Math/CSE 1019C: Discrete Mathematics for Computer Science
Fall 2012

Jessie Zhao
jessie@cse.yorku.ca

Course page:
http://www.cse.yorku.ca/course/1019

- Final Exam
  - Time: Dec 20^{th} , 2pm
  - Location: TM TMEAST
- Assignment 6 is released!

Analysis of Algorithms

- Measures of efficiency:
  - Running time
  - Space used (Not included in this course)
- Efficiency as a function of input size
  - Number of data elements (numbers, points)
  - Number of bits in an input number
    - Example: Find the factors of a number n
    - Example: Determine if an integer n is prime
    - Example: Find the max in A[1..n]

- What operations are counted?
  - Arithmetic (add, subtract, multiply, etc.)
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - Comparison

Analysis of Find-max

- COUNT the number of cycles (running time) as a function of the input size

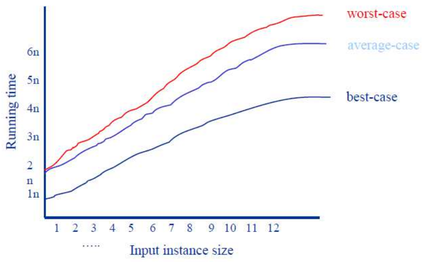
Find-max (A)	cost	times
1. max ← A[1]	c_1	1
2. for j ← 2 to length(A)	c_2	n
3. do if (max < A[j])	c_3	n-1
4. max ← A[j]	c_4	$0 \le k \le n-1$
5. return max	c_5	1

- Running time (upper bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$
- Running time (lower bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3)n$

Best/Worst/Average Case Analysis

- Best case: A[1] is the largest element.
- Worst case: elements are sorted in increasing order
- Average case: ? Depends on the input characteristics
- What do we use?
- Worst case or Average-case is usually used:
  - Worst-case is an upper-bound; in certain application domains (e.g., air traffic control, surgery) knowing the worst-case time complexity is of crucial importance
  - Finding the average case can be very difficult; needs knowledge of input distribution.
  - Best-case is not very useful

Best/Worst/Average Case



- Running time upper bound: $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$
- What are the values of c_i?
  - Machine-dependent
  - Constant to n
- A simpler expression: a + bn
- The running time is Θ(n)
  - a + bn is O(n)
    - ∃ constants C and k such that ∀n>k a + bn ≤ Cn
  - a + bn is Ω(n)
    - ∃ constants C and k such that ∀n>k a + bn ≥ Cn

- Bounds on running time
  - 1. O() is used for upper bounds "grows slower than"
  - 2. Ω() used for lower bounds "grows faster than"
  - 3. Θ() used for denoting matching upper and lower bounds. "grows as fast as"
- The rules for getting the running time are
  - 1. Throw away all terms other than the most significant one
  - 2. Throw away the constant factors.
  - 3. The expression is Θ() of whatever's left.

Sorting and Searching

- Very basic operations
- Used very, very often in real applications
- LOTS of new ideas

Searching an array

- Given an array A[1..n] does there exist a number (key) x?
- Unsorted array: linear search
  - Input: A[1..n]: array of distinct integers; x: an integer.
  - Output: The location of n in A[1..n], or 0 if n is not found.
- LinearSearch(A,x)
  - j=1
  - Loop
    - <loop invariant>: x is not in the scanned subarray.
    - Exit when j>n or x=A[j]
    - j=j+1
  - End loop
  - if j<=n then return j
  - else return 0

Proof using loop invariant

Loop

- Input
- Loop
  - Loop Invariant I
  - Exit when E
  - <code>
- End Loop
- Output

Prove its correctness

- Step 1. Basis case: Input->I
- Step 2. Inductive Step: Assume I(i) is true before the ith interaction. Prove it is true after i+1 iteration.
  I(i)∧˥E->I(i+1)
- Step 3. Show loop terminate and return the correct results.
  I∧E->Output

LinearSearch(A,x)

```
j=1
Loop
    <loop invariant>: x is not in the scanned subarray.
    Exit when j>n or x=A[j]
    j=j+1
End loop
if j<=n then return j
else return 0
```

- Proof by using loop invariant.
  - Basis Case: j=1. No element is scanned.
  - Inductive Step: Assume x is not in the scanned subarray A[1..j-1]. Prove x is not in A[1..j] if the loop does not terminate. Prove the output is correct if the loop terminates.
    - If the loop does not terminate.
      - j<=n and x≠A[j]. Then x is not in A[1..j].
    - If the loop terminates.
      - j>n or x=A[j]
      - If j>n, then j=n+1. By loop invariant, x is not in A[1..n]. The output **0** is correct.
      - If x=A[j], the n j is returned. The output **j** is correct.

