Sorting

CS240 Spring 2019

Concerns

- We began the semester with bogosort
- When sorting, time isn't the only concern:
  - Extra memory requirements
    - *Internal vs. external sorting*
  - Stable vs. unstable sorts
  - Runtime complexity
    - *N^2 vs. $N\log_2 N$ performance*
    - *worst case vs. average case*

Memory

- Memory Requirements
  - In-Place Sort vs Out-of-Place Sort
    - *Does your algorithm need additional memory to sort?*
- Do you need all the data to sort?
  - Internal sorting algorithm
    - *the entire data to be sorted can be held in the computer's main memory.*
  - External sorting algorithm
    - *when the data cannot be stored in memory*
      - example: a very large transaction file which is stored over the network or on disc

Stable Sort vs Unstable Sort

- An algorithm is said to be stable if for any items in the unsorted list that are equivalent, their original order is maintained
- Example
  - sorting only numbers
    - *Stable: [3, 2a, 2b, 1] => [1, 2a, 2b, 3]*
    - *Unstable: [3, 2a, 2b, 1] => [1, 2b, 2a, 3]*

Simple Sorting

- For any sort algorithm, the resulting dataset must be a permutation of the original dataset
  - Must not add or subtract any elements
- $O(n^2)$ - Quadratic Sorts
  - Run through every element of an unsorted array and compare it (possibly swap) with every other element in the array
    - *Generally slow for large data sets*
- Why would you use slower sort?
  - easy and fast to code, performs better on certain kinds of datasets

Bubble

- Bubblesort starts at the beginning
  - if the first element is greater than the second element, it swaps them.
  - It continues doing this for each pair of adjacent elements to the end of the data set.
  - It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- Visualizing Bubble Sort
  - https://visualgo.net/sorting

Bubblesort Pseudocode

```
while(swapped)
    swapped = false
    for i => n
        if a[j] < a[j-1]
            swap a[j,j-1]
            swapped = true
```

BubbleSort

- Average case: $O(n^2)$
- Worst case: $O(n^2)$
- Use Case:
  - Use on a list that is already sorted except for a very small number of elements.
    - *Example: if only one element is not in order, bubble sort will take only 2n time.*
    - *That's linear sort!*

Selection Sort Pseudocode

```
for i = 0:n
   min = i
   for j = i+1:n
      if a[j] < a[min]
         min = j
   swap a[i,min]
end
```

Selection Sort

- Average case: $O(n^2)$
- Worst case:$O(n^2)$
- Use Case:
  - Use on hardware where swapping is an expensive operation
    - *Does no more than n swaps*
  - In practice, the slowest sort

Insertion Sort

```
for (i = 1:n)
    for (j = i; j>0 and a[j]<a[j-1]; j--)
        swap a[j,j-1]
end
```

Insertion

- Insertion sort works by
  - taking an element from the list
  - inserting the element in their correct position into a new sorted list.

Insertion Sort

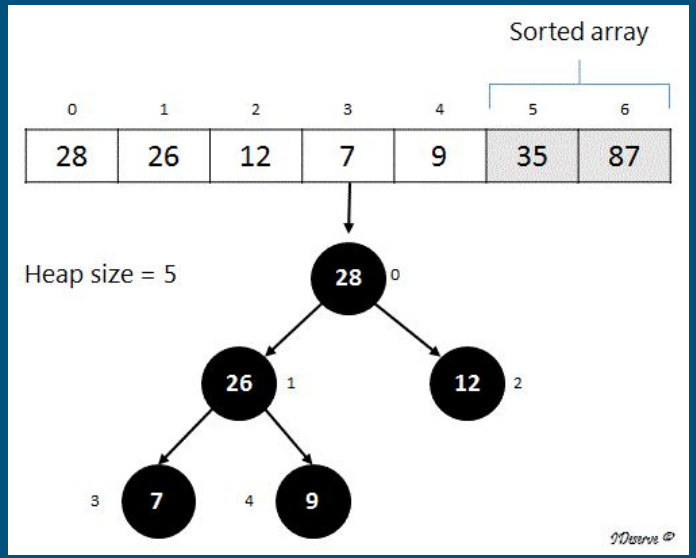
- Average case: n^2
- Worst case: n^2
- Stable Sort
  - Element order stays the same for equal elements
- Use Case:
  - Insertion sort is also called natural sort
    - *Mimics natural sorting of cards*
    - *Generally the fastest of the Quadratic sorts*
      - depending on implementation
    - *Also efficient on nearly sorted, like bubble sort*

Logarithmic Sorts

- nlogn sorts use top down sorting techniques
  - Also known as a divide and conquer sort
- Generally,
  - more difficult to code
  - fast for large data sets, slow for small datasets

Heapsort

- Sort a list by building it into a heap, then deleting the root node and placing it 1 past the last heap index until the heap is empty.
- This is an in place sort
  - Doesn't require an additional list to hold the sorted value



