29.2 Selection Sort & Heapsort

Selection Sort

Selection sort uses the following algorithm:

1. Find the smallest item.
2. Swap that item to the front.
3. Repeat until all items are fixed (there are no inversions).

You can see a demo of the sorting algorithm here.

Selection sort runs in $\Theta(N^2)$ time using an array or similar data structure.

You may have noticed that selection sort seems inefficient, and you'd be right--we look through the entire remaining array each time to find the minimum, examining the same items over and over.

Heapsort

Naive Heapsort

To avoid the inefficiency mentioned above regarding selection sort, we can leverage a max-oriented heap instead of scanning over the array linearly.

Note: Because of the array-based representation of a heap, using a max heap results in a simpler implementation where we can maintain a "sorted" and "unsorted" portion of the array. This will be explained further in the next section.

Then, to heapsort N items, we can insert all the items into a max heap and create and output array. Then, we repeatedly delete the largest item from the max heap and put the

largest item at the end part of the output array.

Naive Heapsort Analysis

The overall runtime of this algorithm is $\Theta(N \log N)$. There are three main components to this runtime:

- Inserting N items into the heap: $O(N \log N)$.
- Selecting the largest item: $\Theta(1)$.
- Removing the largest item: $O(\log N)$.

This is a large improvement over selection sort's $\Theta(N^2)$ runtime.

In terms of memory usage, the output array takes an extra $\Theta(N)$space. This is worse than selection sort, which uses no extra space, but the improvement in runtime far outweighs this downside.

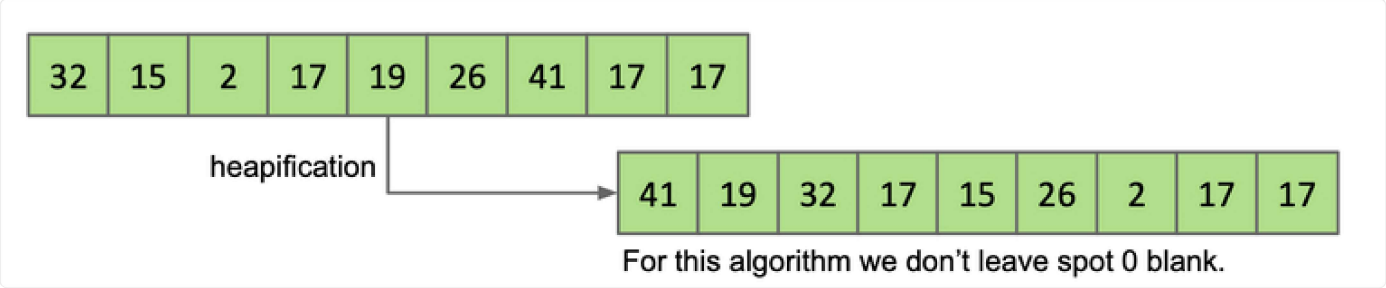
Even more, we can use a trick with heapsort to get rid of the extra output array.

In-place Heapsort

As an alternate approach, we can use the input array itself to form the heap and output array.

Rather than inserting into a new array that represents our heap, we can use a process known as *bottom-up heapification* to convert the input array into a heap. Bottom-up heapification involves moving in reverse level order up the heap, sinking nodes to their appropriate location as you move up.

By using this approach, we avoid the need for an extra copy of the data. Once heapified, we use the naive heapsort approach of popping off the maximum and placing it at the end of our array. In doing so, we maintain an "unsorted" front portion of the array (representing the heap) and a "sorted" back portion of the the array (representing the sorted items so far).



Bottom-up heapfication

You can see a demo of this algorithm here.

In-place Heapsort Runtime

This process overall is still $O(N \log N)$, since bottom-up heapification requires at most N sink-down operations that take at most $\log N$ time each.

Note: it is possible to prove that bottom-up heapficiation is bounded by $\Theta(N)$. However, this proof is out of scope for this class.

In-place Heapsort Memory

Using in-place heapsort, we reduce the memory usage of heapsort to $\Theta(1)$. Since we are reusing the input array, no additional space is used (and remember that we do not count the input when assessing memory complexity).

Previous
29.1 The Sorting Problem

Next
29.3 Mergesort

Last updated 1 year ago