LinearSearch(A,x)

```
j=1
Loop
    <loop invariant>: x is not in the scanned subarray.
    Exit when j>n or x=A[j]
    j=j+1
End loop
if j<=n then return j
else return 0
```

- Running Time?
  - Outside the loop: Constant O(C)
  - The loop: O(n)
  - Overall: O(n)

- Sorted array: Can we do better?
- Binary search: Use the sorted property to eliminate large parts of the array
  - Input: L[1..n]: a sorted array L(i)<L(j) if 1≤i<j≤n; x: an integer.
  - Output: The location of n in A[1..n], or 0 if n is not found.
- BinarySearch(L,x)
  - i=1, j=n
  - Loop
    - <loop invariant>: If x is in L[1..n], then x is in L[i..j].
    - Exit when j<=i
    - mid = ⌊(i+j)/2⌋
    - If (x ≤ L(mid)) then
      - j=mid
    - Else
      - i=mid+1
    - End if
  - End loop
  - if (x=L(i)) then return i
  - else return 0

Running time?
O(log n)

- By preprocessing (sorting) the data into a data structure (sorted array), we were able to speed up search queries.
  Very common idea in Computer Science
- Many other data structures are commonly used: linked lists, trees, hash tables,….
- CSE 2011: Data Structures
- CSE 4101: Advanced Data Structures

Sorting

- Input: A[1..n]: array of distinct numbers
- Output: A[1..n]: a sorted array A(i)<A(j) if 1≤i<j≤n

- Simple algorithm using FindMax
  - 1. j=n
  - 2. while (j>1){
  - 3. maxindex = index of max A[1..j]
  - 4. swap (A[maxindex], A[j])
  - 5. j=j-1
  - 6. }
- Proof and Running time? $O(n^2)$

Sorting: Insertion sort

- We maintain a subset of elements sorted within a list.
  - Initially, think of the first element in the array as a sorted list of length one.
  - One at a time, we take one of the elements (from the original list) and we insert it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before.
  - When the last element has been inserted, the array is completely sorted.

```
for j=2 to length(A)
  do key=A[j]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Insert A[j] into the sorted list A[1..j-1]

```
for j=2 to length(A)
  do key=A[j]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Insert A[j] into the sorted list A[1..j-1]

Loop Invariant: at the start of each for loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order

- Proof:
  - Basis Step: j = 2, the invariant holds because A[1] is a sorted array.


```
for j=2 to length(A)
  do key=A[j]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Insert A[j] into the sorted list A[1..j-1]

Loop Invariant: at the start of each for loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order

- Inductive Step: Assume elements in A[1…j-1] are sorted.
  - The inner while loop moves elements A[j-1], A[j-2], …, A[k] one position right without changing their order.
  - Then the former A[j] element is inserted into kth position so that A[k-1] ≤ A[k] ≤ A[k+1].
  - A[1…j] is sorted.


```
for j=2 to length(A)
  do key=A[j]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Insert A[j] into the sorted list A[1..j-1]

Loop Invariant: at the start of each for loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order

- Termination: the loop terminates, when j=n+1.
  - By loop invariant: "A[1…n] consists of elements originally in A[1…n] but in sorted order"
  - The output A[1..n] is correctly sorted.

Analysis of Insertion Sort

- Let's compute the running time as a function of the input size.

	cost	times
for j←2 to n	c_1	n
do key←A[j]	c_2	n-1
i←j-1	c_3	n-1
while i>0 and A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] ← key	c_7	n-1

- What is the running time?
  $O(n^2)$

Many, many other sorts

- bubble sort $O(n^2)$
- merge sort, quick sort, heap sort
  O(nlogn)
- Linear time sorts (require certain type of input): counting sort, radix sort, bucket sort.

Greedy Algorithm

- Optimization problems
  - Find a solution to the given problem that either minimizes or maximizes the value of some parameters.
- Greedy Algorithm
  - Select the best choice at each step
  - Does the solution always be optimal?

Greedy Algorithm

- Example: Want to make change for ANY amount using the fewest number of coins
- Simple "greedy" algorithm: keep using the largest denomination possible
  - Works for our coins: 1,5,10, 25,100.
  - Does it always work?
  - Fails for the following coins: 1,5,7,10
  - e.g: 14 =10 + 1 +1 +1 +1, 14 = 7 + 7

- Prove the greedy algorithm works for {1,5,10, 25,100}.
  - Lemma 1. Using the fewest coins possible has at most three 25(quarters), two 10(dimes), one 5(nickels), four 1(cents), and can not have two 10 and one 5 together.
    - Proof by contradiction: If we have more than any above numbers, then we can replace them with fewer coins.

- Prove the greedy algorithm works for {1,5,10, 25,100}.
- Proof by contradiction:
  - Assume there is an integer n, such that there is a way to make changes using less coins than the greedy algorithm.
  - Suppose different numbers for 100 (dollars): x dollars for greedy algorithm, and y dollars for the optimal solution
    - By greedy algorithm x>=y
    - If x>y, then we need to make up at least 100 from {1,5,10, 25}. This is impossible by lemma 1.
  - Similarly we can prove the greedy solution and the optimal solution won't have different numbers for {1,5,10, 25}.
  - Q.E.D.

Goals

- Understand existing classic algorithms
- Design simple algorithms
- Prove the correctness of an algorithm
  - Loop Invariant
- Analyze the time complexity of an algorithm