USC Viterbi School of Engineering

CSCI 104
Runtime Complexity

Mark Redekopp
David Kempe
Sandra Batista

Revised: 01/2022

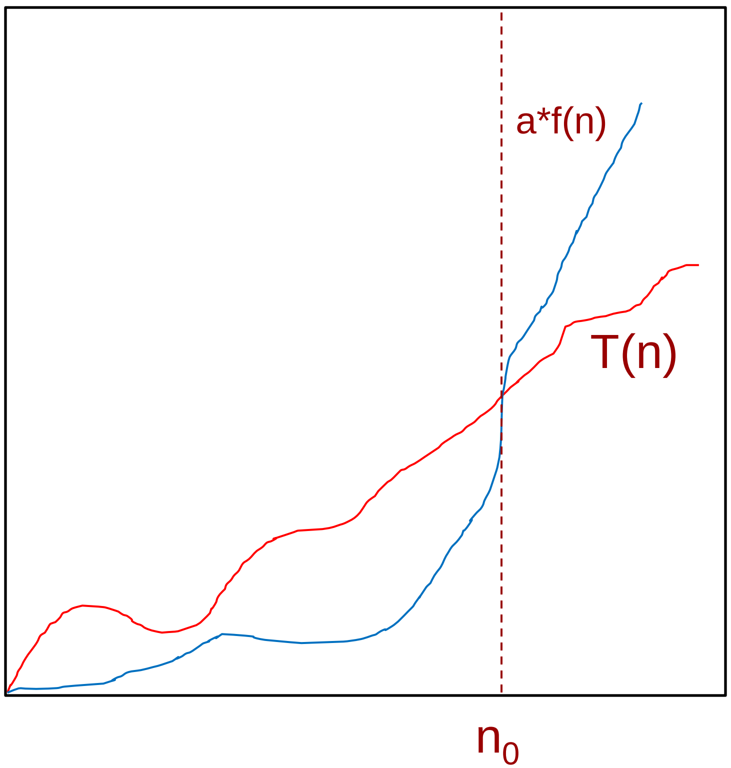
REVIEW FROM CS 170

Steps for Deriving T(n)

- Considering an input of size n that requires the maximum runtime, go through each line of the algorithm or code
- Assume elementary operations such as incrementing a variable occur in constant time
- If sequential blocks of code have runtime T1(n) and T2(n) respectively, then their total runtime will be their sum T1(n)+T2(n)
- When we encounter loops, sum the runtime for each iteration, i, of the loop, $T_i(n)$, to get the total runtime for the loop.
  - Nested loops often lead to summations of summations, etc.

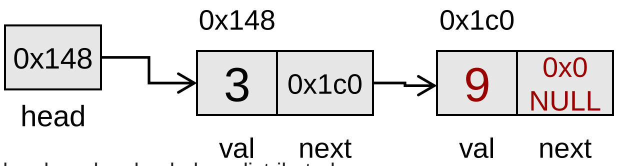
Asymptotic Notation

- 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))



Data Dependent or Not [T(n) or T(n,i)]

- One of the first questions you should ask yourself when starting your analysis is, "Is this code's runtime data-dependent or not (depending on the particular values of the data as opposed to just how many values exist (i.e. n))
- **Example 1**: Finding the size of a linked list (**does** / **does not**) depend on the data in the linked list?
  - Does NOT: We must walk all n items regardless of their value. Thus, the runtime is just a function of n, **T(n)**.
- **Example 2**: Finding if an element exists in the linked lists (**does** / **does not**) depend on the data in the linked list
  - Does: How many items we walk depends on the data values in the list and the data value we are finding. Thus, the runtime is a function of n and the input values, i => **T(n,i)**



Worst Case and Big-Ω

- What's the lower bound on List::find(val)
  - Is it Ω(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 is ***NOT synonymous*** with 'best case'
  - Though many times it mistakenly is assumed as such
- 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
- Note:
  - Big-O and Big-Ω analyses are **ONLY** necessary when the runtime of the algorithm is **data-dependent** (i.e. function of input size (n) AND values (i) => T(n,i)).
  - If the code is **NOT data-dependent** then your analysis is valid for any input and thus is already a tight bound (big- Θ)

Worst Case and Big-Ω

- 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*** means ***REGARDLESS of*** possible inputs the runtime is bound (at-most) by O(f(n))
- Big-Ω 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 Ω(f(n)) for the ***worst-case*** requires you simply try to find ***AN*** input case (i.e. the worst case) that requires ***at least*** f(n) steps
- Cost analogy…

```
int i; j;
for(i=0; i < n; i++){
  if(a[i][0] == 0){
    for(j=0; j<n; j++)
    {
     a[i][j] = i*j;
    }
  }
}
```

Consider the effect of the 'if' statement. Can it be true for each value of i? If we don't want to (or can't) determine this, we can assume it will be true and say that the upper bound for the runtime is $O(n^2)$. To prove it is $\Theta(n^2)$ we'd need to prove there is a possible input matrix that makes the 'if' true on each iteration (i.e. $\Omega(n^2)$).

