CS 261 – Data Structures

Big-Oh and Execution Time: A Review

Big-Oh: Purpose

- A machine-independent way to describe execution time
- Change in execution time relative to change in input size, independent of:
  - hardware used (e.g., PC, Mac, etc.)
  - the clock speed of your processor
  - what compiler you use
  - what language you use

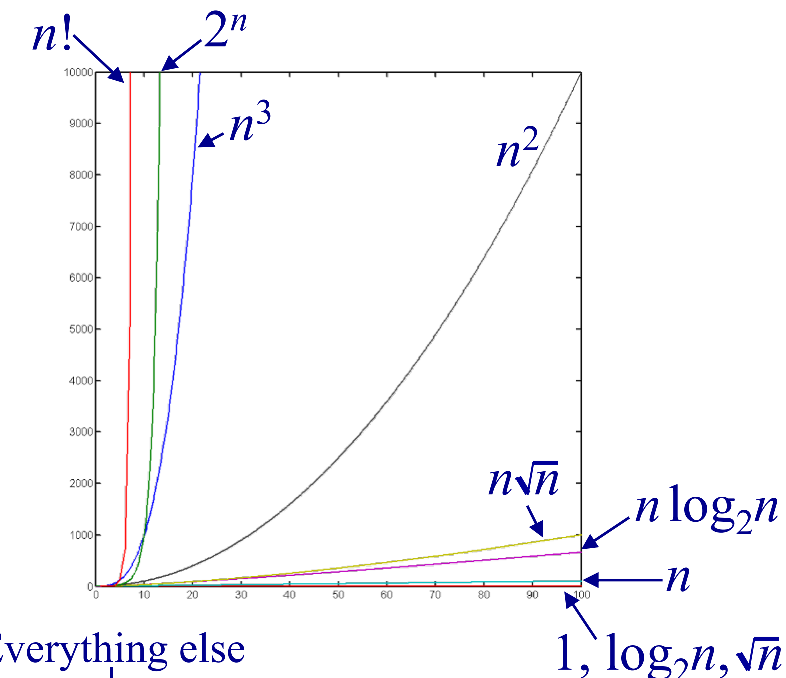
Algorithmic Analysis

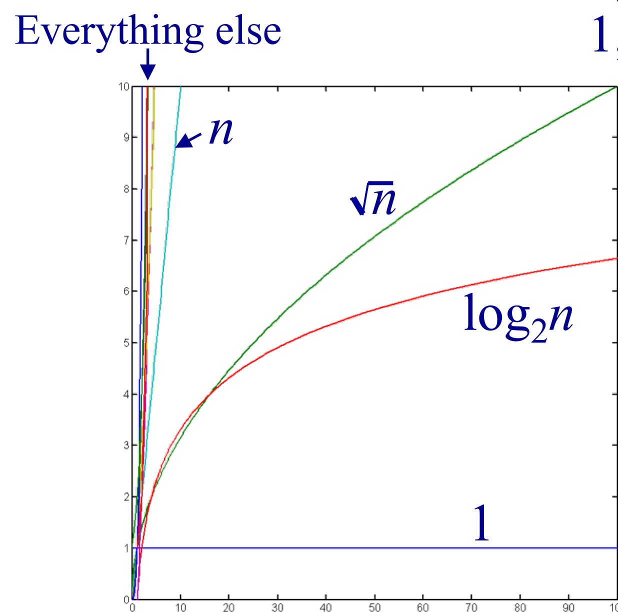
- Suppose that algorithm ***A*** processes ***n*** data elements in time ***t***.

- Algorithmic analysis attempts to estimate how ***t*** is affected by changes in ***n***, when we use ***A***.

Complexity: O(f(n))

Function	Common Name
$n!$	Factorial
2^n (or c^n)	Exponential
n^d, $d > 3$	Polynomial
n^3	Cubic
n^2	Quadratic
$n\sqrt{n}$	
$n \log n$	
n	Linear
$\sqrt{n}$	Root-n
$\log n$	Logarithmic
1	Constant





Determining Big Oh: Simple Loops

- Simple loop: constant-time operations within the loop
- Ask yourself how many times the loop executes as a function of input size.

Determining Big Oh: Simple Loops

- Simple loop: constant-time operations within the loop
- Ask yourself how many times the loop executes as a function of input size.

Example:

```
double minimum(double data[], int n) {
/* Input: data array has at least one element. */
/* Output: returns the smallest value in input array */

   int    i;
   double min = data[0];

   for(i = 1; i < n; i++)
     if(data[i] < min) min = data[i];

   return min;
}
```

O(n) (the circled for loop)

Determining Big Oh: Simple Loops

Not always simple iteration and termination criteria

- Iterations dependent on a **function** of *n*
- Possibility of early exit

Example:

```
int isPrime(int n) {
  int i;
  for(i = 2; i * i < n; i++) {        /*If i is a factor*/
    if (n % i == 0) return 0;         /*then n is not a prime*/
  }                                   /*If loop exits without finding*/
  return 1;                           /*a factor then n is a prime*/
}
```

Determining Big Oh: Simple Loops

Not always simple iteration and termination criteria

– Iterations dependent on a **function** of *n*

– Possibility of early exit

Example:

```
int isPrime(int n) {
  int i;
  for(i = 2; i * i < n; i++) {        /*If i is a factor*/
    if (n % i == 0) return 0;         /*then n is not a prime*/
  }                                   /*If loop exits without finding*/
  return 1;                           /*a factor then is a prime*/
}
```

`i * i < n;` → $O(\sqrt{n})$

Determining Big Oh: Simple Loops

Not always simple iteration and termination criteria

- Iterations dependent on a **function** of *n*
- Possibility of early exit

Example:

```
int isPrime(int n) {
  int i;
  for(i = 2; i * i < n; i++) {        /*If i is a factor*/
     if (n % i == 0) return 0;        /*then n is not a prime*/
  }                                   /*If loop exits without finding*/
  return 1;                           /*a factor then n is a prime*/
}
```

`i * i < n;` → $O(\sqrt{n})$

`return 0;` → But what happens if it exits early?

