CSE 373

OCTOBER 6TH – ALGORITHM ANALYSIS

ASSORTED MINUTIAE

- Checkpoint 1 should be in
- Late submissions still to canvas
- Official grade Monday
- Half of points lost can be reearned

REVIEW

- **Counting operations isn’t the best for determining performance**
  - Need an understanding of how runtime changes relative to input size
  - Asymptotic analysis
  - bigO notation

BIG-O NOTATION

- Informally: bigO notation denotes an upper bound for an algorithms asymptotic runtime
- For example, if an algorithm `A` is `O(log n)`, that means some logarithmic function upper bounds `A`.

BIG-O NOTATION

- Formally, a function $f(n)$ is $O(g(n))$ if there exists a c and n_0 such that:
- For all $n \geq n_0$, $f(n) < c*g(n)$
- To prove a function is O(g(n)), simply find the c and n_0

EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?

EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?
- $4 + 3n = O(n^2)$
- $n + 2 \log n = O(\log n)$?

EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?
- $4 + 3n = O(n^2)$
- $n + 2 \log n = O(\log n)$?
- $\log n = O(n + 2 \log n)$?

REVIEW

- **Practice**
  - Inserting into a sorted linked list

REVIEW

- **Practice**
  - Inserting into a sorted linked list
  - What is the approach?

REVIEW

```
start at the front of the list
```

REVIEW

```
start at the front of the list
while the pointer is less than the insert item:
```

REVIEW

```
start at the front of the list
while the pointer is less than the insert item:
      move to the next node
```

REVIEW

```
start at the front of the list
while the pointer is less than the insert item:
      move to the next node
insert the element, relinking the list around it
```

REVIEW

```
start at the front of the list
while the pointer is less than the insert item:
      move to the next node
insert the element, relinking the list around it
```

- **What is the runtime here?**

REVIEW

```
start at the front of the list
while the pointer is less than the insert item:
      move to the next node
insert the element, relinking the list around it
```

- **What is the runtime here?**
  - Important considerations—best-case or worst-case?

REVIEW

- **Worst-case**

REVIEW

- **Worst-case**
  - What is this case?

REVIEW

- **Worst-case**
  - What is this case?
    - Inserting the new largest element (i.e. at the end of the list)

REVIEW

- **Worst-case**
  - What is this case?
    - Inserting the new largest element (i.e. at the end of the list)
  - What is the runtime?
    - **O(n)**

REVIEW

- **Worst-case**
  - What is this case?
    - Inserting the new largest element (i.e. at the end of the list)
  - What is the runtime?
    - **O(n)** Why?

REVIEW

- **Worst-case**
  - What is this case?
    - Inserting the new largest element (i.e. at the end of the list)
  - What is the runtime?
    - **O(n)** Why?
    - The loop must iterate through all *n* elements to find the correct place

REVIEW

- **Best-case**

REVIEW

- **Best-case**
  - What is this case?

REVIEW

- **Best-case**
  - What is this case?
    - Smallest element, inserting at the beginning

REVIEW

- **Best-case**
  - What is this case?
    - Smallest element, inserting at the beginning
  - What is the runtime?

REVIEW

- **Best-case**
  - What is this case?
    - Smallest element, inserting at the beginning
  - What is the runtime?
    - **O(1)**

REVIEW

- **Best-case**
  - What is this case?
    - Smallest element, inserting at the beginning
  - What is the runtime?
    - **O(1) –** we can add to the front of a linked list in constant time

ANALYSIS

- Loops and iterations can be analyzed

ANALYSIS

- Loops and iterations can be analyzed
- How do we approach recursive functions?

