Lecture 13 - Sorting

Eric A. Autry

Last time: Recursion and Recurrence Relations

This time: Sorting Algorithms

Next time: Use-It-or-Lose-It and Dynamic Programming

HW 5 due this Thursday, the 25th.

Project 1 assigned today and due next Thursday, the 1st.

Last Time: Divide & Conquer

Idea: break our problem into multiple, smaller subproblems.

We then solve these subproblems with recursive calls.

Then recombine the solutions.

$$T(n) = 3T(n/2) + O(n)$$

This recurrence relation comes from an algorithms that makes 3 recursive calls of size $n/2$, and that recombines those solutions in $O(n)$ time.

Sorting

Today we will consider the problem of sorting an array.

- The input array will have a positive integer length.
    - Will not be asked to sort an empty array.
    - Could be asked to sort a singleton array.
- The elements need not be distinct (i.e. repeats allowed).
- The elements can be any real number.

The task for a sorting algorithm is to sort the elements from the input array into ascending order so that the smallest element is at index 0, i.e.,

$$A[0] \leq A[1] \leq \cdots \leq A[k-1] \leq A[k] \leq A[k+1] \leq \cdots \leq A[end]$$

BogoSort

Idea:

- Take the array and randomly permute its elements in $O(n)$.
- Check if sorted in $O(n)$.
- Repeat until sorted...

Best Case: We get lucky and sort it the first time. $O(n)$

Worst Case: We get very unlucky and it never gets sorted.

Average Case:

- There are $n!$ possible permutations of the array.
- So the odds of getting the sorted array are $1/n!$
- Will require $n \cdot n! \in O((n+1)!)$.
- Worse than exponential runtime!

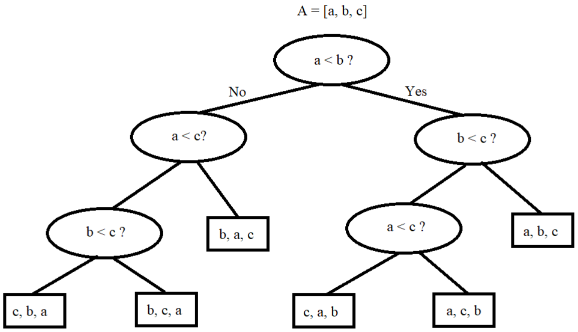
Lower Bound on Sorting

BogoSort is a terrible algorithm for sorting.

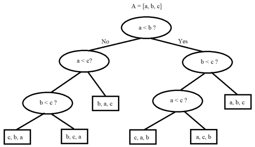
Can we do better? Almost certainly.

What is the best that we can do?

Let's think of sorting as a **comparison tree**:



Lower Bound on Sorting



The number of comparisons we will have to make (and therefore the runtime of the algorithm) will be proportional to the depth of this tree.

Note that each leaf is a possible permutation of the original array. So there are $n!$ leaves.

But a tree of depth d has 2^d leaves. So the depth of the tree is $\log(n!) \geq n \log n$.

Therefore **ALL possible sorting algorithms** will run in

$$\Omega(n \log n).$$

Selection Sort

- Separate the array into a sorted component and an unsorted component. Originally, the entire array is unsorted, and the sorted component is empty.
- Store the index separating the sorted and unsorted components starting at the front of the array, i.e., $k = 0$ originally.
- Iteratively search the unsorted component for the minimum element. $O(s)$ where s is the size of the unsorted component.
- When the minimum of the unsorted array is found, place it at the end of the sorted component (at location k) and increment the index k.

Best/Average/Worst Case:
$n + (n-1) + (n-2) + \cdots + 2 + 1 \in O(n^2)$

Note that this algorithm performs the same regardless of the input, so there is no difference between best and worst cases.

Insertion Sort

- Separate the array into a sorted component and an unsorted component. Originally, the entire array is unsorted, and the sorted component is empty.
- Store the index separating the sorted and unsorted components starting at the front of the array, i.e., $k = 0$ originally.
- Iteratively insert the element at $k + 1$ into the sorted component.
  - Search backwards (starting at position k) through the sorted component until we find where the element should go.
  - Shift the remaining sorted elements 1 index to the right.
  - Insert the element in that position.
  - Increment the index k.

Insertion Sort

Note that each insert+shift costs between:

- 1 work if the inserted element was already in the correct place, and
- k work if the inserted element was the smallest of the sorted component.

Best Case: The array was already sorted. Each insertion cost $O(1)$ work, with n total insertions for a total of $O(n)$ work.

Worst Case: The array is sorted in descending order so that each element has to be moved to the front of the sorted component. Each insertion costs k work, with n total insertions for a total of $1 + 2 + \cdots + n \in O(n^2)$ work.