Best case?   Average case?   **Worst case?**

Determining Big Oh: Nested Loops

Nested loops (dependent or independent) multiply:

```
void insertionSort(double arr[], unsigned int n) {
     unsigned i, j;
     double   elem;

     for(i = 1; i < n; i++) { /*loop for n - 1 times*/

       elem = arr[i]; /* Memorize arr[i]*/
       for (j = i - 1; j >= 0 && elem < arr[j]; j--) {
         arr[j+1] = arr[j]; /*Slide old values up*/
       }
       arr[j+1] = elem; /*put arr[i] in place*/
     }
   }
```

Determining Big Oh: Nested Loops

Nested loops (dependent or independent) multiply:

```
void insertionSort(double arr[], unsigned int n) {
     unsigned i, j;
     double   elem;

     for(i = 1; i < n; i++) { /*loop for n - 1 times*/

       elem = arr[i]; /* Memorize arr[i]*/
       for (j = i - 1; j >= 0 && elem < arr[j]; j--) {
         arr[j+1] = arr[j]; /*Slide old values up*/
       }
       arr[j+1] = elem; /*put arr[i] in place*/
     }
}
```

Worst case (reverse order): $1 + 2 + \ldots + (n\text{-}1) = (n^2 - n) / 2 \rightarrow O(n^2)$

8	7	6	5	4	3	2	1

→

2	3	4	5	6	7	8	1

Determining Big Oh: Recursion

For recursion, ask yourself:

– How many times will the function be executed?

– How much time does it spend on each call?

– Multiply these together

Example:

```
double exp(double a, int n) {
  if (n = 0) return 1; /*Stop*/
  /* Cases of recursive calls */
  if (n < 0) return 1 / exp(a, -n);
  return a * exp(a, n - 1);
}
```

Determining Big Oh: Recursion

For recursion, ask yourself:

- How many times will the function be executed?
- How much time does it spend on each call?
- Multiply these together

Example:

```
double exp(double a, int n) {
   if (n = 0) return 1; /*Stop*/
   /* Cases of recursive calls */
   if (n < 0) return 1 / exp(a, -n);
   return a * exp(a, n - 1);
}
```

complexity:

O(n)

Determining Big Oh: Calling Functions

Big-Oh of a calling function includes the big-Oh of the called function

```
void removeElement(double elem,
                   struct Vector *v){
  int i = vectorSize(v);

  while (i-- > 0)
    if (elem == *(v+i)) {
      vectorRemove(v, i);
      return;
    }
}
```

Determining Big Oh: Calling Functions

Big-Oh of a calling function also considers the big-Oh of the called function

```
void removeElement(double elem,
                   struct Vector *v){
  int i = vectorSize(v);

  while (i-- > 0)
    if (elem == *(v+i)) {
      vectorRemove(v, i);
      return;
    }
}
```

`while (i-- > 0)` → $O(n)$

`if (elem == *(v+i)) {` → $O(1)$

`vectorRemove(v, i);` → $O(n)$ if we need to slide up all elements after e

Determining Big Oh: Calling Functions

Big-Oh of a calling function also considers the big-Oh of the called function

```
void removeElement(double elem,
                        struct Vector *v){
  int i = vectorSize(v);
  while (i-- > 0)
    if (elem == *(v+i)) {
      vectorRemove(v, i);
      return;
    }
}
```

`while (i-- > 0)` → $O(n)$

`vectorRemove(v, i);` → $O(n)$ if we need to slide up all elements after e

Whole loop → $O(n^2)$

Determining Big Oh: Logarithmic

- Example problem: how many guesses to find a particular number in a **sorted** array of numbers
  - To find a number between 0 and 1024, it takes approximately $\log 1024 = \log 2^{10} = 10$ guesses

- In algorithmic analysis, the log of *n* is the number of times you can split *n* in half (binary search, etc)

Summation and the Dominant Component

- Runtime of a sequence of statements
- The largest component dominates
- Constant multipliers are ignored

Example: $O(8n + 3n^2 + 1) = O(n^2)$

Single loop

Nested loop

Constant-time statements

Let’s Practice: What is the O(??)

```
int countReps(double data[],int n, double v) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (data[i] == v)
            count++;
    }
    return count;
}
```

What is the O(??)

```
/* Matrix multiplication */

void matMult (int a[][], int b[][], int c[][], int n) {
   /* assume all matrices have the same size n x n  */
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
         c[i][j] = 0;
         for (k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
}
```

What is the O(??)

```
void selectionSort (double storage [ ], int n) {
  for (int p = n - 1; p > 0; p--) {/*backward index*/
     int indexLargest = 0; /*index of largest elem.*/
     for (int i = 1; i <= p; i++) {/*forward index*/
        if (storage[i] > storage[indexLargest])
           indexLargest = i; /*found largest elem.*/
     }
     if (indexlargest != p)
        swap(storage, indexLargest, p);
  }
}
```

Reading & Worksheets

- http://bigocheatsheet.com

- Worksheet 9: Summing Execution Times
- Worksheet 10: Wall Clock Time Estimation