro de la misma, luego aumenta este índice siempre y cuando el siguiente elemento sea mayor o igual al actual, cuando encuentra un elemento mayor estricto que el siguiente, devuelve ese índice.

Nombre:

indice_del_mayor

Algoritmo 2:

```
fun t(p: array[1..n] of nat) ret y: nat
```

```
      var z: nat
      y, z := 0,1
      while z ≤ n do
            y, z := y+1, s(z,p)+1
      od
end fun
```

¿Qué hace?

Nos da el número de segmentos ordenados dentro de una secuencia de números

¿Cuáles son las precondiciones necesarias para que haga eso?

{PRE: true}

¿Qué orden tiene?

Si el arreglo está ordenado el bucle se ejecuta 1 sola vez. En el peor de los casos el arreglo está ordenado de forma decreciente y la complejidad sería de o(n)

¿Cómo lo hace?

Toma la secuencia y la recorre de izquierda a derecha, aumentando un contador cada vez que entra al bucle en donde se llama a la función s, la cual nos dice hasta qué índice está el primer segmento ordenado y se sigue recorriendo desde este nuevo índice en adelante, así sucesivamente se obtiene la cantidad de segmentos ordenados.

Nombre:

segmentos_ordenados

Algoritmo 3:

```
fun u(p: array[1..n] of nat) ret v: bool
      v := (t(p) ≤ 1)
end fun
```

¿Qué hace?

Nos dice si un arreglo está ordenado

¿Cuáles son las precondiciones necesarias para que haga eso?

{PRE: true}

¿Qué orden tiene?

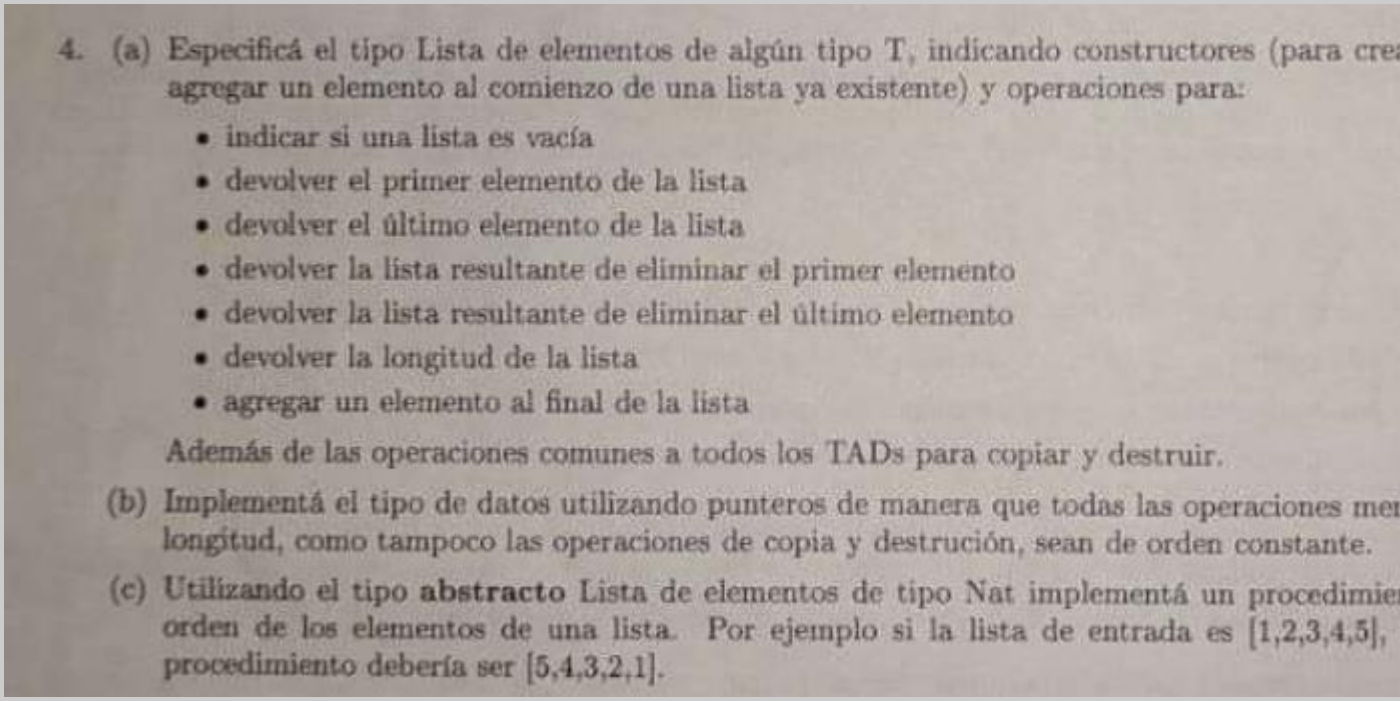
La función t tiene un bucle que en el peor de los casos su complejidad es de o(n) y como por dentro se llama a s, la complejidad total es de o(n)*o(n-v) = o(n^2)

