AAA – Sorting

Ian Sanders

Second Semester, 2024

Current plan

- Analyse various classical algorithms and problems
- Introduce problem
- Discuss naïve approach
- Look at possible improvements:
- Algorithmic improvements
- Assumptions about the data
- Better data structures
- consider correctness, complexity and optimality

Sorting

- Given a list of numbers, arrange them in ascending order
- We will look at different approaches.
- Brute force:
  - Max sort
  - Selection sort
  - Bubblesort
- Decrease and conquer:
  - Insertion sort
- Divide and conquer:
  - Mergesort
  - Quicksort

Max Sort

On each pass: Find the maximum value and place at end.
Repeat

```
Algorithm maxSort($myList$, $n$)
    For $i$ from $n-1$ down to 1
        $maxPos \leftarrow i$
        For $j$ from 0 to $i-1$
            If $myList[j] > myList[maxPos]$
                Then
                    $maxPos \leftarrow j$
        swop($myList[maxPos], myList[i]$)
    Return $myList$
```

4	0	2	1	3	9	5	8	7	6
4	0	2	1	3	6	5	8	7	**9**
4	0	2	1	3	6	5	7	**8**	**9**
4	0	2	1	3	6	5	**7**	**8**	**9**
4	0	2	1	3	5	**6**	**7**	**8**	**9**
					...				

- Correctness
  Induction: will work for list of length 2
  Assume it works for length $k-1$
  Then for length k:
  First run of outer loop will place max value at last position in list, `MyList[k - 1]`
  This leaves us an unsorted sublist from 0 to $k-2$ (i.e. a sublist of length $k-1$)
  Induction hypothesis is that `maxSort` will work on this sublist correctly
- clarity: fairly straightforward
- complexity: always does the same amount of work, i.e. no best or worst case – $O(n^2)$
- optimality: is not optimal

Selection sort

- Start at first position
- Scan remaining list, find smallest value
- Swap current value with smallest
- Move to next position and repeat

```
Algorithm selectionSort($myList$, $n$)
    For $i$ from 0 to $n-2$
        $minPos \leftarrow i$
        For $j$ from $i+1$ to $n-1$
            If $myList[j] < myList[minPos]$
                Then
                    $minPos \leftarrow j$
        swop($myList[minPos], myList[i]$)
    Return $myList$
```

- clarity: fairly straightforward
- correctness: by induction
- complexity: always does the same amount of work – $O(n^2)$

Bubblesort

```
Algorithm bubbleSort($myList$, $n$)
    For $i$ from $n-1$ down to 1
        For $j$ from 0 to $i-1$
            If $myList[j] > myList[j+1]$
                Then
                    Swop($myList[j], myList[j+1]$)
    Return $myList$
```

- Method: Instead of swapping one element each time, do a bunch of swaps as we scan through!
- Complexity:- Amount of work done is O(n^2)
- But what about best, worst case?
- If list is unsorted? O(n^2)
- If list is already sorted? O(n^2)
- If list is in reverse order? O(n^2)
- So best, worst, average case is all the same!
- Can we do better?
- Yes! Keep track if we have done no swaps. If no swaps, list must be sorted, so stop!
- Best case is now O(n)

Optimality

- Max sort, selection sort, bubblesort are all $O(n^2)$
- This is not optimal. Why?
- Intuition: Given a list of n numbers, there are n! ways to arrange them.
- Sorting is about finding the arrangement amongst these that is sorted!
- Recall that each comparison gives us a binary output
- If algorithm takes $f(n)$ steps, then it cannot distinguish more than $2^{f(n)}$ cases
- So we need $2^{f(n)} \geq n! \implies f(n) \geq lg(n!)$
- $n!$ grows like n^n (Stirling's approx)
  $\implies f(n) \geq lg(n^n) = nlgn \implies f(n) \in \Omega(nlg(n))$

Later we will see sorts that are optimal!

