**Algoritmo de ordenación de un array numérico 🡪 SHELLSORT**

* Se hace con una función void que recibe el array numérico (int v[]) y su tamaño n.
* Consta de tres bucles for, y se va modificando el propio array de entrada para ordenarlo.
* Primer bucle: se define un espacio entre números (int gap), que se inicializa a la mitad del array (n/2). Mientras este espacio sea mayor que 0, se reducirá a razón de 2 (gap /= 2).
* Segundo bucle: se recorre el array con un índice i desde la posición indicada por gap hasta el final.
* Tercer bucle: el índice j se inicializa a i-gap (si i=gap en la primera iteración del bucle, j=0). j va recorriendo el array de esta forma:
  + [j X X X … j+gap (centro) X X … X fin]
  + [X j X X … X j+gap X … X fin]

En este bucle se tienen que ir considerando los elementos separados por gap, (j y j+gap), y comprobando si v[j] > v[j+gap]. Si ocurre esto, los dos elementos no están ordenados y hay que intercambiarlos usando una variable auxiliar. j disminuye en gap hasta que se hace negativo.

* Este algoritmo hace más reordenaciones al principio de la ejecución, cuando el array está desordenado. Conforme avanza, hace menos trabajo.

**Algoritmo de ordenación BUBBLE SORT**

* Consta de dos bucles for anidados. Se debe pasar el tamaño del array n.
* En el primero, se recorre el array normalmente con un índice i (de 0 a n).
* En el segundo, el índice j empieza en el final del array (n-1), y decrementa hasta llegar llegar a i+1 (j > i ó j >= i+1).
* Dentro de este bucle, se comprueba si array[j-1] > array[j]. En tal caso, se intercambian con la función swap programada.

**Algoritmo de ordenación MERGE SORT**

* Hace uso de la estrategia divide-and-conquer.
* Proceso de mezclado básico (merge):
  + Se pasan 3 arrays (dos pilas, y la pila de salida donde se mezclan las dos), y los tamaños de las dos primeras pilas (lo suponemos igual m).
  + Debemos definir 3 índices (i, j, k) 🡪 uno para cada pila.
  + Recorremos la primera pila a, y la segunda b con los índices i y j (while i<m && j<m).
  + Si a[i] < b[j], metemos a[i++] en la pila de salida c[k++]. Al revés si no.
  + Fase de limpieza: Metemos dos while, el programa en la ejecución entrará solo en uno (while i<m, c[k++] = a[i++]; y while j<m hacemos lo propio).
* Una vez tenemos el merge, el ordenamiento de un array arbitrariamente largo se realiza definiendo un salto i, que incrementará en múltiplos de 2 hasta alcanzar el tamaño del array n.
* Cuando defines ese bucle, empleamos otro con un índice j (que va desde 0 hasta n-i). Este irá incrementando en dos veces el salto i. Lo que hará este bucle será llamar a merge con las pilas array+j y array+j+i; ambos de tamaño i, y poniendo la mezcla en un array definido en la propia función (w+j).

**Algoritmo de ordenación recursiva: QUICKSORT**

* Tenemos un array numérico de longitud n.
* Para ordenarlo, definimos una función recursiva que toma como argumentos el array, y dos índices izq y der que delimitan una partición (inicialmente 0 y n-1).
* Si izq >= der, entonces el array tiene un solo elemento o no existe (partición sin sentido).
* Definimos un pivote (punto medio de la partición 🡪 (izq + der) / 2). Lo ponemos en la posición de la izquierda, la “primera” dentro de la partición. Esto lo hacemos para recorrer la partición y poner todos los números menores que este a su izquierda, y los mayores a la derecha.
* Inicializamos “last” a “left” (primera posición, pivote). Recorremos la partición con un bucle empezando por la posición “left + 1”. Si v[i] es menor que v[left], es decir, menor que el pivote, lo intercambiamos por ++last (sumarle 1 a last e intercambiar).
* Finalmente, intercambiamos last y left para recolocar el pivote y dejar el array con los elementos menores que el pivote a su izquierda. Esto se debe a que last señala el último intercambio, es decir, la última posición que cumple que es menor que el pivote durante el recorrido de la partición. Al intercambiar last y el pivote, conseguimos la configuración deseada.
* Por último, llamamos recursivamente a la función dos veces:
  + Para la partición [left, pivote-1] y para la de [pivote+1, right].

**Algoritmo reverse con doble índice:**

* Usamos comas en el bucle para manejar los dos índices.
* Necesitamos los índices i, j (int), y otro int c como variable auxiliar.
* El bucle es: for (i=0, j=strlen(s)-1; i < j; ++i, --j)
* Lo que se hace es recorrer la cadena desde el principio y desde el final hasta llegar a la mitad, donde ambos índices se igualan. En cada iteración, intercambiamos s[i] con s[j], es decir, el carácter del principio lo ponemos al final y viceversa.

