CS180 - Asymptotic Analysis

Note Title 9/7/2010

Announcements

- Program due tomorrow
- Next HW is posted, due 1 week from tomorrow (make sure the code works!)
- Review session Feb. 28 (Monday) Midterm on Tuesday, March 1

How to measure speed of a program?

Counting primitive operations

Identify high-level primitive operations, independent of language, compiler, OS, or computer

Operations:
- create variables
- variable assignment
- comparison
- addition
- multiplication...
- boolean operations

$x = x + 1$

Counting operations: (pseudocode)

Algorithm arrayMax(A, n):
    Input: An array A of $n \geq 1$ numbers
    Output: The maximum element of A

```
1  currentMax ← A[0]                      ← 1
2  for i ← 1 to n-1                       ← repeats n-1 times
3      if currentMax < A[i] then          ← 1 comparison
4          currentMax ← A[i]              ← 1 assignment
5  return currentMax                      ← 1 return
```

~~best~~: $1 + (n-1)1 + 1 = n+1$

worst: $1 + (n-1)2 + 1 = 2n$

Asymptotic Notation - Ch. 4

How important is <u>exact</u> number of computations?

In general, any primitive statement depends on a small number of low-level operations, independent of language or computer.

So we'll focus on big-picture, or how the running time grows in proportion to input size (usually n).

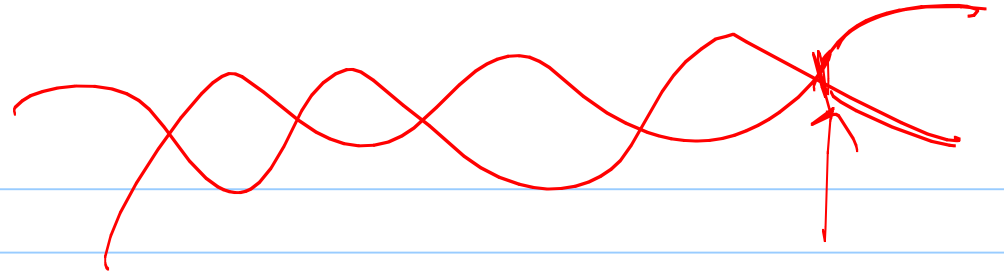
Formalize: Big-Oh notation

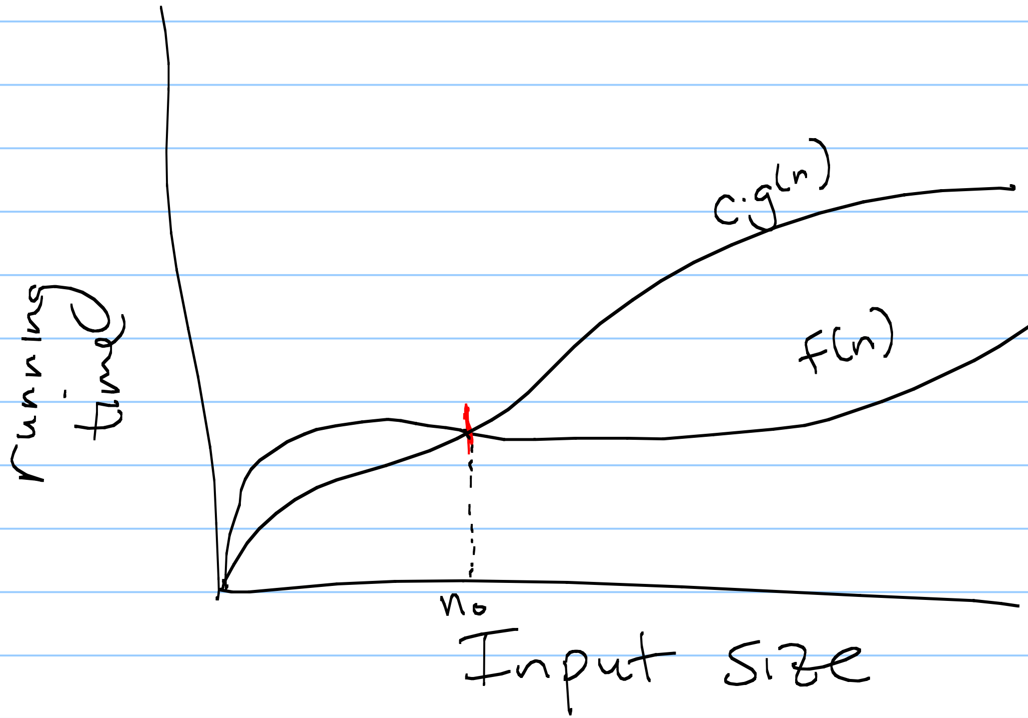
Let $f(n)$ and $g(n)$ be two functions from non-negative integers to reals.

