Recitation 2 - Asymptotic Complexity and Loop Invariants

Sebastian Laudenschlager

sebastian.laudenschlager@colorado.edu

January 26, 2018

Asymptotic Complexity

- Recall the various asymptotic complexity definitions:
- Big O, e.g. $\mathcal{O}(n^2)$
  - Asymptotic upper bound
  - Informal intuition: $f(n) \in \mathcal{O}(g(n))$ means that $f(n) \leq g(n)$, asymptotically.
  - Formally: If $f(n) \in \mathcal{O}(g(n))$, then there exist (positive) constants c and n_0 such that $f(n) \leq cg(n)\ \forall n \geq n_0$.
- Big Omega, e.g. $\Omega(n)$
  - Asymptotic lower bound
  - Informally: $f(n) \in \Omega(g(n)$ means that $f(n) \geq g(n)$, asymptotically.
  - Formally: If $f(n) \in \Omega(g(n))$, then there exist (positive) constants c and n_0 such that $f(n) \geq cg(n)\ \forall n \geq n_0$.
- Theta, e.g. $\Theta(n \log n)$
  - Asymptotic tight bound
  - Informally: $f(n) \in \Theta(g(n))$ means that $f(n) = g(n)$, asymptotically.
  - Formally: If $f(n) \in \Theta(g(n))$, then there exist (positive) constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$.

Example

- Let's look at a simple example:

```
def findMax(A):
    max = 0
    for x in A:
        if x > max:
            max = x
    return max
```

- What asymptotic complexity does this code have?

Another example

- Consider the following code for Insertion Sort:

```
def insertionSort(A):
    for j = 2:len(A):
        key = A[j]
        i = j - 1
        while i > 0 && A[i] > key:
            A[i + 1] = A[i]
            i--
        A[i + 1] = key
```

- Complexity?

Example, now with two variables

- Consider the following code to compute a matvec (matrix-vector multiplication):

```
def matvec(A, x):
    # A = m x n, x = n x 1 => b = ?
    for i = 0:m
        b[i] = 0
        for j = 0:n
            b[i] += A[i][j] * x[j]
```

- What's the complexity here?

One more example, now with recursion

- Consider the following code for Binary Search:

```
def binarySearch(A, l, r, target):
    if (l > r):
        return -1
    mid = l + (r - l) / 2
    if (target == A[mid]):
        return mid
    if (target > A[mid]):
        return binarySearch(A, mid + 1, r, target)
    else:
        return binarySearch(A, l, mid - 1, target)
```

- Complexity?

Example, cont.

- Worst case?
  - Recursively halve array at each call.
  - Base case: $l <= r$ when $l = r = 1$
  - How many times (k) do we halve the array?
  - $\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$
  - Hence, worst case is $O(\log n)$.
- Best case?
  - We find the target value on the first try $\Rightarrow O(1)$.

Loop Invariants

- Often used to prove an algorithm's correctness.
- Three parts:
  - Initialization: Show that your loop invariant holds initially.
  - Maintenance: Show that your loop invariant is maintained at each iteration.
  - Termination: Show that your loop invariant holds after the loop terminates.

Example

- Recall **Insertion Sort** from earlier
- Let's show the correctness of **Insertion Sort** using the following loop invariant:
- At the beginning of each iteration of the `for`, the subarray $A[1, \ldots, j-1]$ consists of the original elements $A[1, \ldots, j-1]$ in sorted order.

Initialization

- Before the first iteration (when $j = 2$), $A[1, \ldots, j-1]$ consists of just $A[1]$, which is trivially in sorted order.

Maintenance

- Inductively, we assume that at the beginning of an iteration, $A[1, \ldots, j-1]$ is sorted.
- For the j-th iteration, we simply insert $A[j]$ into its correct position in the subarray $A[1, \ldots, j]$.
- At the end of the iteration, we now have that $A[1, \ldots, j]$ is in sorted order, and thus the loop invariant is preserved.

Termination

- The for loop terminates when $j = n + 1$.
- At this point, we know that the subarray $A[1, \ldots, j-1] = A[1, \ldots, n]$ contains the original elements in $A[1, \ldots, n]$ in sorted order.
- Thus, when the loop terminates, the whole array is in sorted order.

The end

- Any questions?
- Have a good weekend and see you next week.