Insertion sort

A decrease and conquer sorting algorithm

```
Algorithm insertionSort($myList$, $n$)
    For $i$ from 1 to $n-1$
        $x \leftarrow myList[i]$
        $j \leftarrow i-1$
        While ($j >= 0$) and $myList[j] > x$
            $myList[j+1] \leftarrow myList[j]$
            $j \leftarrow j-1$
        $myList[j+1] \leftarrow x$
    Return $myList$
```

Method

- Maintain sorted sublist
- For each new value, insert it into correct location in sublist
- After each insertion, sublist grows by one
- But always maintains sorted order!

X denotes number to be inserted into correct place in sorted sublist.

	X								
4	0	2	1	3	9	5	8	7	6
		X							
0	4	2	1	3	6	5	8	7	9
			X						
0	2	4	1	3	6	5	8	7	9
				X					
0	1	2	4	3	6	5	8	7	9
					...				

- clarity: not too difficult but requires a little bit of insight.
- Correctness: by induction
  1 element array is sorted
  assume sorted for list of length $n - 1$
  Prove list of n is sorted i.e. argue that the last element is inserted into its correct position, and that all elements before it are smaller (and sorted) and all elements after are larger (and sorted)

Complexity

- Best case:
  inner loop never runs
  1 comparison each time so $n-1$ comparisons
  Already in sorted order!
- Worst case:
  inner loop runs max times
  $1+2+\ldots+(n-1) \in O(n^2)$ comparisons
  List in reverse order
- Average case same as worst. See notes

A typical divide and conquer algorithm

```
Algorithm DivideAndConquer($Inputdata$)
    IF $limitingCase$
        Then
            $Answer \leftarrow$ DirectSolution($Inputdata$)
        Else
            Divide($Inputdata, I_1, I_2, ..., I_n$)
            $A_1 \leftarrow$ DivideAndConquer($I_1$)
            $A_2 \leftarrow$ DivideAndConquer($I_2$)
            ...
            $A_n \leftarrow$ DivideAndConquer($I_n$)
            $Answer \leftarrow$ Combine($A_1, A_2, ..., A_n$)
```

Mergesort

- A recursive procedure:
- Split list in two, sort left and right sublists
- Then merge the two sorted sublists.

```
Algorithm mergeSort($left$, $right$)
Input: $left$, $right$ where $left$ and $right$ are indices
    into an array $myList$ of $n$ entries
Output: a portion of the array $myList$ where the
    values in $myList$ are such that
    $myList[left] \leq myList[left+1] \leq \ldots \leq myList[right]$
    IF $right - left > 0$
        Then
            $mid \leftarrow \lfloor (left + right)/2 \rfloor$
            mergeSort($left, mid$)
            mergeSort($mid + 1, right$)
            merge($left, mid, mid + 1, right$)
```

```
Detail of the merge
        For $i$ from $mid$ down to $left$
            $temp[i] \leftarrow myList[i]$
        For $j$ from $mid + 1$ to $right$
            $temp[right + mid + 1 - j] \leftarrow myList[j]$
        $i \leftarrow left$
        $j \leftarrow right$
        For $k$ from $left$ to $right$
            If $temp[i] < temp[j]$
                Then
                    $myList[k] \leftarrow temp[i]$
                    $i \leftarrow i + 1$
                Else
                    $myList[k] \leftarrow temp[j]$
                    $j \leftarrow j - 1$
```

The merge

- Crux of algorithm is merge
- i.e. given two sorted sublists, combine into one sorted list
- Can be done in linear time because sublists are sorted!

Two sorted sub lists....

myList	...	21	29	99	10	42	92	...

The temporary storage...
Note second sublist is in reverse order

temp	...	21	29	99	92	42	10	...

Compare left and right ends of temp list and work inwards.

Merged (and now sorted) sub list

myList	...	10	21	29	42	92	99	...