Helpful 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
- $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1} = \theta(c^n)$
  - This is called the geometric series
- $\sum_{i=1}^{n} \frac{1}{i} = \theta(\log n)$
  - This is called the harmonic series

Runtime Practice #1

- 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!!
- Find T(n) for this example

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

Hint: Geometric series

- $\sum_{i=0}^{\_\_\_\_} \sum_{j=0}^{\_\_\_\_} \theta(1)$
- =
- $=\sum_{i=0}^{n-1} a^i = \frac{a^n - 1}{a-1}$
- So our answer is…

Runtime Practice #2

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

```
for(int i=0; i < n; i++){
   if (a[i] == 0){
     for (int j = 0; j < i; j++){
          a[i] = i*j;
     }
   }
}
```

Hint: Arithmetic series

- $T(n) =$ ________________________ May start with big-O if we aren't sure how many times the if statement will execute just to get a handle on the upper bound of the worst case. But to get a tight bound, we will need to think carefully and determine how many times it really executes
- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1)\right) + \sum_{i}\left(\theta(i)\right)$ Distribute to deal with 'if' separately. Not sure which values of i will trigger the for loop that incurs i steps
  - In the worst case, how many times can the 'if' statement be true? ________________
- $T(n) =$

Runtime Practice #3

- $T(n) =$

```
for(int i=0; i < n; i++){
   if (i == 0){
      for (int j = 0; j < n; j++){
         a[i][j] = i*j;
      }
   }
}
```

- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1) + O\left(\sum_{j=0}^{n-1}\theta(1)\right)\right)$ Use big-O to start if we are unsure of how many times if statement executes
  - Important: How many times will the 'if' statement be true?
- $T(n) = \sum_{i=0}^{n-1}(\theta(1)) + \sum_{i}\sum_{j=0}^{n-1}\theta(1)$
  - The 'if' statement only triggers once! So the inner loop executes only once
- $T(n) =$

Runtime Practice #4

You must use your analytical skills to determine how many times the 'if' will trigger and then sum the inner operations that many times.

```
for (int i = 1; i <= n; i++)
{  int m = sqrt(n);
   if( i % m == 0){
     for (int j=0; j < n; j++)
       cout << j << " ";
   }
   cout << endl;
}
```

- $T(n) = \sum_{i=1}^{n} \left(\theta(1) + O\left(\sum_{j=0}^{n-1} \theta(1)\right)\right)$
  - big-O indicates we have not considered the 'if' statement but are setting an upper bound
- $T(n) = \sum_{i=1}^{n} \theta(1) + \sum_{i} \sum_{j=0}^{n-1} \theta(1)$ but we need to user our own analysis skills to find the actual values of i that will cause the 'if' to be true?
  - Use some actual values of n (e.g. n=9 or 16). Write out a table to find the pattern.
  - If n=9, the 'if' will trigger ___ times for i = _________________
  - If n=16, the 'if' will trigger ___ times for i = ________________
  - The dummy variable of a summation must increment ____ at a time
  - Thus, make a table with some dummy variable (k) that increments 1 at a time and find a relationship to the actual variable, i, for when the if statement will trigger.
  - Solve for upper bound of k
    - Stop when i = __, but i = ____ so we stop when ____________thus solve for k to find that the upper-bound for k = ______

k	1	2	3	…	Arbitrary k	Stop when k =??
i				…	i = _______	Stop when i = _______

- $T(n) =$

Key Skill

- The dummy variable (say k) of a summation runs from 1 to an UPPER_BOUND incrementing 1 at a time
- Often our code performs work at some other interval such as $i = \{1\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}\ \ldots\}$ (or actual values that are not incrementing by 1 at a time)
- You must use your own analytical abilities to find a relationship that converts the dummy variable (k=1,2,3,…) to the actual values [eg. i = f(k) = $k\sqrt{n}$], usually by making a table of the dummy variable (k) and the actual code values/variables (i)

k	1	2	3	…	Arbitrary k	Stop when k =??
i=f(k)				…	$i =$ ________	Stop when i = ________

