Announcements

• HW5 out this Wednesday,
  – Due on Wednesday, April 27\textsuperscript{th} before class

• Final on Monday May 9\textsuperscript{th}, from 9am to 12pm, in class
  – Same format as Midterm
Today

- Quick review of linked lists
- Binary Trees
- Complexity Analysis
Introduction to Complexity Analysis
Measuring Algorithms

• In Computer Science, we are interested in finding a function that defines the quantity of some resource consumed by a particular algorithm

• This function is often referred to as a complexity of the algorithm

• The resources we usually investigate are
  – running time
  – memory requirements
Measuring Algorithms

• We want to express complexity in the most general way possible

• Running time and space typically depend on input size

For varying input sizes, we can write time and space requirements as functions of $n$.

• Algorithms run on different machines

For varying implementation, we use a description independent from constant factors.
Example

Given an array $X$ of 10 elements of type $\text{int}$

$X \begin{bmatrix} 7 & 1 & 44 & 2 & 34 & 9 & 12 & 7 & 33 & 12 \end{bmatrix}$

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?

- How much space (SP) in memory is used by that algorithm?

```c
int X[10];
int i, sum = X[0];
for(i=1; i<10;i++){
    sum += X[i];
}
```
Example

Given an array $X$ of 10 elements of type `int`

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>1</th>
<th>44</th>
<th>2</th>
<th>34</th>
<th>9</th>
<th>12</th>
<th>7</th>
<th>33</th>
<th>12</th>
</tr>
</thead>
</table>

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?

- How much space (SP) in memory is used by that algorithm?

```
int X[10];
int i, sum = X[0];
for (i=1; i<10; i++){
    sum += X[i];
}
```

Machine 1

- Addition → 2 seconds
- int → 4 bytes

$RT = 9 \times 2 = 18$
$SP = 10 \times 4 + 2 \times 4 = 48$

Machine 2

- Addition → 3 seconds
- int → 8 bytes

$RT = 9 \times 3 = 27$
$SP = 10 \times 8 + 2 \times 8 = 96$

... Machine 2

- Addition → 2 seconds
- int → 8 bytes

$RT = 9 \times 2 = 18$
$SP = 10 \times 8 + 2 \times 8 = 96$
Example

Given an array of 10 elements of type `int`

<table>
<thead>
<tr>
<th>X</th>
<th>7</th>
<th>1</th>
<th>44</th>
<th>2</th>
<th>34</th>
<th>9</th>
<th>12</th>
<th>7</th>
<th>33</th>
<th>12</th>
</tr>
</thead>
</table>

Complexity analysis

• What is the running time (RT) of an algorithm that sums the elements in the array?
  
  \[ \text{RT} = 9 \times 2 = 18 \]

• How much space (SP) in memory is used by that algorithm?
  
  \[ \text{SP} = 10 \times 4 + 2 \times 4 = 48 \]

\[ \text{RT} = 9 \times 3 = 27 \]

\[ \text{SP} = 10 \times 8 + 2 \times 8 = 96 \]

\[ \text{RT} = 9 \times 2 = 18 \]

\[ \text{SP} = 10 \times 8 + 2 \times 8 = 96 \]

This is not general!
Performance of machines, not of algorithm!

What if array has \( n \) elements?

We want to express complexity of algorithm in terms of

- \( n \) : number of elements in array (variable)
- \( c \) : number of seconds to execute addition (constant)
- \( b \) : number of bytes to store elements (constant)
Example

Given an array of 10 elements of type int

| X | 7 | 1 | 44 | 2 | 34 | 9 | 12 | 7 | 33 | 12 |

Complexity analysis

• What is the running time (RT) of an algorithm that sums the elements in the array?

• How much space (SP) in memory is used by that algorithm?

We want to express complexity of algorithm in terms of

— n: number of elements in array (variable)
— c: number of seconds to execute addition (constant)
— b: number of bytes to store elements (constant)

$$RT = c(n-1)$$

$$SP = b(n+2)$$

```c
int X[10];
int i, sum = X[0];
for (i=1; i<10; i++){
    sum += X[i];
}
```
Big – O Notation

GOAL: estimate the order of the function $f(n)$ that represents RT or SP in terms of $n$

\[ f(n) = O(g(n)) \]

\[ f(n) = O(g(n)) \quad n \to \infty \]

\[ \iff \]

\[ \exists C > 0 \text{ and } n_0 : \]

\[ |f(n)| \leq C|g(n)| \quad \forall n > n_0 \]
Big – O Notation

GOAL: estimate the order of the function \( f(n) \) that represents RT or SP in terms of \( n \)

\[
f(n) = O(g(n))_{n \to \infty}
\]

\[
\Leftrightarrow
\]

\[
\exists C > 0 \text{ and } n_0 : 
\]

\[
|f(n)| \leq C|g(n)| \quad \forall n > n_0
\]

f(n) equals oh of g(n) as \( n \) tends to infinity

if and only if

there exists a positive constant \( C \) and a value \( n_0 \) such that

for all \( n \) greater than \( n_0 \), the absolute value of \( f(n) \) is smaller than \( C \) times the absolute value of \( g(n) \)
GOAL: estimate the order of the function $f(n)$ that represents RT or SP in terms of $n$

$$f(n) = O(g(n))$$

$$f(n) = O(g(n))$$

$$\iff$$

$$\exists C > 0 \text{ and } n_0 :$$

$$|f(n)| \leq C|g(n)| \quad \forall n > n_0$$

In other words, big-O means less than some constant scaling
 When analyzing complexity with big-O notation, we always consider the WORST CASE SCENARIO
Big-O notation: Examples

