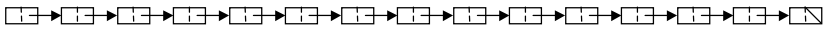
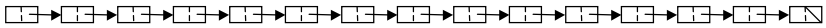


CSE 373: Sorting

(Shellsort, Heapsort, and Mergesort)

Chapter 7



Drawback of Adjacent Swap Sorts

- Each swap only fixes a single inversion
- Thus, elements that are far out of place must be swapped with many values instead of being moved into place more directly:

  4 5 7 8 9 2

- This is the motivation for Shellsort (*named after its inventor, Donald Shell*): try to move values to their general area quickly, then fix them up

Shellsort

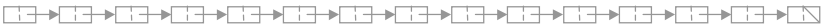
- Uses p phases
- The phases are characterized by an *increment sequence* of integers: $h_1, h_2, h_3, \ldots, h_p$:
  - Typically, $h_i > h_{i+1}$
  - $h_p = 1$ (last phase is insertion sort)
- In phase k, we compare and swap values that are h_k positions apart until they are sorted
- This essentially performs h_k independent insertion sorts in phase k

Shellsort Example

h = 5, 3, 2, 1 *input* = 7, 4, 8, 6, 9, 2, 5, 3

7	4	8	6	9	2	5	3
2	4	3	6	9	7	5	8
2	4	3	6	9	7	5	8
2	4	3	5	8	7	6	9
2	4	3	5	8	7	6	9
2	4	3	5	6	7	8	9
2	3	4	5	6	7	8	9

Increment Sequences



- Designing increment sequences:
  - Running time is proportional to the number of increments, so we don't want too many
  - But just having one would give us insertion sort
- Worst-case running time:

  $\Sigma_i (h_i (n/h_i)^2)$: h_i insertion sorts of n/h_i elements each;
  (recall: insertion sort has worst-case of $O(n^2)$)

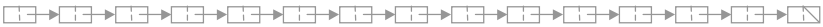
Common Increment Sequences

- Shell's original sequence:

  $h = n/2, n/4, n/8, \ldots, 2, 1$
  - probably the most intuitive sequence
  - but, it has a worst-case of $O(n^2)$
- Hibbard's sequence:

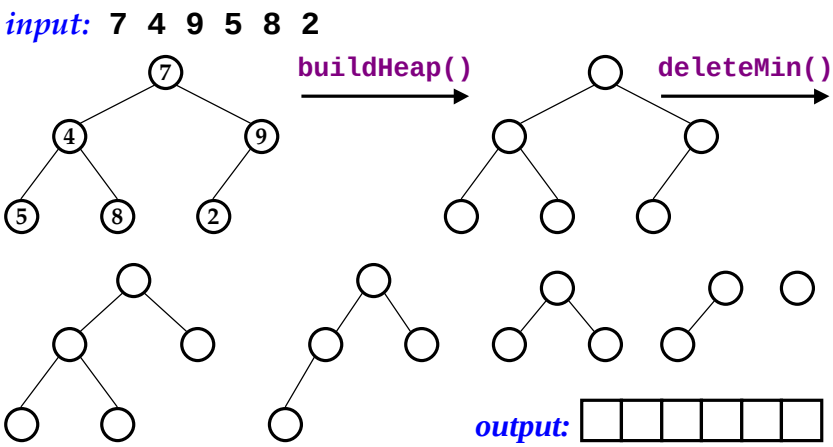
  $h = 2^k-1, \ldots, 15, 7, 3, 1$
  - adjacent numbers are relatively prime
  - leads to a worst-case of $O(n^{1.5})$

Heapsort



- Naive algorithm:
  - Run `buildHeap()` on the input array
  - Call `deleteMin()` n times, storing the results in an output array
- Running Time?
- Disadvantage?
- How can we fix this?

Heapsort Example



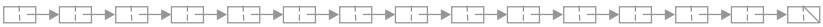
Improved Heapsort

- Use the heap's array to store the sorted values
- *Recall:* a *k*-element heap uses the first *k* positions of its implementing array
- Thus, whenever we delete an element from the heap, store it at the end of the array
- What does this give us?
- How to fix it?

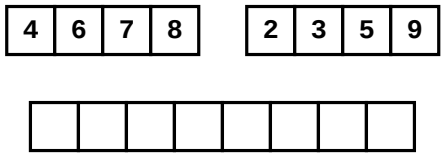
Treesort?

- BSTs can obviously be used to sort input
  - `insert()` all values
  - traverse tree in-order, copying to output array
- This is rarely done in practice (unless a tree is already being used to store the data)
  - asymptotically similar to Heapsort
  - *but* trees require more memory
  - *and* can't be done using only input array memory
  - might as well use Heapsort

The Merge Operation



- Given two sorted lists, `merge()` combines them into a single sorted list:



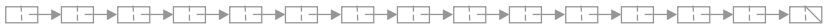
- Running time of `merge()`?

