**I.1. Diseñar un algoritmo de búsqueda ternaria en un vector. La idea es partir el vector en 3 partes y activar recursivamente la función de búsqueda en cada una de las 3 partes. Analizar el costo de algoritmo en comparación con el de búsqueda binaria.**

Hacemos como en la búsqueda binaria, usando técnica divide y vencerás, vamos partiendo el vector en bloques de 3. Y así sucesivamente cada bloque, hasta que o bien sólo tengamos bloques de 1 elemento, y sea ese el elegido, o sea el “pivote” o primer elemento de cada bloque.

```
int busquedaTernaria (int * array, int inicio, int fin, int numeroBusqueda){
	int tamanioActual = fin - inicio;
	int pCentro, pIzda, pDcha;

	//Partimos el vector en 3 partes
	int tamParte = (fin - inicio)/3;
	pIzda = inicio;
	pCentro = inicio+tamParte;
	pDcha = pCentro+tamParte;

	cout<<"\nPos izda: "<<pIzda;
	cout<<"\nPos central: "<<pCentro;
	cout<<"\nPos Dcha: "<<pDcha;
	cout<<endl;

	if (tamanioActual==1) { //Caso base
		cout<<"\nCaso base";
		return fin+1;
	}

	//Ahora lo tenemos dividido por 3 pivotes, vemos en que parte estará, en
que subconjunto, ya que el array
	//suponemos está ordenado
	if (numeroBusqueda >= array[pDcha]) { //Esta en el último subconjunto
		if (numeroBusqueda == array[pDcha])
				return pDcha+1;
		else{
				busquedaTernaria(array,(pDcha+1),fin,numeroBusqueda);
		}
	}
	else{
		if (numeroBusqueda >= array[pCentro]) { //Esta en el centro
				if (numeroBusqueda == array[pCentro])
					return pCentro+1;
				else

busquedaTernaria(array,(pCentro+1),fin,numeroBusqueda);
		}
		else{ //Esta en la parte izda
				if (numeroBusqueda == array[pIzda])
```

```
                                return pIzda+1;
                            else

busquedaTernaria(array,(pIzda+1),fin,numeroBusqueda);
                }
        }


}
```

Como mucho, haremos (n/3) veces la operación recursiva, esto conlleva a tener un orden de eficiencia O($log_3$ n), mientras que una búsqueda binaria sería O($log_2$ n)

**I.2. Dados n valores reales (v1,...,vn) diseñar un algorítmo que calcule el valor de la diferencia d entre los dos valores más cercanos:**
**d = min | vi - vj |**
**1≤i, j ≤ n , i ≠ j**
**Sugerencia : Usar el algorítmo de partición del quicksort.**

**II.1. Deseamos almacenar en una cinta magnética de longitud L un conjunto de n programas**
**{P1, P2, ... , Pn}. Sabemos que cada Pi necesita un espacio ai de la cinta y que**
**( ∑ ai ) > L**
**1≤i ≤n**
**Construir un algoritmo que seleccione aquel subconjunto de programas que hace que el número de programas almacenado en la cinta sea máximo.**

Tenemos una función de selección que funciona de la siguiente manera:
*Para cada elemento/programa Pi*

1. *Si es el primer elemento*
    a. *Si ai de Pi es menor (tamanioCintaRestante/2)+1*
        i. *Introducir en la cinta*
        ii. *Recalcular el tamanio restante de la cinta*
    ***Si no** descartamos la selección*
2. ***Si no** es el primer elemento*
    *B. Si ai de Pi es **menor** (tamanioCintaRestante/2)+1*
        *Introducir Pi en la cinta*
        *Recalcular el tamanio restante de la cinta*
    ***Si no***
        *Si la media geométrica es menor (tamanioCintaRestante/2)+1*
            *Si ai de Pi es menor o igual en tamanioCintaRestante*
                *Introducimos Pi en la cinta*
                *Recalcular el tamanio restante de la cinta*

*Calcular/Actualizar la **media geométrica** de los ai, de cada Pi recorrido*

Con lo de la media geométrica, lo que he intentado obtener es tener una ponderación aproximada de cuánto será el tamaño del siguiente programa. Con esto, si tenemos una media baja, y un programa un poco más grande, lo introducimos en la cinta porque con una probabilidad alta tendremos que el siguiente programa será pequeño.

Respecto al **tamanioRestante/2 + 1,** he intentado dar preferencia en la selección a los programas más pequeños respecto a los que ocupen más espacio, para así poder tener una mayor cantidad de programas en cinta.