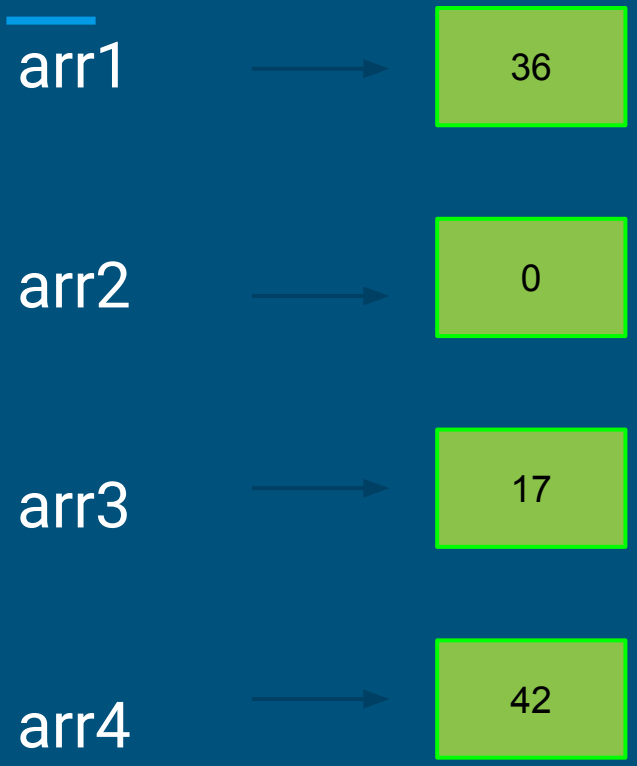
Heapsort

```
void HeapSort(int* list, int size) {
	MinHeap toSort(list, size);
	toSort.buildHeap();
	for(int i = size - 1; !toSort.empty(); i--){
		list[i] = toSort.removeRoot();
	}
}
```

Cost of Heapsort

- Complexity
  - Total Comparisons: NlogN
  - Total Swaps: NlogN
  - Heap Sort is T(n) = c(NlogN)
- Unstable Sort

What do you notice?



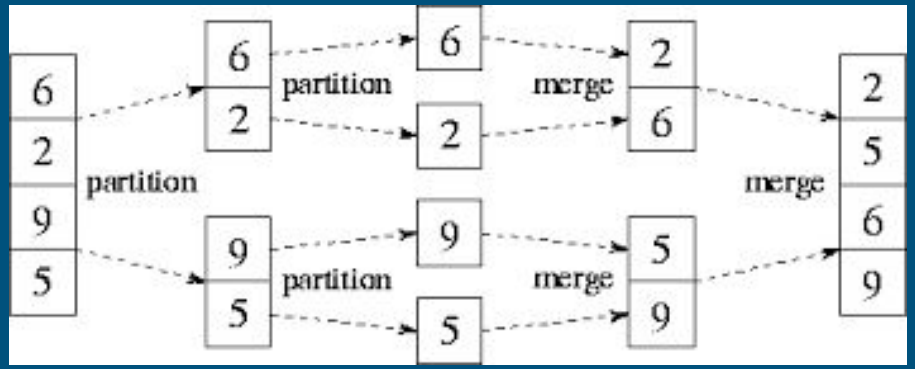
Each of these is a sorted array

MergeSort

- Two properties of lists allow mergesort to sort quickly:
  - A list containing 1 item is a sorted list
  - Merging 2 sorted lists into a new sorted list is an linear operation
- Splitting a list into two lists is a logn operation
  - Think of binary search

MergeSort

- Average case: nlogn
- Worst case: nlogn
- Stable Sort
- Mergesort has two parts
  - Divide list into two lists
  - Merge two sorted lists




a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
2
4
0
6
8
5
1
7
3
9
b[0]
b[1]
b[2]
b[3]
b[4]
b[5]
b[6]
b[7]
b[8]
b[9]

Merging 2 sorted arrays

- Given the pseudocode on the right, how many times are we going to loop?
  - no more than left->size + right->size
- Merging 2 sorted arrays is a linear operation
  - O(m+n)
    - *where m,n are the size of the 2 arrays*

```
merge( left, right )
        new sorted_array[ left->size + right->size ]
        i, j, k = 0;
        while ( i != left->size() and j != right->size() )
                if(left[i] > right[j] )
                        sorted_array[k] = right[j])
                        increment j and k
                else
                        sorted_array[k] = left[i])
                        increment i and k
        while ( left or right has elements )
                sorted_array[i] = remaining_elements;
                increment i
        return sorted_array
```

Classwork

Merge Function

```
mergesort( array a  )
        if ( a->size == 1 ) return a
        array_left = {a[0] ... a[n/2]}
        array_right = {a[n/2+1] ... a[n]}
        array_left = mergesort(array_left)
        array_right = mergesort(array_right)
        sorted_array = merge( array_left, array_right)
        return sorted_array;
```

What do you notice?