Mergesort

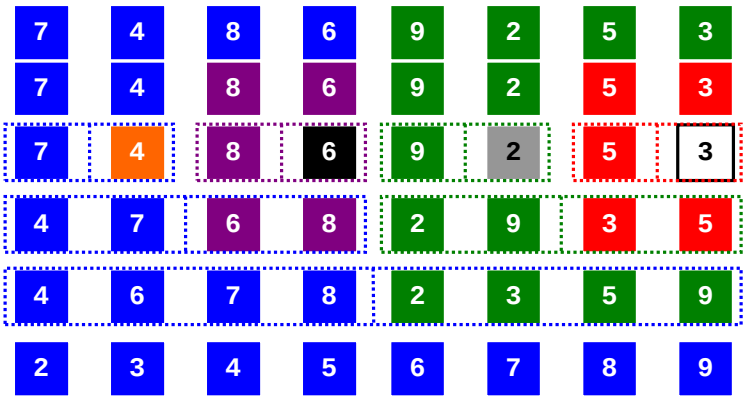
Elegant recursive sorting algorithm:

- if the input is one element, it's sorted; return
- otherwise, split the input into two equal-sized lists
- call `Mergesort()` recursively on each list
- `merge()` the sorted lists that are returned

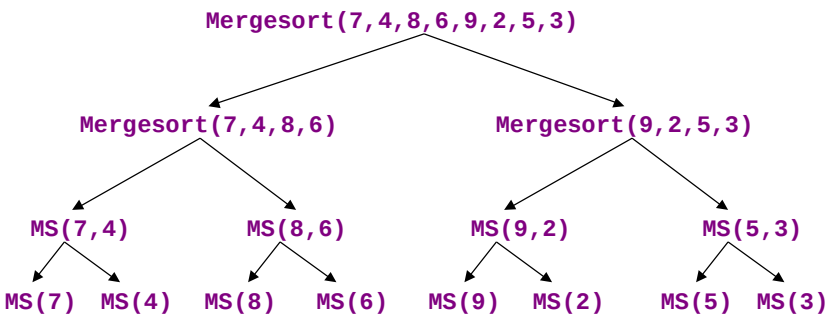
Mergesort Example



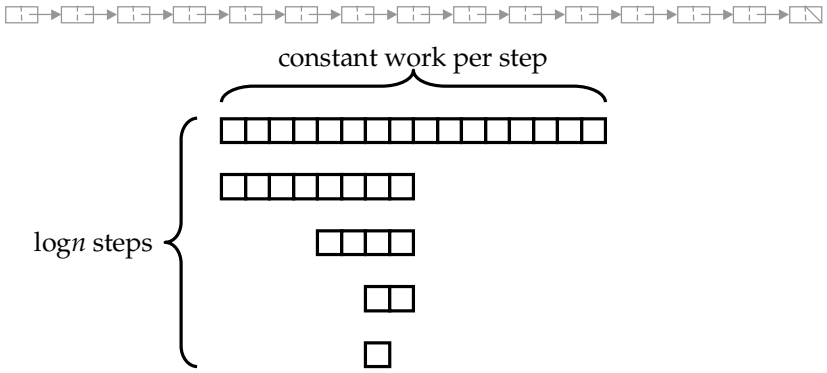
input = 7, 4, 8, 6, 9, 2, 5, 3



Mergesort Call Tree

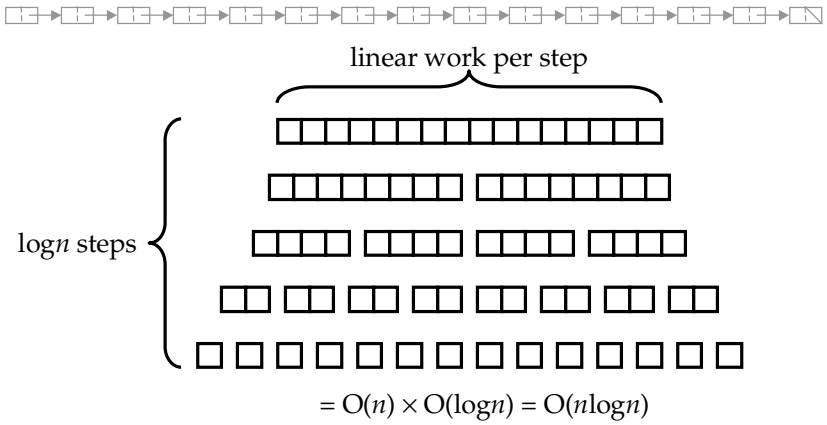


Binary Search Running Time



$= O(1) \times O(\log n) = O(\log n)$

Mergesort Running Time



$= O(n) \times O(\log n) = O(n \log n)$

Disadvantages?