- Then use that relationship to find the UPPER_BOUND of the dummy variable
  - In the previous example, we stopped when i = n, thus we would stop when our dummy variable is $\sqrt{n}$. This then is the upper bound.
- The key skill is to relate the dummy variable to the actual variable values and then find the UPPER BOUND of the dummy variable

Runtime Practice #6

- 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=____
  - After 2nd iteration => x=____
  - After 3rd iteration => x=____
  - Say kth iteration is last => x = ______ = 1
  - Solve for k
  - k = __________ iterations
  - θ(_____________)

```
#include <iostream>
using namespace std;
const int n = /* Some constant */;

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;
}
```

PRE-SUMMER 2021 SLIDES

Motivation

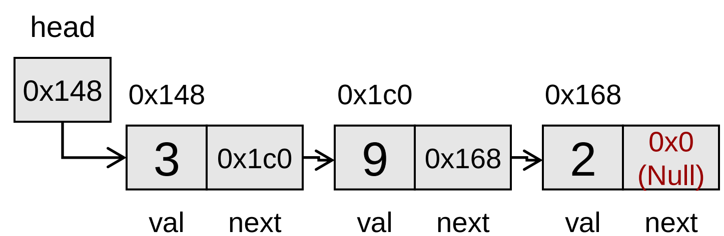
- You are given a large data set with n = 500,000 genetic markers for 5000 patients and you want to examine that data for genetic markers that maybe correlated to a disease that the patients have.
- You are given two algorithms, Algorithm A and Algorithm B, to solve this problem. You are given the implementation, code, and description of each algorithm.
- You need a solution as soon as possible to give medical professionals more data to advise patients and apply for grants for more funding.
- How would you determine which algorithm runs faster?

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 of an algorithm is not just based on input size (n), BUT input size (n) and its value (i)

Time Complexity Analysis

- Case Analysis is when you determine which input must be used to define the runtime function, T(n), for inputs of size n

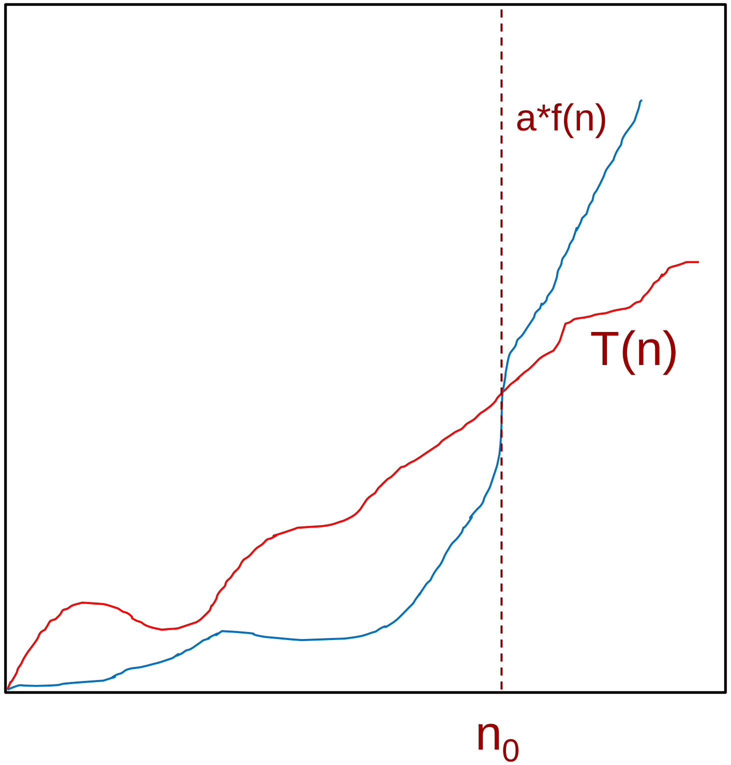
- **Best-case analysis**: Find the input of size n that takes the **minimum** amount of time.
- **Average-case analysis**: Find the runtime for all inputs of size n and take the average of all of the runtimes. (This assumes a distribution over the inputs, but uniform is a reasonable choice.)
- **Worst-case analysis**: Find the input, i, of size n that takes the **maximum** amount of time.
- Our focus will be on worst-case analysis, but for many examples, the runtime is the same on any input of size n. Please consider this as we study them.

