CSCI 104
Runtime Complexity

Mark Redekopp
David Kempe
Runtime

• It is hard to compare the run time of an algorithm on actual hardware
  – Time may vary based on speed of the HW, etc.
    • The same program may take 1 sec. on your laptop but 0.5 second on a high performance server
• If we want to compare 2 algorithms that perform the same task we could try to count operations (regardless of how fast the operation can execute on given hardware)...
  – But what is an operation?
  – How many operations is: i++ ?
  – i++ actually requires grabbing the value of i from memory and bringing it to the processor, then adding 1, then putting it back in memory. Should that be 3 operations or 1?
  – Its painful to count 'exact' numbers operations
• Big-O, Big-Ω, and Θ notation allows us to be more general (or "sloppy" as you may prefer)
Complexity Analysis

• To find upper or lower bounds on the complexity, we must consider the set of all possible inputs, I, of size, n
• Derive an expression, T(n), in terms of the input size, n, for the number of operations/steps that are required to solve the problem of a given input, i
  – Some algorithms depend on i and n
    • Find(3) in the list shown vs. Find(2)
  – Others just depend on n
    • Push_back / Append
• Which inputs though?
  – Best, worst, or "typical/average" case?
• We will always apply it to the "worst case"
  – That's usually what people care about

Note: Running time is not just based on an algorithm, BUT algorithm + input data
Big-O, Big-Ω

- \( T(n) \) is said to be \( O(f(n)) \) if...
  - \( T(n) < a*f(n) \) for \( n > n_0 \) (where \( a \) and \( n_0 \) are constants)
  - Essentially an upper-bound
  - We'll focus on big-O for the worst case

- \( T(n) \) is said to be \( Ω(f(n)) \) if...
  - \( T(n) > a*f(n) \) for \( n > n_0 \) (where \( a \) and \( n_0 \) are constants)
  - Essentially a lower-bound

- \( T(n) \) is said to be \( Θ(f(n)) \) if...
  - \( T(n) \) is both \( O(f(n)) \) AND \( Ω(f(n)) \)
Worst Case and Big-Ω

• What's the lower bound on `List::find(val)`
  – Is it $\Omega(1)$ since we might find the given value on the first element?
  – Well it could be if we are finding a lower bound on the 'best case'

• Big-Ω does **NOT** have to be synonymous with 'best case'
  – Though many times it mistakenly is

• You can have:
  – Big-O for the best, average, worst cases
  – Big-Ω for the best, average, worst cases
  – Big-Θ for the best, average, worst cases
Worst Case and Big-$\Omega$

- The key idea is an algorithm may perform differently for different input cases
  - Imagine an algorithm that processes an array of size $n$ but depends on what data is in the array
- Big-$O$ for the worst-case says **ALL** possible inputs are bound by $O(f(n))$
  - Every possible combination of data is at MOST bound by $O(f(n))$
- Big-$\Omega$ for the worst-case is attempting to establish a lower bound (at-least) for the worst case (the worst case is just one of the possible input scenarios)
  - If we look at the first data combination in the array and it takes $n$ steps then we can say the algorithm is $\Omega(n)$.
  - Now we look at the next data combination in the array and the algorithm takes $n^{1.5}$. We can now say worst case is $\Omega(n^{1.5})$.
- To arrive at $\Omega(f(n))$ for the worst-case requires you simply to find **AN** input case (i.e. the worst case) that requires at least $f(n)$ steps
Deriving T(n)

• Derive an expression, T(n), in terms of the input size for the number of operations/steps that are required to solve a problem
• If is true => 4
• Else if is true => 5
• Worst case => T(n) = 5

```cpp
#include <iostream>

using namespace std;

int main()
{
    int i = 0;
    x = 5;

    if(i < x)
    {
        x--;  1
    }
    else if(i > x)
    {
        x += 2;  1
    }
    return 0;
    1

    1
    1
    1
    1
    1
    1
    1
```
Deriving $T(n)$

- Since loops repeat you have to take the sum of the steps that get executed over all iterations

- $T(n) =$

  $= \sum_{i=0}^{n-1} 5 = 5 \times n$

- Or you can setup a relationship like:

- $T(n) = T(n - 1) + 5$
  $= T(n - 2) + 5 + 5$
  $= \sum_{i=0}^{n-1} 5 = 5 \times n$
  $= \sum_{i=0}^{n-1} O(1) = O(n)$

```cpp
#include <iostream>
using namespace std;

int main()
{
    for(int i=0; i < N; i++){
        x = 5;
        if(i < x){
            x--;
        }
        else if(i > x){
            x += 2;
        }
    }
    return 0;
}
```
Common Summations

- \[ \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2) \]
  - This is called the arithmetic series

- \[ \sum_{i=1}^{n} \theta(i^p) = \theta(n^{p+1}) \]
  - This is a general form of the arithmetic series

- \[ \sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1} = \theta(c^n) \]
  - This is called the geometric series

