Classic Algorithms: Sort II

Advanced Analysis of Algorithms – COMS3005A

Steve James

Steven.James@wits.ac.za

Course notes

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 list of numbers, arrange them in ascending order

- Brute force:
  - Max sort
  - Selection sort
  - Bubblesort

LAST TIME

- Decrease and conquer:
  - Insertion sort

Exploit relationship between solution to instance, and solution of smaller instance of same problem

- Divide and conquer:
  - Mergesort
  - Quicksort

Divide problem into independent subproblems, then solve them with the same technique!

Insertion sort

- 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! Loop invariant
- 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

Algorithm 5 insertionSort($myList$, n)

The insertion sort algorithm

Input: $myList, n$ where $myList$ is an array with n entries (indexed $0 \ldots n-1$)

Output: $myList$ where the values in $myList$ are such that

$myList[0] \leq myList[1] \leq \ldots \leq myList[n-2] \leq myList[n-1]$

```
For i from 1 to n − 1
    x ← myList[i]
    j ← i − 1
    While (j >= 0) and myList[j] > x
        myList[j + 1] ← myList[j]
        j ← j − 1
    myList[j + 1] ← x
Return myList
```

Best case: doesn't loop at all!
Worst case: loops max times!

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

Mergesort

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

Algorithm 7 mergeSort($left$, $right$)

The mergesort algorithm

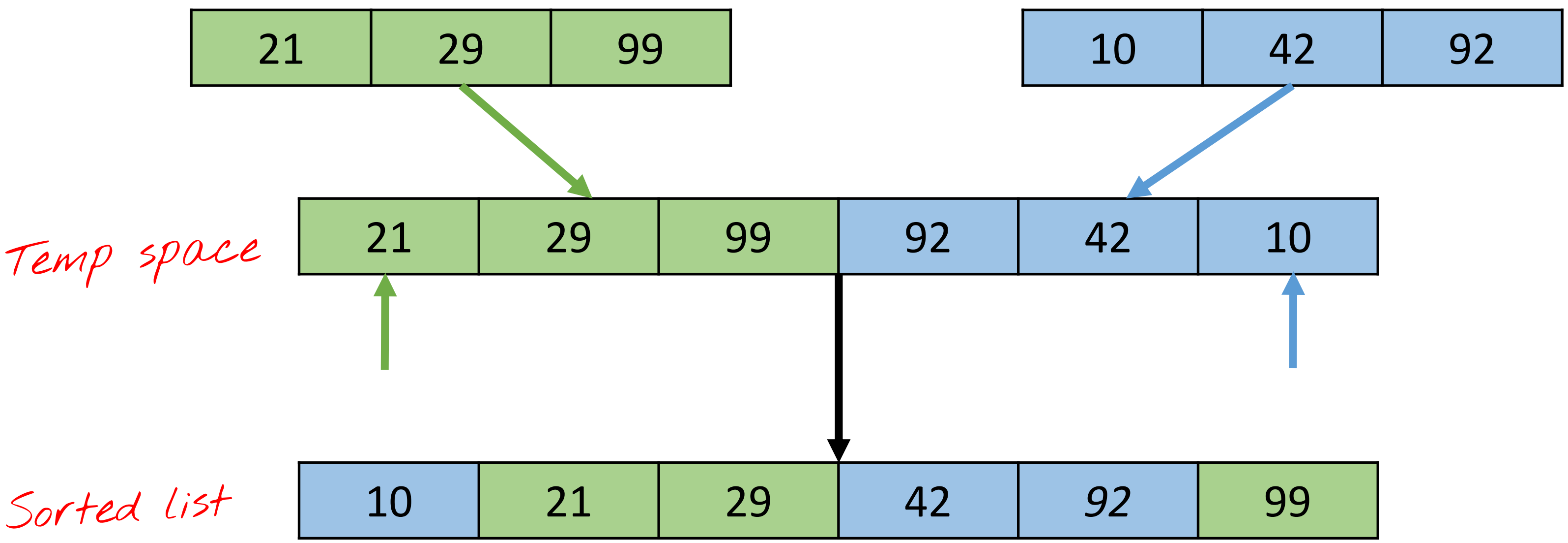
Input: $left$, $right$ where $left$ and $right$ are indices into an array $myList$ of n entries (initially indexed $0 \dots n-1$)

Output: a portion of the array $myList$ where the values in $myList$ are such that $myList[left] \leq myList[left+1] \leq \dots \leq myList[right-1] \leq myList[right]$

IF $right - left > 0$
  Then
    $mid \leftarrow \lfloor (left + right)/2 \rfloor$
    mergeSort($left$,mid)
    mergeSort($mid + 1$,$right$)
    merge($left$,mid,$mid + 1$,$right$)

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!



Correctness

- By induction
- First, must show that merge is correct:
  - Can show that at iteration k, k smallest elements have been copied from temp to array. 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 size $\frac{n}{2}$. 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

$$g(n) = 2 \times g\left(\frac{n}{2}\right) + n \Rightarrow g(n) \in O\big(n\lg(n)\big)$$

- But see notes for actual proof

- Note: requires extra space for copying values for merge
  - $O(n)$ space complexity

Quicksort

- Select value in list: the pivot
- Guarantee that:
  - Elements left of pivot are smaller
  - Elements right of pivot are larger
  - Therefore, pivot is in correct final spot

Not necessarily sorted!

- Recursively sort left and right sublists!

Algorithm 9 quickSort($left$, $right$)

The quicksort algorithm

Input: $left$, $right$ where $left$ and $right$ are indices into an array of n entries (initially indexed $0 \ldots n-1$)

Output: a portion of the array $myList$ where the values in $myList$ are such that $myList[left] \leq myList[left+1] \leq \ldots \leq myList[right-1] \leq myList[right]$

```
IF right > left
   Then
      i ← partition(left, right)
      quickSort(left,i − 1)
      quickSort(i + 1,right)
```

Partition

- Crux of algorithm is the partition subroutine
- 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 $left$

Other ways to pick pivot (e.g. randomly!)

Complexity

- Complexity of partition is $O(r - l) \rightarrow$ linear
- Partitions, then sorts left and right sublists
  $$g(n) = \Theta(n) + g(size\ left) + g(size\ right)$$
- Best case: when partitioning divides the list exactly in half
  $$g(n) = g\left(\frac{n}{2}\right) + g\left(\frac{n}{2}\right) + n \Rightarrow g(n) \in O\big(n\lg(n)\big)$$
- Worst case: when lists completely unbalanced
  $$g(n) = g(n-1) + g(0) + n \Rightarrow g(n) \in O(n^2)$$

Rightmost element doesn't move (i.e. list is sorted)

Conclusion

- Looked at non-optimal sorting algorithms
  - Selection sort, bubblesort, insertion sort
- Looked at divide and conquer algorithms
  - Mergesort, quicksort
- $n\lg(n)$ 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!

Suggests hybrid sorts!