

Sleep Sort – The King of Laziness / Sorting while Sleeping

Program Name- Sleep Sort

Project Category- Sorting

Programming Paradigm Used- Threads and Multithreading, Operating System Concepts

Explanation-

We start by introducing few Operating Systems terms which is needed to understand this algorithm.

1) Thread - A 'thread' in layman terms can be understood as a set of instructions for a process to be performed. For an excellent discussion on 'threads' see -

<http://stackoverflow.com/questions/5201852/what-is-a-thread-really>

2) Multithreading- It is a process of executing multiple threads simultaneously.

In this algorithm we create different **threads** for each of the numbers in the input array and then each thread sleeps for an amount of time which is proportional to the array elements.

Hence, the thread having the least amount of sleeping time wakes up first and the number gets printed and then the second least element and so on. The largest element wakes up after a long time and then the element gets printed at the last. Thus the output is a sorted one.

All this multithreading process happens in background and at the core of the OS. We do not get to know anything about what's happening in the background, hence this is a "**mysterious**" sorting algorithm.

Implementation-

Note- To use sleep sort we need multithreading functions, such as → `_beginthread()` and `WaitForMultipleObjects()` . Hence we need to include `windows.h` to use these functions. This won't compile on G4G IDE or any other online IDE. You must run it in your PC (Note this code is for WINDOWS and not for LINUX).

To perform a sleep sort we need to create threads for each of the number in the input array. We do this using the function - `_beginthread()`.

`_beginthread()` is a C run-time library call that creates a new 'thread' for all the integers in the array and returns that thread.

Each of the 'thread' sleeps for a time proportional to that integer and print it after waking up.

We pass three parameters to `_beginthread()` :-

- 1) `start_address` --> start address of the routine/function which creates a new thread. .
- 2) `stack_size` --> Stack Size of the new thread (which is 0).
- 3) `arglist` --> Address of the argument to be passed.

The return value of `_beginthread()` function is a handle to the thread which is created. So we must accept is using the datatype- '`HANDLE`' which is included in `windows.h` header.

'`HANDLE`' datatype is used to represent an event/thread/process etc. So '`HANDLE`' datatype is used to declare/define a thread.

We store the threads in an array - `threads[]` which is declared using '`HANDLE`' datatype.

In each of the threads we assign two instructions→

1) Sleep → Sleep this thread till `arr[i] milliseconds` (where `arr[i]` is the array element which this thread is associated to). We do this using `Sleep()` function . The `Sleep(n)` function suspends the activity associated with this thread till '`n`' milliseconds. Hence if we write `Sleep(1000)`, then it means that the thread will sleep for 1 second (1000 milliseconds = 1 second)

2) Print → When the thread '`wakes`' up after the sleep then print the array element – `arr[i]` which this thread is associated to.

After creating the threads we will process these threads. We do this using `WaitForMultipleObjects()`.

`WaitForMultipleObjects()` is a function which processes the threads and has four arguments-

- 1) `no_of_threads` --> Number of threads to be processed.
- 2) `array_of_threads` --> This is the array of threads which should be processed. This array must be of the type-'`HANDLE`'.
- 3) `TRUE` or `FALSE` --> We pass `TRUE` if all the threads in the array are to be processed else `FALSE`.
- 4) `time_limit` --> The threads will be processed until this time limit is crossed. So if we pass a `0` then no threads will be processed, otherwise if we pass an `INFINITE`, then the program will stop only

when all the threads are processed. We can put a cap on the execution time of the program by passing the desired time limit.

