Classic Algorithms: Search

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

Search

- Given a list of numbers, find a particular key

- Assumptions
  - Key is in the list
  - Key appears exactly once

See notes for when key is not in list

- Linear search
- Binary search

Very quickly!

Basic linear search

Algorithm 1 linearSearch($myList$, n, key)

Input: $myList, n, key$ where $myList$ is an array with n entries (indexed $0 \ldots (n-1)$), and key is the item sought.

Output: $index$ the location of key in $myList$.

$index \leftarrow 0$
While $key \neq myList[index]$
    $index \leftarrow index + 1$
Return $index$

Correctness

- If the key is in the list, will linear search find it?
- Yes
- Sketch proof:
  - Yes.

Direct proof

- See notes for inductive proof (especially when key may not be in list)

Complexity

10	21	42	29	92	61

- Measure in terms of comparisons
- Best case
  - Key is 10 – 1 comparison $O(1)$
- Worst case:
  - Key is 61 – length of list comparisons $O(n)$

Average complexity

10	21	42	29	92	61

- Key could be any number in list
- All numbers equally likely:
  - $P(index\ i\ is\ key) = \frac{1}{n}$
- Number comparisons if key is at index i
  - $i + 1$
- So average work is $\frac{1}{n}\sum_i (i + 1\ ) = O(n)$

Optimality

- Is there a better algorithm out there?
  - i.e. given the same assumptions, can some algorithm perform fewer comparisons?

Lower bound on comparisons to solve the problem

- Answer: No, linear is optimal
  - For an unordered list, any correct algorithm must check $O(n)$ values.
  - Adversarial argument: algorithm must check elements, but could be in any order
  - Adversary could construct a list so that the key is always the last value checked!

Binary search

- Assume list is sorted
- Use this to divide the problem → limit search space

Algorithm 3 bisectionSearch(myList,n,key)

Input: $myList, n, key$ where $myList$ is an array with n entries (indexed $0 \ldots (n-1)$), and key is the item sought.
The values stored in $myList$ are such that
$myList[0] \leq myList[1] \leq \ldots \leq myList[n-2] \leq myList[n-1]$
Output: mid the location of key in $myList$.

$low \leftarrow 0$
$high \leftarrow n-1$
$mid \leftarrow \lfloor (low + high)/2 \rfloor$
While $key \neq myList[mid]$
    If $key < myList[mid]$
        Then $high \leftarrow mid - 1$
        Else $low \leftarrow mid + 1$
    $mid \leftarrow \lfloor (low + high)/2 \rfloor$
return mid

Correctness

- If the key is in the sorted list, will binary search find it?
- Yes, by the rules of arithmetic
- See notes for slightly longer discussion!

Complexity

10	21	29	42	61	92	99

- Measure in terms of comparisons
- Best case
  - Key is 42 – 1 comparison $O(1)$
- Worst case:
  - Key is last number checked (e.g. 10)

Worst case complexity

10	21	29	42	61	92	99

- For list of length n, binary search will do at most $g(n) = 1 + g(\lfloor n/2 \rfloor)$ comparisons
- We also have the boundary condition $g(1) = 1$
- Solve this recurrence equation with ansatz!
- Solution: $g(n) = \lfloor \lg(n) \rfloor + 1$

lg = Base 2

Optimality

- Is there a better algorithm out there?
  - i.e. given the same assumptions, can some algorithm perform fewer comparisons?
- Answer: No, binary search on sorted list is optimal
  - See discussion in notes
- To-do: read analysis of linear + binary search for when key may not be in list

Classic Algorithms: Sort I

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
- Decrease and conquer:
  - Insertion sort
- Divide and conquer:
  - Mergesort
  - Quicksort

NEXT TIME

Selection sort

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

Max Sort

- Opposite of selection sort
  - Find max value and place at end. Repeat

Algorithm 1 maxSort($myList$, n)

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

Num comparisons = (n-1) + (n-2) + ... = n(n-1)/2 = O(n^2)

Correctness

- Yes, will sort list
- 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 $(k - 1)$
  - This leaves us an unsorted sublist from 0 to $k - 2$ (i.e. a sublist of length $k - 1$
  - Induction hypothesis is that max sort will work on this sublist correctly
  - QED

Bubblesort

- Instead of swapping one element each time, do a bunch of swaps as we scan through!

Algorithm 3 bubbleSort($myList$, n)

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 n − 1 down to 1
    For j from 0 to i − 1
        If myList[j] > myList[j + 1]
            Then
                Swap(myList[j], myList[j + 1])
Return myList
```

6 5 3 1 8 7 2 4

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 not, 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 $2^{f(n)} \geq n! \Rightarrow f(n) \geq \lg(n!)$
  - $n!$ grows like n^n (Stirling's approx)

    $\Rightarrow f(n) \geq \lg(n^n) = nlg(n) \Rightarrow f(n) \in \Omega(n \lg(n))$
- In next section, we will see sorts that are optimal!