ANALYSIS

- **Loops and iterations can be analyzed**
- **How do we approach recursive functions?**
  - Let’s consider a recursive algorithm that reverses a list

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;
        else if(L.next == null) return L;
        else
                Node front = L
                Node rest = L.next
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;
        else if(L.next == null) return L;
        else
                Node front = L
                Node rest = L.next
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                          non-recursive
        else if(L.next == null) return L;
        else
                Node front = L
                Node rest = L.next
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                          non-recursive
        else if(L.next == null) return L;              non-recursive
        else
                Node front = L
                Node rest = L.next
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                       non-recursive
        else if(L.next == null) return L;           non-recursive
        else
                Node front = L                      non-recursive
                Node rest = L.next
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                        non-recursive
        else if(L.next == null) return L;            non-recursive
        else
                Node front = L                       non-recursive
                Node rest = L.next                   non-recursive
                L.next = null
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                       non-recursive
        else if(L.next == null) return L;           non-recursive
        else
                Node front = L                      non-recursive
                Node rest = L.next                  non-recursive
                L.next = null                       non-recursive
                Node restRev = reverse(rest)
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                     non-recursive
        else if(L.next == null) return L;         non-recursive
        else
                Node front = L                    non-recursive
                Node rest = L.next                non-recursive
                L.next = null                     non-recursive
                Node restRev = reverse(rest)      recursive
                appendToEnd(front,restRev)
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                         non-recursive
        else if(L.next == null) return L;             non-recursive
        else
                Node front = L                        non-recursive
                Node rest = L.next                    non-recursive
                L.next = null                         non-recursive
                Node restRev = reverse(rest)          recursive
                appendToEnd(front,restRev)            non-recursive
```

- **We know how to analyze everything but the recursive step, so break the algorithm into its two parts, recursive and non-recursive**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                        non-recursive
        else if(L.next == null) return L;            non-recursive
        else
                Node front = L                       non-recursive
                Node rest = L.next                   non-recursive
                L.next = null                        non-recursive
                Node restRev = reverse(rest)         recursive
                appendToEnd(front,restRev)           non-recursive
```

- **What is the runtime of the non-recursive work?**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                          non-recursive
        else if(L.next == null) return L;              non-recursive
        else
                Node front = L                         non-recursive
                Node rest = L.next                     non-recursive
                L.next = null                          non-recursive
                Node restRev = reverse(rest)           recursive
                appendToEnd(front,restRev)             non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Depends on the case!

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                      non-recursive
        else if(L.next == null) return L;          non-recursive
        else
                Node front = L                     non-recursive
                Node rest = L.next                 non-recursive
                L.next = null                      non-recursive
                Node restRev = reverse(rest)       recursive
                appendToEnd(front,restRev)         non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Depends on the case! There are two base cases, n = 0 and n = 1, but let's look at the n > 1 case first

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                       non-recursive
        else if(L.next == null) return L;           non-recursive
        else
                Node front = L                      non-recursive
                Node rest = L.next                  non-recursive
                L.next = null                       non-recursive
                Node restRev = reverse(rest)        recursive
                appendToEnd(front,restRev)          non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Depends on the case! There are two base cases, n = 0 and n = 1, but let's look at the n > 1 case first
  - Suppose that `appendToEnd` takes O(n) time

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                        non-recursive
        else if(L.next == null) return L;            non-recursive
        else
                Node front = L                       non-recursive
                Node rest = L.next                   non-recursive
                L.next = null                        non-recursive
                Node restRev = reverse(rest)         recursive
                appendToEnd(front,restRev)           non-recursive
```

- **What is the runtime of the non-recursive work?**
    - Let's look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                   non-recursive
        else if(L.next == null) return L;       non-recursive
        else
                Node front = L                  non-recursive O(1)
                Node rest = L.next              non-recursive
                L.next = null                   non-recursive
                Node restRev = reverse(rest)    recursive
                appendToEnd(front,restRev)      non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Let’s look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                     non-recursive
        else if(L.next == null) return L;         non-recursive
        else
                Node front = L                    non-recursive O(1)
                Node rest = L.next                non-recursive O(1)
                L.next = null                     non-recursive
                Node restRev = reverse(rest)      recursive
                appendToEnd(front,restRev)        non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Let’s look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                       non-recursive
        else if(L.next == null) return L;           non-recursive
        else
                Node front = L                      non-recursive O(1)
                Node rest = L.next                  non-recursive O(1)
                L.next = null                       non-recursive O(1)
                Node restRev = reverse(rest)        recursive
                appendToEnd(front,restRev)          non-recursive
```