Example-

We describe sleep sort algorithm using the below example-

Let's assume (for convenience) we have a computer that's so slow it takes 3 seconds to work through each element:

INPUT: 8 2 9

```
3s: sleep 8
6s: sleep 2
8s: "2" (2 wakes up so print it)
9s: sleep 9
11s: "8" (8 wakes up so print it)
18s: "9" (9 wakes up so print it)
```

OUTPUT: 2 8 9

Limitations-

- 1) This algorithm won't work for *negative numbers* as a thread cannot sleep for a negative amount of time.
- 2) Since this algorithm depends on the input elements, so a huge number in the input array causes this algorithm to *slow* down drastically (as the thread associated with that number has to sleep for a long time). So even if the input array element contains only 2 elements, like- {1, 100000000}, then also we have to wait for a much longer duration to sort.
- 3) This algorithm doesn't produce a *correct sorted* output every time. This generally happens when there is a very small number to the left of a very large number in the input array.
For example – {34, 23, 1, 12253, 9}.
The output after sleep sorting is {9, 1, 23, 34, 1223}

A wrong output also occurs when the input array is reverse sorted initially, like- {10, 9, 8, 7, 6, 5}.

The reason for such an unexpected output is because *some time is taken between scanning through each element as well as some other OS operations (like inserting each threads in a priority queue for scheduling)*. We cannot simply ignore the time taken by all these things.

We describe this using the below example-

Let's assume (for convenience) we have a computer that's so slow it takes 3 seconds to work through each element:

INPUT: 10 9 8 7 6 5

```
3s: sleep 10
6s: sleep 9
9s: sleep 8
12s: sleep 7
13s: "10" (10 wakes up so print it)
15s: sleep 6
15s: "9" (9 wakes up so print it)
17s: "8" (8 wakes up so print it)
18s: sleep 5
19s: "7" (7 wakes up so print it)
21s: "6" (6 wakes up so print it)
23s: "5" (5 wakes up so print it)
```

OUTPUT: 10 9 8 7 6 5

The above output is just an example.

Obviously, modern-day computers computer are not so slow (to take 3 seconds to scan through each element).

In reality running sleep sort on a modern computer on the above array gives the output – {9, 5, 7, 10, 8, 6}

How to fix this ?-

- 1) We can fix this by repeatedly sleep sorting on the new output until the output becomes sorted. Every time it will sort the elements more accurately.
- 2) The wrong output as discussed earlier happens due to the time taken by other OS works and scanning through each element.

In our program we have used the function `Sleep(arr[i])`, which means that each thread associated with the array elements sleep for '`arr[i]`' milliseconds. Since milliseconds is a very small quantity and other OS tasks can take more time than '`arr[i]`' milliseconds which ultimately can cause an error in sleep sorting. Increasing the sleeping time by even 10 times can give a sorted output as the OS tasks will finish all its task in between this much sleep, hence not producing any errors.

Had we used `Sleep(10*arr[i])` instead of just `Sleep(arr[i])` then we will certainly get a more precise output than the latter one. For example the input array – {10, 9, 8, 7, 6, 5} will give the correct sorted output – {5, 6, 7, 8, 9, 10} if we use `Sleep(10*arr[i])` instead of just `Sleep(arr[i])` .

However it is still possible that `Sleep(10*arr[i])` will give wrong results for some test cases. To make it more precise increase the sleep time more , say something like - `Sleep(20*arr[i])`.

Hence the bottomline is that more the sleep time, more accurate the results are. (Sounds interesting, eh?) . But again that would increase the runtime of this algorithm.

Exercise to the readers-

1) The above algorithm tries to sort it in ascending order. Can you sort an input array in descending order using sleep sort. Think upon it.

2) Is it a comparison based sorting algorithm ? How many comparisons this algorithm makes ?

[*Answer : No, it makes zero comparisons*]

3) Can we do sleeping sort without using `windows.h` header and without using `Sleep()` function?

[One idea can be to create a priority queue where the elements are arranged according to the **time left before waking up** and getting printed. The element at the front of the priority queue will be the first one to get waked up. However the implementation doesn't look easy. Think on it.]