¿Cómo lo hace?

Se llama a t, si el valor que devuelve es menor o igual a 1 el resultado es verdadero, puesto a que nos indica que la secuencia solo tiene un segmento ordenado.

Nombre:

esta_ordenado



4. (a) Especificá el tipo Lista de elementos de algún tipo T, indicando constructores (para cre agregar un elemento al comienzo de una lista ya existente) y operaciones para:
   - indicar si una lista es vacía
   - devolver el primer elemento de la lista
   - devolver el último elemento de la lista
   - devolver la lista resultante de eliminar el primer elemento
   - devolver la lista resultante de eliminar el último elemento
   - devolver la longitud de la lista
   - agregar un elemento al final de la lista

   Además de las operaciones comunes a todos los TADs para copiar y destruir.

   (b) Implementá el tipo de datos utilizando punteros de manera que todas las operaciones mer longitud, como tampoco las operaciones de copia y destrución, sean de orden constante.

   (c) Utilizando el tipo **abstracto** Lista de elementos de tipo Nat implementá un procedimie orden de los elementos de una lista. Por ejemplo si la lista de entrada es [1,2,3,4,5], procedimiento debería ser [5,4,3,2,1].

a)

```
Spec List of T where

    Constructors

        fun empty_list() ret l: List of T
        {- Crea una lista vacía -}

        proc addl(l: List of T, e: T)
        {- Agrega un elemento al comienzo de la lista -}

        fun copy(l: List of T) ret l2: List of T
        {- Copa la lista l en la lista l2 -}

    Destroy

        proc destroy(l: List of T)
        {- Libera memoria en caso de ser necesario -}

    Operations

        fun is_empty(l: List of T) ret b: bool
        {- Nos dice si una lista está vacía -}

        fun first_elem(l: List of T) ret e: T
        {- Devuelve el primer elemento de la lista -}
        {- PRE: not is_empty(l) -}
```

```
        fun last_elem(l: List of T) ret e: T
        {- Devuelve el último elemento de la lista -}
        {- PRE: not is_empty(l) -}

        proc elim_first_elem(l: List of T)
        {- Elimina el primer elemento -}

        proc elim_last_elem(l: List of T)
        {- Elimina el último elemento -}

        fun length(l: List of T) ret len: nat
        {- Largo de la lista -}

        proc addr(l: List of T, e: T)
        {- Agrega un elemento al final de la lista -}

end spec
```

b)

```
Implement List of T where

    type Node of T = tuple
                        elem: T
                        next: pointer to (Node of T)
                        prev: pointer to (Node of T)
                    end tuple

    type List of T = tuple
                        first: pointer to (Node of T)
                        last: pointer to (Node of T)
                    end tuple

    Constructors

        fun empty_list() ret l: List of T
            l->first := null
            l->last := null
        end fun

        proc addl(l: List of T, e: T)
            var new_node: pointer to (Node of T)
            alloc(new_node)
            new_node->elem := e
            new_node->next := null
            new_node->prev := null

            if (is_empty(l)) then
```