- **What is the runtime of the non-recursive work?**
  - Let's look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                          non-recursive
        else if(L.next == null) return L;              non-recursive
        else
                Node front = L                         non-recursive O(1)
                Node rest = L.next                     non-recursive O(1)
                L.next = null                          non-recursive O(1)
                Node restRev = reverse(rest)           recursive
                appendToEnd(front,restRev)             non-recursive O(n)
```

- **What is the runtime of the non-recursive work?**
  - Let’s look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                         non-recursive O(1)
        else if(L.next == null) return L;             non-recursive O(1)
        else
                Node front = L                        non-recursive O(1)
                Node rest = L.next                    non-recursive O(1)
                L.next = null                         non-recursive O(1)
                Node restRev = reverse(rest)          recursive
                appendToEnd(front,restRev)            non-recursive O(n)
```

- **What is the runtime of the non-recursive work?**
  - Let’s look at each piece

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                      non-recursive O(1)
        else if(L.next == null) return L;          non-recursive O(1)
        else
                Node front = L                     non-recursive O(1)
                Node rest = L.next                 non-recursive O(1)
                L.next = null                      non-recursive O(1)
                Node restRev = reverse(rest)       recursive
                appendToEnd(front,restRev)         non-recursive O(n)
```

- **What is the runtime of the non-recursive work?**
    - Here, n is the size of the list starting at L

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                    non-recursive O(1)
        else if(L.next == null) return L;        non-recursive O(1)
        else
                Node front = L                   non-recursive O(1)
                Node rest = L.next               non-recursive O(1)
                L.next = null                    non-recursive O(1)
                Node restRev = reverse(rest)     recursive
                appendToEnd(front,restRev)       non-recursive O(n)
```

- **What is the runtime of the non-recursive work?**
  - This is O(n) total, which means we can upper bound the non-recursive work by $c_0 + c_1 * n$

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                   non-recursive O(1)
        else if(L.next == null) return L;       non-recursive O(1)
        else
                Node front = L                  non-recursive O(1)
                Node rest = L.next              non-recursive O(1)
                L.next = null                   non-recursive O(1)
                Node restRev = reverse(rest)    recursive
                appendToEnd(front,restRev)      non-recursive O(n)
```

- **What is the total runtime then?**

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                     non-recursive O(1)
        else if(L.next == null) return L;         non-recursive O(1)
        else
                Node front = L                    non-recursive O(1)
                Node rest = L.next                non-recursive O(1)
                L.next = null                     non-recursive O(1)
                Node restRev = reverse(rest)      recursive
                appendToEnd(front,restRev)        non-recursive O(n)
```

- **What is the total runtime then?**
  - Let the functions runtime be denoted as T(n), where n is the number of elements

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                   non-recursive O(1)
        else if(L.next == null) return L;       non-recursive O(1)
        else
                Node front = L                  non-recursive O(1)
                Node rest = L.next              non-recursive O(1)
                L.next = null                   non-recursive O(1)
                Node restRev = reverse(rest)    recursive
                appendToEnd(front,restRev)      non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1 * n +$ *recursive work*

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                         non-recursive O(1)
        else if(L.next == null) return L;             non-recursive O(1)
        else
                Node front = L                        non-recursive O(1)
                Node rest = L.next                    non-recursive O(1)
                L.next = null                         non-recursive O(1)
                Node restRev = reverse(rest)          recursive
                appendToEnd(front,restRev)            non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1 * n +$ *recursive work*
  - What is the recursive work?

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                        non-recursive O(1)
        else if(L.next == null) return L;            non-recursive O(1)
        else
                Node front = L                       non-recursive O(1)
                Node rest = L.next                   non-recursive O(1)
                L.next = null                        non-recursive O(1)
                Node restRev = reverse(rest)         recursive
                appendToEnd(front,restRev)           non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1*n +$ *recursive work*
  - What is the recursive work? `rest` is size n-1

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                       non-recursive O(1)
        else if(L.next == null) return L;           non-recursive O(1)
        else
                Node front = L                      non-recursive O(1)
                Node rest = L.next                  non-recursive O(1)
                L.next = null                       non-recursive O(1)
                Node restRev = reverse(rest)        recursive
                appendToEnd(front,restRev)          non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1 * n + T(n-1)$