- \( f(n) = 3n^4 + 7n^2 - 5n + 8 \)
  \[
  |3n^4 + 7n^2 - 5n + 8| \leq 3n^4 + 7n^2 + |5n| + 8 \\
  \leq 3n^4 + 7n^4 + 5n^4 + 8n^4 \\
  \leq 23n^4 \\
  |f(n)| \leq C|g(n)| \\
  \]

- \( f(n) = O(n^4) \)

- What is the running time (RT) of an algorithm that sums \( n \) elements in an array?

- \( C(n-1) = O(n-1) = O(n) \)
Big – O: common cases

- The algorithm requires the same fixed number of steps regardless of the size of the task
- **Example**: insert an element in front of a linked list

```c
int addNodeFront(int val, node *head)
{
    1) node *newNode = malloc(sizeof(node));
    2) newNode->value = val;
    3) newNode->next = head->next;
    4) head->next = newNode;
}
```

No matter how long the list is, this operation always requires 4 steps
$O(4) = O(1)$
Big – O : common cases

- The algorithm requires the same fixed number of steps regardless of the size of the task.
- **Example**: insert an element in front of a linked list

```c
int addNodeFront(int val, node *head)
{
    1) node *newNode = malloc(sizeof(node));
    2) newNode->value = val;
    3) newNode->next = head->next;
    4) head->next = newNode;
}
```

No matter how long the list is, this operation always requires 4 steps:

RT = O(4) = O(1)
Big – O : common cases

- The algorithm requires a number of steps proportional to the size of the task

- **Examples:**
  - Travers a linked list or an array with $n$ elements;
  - Find the maximum and minimum element in a list or array

```c
for(i=0 ; i < n; i++){
    if(arr[i] < minVal)
        minVal = arr[i];
    if(arr[i] > maxVal)
        maxVal = arr[i];
}
```

- RT = O(2n) = O(n)
- SP = O(n+2) = O(n)
Big – O : common cases

- The number of operations is proportional to the size of the task squared.

**Example**: Finding duplicates in an unsorted list of \( n \) elements

```c
for(i=0 ; i < n; i++){
    for(j=0 ; j < n; j++){
        if( (i!=j) && arr[i] == arr[j] )
            dup[i][j] = 1;
    }
}
```

\[
RT = O(4n^2+n) = O(n^2)
\]

- Increment \( i \) \( n \) times
- Increment \( j \) \( n^2 \) times
- Check \( i\neq j \) \( n^2 \) times
- Check \( arr[i]==arr[j] \) \( n^2 \) times
- Set \( dup[i][j]=1 \) \( (n-1)*(n-1) \) times
Big – O : common cases

O(log(n)) - logarithmic time

• **Example**: Find operation in a balanced binary tree with \( n \) nodes

\[
\text{height of tree} = 3 = \left\lfloor \log_2(n) \right\rfloor
\]

\[
RT = \log_2(n) + 1 = O(\log_2(n))
\]

\[n = 15\]

RT = \log_2(15) + 1 = O(\log_2(15))
Big – O: common cases

O(n log(n)) – “n log(n)” time

• Examples: sorting algorithms (will see in next class)
  – quicksort
  – mergesort
Big – O : common cases

\[ O(a^n) – \text{exponential time} \quad a > 1 \]

- **Example**: Recursive Fibonacci implementation

```c
int fib(int n) {
    switch(n) {
    case 0:
        return(0);
    case 1:
        return(1);
    default:
        return(fib(n-1) + fib(n-2));
    }
}
```

How many times is `fib()` called?

Cost of `fib()` without return statement = 2 = \( O(1) \)

\[
RT(n) = RT(n-1) + RT(n-2) + O(1)
\]

\[
RT = O(a^n)
\]

\[
a^n = a^{n-1} + a^{n-2}
\]

\[
a^2 = a + 1
\]

\[
a = \frac{1 + \sqrt{5}}{2} \approx 1.6
\]
Big–O : Relationship among common cases

\[
O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)
\]

**Example**: big-O when a function is the *sum of several statements*

```c
int i=0;
for(i=0 ; i < n; i++){
    for(j=0 ; j < n; j++){
        if( (i!=j) && arr[i] == arr[j] )
            increment i
            increment j
            check i!=j
            check arr[i]==arr[j]
            dup[i][j] = 1;
    }
}
```

\[
RT = O(4n^2+n) = O(n^2)
\]

Longest operation dominates (worst case)