- \[ \sum_{i=1}^{n} \frac{1}{i} = \theta(\log n) \]
  - This is called the harmonic series
Skills You Should Gain

• To solve these running time problems try to break the problem into 2 parts:
  • FIRST, setup the expression (or recurrence relationship) for the number of operations
  • SECOND, solve
    – Unwind the recurrence relationship
    – Develop a series summation
    – Solve the series summation
Loops

• Derive an expression, \( T(n) \), in terms of the input size for the number of operations/steps that are required to solve a problem

\[
T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \theta(1) = \sum_{i=0}^{n-1} \theta(n) = \Theta(n^2)
\]

```cpp
#include <iostream>

using namespace std;

const int n = 256;

unsigned char image[n][n]

int main()
{
    for(int i=0; i < n; i++){
        for(int j=0; j < n; j++){
            image[i][j] = 0;
        }
    }
    return 0;
}
```
Matrix Multiply

- Derive an expression, \( T(n) \), in terms of the input size for the number of operations/steps that are required to solve a problem
- \( T(n) = \)

\[
= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \theta(1) = \theta(n^3)
\]

```
#include <iostream>
using namespace std;
const int n = 256;
int a[n][n], b[n][n], c[n][n];
int main()
{
    for(int i=0; i < n; i++){
        for(int j=0; j < n; j++){
            c[i][j] = 0;
            for(int k=0; k < n; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
    return 0;
}
```
Sequential Loops

- Is this also $n^3$?
- No!
  - 3 for loops, but not nested
  - $O(n) + O(n) + O(n) = 3*O(n) = O(n)$

```cpp
#include <iostream>

using namespace std;

const int n = 256;
unsigned char image[n][n]
int main()
{
    for(int i=0; i < n; i++)
    {
        image[0][i] = 5;
    }
    for(int j=0; j < n; j++)
    {
        image[1][j] = 5;
    }
    for(int k=0; k < n; k++)
    {
        image[2][k] = 5;
    }
    return 0;
}
```
Counting Steps

• It may seem like you can just look for nested loops and then raise n to that power
  – 2 nested for loops => $O(n^2)$

• But be careful!!

• You have to count steps
  – Look at the update statement
  – Outer loop increments by 1 each time so it will iterate N times
  – Inner loop updates by dividing x in half each iteration?
  – After 1st iteration => $x=n/2$
  – After 2nd iteration => $x=n/4$
  – After 3rd iteration => $x=n/8$
  – Say $k^{th}$ iteration is last => $x = n/2^k = 1$
  – Solve for $k$
  – $k = \log_2(n)$ iterations
  – $O(n\log(n))$

```cpp
#include <iostream>
using namespace std;

const int n = 256;

int main()
{
    for(int i=0; i < n; i++){
        int y=0;
        for(int x=n; x != 1; x=x/2){
            y++;
        }
        cout << y << endl;
    }
    return 0;
}
```
Analyze This

- Count the steps of this example?

- \( T(n) = T(n-1) + n-1 \)
- \( 0 + 1 + ... + n-2 + n-1 \)
- \( (n-1)n/2 \)

```cpp
#include <iostream>
using namespace std;
const int n = 256;
int a[n];
int main()
{
    for(int i=0; i < n; i++){
        a[i] = 0;
        for(int j=0; j < i; j++){
            a[i] += j;
        }
    }
    return 0;
}
```
Analyze This

• Count the steps of this example?

\[ \sum_{i=0}^{\lfloor \lg(n) \rfloor} \sum_{j=0}^{2^i} 1 \]
\[ = \sum_{i=0}^{\lfloor \lg(n) \rfloor} 2^i \]

• Use the geometric sum eqn.

\[ = \sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a} \]

• So our answer is...

\[ \frac{1-2^{\lfloor \lg(n) \rfloor+1}}{1-2} = \frac{1-2*2^n}{-1} = O(n) \]

```cpp
for (int i = 0; i <= log2(n); i ++)
    for (int j=0; j < (int) pow(2,i); j++)
        cout << j;
```
Another Example

- Count steps here...
  - Think about how many times if statement will evaluate true

- \( T(n) = \sum_{i=0}^{n-1}(\theta(1) + O(n)) \)
- \( T(n) = \)

```c++
for (int i = 0; i < n; i++)
{
    cout << "i: " ;
    int m = sqrt(n);
    if ( i % m == 0 ){
        for (int j=0; j < n; j++)
            cout << j << " " ;
    }
    cout << endl;
}
```
Another Example

- Count steps here...
  - Think about how many times if statement will evaluate true

- \( T(n) = \sum_{i=0}^{n-1} (\theta(1) + O(n)) \)
- \( T(n) = \sum_{i=0}^{n-1} \theta(1) + \sum_{k=1}^{\sqrt{n}} \sum_{j=1}^{n} \theta(1) \)
- \( T(n) = \theta(n) + \sum_{k=1}^{\sqrt{n}} \theta(n) \)
- \( T(n) = \theta(n) + \theta(n \cdot \sqrt{n}) \)
- \( T(n) = \theta(n^{3/2}) \)

```cpp
for (int i = 0; i < n; i++)
{
    cout << "i: " ;
    int m = sqrt(n);
    if( i % m == 0){
        for (int j=0; j < n; j++)
            cout << j << " ";
    }
    cout << endl;
}
```
What about Recursion