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                        non-recursive O(1)
        else if(L.next == null) return L;            non-recursive O(1)
        else
                Node front = L                       non-recursive O(1)
                Node rest = L.next                   non-recursive O(1)
                L.next = null                        non-recursive O(1)
                Node restRev = reverse(rest)         recursive
                appendToEnd(front,restRev)           non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1*n + T(n-1)$
  - This is the recurrence! It's a function that uses itself in its definition

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                         non-recursive O(1)
        else if(L.next == null) return L;             non-recursive O(1)
        else
                Node front = L                        non-recursive O(1)
                Node rest = L.next                    non-recursive O(1)
                L.next = null                         non-recursive O(1)
                Node restRev = reverse(rest)          recursive
                appendToEnd(front,restRev)            non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1 * n + T(n-1)$
  - This is the recurrence! It's a function that uses itself in its definition
  - Fibonnacci numbers are an example

ANALYSIS

```
reverse(Node L):
        if(L==null) return L;                          non-recursive O(1)
        else if(L.next == null) return L;              non-recursive O(1)
        else
                Node front = L                         non-recursive O(1)
                Node rest = L.next                     non-recursive O(1)
                L.next = null                          non-recursive O(1)
                Node restRev = reverse(rest)           recursive
                appendToEnd(front,restRev)             non-recursive O(n)
```

- **What is the total runtime then?**
  - $T(n) = c_0 + c_1 * n + \mathit{T(n-1)}$
  - This is the recurrence! It's a function that uses itself in its definition
  - Fibonnacci numbers are an example. **What's missing?**

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **How do we solve this recurrence?**

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **How do we solve this recurrence?**
  - We can unroll it and see if a pattern emerges
  - $T(n) = c_0 + c_1*n + T(n-1)$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1{*}n + T(n-1)$ when $n > 1$
- **How do we solve this recurrence?**
  - We can unroll it and see if a pattern emerges
  - $T(n) = c_0 + c_1{*}n + T(n-1)$
  - $T(n) = c_0 + c_1{*}n + c_0 + c_1{*}(n-1) + T(n-2)$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **How do we solve this recurrence?**
  - We can unroll it and see if a pattern emerges
  - $T(n) = c_0 + c_1*n + T(n-1)$
  - $T(n) = c_0 + c_1*n + c_0 + c_1*(n-1) + T(n-2)$
  - $T(n) = c_0 + c_1*n + c_0 + c_1*(n-1) + c_0 + c_1*(n-2) + T(n-3)$
  - $T(n) = 3c_0 + c_1*(n+(n-1)+(n-2)) + T(n-3)$
  - What are the patterns?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **What are the patterns?**
  - Each time we add 1 c_0
  - Each time we add ‘n’ c_1

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **What are the patterns?**
  - Each time we add 1 c_0
  - Each time we add ‘n’ c_1
  - But n is getting reduced by one every time

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **What are the patterns?**
  - Each time we add 1 c_0
  - Each time we add ‘n’ c_1
  - But n is getting reduced by one every time
  - How many times does this call itself?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **What are the patterns?**
  - Each time we add 1 c_0
  - Each time we add ‘n’ c_1
  - But n is getting reduced by one every time
  - How many times does this call itself?
    - n-1, because 1 is a base case

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **What are the patterns?**
  - Each time we add 1 c_0
  - Each time we add ‘n’ c_1
  - But n is getting reduced by one every time
  - How many times does this call itself?
    - n-1, because 1 is a base case
  - What then is the closed form of this recurrence?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **Closed form?**

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) =$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 +$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + \Sigma\, i * c_1$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1*n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1$
  - Is this all?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1 \mathbf{+ d_1}$
  - Is this all?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1 + d_1$
  - What is the upper bound of this function?