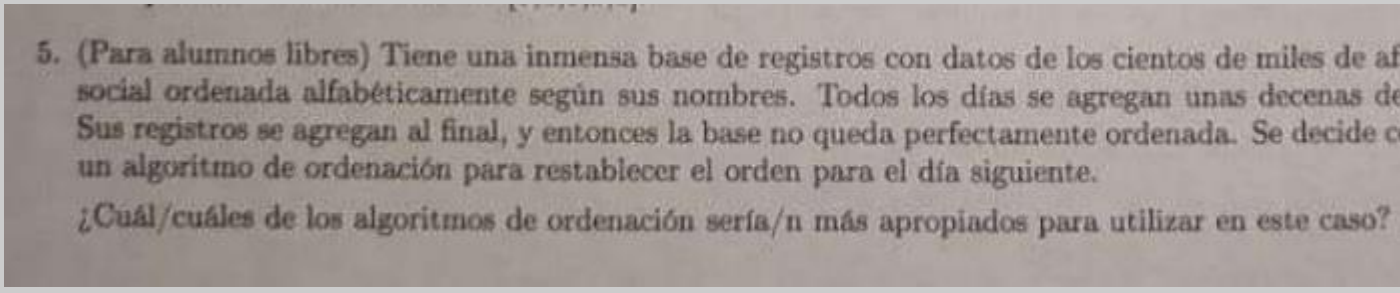
```
                l->first := new_node
                l->last := new_node
          else
                l->first->prev := new_node
                new_node->next := l->first
                l->first := new_node
          fi
    end proc

    fun copy(l: List of T) ret l2: List of T
          l2 := empty_list()
          if (not is_empty(l)) then
                var current: pointer to (Node of T)
                var new_node: pointer to (Node of T)

                current := l->first
                alloc(new_node)
                new_node->elem := current->elem
                new_node->next := null
                new_node->prev := null
                l2->first := new_node
                l2->last := new_node
                current := current->next
                while(current != null) do
                      var temp_node: pointer to (Node of T)
                      alloc(temp_node)
                      temp_node->elem := current->elem
                      temp_node->next := null
                      temp_node->prev := l2->last
                      l2->last->next := temp_node
                      l2->last := temp_node
                      current := current->next
                od
          fi
    end fun

Destroy

    proc destroy(l: List of T)
          if(not is_empty(l)) then
                var killme: pointer to (Node of T)
                var current: pointer to (Node of T)
                current := l->first
                while(current != null) do
                      killme := current
                      current := current->next
                      free(killme)
                od
```

```
            l->first := null
            l->last := null
        fi
    end proc

Operations

    fun is_empty(l: List of T) ret b: bool
       b := l->first == null & l->last == null
    end fun

    fun first_elem(l: List of T) ret e: T
        e := l->first->elem
    end fun

    fun last_elem(l: List of T) ret e: T
        e := l->last->elem
    end fun

    proc elim_first_elem(l: List of T)
        if (not is_empty(l)) then
            var killme: pointer to (Node of T)
            killme := l->first
            l->first := l->first->next
            free(killme)
        fi
        if (l->first := null) then
            l->last := null
        fi
    end proc

    proc elim_last_elem(l: List of T)
        if (not is_empty(l)) then
            var killme: pointer to (Node of T)
            killme := l->last
            l->last := l->last->prev
            l->last->next := null
            free(killme)
        fi
    end proc

    fun length(l: List of T) ret len: nat
        len := 0
        if (not is_empty(l)) then
            var current: pointer to (Node of T)
            while(current != null) do
                len := len + 1
                current := current->next
```

```
                od
            fi
        end fun

        proc addr(l: List of T, e: T)
            var new_node: pointer to (Node of T)
            alloc(new_node)
            new_node->elem := e
            new_node->next := null
            new_node->prev := null

            if (is_empty(l)) then
                l->first := new_node
                l->last := new_node
            else
                new_node->prev := l->last
                l->last := new_node
            fi
        end proc

end implement
```

c)

```
proc invert(l: List of Nat)
    var l_aux: List of Nat
    l_aux := copy(l)
    while(not is_empty(l_aux))
        elim_first_elem(l)
        addr(l, last_elem(l_aux))
        elim_last_elem(l_aux)
    do
end proc
```



5. (Para alumnos libres) Tiene una inmensa base de registros con datos de los cientos de miles de af
social ordenada alfabéticamente según sus nombres. Todos los días se agregan unas decenas de
Sus registros se agregan al final, y entonces la base no queda perfectamente ordenada. Se decide c
un algoritmo de ordenación para restablecer el orden para el día siguiente.

¿Cuál/cuáles de los algoritmos de ordenación sería/n más apropiados para utilizar en este caso?

Insertion_sort si los nuevos afiliados están parcialmente ordenados, puesto que mientras más ordenado este la base mejor para este algoritmo, si los nuevos afiliados están completamente desordenados, es mejor usar quick_sort dado que al separar todos los menores de un lado y todos los mayores de otro logra un mejor ordenamiento que estar intercalando ambas mitades.