Time Complexity-

Although there are many conflicting opinions about the time complexity of sleep sort, but we can approximate the time complexity using the below reasoning-

Since `Sleep()` function and creating multiple threads is done internally by the OS using a priority queue (used for scheduling purposes). Hence inserting all the array elements in the priority queue takes $O(N \log N)$ time. Also the output is obtained only when all the threads are processed, i.e- when all the elements 'wakes' up. Since it takes $O(arr[i])$ time to wake the i^{th} array element's thread. So it will take a maximum of $O(\max(input))$ for the largest element of the array to wake up. Thus the overall time complexity can be assumed as $O(N \log N + \max(input))$, where, N = number of elements in the input array and `input` = input array elements

Auxiliary Space-

All the things are done by the internal priority queue of the OS. Hence auxiliary space can be ignored.

Conclusion-

Sleep Sort is related to Operating System more than any other sorting algorithm. This sorting algorithm is a perfect demonstration of multi-threading and scheduling done by OS.

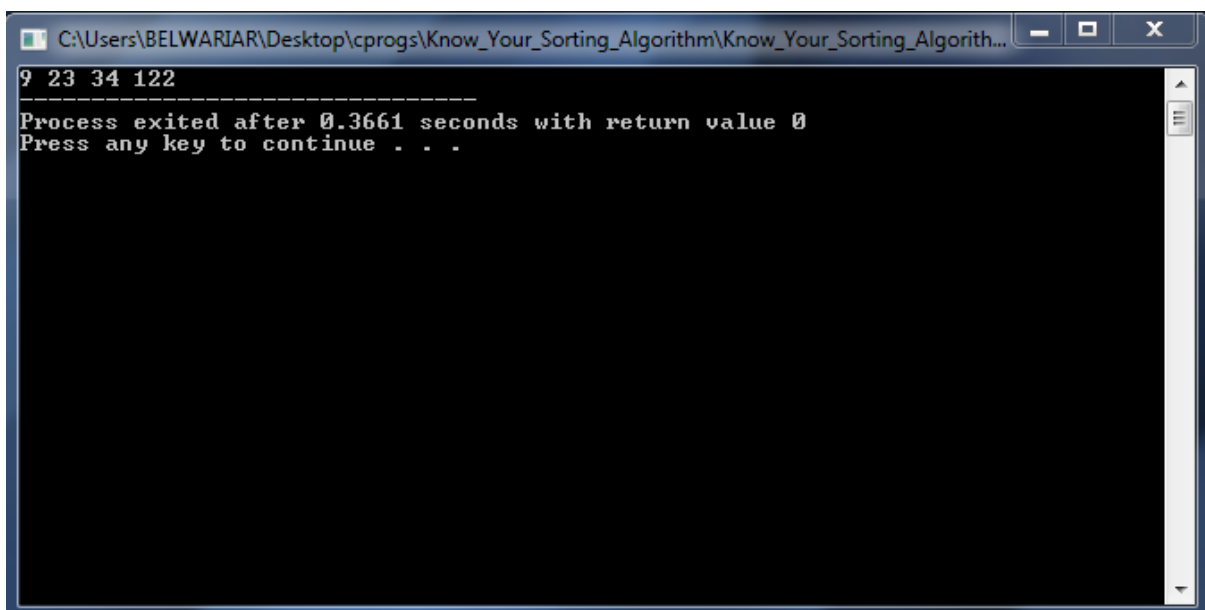
The phrase "**Sorting while Sleeping**" itself sounds very unique.

Overall it is a fun, lazy, weird algorithm. But as rightly said by someone- *“If it works then it is not lazy”*.

Note-

As said above the program won't compile on G4G IDE or any other online IDE as this needs windows.h header.

Screenshot of the Output (compiled on WINDOWS 7):-



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\BELWARIAR\Desktop\cprogs\Know_Your_Sorting_Algorithm\Know_Your_Sorting_Algorith... The window contains the following text: 9 23 34 122, followed by a dashed line, then "Process exited after 0.3661 seconds with return value 0", and finally "Press any key to continue . . .".

```
C:\Users\BELWARIAR\Desktop\cprogs\Know_Your_Sorting_Algorithm\Know_Your_Sorting_Algorith...
9 23 34 122
-----
Process exited after 0.3661 seconds with return value 0
Press any key to continue . . .
```