Steps for Performing Runtime Analysis of Algorithms

- We perform **worst-case analysis** in determining the runtime function on inputs of size n, T(n).
- To do so, we need to find at least one input of size n that will require the **maximum** runtime of the algorithm.
  - In many of the examples we will examine, the algorithm will take the same amount of running time on any input (i.e. only depend on n)
- Using that input, express the runtime of the algorithm (on that input case) as a function of n, T(n).
  - This is done by **stepping through the code and counting the steps** that will be done.
- Once we have a function for the runtime, T(n), we apply asymptotic notation to that function in order to find the order of growth of the runtime function, T(n).

Asymptotic Notation

- 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 Ω(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

- Note:
  - Big-O and Big-Ω analysis are **ONLY** necessary when the runtime of the algorithm is **data-dependent** (i.e. function of inputs / T(n,i)).
  - If the code is **NOT data-dependent** then your analysis is valid for any input and thus is already a tight bound (big- Θ)

Worst Case and Big-Ω

- 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 for ***REGARDLESS of*** possible inputs the runtime is bound (at-most) by O(f(n))
- Big-Ω 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
- Cost analogy…

```
int i; j;
for(i=0; i < n; i++){
  if(a[i][0] == 0){
    for(j=0; j<n; j++)
    {
     a[i][j] = i*j;
    }
  }
}
```

Consider the effect of the 'if' statement. Can it be true for each value of i? If we don't want to (or can't) determine this we can assume it will be true and say that the upper bound for the runtime is $O(n^2)$. To prove it is $\Theta(n^2)$ we'd need to prove there is a set of inputs for the a matrix that makes the 'if' true on each iteration (i.e. $\Omega(n^2)$).

Steps for Deriving T(n)

- Considering an input of size n that requires the maximum runtime, go through each line of the algorithm or code
- Assume elementary operations such as incrementing a variable occur in constant time
- If sequential blocks of code have runtime T1(n) and T2(n) respectively, then their total runtime will be their sum T1(n)+T2(n)
- When we encounter loops, sum the runtime for each iteration of the loop, Ti(n), to get the total runtime for the loop.
  - Nested loops often lead to summations of summations, etc.

Helpful 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
- $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1} = \theta(c^n)$
  - This is called the geometric series
- $\sum_{i=1}^{n} \frac{1}{i} = \theta(\log n)$
  - This is called the harmonic series

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 "steps"
- Else if is true => 5 "steps"
- Worst case => T(n) = $\theta(1)$

```
#include <iostream>

using namespace std;


int main(int argc, char* argv[])
{

  int i = argc;        1

  int x = 5;           1

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

Deriving T(n)

- Since loops repeat you have to take the sum of the steps that get executed over all iterations
- $T(n) =$
- $= \sum_{i=0}^{n-1} 4 = 4 + 4 + \cdots 4 = 4 * n$
  $= \theta(n)$

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

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

This code does nothing useful and is just illustrative

Skills To Gain

- To solve these runtime problems try to break the problem into 3 parts:
- FIRST, setup the expression (or recurrence relationship) for the number of operations, T(n)
- SECOND, solve to get a closed form for T(n)
  - Unwind the recurrence relationship
  - Develop a series summation
  - Solve the series summation
- THIRD, determine the asymptotic bound for T(n)

Loops 1

- Derive an expression, T(n), in terms of the input size for the number of operations/steps that are required to solve a problem
- $T(n) =$
- $= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \theta(1) = \sum_{i=0}^{n-1} \theta(n) = \Theta(n^2)$

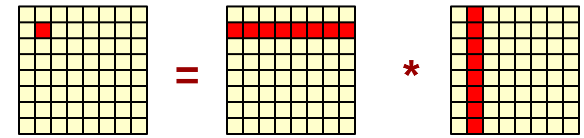
```
#include <iostream>

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;
}
```

USC Viterbi School of Engineering

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
- $T(n) =$

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



C A B

Traditional Multiply

```
#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?
- ______________
  - 3 for loops, ______________

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

const int n = /* large constant */;

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;
}
```

Runtime Practice #1

- 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!!
- Find T(n) for this example

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

Hint: Geometric series

- $\sum_{i=0}^{\_\_\_\_\_} \sum_{j=0}^{\_\_\_\_\_} \theta(1)$
- =
- Use the geometric sum eqn.
- $= \sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$
- So our answer is…

Runtime Practice #2

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

```
for(int i=0; i < n; i++){
   if (a[i][0] == 0){
     for (int j = 0; j < i; j++){
          a[i][j] = i*j;
     }
   }
}
```

Hint: Arithmetic series

- $T(n) =$ ________________________ May start with big-O and not worry about input values affecting how many times if statement executes
- $T(n) = \sum_{i=0}^{n-1}(\theta(1)) + \sum_{i}(\theta(i))$ Distribute to deal with 'if' separately. Not sure which values of i will trigger the for loop that incurs i steps
  - In the worst case, how many times can the 'if' statement be true? ________________
- $T(n) =$

Runtime Practice #3