Average Case: Each inserted element belongs in the middle of the sorted component. Each insertion costs $k/2$ work, with n total insertions for a total of $\frac{1}{2}(1 + 2 + \cdots + n) \in O(n^2)$ work.

Bubble Sort

- Iterate through the array.
- Compare every two adjacent elements.
- If they are out of order, swap them.
- Repeat until no more swaps are made.

Rabbits: note that large elements near the start of the array get moved very far in a single iteration.

- The kth largest element gets placed at the kth iteration.
- So we really only need to iterate through the first $n - k$ elements at each pass.

Turtles: note that small elements near the end of the array move only one location each iteration.

Bubble Sort

Best Case: If the array is already sorted, no swaps are made and only one pass of $O(n)$ is required.

- Note that if a turtle is m positions out of place, then it will require m iterations to be properly placed.

Worst Case: The smallest element was at the end of the array. This will require n iterations, each of size $n-k$. So the total work is $n+(n-1)+(n-2)+\cdots+1 \in O(n^2)$.

Average Case: The turtles are $n/2$ locations out of place. This will require $n/2$ iterations, each of size $n-k$. So the total work is

$$n+(n-1)+(n-2)+\cdots+n/2$$

$$=(n+(n-1)+\cdots+2+1)-((n/2-1)+\cdots+1)$$

$$=\frac{n(n+1)}{2}-\frac{(n/2)(n/2+1)}{2}=\frac{3}{8}n^2+\frac{1}{4}n \in O(n^2)$$

MergeSort

Divide & Conquer Approach

- Base Cases: If the array has 1 element, it is sorted. If the array has 2 elements, swap if needed and return.
- Split the array into two halves.
- Recursively sort each half.
- Merge the **already sorted** halves.
  - Iterate through them simultaneously.
  - Compare their smallest elements.
  - The smaller of the two gets removed and inserted into the merged array.
  - This merge takes $O(n)$ time when merging two arrays of size $n/2$.

$$T(n) = 2T(n/2) + O(n).$$

Runtime is $O(n \log n)$.

Quick Sort

Divide & Conquer Approach

- Base Cases: If the array has 1 element, it is sorted.
- Choose a pivot element.
- Partition the array based on the pivot. Put everything smaller than the pivot in front and everything larger than the pivot in back. $O(n)$
- Recurse on each partition.

Note that it is always best to split the array in half because this minimizes the size of the largest recursive call.

This will depend on our choice of pivot.

In the Best Case (and Average Case):

$$T(n) = 2T(n/2) + O(n) \quad \Rightarrow \quad T(n) \in O(n \log n).$$

In the Worst Case: our pivot was the smallest/largest element and one of the partitions is of size $n-1$. This is effectively Selection Sort and runs in $O(n^2)$.

Quick Sort

How do we choose the pivot?

- Pick the first element in the list. This works very poorly on already sorted inputs.
- Pick the last element in the list. This also works very poorly on already sorted inputs.
- Pick a random element in the list.
- Partition based on the median of the first, last, and middle elements.

No matter which choice we make, it is still possible (though in practice very rare) that we will get unlucky and have $O(n^2)$ running time.

Heap Sort

Think back to Selection Sort. We iteratively picked the minimum element out of the unsorted list.

Min-heaps are really good at iteratively picking the minimum!

- Put the entire unsorted array into a min-heap in $O(n)$ time.
- Track the index k separating the sorted and unsorted components of the array.
- Pop the minimum element from the heap and put it at index k, the end of the sorted component of the array.
- Increment k and repeat until everything is in place.

Each pop takes $O(\log n)$ time, and there are n of them. So the runtime is

$$n + n\log n \in O(n\log n).$$

Notes on Implementation

When coding comparsions, **do not** use:

```
if A[i] > A[j]:
    swap
if A[i] < A[j]:
    don't swap
```

Instead, you should use:

```
if A[i] > A[j]:
    swap
else:
    don't swap
```

Remember that repeats are allowed, so the first method would have to explicitly check if they are equal.

But if they are equal, then it doesn't matter if they are swapped.

Notes on Implementation

Note that the input arrays do not have to be powers of 2, or even multiples of 2.

If the array has an odd length and we go to split it into two halves, we need to compute the middle index as

$$k = \left\lfloor \frac{n}{2} \right\rfloor$$

How can we compute this value?

- In most programming languages, we can use integer division. If we divide two **integers**, the result will be an **integer**, i.e.,

  ```
  (int) 3 / (int) 2 = (int) 1
  ```

- In Python version 3.0 and later, integer division requires the '//' operator:

  ```
  (int) 3 // (int) 2 = (int) 1
  ```