Código fuente:

```
/**
   Devuelve true o false, depediendo de si es factible o no introducir el elemento
o no.
   @param elemento elemento a introducir. Contiene el tamaño que ocupa ese
programa
   @param tamanio tamaño restante en la cinta
 */
bool factible(int elemento, int tamanio){
      if (elemento <= (tamanio/2)+1) //Si el elemento es menor que la mitad
restante lo metemos. El objetivo es intentar meter el mayor número posible de
programas en la cinta
            return true;
      else return false;
}

/**
  Devuelve true o false, si selecciona el programa para introducir dentro de la
cinta. Funcion de seleccion.
  @param pi programa a evaluar su seleccion
  @tamanioRest tamanio restante en la cinta
  @media media geometrica de los valores de los programas, espacio que ocupan
  @j posicion actual por la que va la cinta. Último programa almacenado
  */

bool seleccion(int pi, int tamanioRest,int media,int j){

      //Funcion factible
      if (factible(pi,tamanioRest)) {
      if (pi <= tamanioRest) //Entra dentro de la cinta
            return true;
      }
      else{
            if (media<(TAM/2)+1 && j>0) //Lo introducimos, probablemente el
siguiente sea menor y podamos introducirlo tambien
            {
                        if (pi<=tamanioRest) { //Si cabe
                        return true;
                        }
                        else return false;
```

```
            }
        }
        return false;
}
```

.....

```
int tamanioRest = TAM/2;
        int j=0;
        int media= 1; //Media de valores
        for (int i=0; i<TAM; i++) {
                media = sqrt(valores[i]*media);
                if (seleccion(valores[i],tamanioRest,media,j)) {
                                cintaL[j]=valores[i];
                                tamanioRest -= valores[i]; //Actualizamos el tamaño
                                j++;
                                cout<<"\nActualizando el tamaño: "<<tamanioRest<<endl;
                }
        }
```

La eficiencia en este caso sería de O(n), porque sería recorrer el bucle de programas, y allí ir seleccionando cada programa.

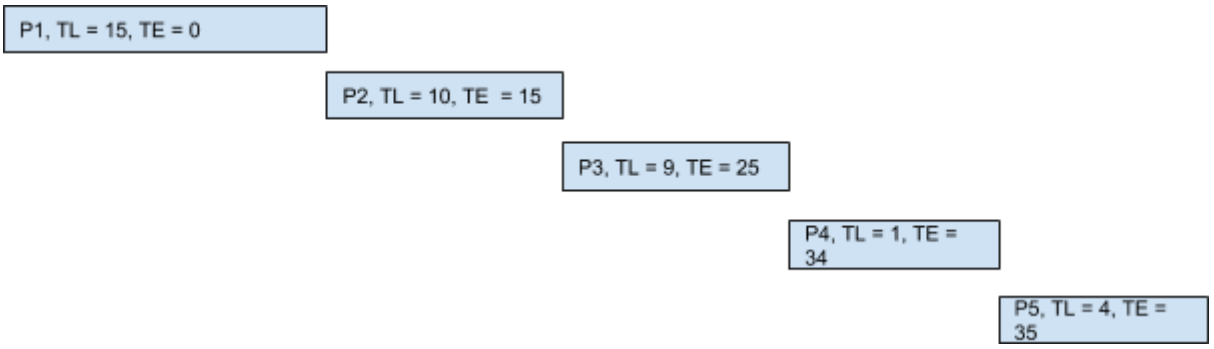
**Sería un algoritmo mucho más eficiente si pudiésemos aplicar un algoritmo de ordenación por tamaños de menor a mayor, entonces simplemente sería un algoritmo voraz de ir introduciendo programas mientras el tamaño fuera suficiente, pero he asumido que los programas van entrando a una especie de cola de trabajo y no sabemos su final, ni cuál será el siguiente programa.**

**II.2. Diseñar un algoritmo para almacenar N programas de longitud li, 1≤li≤H en una cinta**
**magnética de forma que el tiempo medio de lectura de los mismos sea mínimo. Se supone**
**que los programas se leen con igual frecuencia y que antes de leer cada programa se rebobina la cinta. Se entiende por tiempo medio de lectura:**