→ We say $f(n)$ is $O(g(n))$ if there exists a constant c and integer $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

$f(n)$ is big-Oh of $g(n)$

Picture





Ex: $4n - 2$ is $O(n)$.

($f(n)$ → $4n-2$, $g(n)$ → n)

Why? Find c & n_0, s.t. $\forall n > n_0$,

$$4n - 2 \leq c \cdot n$$

Let $c = 5$, $n_0 = 0$

linear time

Ex: running time of arrayMax is $O(n)$:

Algorithm arrayMax(A, n):
Input: An array A of $n \geq 1$ numbers
Output: The maximum element of A

```
currentMax ← A[0]
for i ← 1 to n-1
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

$2n$ is $O(n)$

Ex: $20n^3 + 10n\log_2 n + 5$

is $O(n^3)$

$20n^3 + 10n\log_2 n + 5 < 35n^3$

So let $c = 35$, $n_0 = 1$

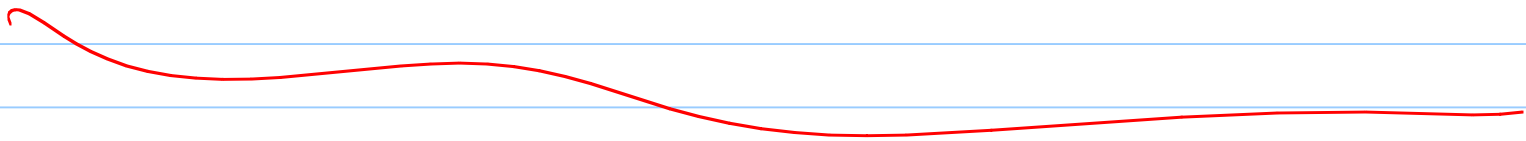
Any polynomial: $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$

highest power is what we care about

so $f(n) = O(n^k)$

Ex: 2^{100} is $O(1)$

Let $c = 2^{100} + 1$



Here: $O(1)$ – constant time

$O(n)$ – linear time
(for loops)

$O(n^2)$ – quadratic time

Ch 4 in book: Rules & Examples

For any data structure, we'll provide big-O bounds on our functions.

This is one way to compare which data structure will work best.

Useful things to remember:

- $$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b)$$

(loops often produce these!)

Example:

```
for (int i=0; i<n; i++)
    x = x+1;
```

or

```
for (int i=0; i<n; i++)
    A[i] = i;
```

$$\sum_{i=0}^{n-1} 1 = n$$

$$\sum_{i=0}^{n-1} 2 = \underbrace{2+2+\cdots+2}_{n \text{ times}} = 2n$$

Another:

for any $n \geq 1$,

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

$\sum_{j=1}^{i} 1 = i$

$\sum_{i=1}^{n} i = 1+2+3+4+ \ldots +n$

When might this come in handy?