```
for(int i=0; i < n; i++){
   if (i == 0){
      for (int j = 0; j < n; j++){
         a[i][j] = i*j;
      }
   }
}
```

You must use your analytical skills to determine how many times the 'if' will trigger and then sum the inner operations that many times.

- $T(n) =$
- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1) + O\left(\sum_{j=1}^{n}\theta(1)\right)\right)$ Use big-O since unsure of how many times if statement executes
  - Important: How many times will the 'if' statement be true?
- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1)\right) + \sum_{i}\sum_{j=1}^{n}\theta(1)$
  - The 'if' statement only triggers once! So the inner loop executes only once
- $T(n) =$

USC Viterbi School of Engineering

Runtime Practice #4

```
for (int i = 1; i <= n; i++)
{  int m = sqrt(n);
   if( i % m == 0){
     for (int j=0; j < n; j++)
       cout << j << " ";
   }
   cout << endl;
}
```

- $T(n) = \sum_{i=1}^{n}\left(\theta(1) + O\left(\sum_{j=0}^{n-1}\theta(1)\right)\right)$
  - big-O indicates we have not considered the 'if' statement but are setting an upper bound
- $T(n) = \sum_{i=1}^{n}\theta(1) + \sum_{i}\sum_{j=0}^{n-1}\theta(1)$ but we need to user our own analysis skills to find the actual values of i that will cause the 'if' to be true?
  - Use some actual values of n (e.g. n=9 or 16). Write out a table to find the pattern.
  - If n=9, the 'if' will trigger ___ times for i = _________________
  - If n=16, the 'if' will trigger ___ times for i = ________________
  - The dummy variable of a summation must increment ____ at a time
  - Thus, make a table with some dummy variable (k) that increments 1 at a time and find a relationship to the actual variable, i, for when the if statement will trigger.
  - Solve for upper bound of k
    - Stop when i = __, but i = ____ so we stop when __________thus solve for k to find that the upper-bound for k = _______

k	1	2	3	…	Arbitrary k	Stop when k =??
i				…	i = _______	Stop when i = _______

- $T(n) =$

Key Skill

- The dummy variable (say k) of a summation runs from 1 to an UPPER_BOUND incrementing 1 at a time
- Often our code does work at some other interval such as $i = \{1\sqrt{n}, 2\sqrt{n}, 3\sqrt{n} \ \ldots\}$ (or actual values that are not incrementing by 1 at a time)
- You must use your own analytical abilities to find a relationship that converts the dummy variable (k=1,2,3,…) to the actual values [eg. i = f(k) = $k\sqrt{n}$], usually by making a table of the dummy variable (k) and the actual code values/variables (i)

k	1	2	3	…	Arbitrary k	Stop when k =??
i				…	$i =$ _______	Stop when i = _______

- Then use that relationship to find the UPPER_BOUND of the dummy variable
  - In the previous example, we stopped when i = n, thus we would stop when our dummy variable is $\sqrt{n}$. This then is the upper bound.
- The key skill is to relate the dummy variable to the actual variable values and then find the UPPER BOUND of the dummy variable

Runtime Practice #5

- $T(n) =$