Choose a ‘pivot’, which can be any index in an array, and move all values ‘greater than’ to one side, and all values, ‘less than’ to the other

Quicksort

- Quicksort is a divide and conquer algorithm
  - relies on **partitioning**
- Using a pivot, all elements smaller than the pivot are moved before it and all greater elements are moved after it.
  - The lesser and greater sublists are then recursively sorted.

Quicksort

```
quicksort(array, low, high):
    if low < high:
      sorted = partition(array, low, high)
      quicksort(array, low, sorted - 1)
      quicksort(array, sorted + 1, high)
```


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
0
1
2
3
6
4
5
8

Implementing a Partition

- Dividing the array along a pivot is called partitioning
- ‘high’ and ‘low’ are the indexes you are working within the array
  - Remember, quicksort is in place, so we divide up the existing array
- The comparison is with the pivot, the swap is with the marker and the current index

```
partition(array, low, high)
    pivot = choosePivot(low, high)
    swap array[pivot, high]
    marker = low
    for i = low to (high−1)
        if array[i] < array[high]
            swap array[i, marker]
            marker = marker + 1
    swap array[marker, high]
    return marker
```

Partition Algorithm

- There are several different algorithms you can use to partition on a pivot
  - Each with pros and cons
- The video uses a pivot swap where you keep swapping the pivot until it is the correct position
- The partition pseudocode swaps values lesser and greater than the pivot, then puts the pivot into place

Classwork

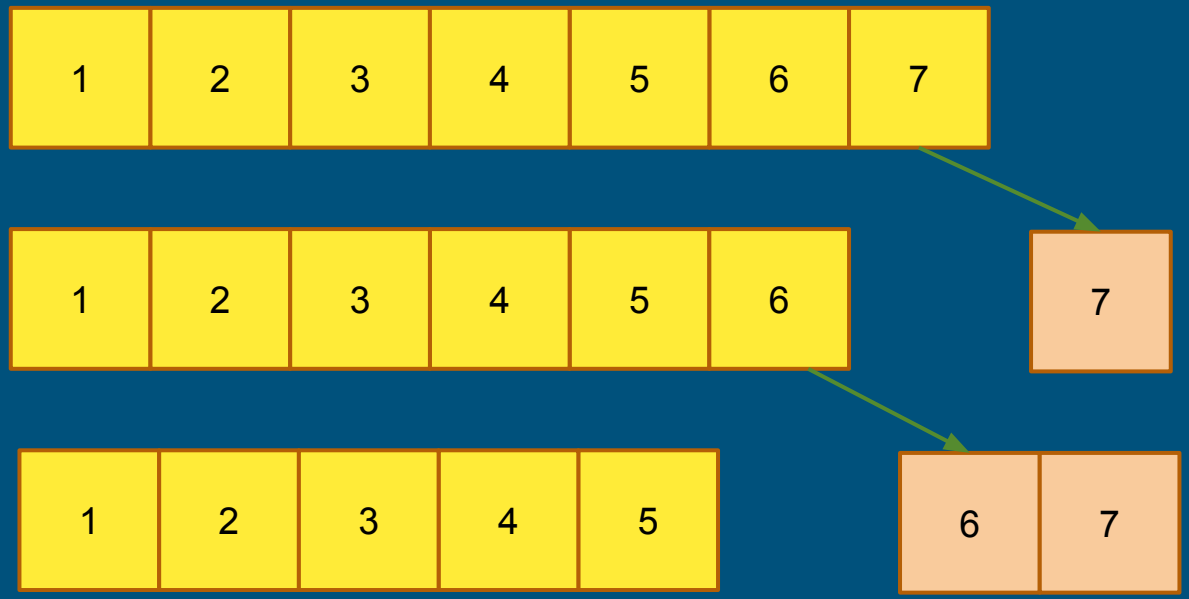
Quicksort

```
int pivotA(low, high):
  return low

int pivotB(low, high):
  //findMedian() loops through to find
  // the median value in the array
  return findMedian(low,high)

int pivotC(low, high):
        return random(low, high)
```

What do you notice?



Must compare every element against every other element

Choosing a bad Pivot

- consistently poor choices of pivots can result in dramatically slower $O(n^2)$
  - Pivot on first/last element on already sorted or nearly sorted lists?
  - Always choosing the lowest or highest value?

Choosing a pivot

- Options for choosing a good pivot
  - Finding the median, use as the pivot O(n log n).
    - *Finding the median however, is an O(n) operation, therefore O(n2)*
  - Select the center,
    - *1/n chance for worst case complexity*
  - Randomly select the pivot
    - *drastically reduce worst case possibility*

choosePivot

```
choosePivot(low, high)
    int pivot = low + random() % (high - low);
    return pivot
```

QuickSort

- Average case: O(nlogn)
- Worst case:O(n^2)
- Unstable Sort
- The fastest sort algorithm in practice.
- Use Case:
  - Any large, unsorted dataset
  - in place sort, Unlike mergesort, n space usage