```
for (i = 1 to n)
  for (j = 1 to i)
    x = x + 1;
```

Stacks and queues (array-based)

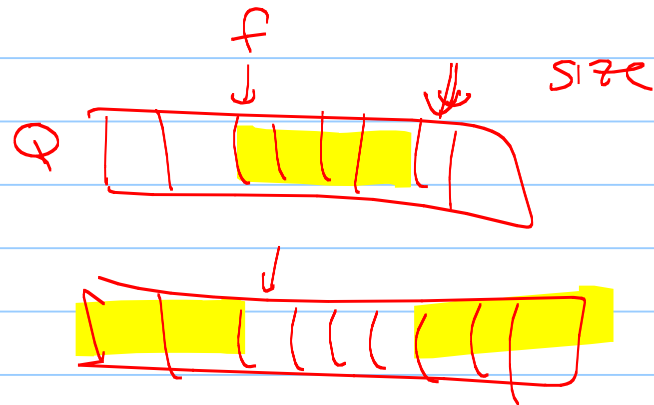
Stack

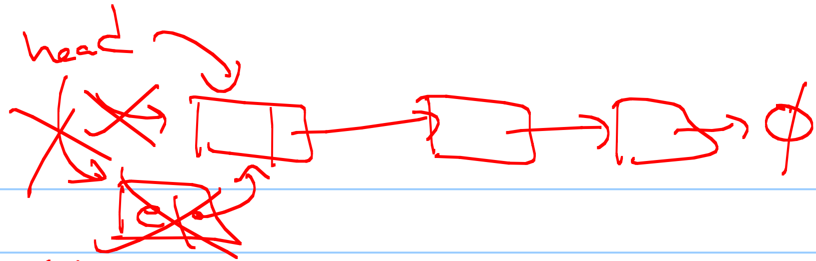
- push(e) : O(1)
- pop : O(1)
- size : O(1)
- empty : O(1)
- destructor: O(1)

} dream data structure (in terms of speed)

Queue

- push(e) : O(1)
- pop() : O(1)
- size : O(1)
- empty : O(1)





SLinked List:

insertFront(e): $O(1)$

removeFront(): $O(1)$

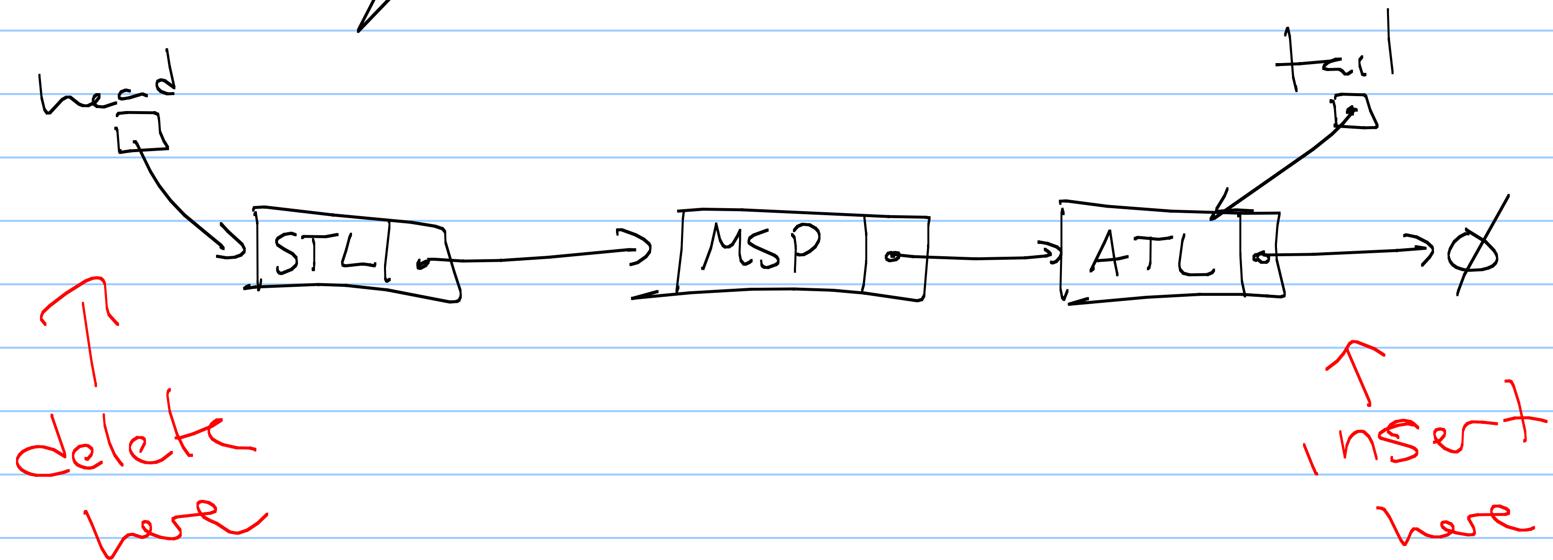
empty: $O(1)$

destructor: $O(n)$

while (!empty()) ← loop repeats n times

removeFront(); ← $O(1)$

Linked queues



not going to use SLinkedList (or DLinkedList)

```
template <typename Object>
class LinkedQueue {

    private:

        class QNode {
          private:
            Object _data;
            QNode _next;

        }

        QNode * _head;
        QNode * _tail;
        int _size;
```

more on Monday!