```
for(int i=1; i <= n; i++){
    for (int j = 0; j < n; j += i){
        a[i][j] = i*j;
    }
}
```

Hint: Harmonic series

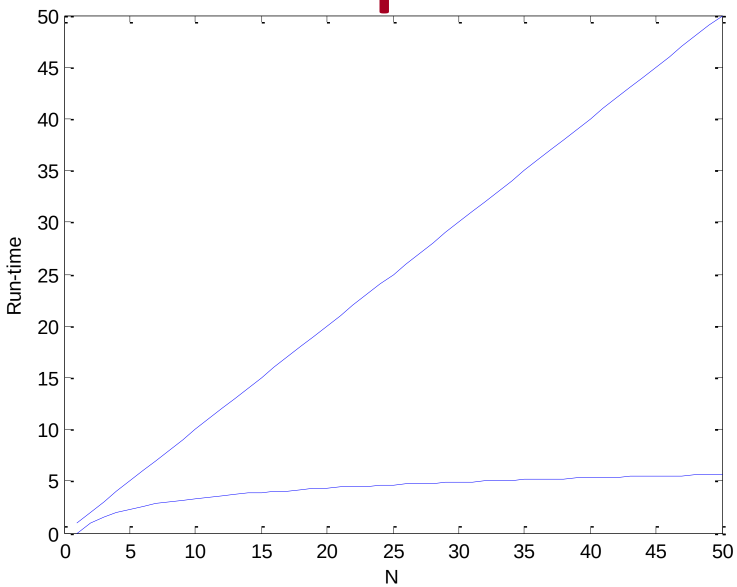
- $T(n) = \sum_{i=1}^{n}\left(\theta(1) + \sum_j \theta(1)\right) = \theta(n) + \sum_{i=1}^{n}\sum_j \theta(1)$
- Manually, determine how many times the j-loop iterates:
  - When i=1, j takes on values: ______________________________ [Total = _____ iters]
  - When i=2, j takes on values: ______________________________ [Total = _____ iters]
  - When i=3, j takes on values: ______________________________ [Total = _____ iters]
- $T(n) = \theta(n) +$

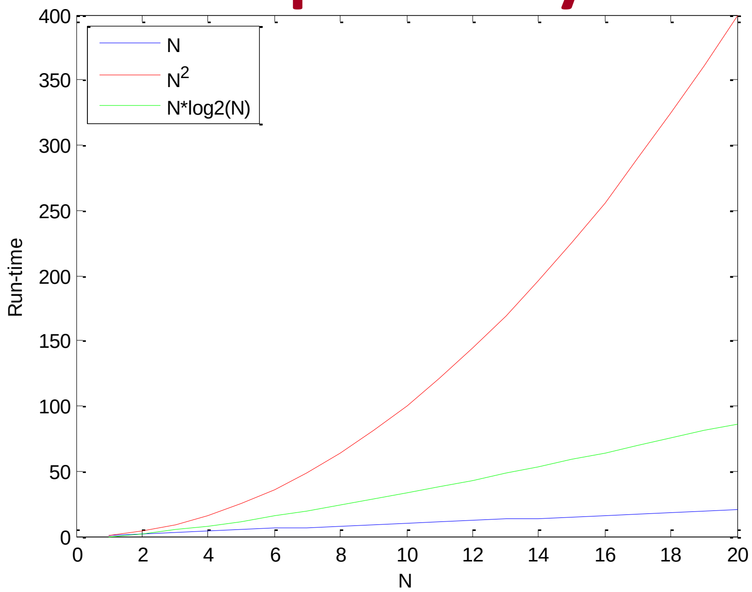
Runtime Practice #6

- 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 1^{st} iteration => x=____
  - After 2^{nd} iteration => x=____
  - After 3^{rd} iteration => x=____
  - Say k^{th} iteration is last => x = ______ = 1
  - Solve for k
  - k = __________ iterations
  - θ(_____________)

```
#include <iostream>
using namespace std;
const int n = /* Some constant */;

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;
}
```

Importance of Complexity





N	O(1)	O($\log_2 n$)	O(n)	O($n*\log_2 n$)	O(n^2)	O(2^n)
2	1	1	2	2	4	4
20	1	4.3	20	86.4	400	1,048,576
200	1	7.6	200	1,528.8	40,000	1.60694E+60
2000	1	11.0	2000	21,931.6	4,000,000	#NUM!

EXTRAS

Runtime Practice #7

- $T(n) = \sum_{i=1}^{n}\left(\theta(1) + O\left(\sum_{j=1}^{i}\theta(1)\right)\right)$

```
for(int i=0; i < n; i++){
    if ((i% 2) == 0){
      for (int j = 0; j < i; j++)
          a[i][j] = i*j;
    }
    else { a[i][0] = i; }
}
```

- Important: How many times will the 'if' statement be true?
- $T(n) = \sum_{i=1}^{n}\left(\theta(1)\right) + \sum_{i}\sum_{j=1}^{i}\theta(1)$
  - Find a relationship between a dummy variable, k, that increments by 1 and the values of i that cause the if statement to trigger

k	**1**	**2**	**3**	**...**	**Arbitrary k**	**Stop when k = (n/2)+1**
i	0	2	4	...	i = ________	Stop when i = ______

- $T(n) =$

Recall: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$

Runtime Practice #8

