Name: Cheow Wei Ren

Admin No: 213033T

Group No: IT1553-02

Fibonacci Search: 7/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| 1 | log n{\displaystyle n^{2}} | log n{\displaystyle n^{2}} | 1 | Yes | Interval |

Description:

The reason behind using Fibonacci numbers takes a while to understand, but its smooth sailing from then on.

Fibonacci Search take advantage of the Fibonacci sequence to separate the array into halves, in a way similar to Binary Search. Each half is further divided, with the half where the search value is located being used for further division.

Since the Fibonacci sequence can be broken down to **1** and 2, no element is missed.

Effectiveness:

While Binary Search separates the array more evenly, Fibonacci Search uses the + and - operations, which use less resources than × and ÷ operations, and was the preferred method in legacy systems where bitwise operations produced too much overhead.

Cocktail Sort: 3/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n | n2{\displaystyle n^{2}} | n2{\displaystyle n^{2}} | 1 | Yes | Exchanging |

Description:

A two-way bubble sort. Once you understand bubble sort, cocktail sort isn’t too difficult to understand; it is simply alternating between bubbling largest values to one end of the array, and smallest values to the other.

Effectiveness:

Very useful in situations where large values are near one end of the array, and small values are near the other end.

*May* be useful with quick sort; the array is quickly separated into smaller and greater halves.

Heap Sort: 8/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n log n | n log n{\displaystyle n^{2}} | n log n{\displaystyle n^{2}} | 1 | No | Selection |

Description:

Visualisation needed before reaching full understanding of this sort.

Builds a max heap; an array which can be arranged into a binary tree, where all parent nodes are larger than their child nodes. A max heap is built by swapping the value of the root node with their child nodes, until the conditions for a max heap is satisfied.

Once built, the root element (largest value in array) is swapped into the last position of the array. This element is removed from the binary tree calculation, and is no longer moved around. The max heap is built again by swapping out the new unordered root element.

This continues until there are no longer any elements in the max heap.

Effectiveness:

While Heap Sort is generally slower (and unstable), the time complexity of Heap Sort is always n log n (beating Quick Sort’s worst time complexity), and has a space complexity of 1 (beating Merge Sort).

Tree Sort: 6/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n log n | n log n{\displaystyle n^{2}} | n2\* {\displaystyle n^{2}}\*\*\*\* | n | Yes | Insertion |

\*A balanced tree will have a worst time complexity of n log n

Description:

Easy in concept, difficult implementation.

The array is moved into a binary search tree, where the values on the node are in the order leftChildNode < parentNode < rightChildNode. After which, the tree is traversed in order, and the values are used to repopulate the array.

However, a binary search tree cannot include elements with the same value. As such, the values have to be stored in a list if they have the same value (**this was not in the tutorial**).

Effectiveness:

Not the best sorting algorithm, especially given the existence of Heap Sort. However, heap sort is not a stable sort; with a balanced tree, it can sacrifice space complexity and be used in situations where stable sorts are needed.

Gnome Sort: 4/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n | n2 | n2 | 1 | Yes | Exchanging |

Description:

Combination of insertion and bubble sort, with some changes. Values are added into a ‘sorted’ subarray (in a manner similar to Insertion Sort) and sorted in the subarray through Bubble Sort.

If a value in the ‘unsorted’ subarray is not in the right position, they are brought down to the ‘sorted’ subarray and bubbled until they are in the right position within said subarray.

This continues until the comparing index reaches the end of the array, which signifies all values in the list have been sorted; the ‘unsorted’ subarray is now empty.

Effectiveness:  
The closer the array is to being sorted, the closer the time complexity is to n. This means it can leverage on the advantages of comb sort (and solve its disadvantages).

Comb Sort quickly moves elements close to their sorted indexes, but takes a long time to fully sort the array. Since the array can become nearly sorted quickly, following up with Gnome Sort can help speed things up immensely.

Comb Sort: 3/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n log n | n2 / 2p\* | n2 | 1 | No | Exchanging |

\*p being number of increments

Description:

Values are compared between one another, and swapped if out of order.

There is a gap between indexes compared (e.g. indexes 0 and 5 are compared, indexes 1 and 6 are compared, and so on). This ensures indexes are quickly moved near to their sorted indexes.

The list is continuously run through, as the gap between indexes decreases, until the array is sorted.

Effectiveness:

Comb Sort can quickly **nearly** sort an array, but takes a while to completely sort it.

As mentioned above, Gnome Sort can be used in conjunction with Comb Sort to rapidly increase sorting speed. Since Comb Sort nearly sorts the array, Gnome Sort has a way easier time sorting through it.

As for the standard Comb Sort, while it may perform rather poorly near the end, at least it still sorts, unlike the next sorting algorithm…

Bogo Sort: 5/10 Difficulty

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best Time | Average Time | Worst Time | Memory | Stable? | Method |
| n | n × n! | ∞ | 1 | Yes | Exchanging |

Description:

Unlike any sort I have seen; probably why it took longer to understand. Understanding how to check if the list was sorted took longer than the ‘sorting’ process itself.

First, the sort checks if the list is sorted. If the list is unsorted, the list is randomly shuffled. This continues until the list is sorted.

Effectiveness (or lack thereof):

The (only) redeeming(?) factor of Bogo Sort is that repeating values help increase the possibility of the array being sorted.

The only reason one would use this sort is to explain the types of sorts to **not** use. Educational purposes matter, after all.