Correctness

- By induction
- First, must show that merge is correct:
  Consider two sorted sublists
  Write them back to back in temp
  Compare values at begining and end of temp
  Write the smallest into myList
  Shift pointers appropriately
  At iteration k, k smallest elements have been copied from temp to myList in the required order.
  So by end of merge, all elements between left and right are in sorted order
- Base case: $n = 1$ is a sorted list
- Induction hypothesis: mergesort will sort any list with length $< n$
- So for list of length n, we call mergesort on two lists of $n/2$ size. Sorted by induction hypothesis. And we showed merge works. So list is sorted!

Complexity

- Mergesort does same thing regardless of input
- Split list in half, sort both halves, then merge
  $g(n) = 2 \times g(n/2) + n \implies g(n) \in O(nlg(n))$
- But see notes for actual proof
- Note: requires extra space for copying values for merge
- $O(n)$ space complexity
- Mergesort is $\Theta(n \log_2 n)$.
- Optimality – Baase says "Mergesort is very close to optimal", generally considered to be so.

Quicksort

- Again divide and conquer
- Select some value in list: call this the *pivot*
- Guarantee that:
  - Elements left of pivot are smaller
  - Elements right of pivot are larger
  - Therefore, pivot is in correct final spot
- Recursively sort left and right sublists!

Quicksort

```
Algorithm quickSort($left$, $right$)
    IF $right > left$
        Then
            $i \leftarrow$ partition($left$, $right$)
            quickSort($left, i - 1$)
            quickSort($i + 1, right$)
```

The crux of the algorithm is the `partition`.

Detail of the partition

```
$v \leftarrow myList[right]$
$i \leftarrow left$
$j \leftarrow right$
While $i < j$
    While $myList[i] < v$
        $i \leftarrow i + 1$
    While $j > i$ and $myList[j] \geq v$
        $j \leftarrow j - 1$
    If $j > i$
        Then
            $t \leftarrow myList[i]$
            $myList[i] \leftarrow myList[j]$
            $myList[j] \leftarrow t$
$t \leftarrow myList[i]$
$myList[i] \leftarrow myList[right]$
$myList[right] \leftarrow t$
```

Partition

Crux of algorithm is the partition

- Select myList[right] as element that will be moved to correct place at end
- Scan from left until element $\geq$ myList[right] found
- Scan from right until element $<$ myList[right] found
- Swap elements
- Continue until left and right pointers cross
- Swap myList[right] with element at left

Unsorted list

	l					r
myList	21	29	99	10	42	92

Pick myList[5] $=$ 92 as pivot.

	i				j	$p = r$
myList	21	29	99	10	42	92

Search from left for a number $\geq$ pivot, find 99
Search from right for a number $<$ pivot, find 42.

			i		j	$p = r$
myList	21	29	99	10	42	92

Swap and continue comparing.

			i		j	$p = r$
myList	21	29	42	10	99	92

From left find 99.

From right find 10.

Pointers have crossed.

				j	i	p = r
myList	21	29	42	10	99	92

Swop number at left pointer (i) with pivot (right).

myList	21	29	42	10	92	99

Pivot is now in its correct position.

	$<$	$<$	$<$	$<$	✓	$>$
myList	21	29	42	10	92	99

Recursively quicksort left and right sublists

Complexity

- Complexity of partition is $O(r - l) \implies$ linear
- Partitions, then sorts left and right sublists
  $g(n) = \Theta(n) + g(\mathit{sizeleft}) + g(\mathit{sizeright})$
- Best case: when partitioning divides the list exactly in half
  $g(n) = g(n/2) + g(n/2) + n \implies g(n) \in O(n lg(n))$
- Worst case: when lists completely unbalanced
  $g(n) = g(n-1) + g(0) + n \implies g(n) \in O(n^2)$
- Worst case is when list is already sorted.

Conclusion

- Looked at nonoptimal sorting algorithms
- Selection sort, bubblesort, insertion sort
- Looked at divide and conquer algorithms
- Mergesort, quicksort
- $nlgn$ is optimal for comparison sorts
- In practice, mergesort, quicksort both good candidates
- Insertion sort good for small lists
- Many other sorts out there
  See the Honours Algorithms course!