```
for(int i=1; i <= n; i*=2){
    for (int j = 0; j < i; j++){
        a[i][j] = i*j;
    }
}
```

Hint: Geometric series

- $T(n) =$
- $T(n) = \sum_i \left(\sum_{j=0}^{i-1} \theta(1) \right) =$
  $= \sum_i \left(\theta(i) \right)$
- The number of iterations of the outer loop requires derivation:

Iter, k	1	2	3	4	…	k	Stop at: ($\log_2 n$)
i after iteration	2	4	8	16	…	2^k	Stop at: n

- $T(n) = \sum_{k=1}^{log2(n)} \theta\left(2^k\right)$
- $T(n) = \theta\left(\frac{2^{log2(n)+1}-1}{2-1}\right) = \theta\left(\frac{2^{log2(n)}2^1-1}{1}\right) =$
  $\theta(2n-1) = \theta(n)$

Iterative Binary Search

- Assume n is total array size and let L = (end-start)
  - L = # of items to be searched
- $T(n) = \sum_k \theta(1)$
  - k is the # of iterations required
- After 1st iteration L = n/2
- After 2nd iteration L = n/4
- After 3rd iteration L = n/8
- …
- After kth iteration L = $n/2^k$
- We stop when we reach size 0 or 1…when k = $\log_2(n)$
- $T(n) = \sum_{k=1}^{log_2(n)} \theta(1) = \theta(log_2(n))$

```
int main()
{  int data[4] = {1, 6, 7, 9};
   it_bsearch(3,data, 4);
}

int it_bsearch(int target,
               int data[],int len)
{
  int start = 0, end = len, mid;

  while (start < end) {
     mid = (start+end)/2;
     if (data[mid] == target){
         return mid;
     } else if ( target < data[mid]){
         end = mid-1;
     } else {
         start = mid+1;
     }
  }
  return -1;
}
```

SOLUTIONS

Sequential Loops

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

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

const int n = /* large constant */;

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;
}
```

Runtime Practice #1

- 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!!
- Find T(n) for this example

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

Hint: Geometric series

- $\sum_{i=0}^{\lg(n)} \sum_{j=0}^{2^i-1} \theta(1)$
- $=\sum_{i=0}^{\lg(n)} \theta(2^i)$
- Use the geometric sum eqn.
- $=\sum_{i=0}^{n-1} a^i = \frac{a^n-1}{a-1}$
- So our answer is…
- $\frac{2^{\lg(n)+1}-1}{2-1} = \frac{2*n-1}{1} = \theta(n)$

Runtime Practice #2

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

```
for(int i=0; i < n; i++){
   if (a[i][0] == 0){
     for (int j = 0; j < i; j++){
          a[i][j] = i*j;
     }
   }
}
```

Hint: Arithmetic series

- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1) + O\left(\sum_{j=0}^{i-1}\theta(1)\right)\right)$ May start with big-O and not worry about input values affecting how many times if statement executes
- $T(n) = \sum_{i=0}^{n-1}(\theta(1)) + \sum_{i}(\theta(i))$ Distribute to deal with 'if' separately. Not sure which values of i will trigger the for loop that incurs i steps
  - In the worst case, how many times can the 'if' statement be true? Each iteration (i.e. all n values of i)
- $T(n) = \sum_{i=0}^{n-1}(\theta(1)) + \sum_{i=0}^{n-1}(\theta(i))$
- $T(n) = \theta(n) + \sum_{i=0}^{n-1}(\theta(i)) = \theta(n) + \theta\left(\frac{n(n-1)}{2}\right) = \theta(n^2)$

Runtime Practice #3

```
for(int i=0; i < n; i++){
   if (i == 0){
      for (int j = 0; j < n; j++){
         a[i][j] = i*j;
      }
   }
}
```

You must use your analytical skills to determine how many times the 'if' will trigger and then sum the inner operations that many times.

- $T(n) =$
- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1) + O\left(\sum_{j=0}^{n-1}\theta(1)\right)\right)$ Use big-O since unsure of how many times if statement executes
  - Important: How many times will the 'if' statement be true?
- $T(n) = \sum_{i=0}^{n-1}\left(\theta(1)\right) + \sum_{i}\sum_{j=0}^{n-1}\theta(1)$
  - The 'if' statement only triggers once! So the inner loop executes only once
- $T(n) = \theta(n) + 1 \cdot \sum_{j=0}^{n-1}\theta(1) = \theta(n) + \theta(n) = \theta(n)$

Runtime Practice #4

```
for (int i = 1; i <= n; i++)
{  int m = sqrt(n);
   if( i % m == 0){
     for (int j=0; j < n; j++)
       cout << j << " ";
   }
   cout << endl;
}
```