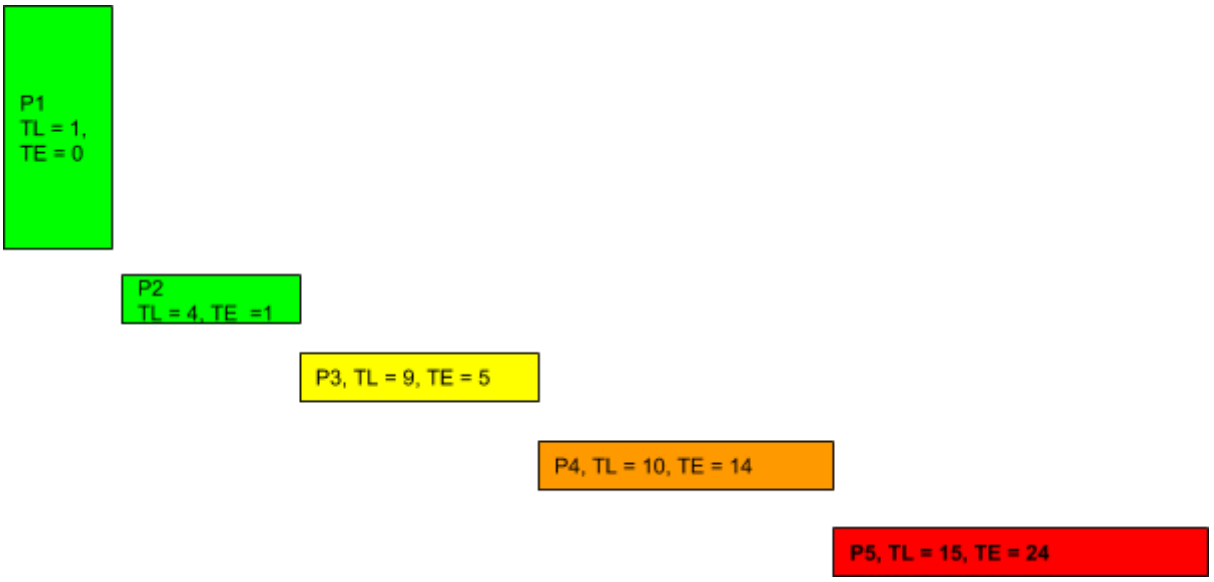
$$T = (1 / N) \sum_{k=1}^{N} \sum_{i=1}^{k} l_i$$

Como bien es sabido, para favorecer el tiempo de lectura o de respuesta, debemos seleccionar antes los procesos más cortos, ya que así, los procesos con un tiempo de lectura mayor quedarán los últimos, y tendrán un tiempo “extra” de haber estado en la cola de espera menor que si lo hubiéramos hecho al revés, primero los procesos largos y después los pequeños. Por ejemplo:

Tenemos 5 procesos de tl (tiempo de lectura): 15, 10, 9, 1, 4
Si los ejecutamos tal cual están, uno detrás de otro tendríamos:




Siento TL, el tiempo de lectura que necesita el proceso, y TE el tiempo de espera, vemos que en total, hasta que se comienza a leer el último proceso han pasado 35 segundos.
Si estos mismos 5 procesos los ejecutamos de menor a mayor tiempo de lectura requerido:




Vemos como el tiempo que pasa desde que empieza a leerse el primer proceso hasta el comienzo de la lectura del último son 24 segundos, estaríamos tardando 10 segundos menos, usando este algoritmo de *los más cortos primero*.

He supuesto que el tiempo de rebobinar la cinta no es muy relevante, ya que es una constante, por eso no lo pongo.

Nos quedaría entonces el siguiente algoritmo:

Cinta C, conjunto solución S, procesos Pi, Función selección F.
**Mientras haya procesos en la cinta C**
¿Hay más de un proceso?
Si → Px = F (P) //Proceso más pequeño
No → Px = Pi //Proceso que toque, es el último
Leer (Px)
C = C - Px //Quitamos el proceso de la cinta

Rebobinar cinta

Aquí el algoritmo implementado en **C++**:

```
/**
   @brief Devuelve la posicion del valor más pequeño del vector
   @param cinta cinta con los programas
   return posicion del elemento mas pequeño
 */
int seleccion(vector<int> cinta){
	int menor = cinta[0];
	int posMenor = 0;
	if (cinta.size()>1) {
		for (int i=1; i<cinta.size(); i++) {
				if (cinta[i]<menor) {
					menor = cinta[i];
					posMenor = i;
				}
		}
	}

	return posMenor;
}

/**
* @brief Funcion leer. Lee los programas de la cinta de procesos
* @param cinta cinta de procesos
*/
void leer (vector<int> cinta){

	float TLecturaMedio = 0.0; //Tiempo de lectura medio
	float TN = (float)(cinta.size());
	while(cinta.size()>0){ //Mientras haya elementos en la cinta
		int posLectura = seleccion(cinta); //Funcion de seleccion

		if (cinta.size()>1) //No es el último tiempo
			TLecturaMedio += cinta[posLectura];

		cinta.erase(cinta.begin()+posLectura); //Borramos de la cinta
		//Volvemos al principio, releyendo

	}
	cout<<"\nTML :"<<((float)TLecturaMedio/(float)TN)<<endl;
}
```