**NOTA: conversión a hexadecimal en C 🡪** n%16 – 10 + ‘A’. Esto se debe a que se coge el digito de n de las “unidades” en cada iteración y, si es mayor o igual que 10, al corresponderle una letra, se le suma al ASCII de A el número correspondiente (0 si es 10, 1 si 11…). Por eso hay que restarle 10. Para convertir de entero a cadena normal, hay que sumarle al número el ASCII de 0 (te saca el ASCII del número a convertir).

**Función strindex(cad1, cad2):**

* Devuelve el índice a partir del que la cadena 1 contiene a la 2, o -1 si no la contiene.
* Para implementarla, hay que recorrer la cadena 1 (for externo).
* Incluimos un segundo for interno con dos índices:
  + Índice j para recorrer cadena 1 a partir del índice i.
  + Índice k para recorrer la cadena 2 desde 0 hasta que cad2[k] sea ‘\0’.
* Este segundo for se encargará de ir aumentando j y k mientras cad2[k] no sea nulo y cad1[j]==cad2[k].
* Luego hay que añadir una comprobación para ver si k ha subido de 0 y si se ha recorrido toda la cadena 2, es decir, si tras ese for vacío se tiene que cad2[k] == ‘\0’. Si es así, se devuelve el índice i.

**Conversión de cadena a punto flotante double:**

* Se procede al principio como en la conversión a entero 🡪 ignorar espacios, guardar signo, recorrer cadena mientras s[i] sea un dígito, ir actualizando el resultado como 10.0\*result + (s[i] – ‘0’).
* Una vez para este bucle, es posible tener un punto decimal, por lo que si el siguiente carácter es ‘.’, lo ignoramos y seguimos recorriendo s mientras s[i] sea dígito.
* En este recorrido, actualizamos result de la misma forma, pero inicializamos una variable int power = 1; incrementándola a razón de 10 en cada iteración.
* Devolveremos (result/power) \* sign.

**Calculadora con notación polaca (‘2 1 -’ en vez de ‘2-1’)**

* IDEA: Ir leyendo una línea. Mientras encontremos números, se van metiendo en una pila (push). Al encontrar un operador, se hacen dos pop y se opera con los números, apilando de nuevo el resultado de la operación. Así hasta encontrar el fin de línea, momento en el que se devuelve el resultado final.
* Opciones para ir leyendo una línea y que el programa devuelva el número y, en la siguiente llamada, recuerde por dónde iba:
  + Usar getch y ungetch con un buffer. El programa va leyendo caracteres sacados de getch (getchar o buffer si hay algo en él), y si lee de más (cuando se quiere que todos sean dígitos, tiene que leer el que no es para la comprobación), cierra la cadena con null y “deslee” el último carácter con ungetch. Esta función mete el último carácter leído en un buffer, para que getch lo devuelva en su próxima llamada.
  + Usar getline definiendo como variables externas a todas las funciones la línea y el índice de la línea. Para “desleer” un carácter leído, se decrementa este índice.
  + Usar getline, pero poner el índice de la línea como variable estática interna en getop. Reiniciarla si tenemos una nueva línea (si llegamos a un fin de línea, llamamos otra vez a getline y vemos si hay otra línea para reiniciar índice).

**Random allocation:**

* Algoritmo para que, dado un conjunto de n objetos ordenados, poder asignarle a cada uno una posición aleatoria dentro del conjunto, manteniendo exactamente los n objetos.
* Recorremos el conjunto (normalmente un array) con i=0; i<n; i++.
* Buscamos un objeto en un índice j aleatorio mayor o igual que i (los de antes de i estarán ya “barajados”), y menor que n. Es decir, j pertenece al intevalo [i, n-1].
* Cambiamos el objeto conjunto[j] por el de conjunto[i] con una variable temporal.

**Algoritmo de Dijsktra en C++:**

* Encontrar el camino más corto del nodo s al nodo d en un grafo dirigido.
* Conjunto de cerrados: nodos cuyo camino más corto hasta d ya se ha almacenado.
* Conjunto de abiertos: nodos que son alcanzables desde el nodo que hayamos evaluado.
* Iteratividad: Evaluar el conjunto de abiertos, elegir un nodo k, si es d paramos. Si no es d, vemos si está en cerrados o abiertos. Si está en cerrados, ya hemos encontrado el mejor camino (lo ignoramos). Si no está en abiertos ni en cerrados, lo metemos en abiertos junto con el coste de alcanzar k. Si lo está, sumamos el coste de llegar hasta el nodo en el que estamos (n) con el coste de ir de n a k. Si es menor que el que hay en abiertos, lo sustituimos.