- $T(n) = \sum_{i=1}^{n} \left(\theta(1) + O\left(\sum_{j=0}^{n-1} \theta(1)\right)\right)$
  - big-O indicates we have not considered the 'if' statement but are setting an upper bound
- $T(n) = \sum_{i=1}^{n} \theta(1) + \sum_{i} \sum_{j=0}^{n-1} \theta(1)$ but we need to user our own analysis skills to find the actual values of i that will cause the 'if' to be true?
  - Use some actual values of n (e.g. n=9 or 16). Write out a table to find the pattern.
  - If n=9, the 'if' will trigger 3 times for i = 3, 6, 9
  - If n=16, the 'if' will trigger 4 times for i = 4, 8, 12, 16
  - The dummy variable of a summation must increment 1 at a time
  - Thus, make a table with some dummy variable (k) that increments 1 at a time and find a relationship to the actual variable, i, for when the if statement will trigger.
  - Solve for upper bound of k
    - Stop when i = n, but i = $k\sqrt{n}$ so we stop when $k\sqrt{n} = n$ thus solve for k to find that the upper-bound for $k = \sqrt{n}$

k	1	2	3	…	Arbitrary k	Stop when k =??
i	$1\sqrt{n}$	$2\sqrt{n}$	$3\sqrt{n}$	…	$i = k\sqrt{n}$	Stop when i = n

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

Runtime Practice #5

- $T(n) =$

```
for(int i=1; i <= n; i++){
    for (int j = 0; j < n; j += i){
        a[i][j] = i*j;
    }
}
```

Hint: Harmonic series

- $T(n) = \sum_{i=1}^{n}\left(\theta(1) + \sum_{j}\theta(1)\right) = \theta(n) + \sum_{i=1}^{n}\sum_{j}\theta(1)$
- Manually, determine how many times the j-loop iterates:
  - When i=1, j takes on values: 0, 1, 2, 3, … , n-1 [Total = n iters]
  - When i=2, j takes on values: 0, 2, 4, 6, … , n-2 or n-1 [Total = n/2 iters]
  - When i=3, j takes on values: 0, 3, 6, 9, … [Total = n/3 iters]
- $T(n) = \theta(n) + \left[\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n}\right]\theta(1)$

$= \theta(n) + \left[\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right]\theta(n)$

$= \theta(n) + \left(\sum_{i=1}^{n}\frac{1}{i}\right)\cdot\theta(n) = \theta(n) + \log n \cdot \theta(n) = \theta(n \cdot \log n)$

Runtime Practice #6

- 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
  - $\theta(n*\log(n))$

```
#include <iostream>
using namespace std;
const int n = /* Some constant */;

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;
}
```

Runtime Practice #7

```
for(int i=0; i < n; i++){
    if ((i% 2) == 0){
      for (int j = 0; j < i; j++)
          a[i][j] = i*j;
    }
    else { a[i][0] = i; }
}
```

- $T(n) = \sum_{i=1}^{n}\left(\theta(1) + O\left(\sum_{j=1}^{i}\theta(1)\right)\right)$
- Important: How many times will the 'if' statement be true?
- $T(n) = \sum_{i=1}^{n}\left(\theta(1)\right) + \sum_{i}\sum_{j=1}^{i}\theta(1)$
  - Find a relationship between a dummy variable, k, that increments by 1 and the values of i that cause the if statement to trigger

k	**1**	**2**	**3**	**…**	**Arbitrary k**	**Stop when k = (n/2)+1**
i	0	2	4	…	$i = 2(k-1)$	Stop when i = n

- $T(n) = \theta(n) + \sum_{k=1}^{\frac{n}{2}+1}\sum_{j=1}^{2(k-1)}\theta(1) = \theta(n) + \sum_{k=1}^{\frac{n}{2}+1}\theta(2k-2) =$

$$\theta(n) + 2 \cdot \sum_{k=1}^{\frac{n}{2}+1}\theta(k) = \theta(n) + 2 \cdot \theta\left(\left[\frac{n}{2}+1\right]^2\right) = \theta(n^2)$$

Recall: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$

Runtime Practice #8

```
for(int i=1; i <= n; i*=2){
    for (int j = 0; j < i; j++){
        a[i][j] = i*j;
    }
}
```

Hint: Geometric series

- $T(n) =$
- $T(n) = \sum_i \left(\sum_{j=0}^{i-1} \theta(1) \right) =$
  $= \sum_i \left(\theta(i) \right)$
- The number of iterations of the outer loop requires derivation:

Iter, k	1	2	3	4	…	k	($\log_2 n$)
i after iteration	2	4	8	16	…	2^k	n

- $T(n) = \sum_{k=1}^{log2(n)} \theta\left(2^k\right)$
- $T(n) = \theta\left(\frac{2^{log2(n)+1}-1}{2-1}\right) = \theta\left(\frac{2^{log2(n)}2^1-1}{1}\right) =$
  $\theta(2n-1) = \theta(n)$