• Assume N items in the linked list

• $T(n) = 1 + T(n-1)$

• $= 1 + 1 + T(n-2)$

• $= 1 + 1 + 1 + T(n-3)$

• $= n = O(n)$

```cpp
void print(Item* head)
{
    if(head==NULL) return;
    else {
        cout << head->val << endl;
        print(head->next);
    }
}
```
Binary Search

• Assume N items in the data array
• \( T(n) = \)
  – \( O(1) \) if base case
  – \( O(1) + T(n/2) \)
• \( = 1 + T(n/2) \)
• \( = 1 + 1 + T(n/4) \)
• \( = k + T(n/2^k) \)
• Stop when \( 2^k = n \)
  – Implies \( \log_2(n) \) recursions
• \( O(\log_2(n)) \)

```c
int bsearch(int data[],
            int start, int end,
            int target)
{
    if(end >= start)
        return -1;
    int mid = (start+end)/2;
    if(target == data[mid])
        return mid;
    else if(target < data[mid])
        return bsearch(data, start, mid, target);
    else
        return bsearch(data, mid, end, target);
}
```
AMORTIZED RUNTIME
Example

• You love going to Disneyland. You purchase an annual pass for $240. You visit Disneyland once a month for a year. Each time you go you spend $20 on food, etc.
  – What is the cost of a visit?

• Your annual pass cost is spread or "amortized" (or averaged) over the duration of its usefulness

• Often times an operation on a data structure will have similar "irregular" costs that we can then amortize over future calls
Amortized Array Resize Run-time

- What is the run-time of insert or push_back:
  - If we have to resize?
  - $O(n)$
  - If we don't have to resize?
  - $O(1)$

- Now compute the total cost of a series of insertions using resize by 1 at a time

- Each insert now costs $O(n)$... not good
Amortized Array Resize Run-time

- What if we resize by adding 5 new locations each time
- Start analyzing when the list is full...
  - 1 call to insert will cost: 5
  - What can I guarantee about the next 4 calls to insert?
    - They will cost 1 each because I have room
  - After those 4 calls the next insert will cost: 10
  - Then 4 more at cost=1
- If the list is size n and full
  - Next insert cost = n
  - 4 inserts after than = 1 each
  - Cost for 5 inserts = n+5
  - Runtime = cost / insert = (n+5)/5 = O(n)
Consider a Doubling Size Strategy

• Start when the list is full and at size n
• Next insertion will cost?
  – O(n+1)
• How many future insertions will be guaranteed to be cost = 1?
  – n-1 insertions
  – At a cost of 1 each, I get n-1 total cost
• So for the n insertions my total cost was
  – n+1 + n-1 = 2*n
• Amortized runtime is then:
  – Cost / insertions
  – O(2*n / n) = O(2)
    = O(1) = constant!!!
Another Example

• Let's say you are writing an algorithm to take a $n$-bit binary combination (3-bit and 4-bit combinations are to the right) and produce the next binary combination

• Assume all the cost in the algorithm is spent changing a bit (define that as 1 unit of work)

• I could give you any combination, what is the worst case run-time? Best-case?
  – $O(n)$ => 011 to 100
  – $O(1)$ => 000 to 001
Another Example

• Now let's consider the program that generates all the combinations sequentially (in order)
  – Starting at 000 => 001 : cost = 1
  – Starting at 001 => 010 : cost = 2
  – Starting at 010 => 011 : cost = 1
  – Starting at 011 => 100 : cost = 3
  – Starting at 100 => 101 : cost = 1
  – Starting at 101 => 110 : cost = 2
  – Starting at 111 => 000 : cost = 3
  – Total = 14 / 8 calls = 1.75

• Repeat for the 4-bit
  – 1 + 2 + 1 + 3 + 1 + 2 + 1 + 4 + ...
  – Total = 30 / 16 = 1.875

• As n gets larger...Amortized cost per call = 2
Importance of Complexity

<table>
<thead>
<tr>
<th>N</th>
<th>O(1)</th>
<th>O(log₂n)</th>
<th>O(n)</th>
<th>O(n*log₂n)</th>
<th>O(n²)</th>
<th>O(2ⁿ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>4.3</td>
<td>20</td>
<td>86.4</td>
<td>400</td>
<td>1,048,576</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>7.6</td>
<td>200</td>
<td>1,528.8</td>
<td>40,000</td>
<td>1.60694E+60</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>11.0</td>
<td>2000</td>
<td>21,931.6</td>
<td>4,000,000</td>
<td>#NUM!</td>
</tr>
</tbody>
</table>