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1 + d_1$
  - What is the upper bound of this function?
    - $O(n^2)$

ANALYSIS

- **Recurrence relation for reverse**
  - $T(n) = d_0$ when $n = 0$
  - $T(n) = d_1$ when $n = 1$
  - $T(n) = c_0 + c_1 * n + T(n-1)$ when $n > 1$
- **Closed form?**
  - $T(n) = (n-1) * c_0 + (n-1)*(n)/2 * c_1 + d_1$
  - What is the upper bound of this function?
    - $O(n^2)$ the $O(n)$ appendToEnd is what costs us

ANALYSIS

- **While this process is important, we can save some steps if all we care about is the upper bound**
  - bigO notation eliminates the need for constants
  - Lots of our messing around with c_0 and c_1 doesn't come through to the solution
  - Rather than saying $T(n) = c_0 + n*c_1 + T(n-1)$, we can observe that $c_0 + n*c_1$ is in $O(n)$
  - Simplify to $T(n) = O(n) + T(n-1)$

ANALYSIS

- **Let's consider binary search again**

ANALYSIS

- **Let’s consider binary search again**
  - We mentioned last week that it was O(log n)

ANALYSIS

- **Let's consider binary search again**
  - We mentioned last week that it was O(log n)
  - Can you use recurrence relations to show this for a recursive implementation?

ANALYSIS

- **Let's consider binary search again**
  - We mentioned last week that it was O(log n)
  - Can you use recurrence relations to show this for a recursive implementation?

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
	if(hi < lo) return null;
	mid = hi/2 + lo/2
	if(A[mid] > value)
		return BinarySearch(array,value,mid,hi)
	else if(A[mid] < value)
		return BinarySearch(array,value,lo,mid)
	else return mid
```

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
        if(hi < lo) return null;
        mid = hi/2 + lo/2
        if(A[mid] > value)
                return BinarySearch(array,value,mid,hi)
        else if(A[mid] < value)
                return BinarySearch(array,value,lo,mid)
        else return mid
```

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
        if(hi < lo) return null;
        mid = hi/2 + lo/2
        if(A[mid] > value)
                return BinarySearch(array,value,mid,hi)
        else if(A[mid] < value)
                return BinarySearch(array,value,lo,mid)
        else return mid
```

- **What steps do we need to take?**

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
        if(hi < lo) return null;
        mid = hi/2 + lo/2
        if(A[mid] > value)
                return BinarySearch(array,value,mid,hi)
        else if(A[mid] < value)
                return BinarySearch(array,value,lo,mid)
        else return mid
```

- **What steps do we need to take?**
  - Break down into recursive and non-recursive

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
       if(hi < lo) return null;
       mid = hi/2 + lo/2
       if(A[mid] > value)
              return BinarySearch(array,value,mid,hi)
       else if(A[mid] < value)
              return BinarySearch(array,value,lo,mid)
       else return mid
```

- **What steps do we need to take?**
  - Break down into recursive and non-recursive
  - Calculate the non-recursive runtimes

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
       if(hi < lo) return null;
       mid = hi/2 + lo/2
       if(A[mid] > value)
              return BinarySearch(array,value,mid,hi)
       else if(A[mid] < value)
              return BinarySearch(array,value,lo,mid)
       else return mid
```

- **What steps do we need to take?**
  - Break down into recursive and non-recursive
  - Calculate the non-recursive runtimes
  - Produce the recurrence

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
       if(hi < lo) return null;
       mid = hi/2 + lo/2
       if(A[mid] > value)
              return BinarySearch(array,value,mid,hi)
       else if(A[mid] < value)
              return BinarySearch(array,value,lo,mid)
       else return mid
```

- **What steps do we need to take?**
  - Break down into recursive and non-recursive
  - Calculate the non-recursive runtimes
  - Produce the recurrence
  - Roll out the recurrence to observe a pattern

ANALYSIS

```
BinarySearch(Integer[] array, Integer value, int lo, int hi)
       if(hi < lo) return null;
       mid = hi/2 + lo/2
       if(A[mid] > value)
              return BinarySearch(array,value,mid,hi)
       else if(A[mid] < value)
              return BinarySearch(array,value,lo,mid)
       else return mid
```

- **What steps do we need to take?**
  - Break down into recursive and non-recursive
  - Calculate the non-recursive runtimes
  - Produce the recurrence
  - Roll out the recurrence to observe a pattern
  - Upper bound the closed form

ANALYSIS

- What is the recurrence we produced?

ANALYSIS

- **What is the recurrence we produced?**
    - $T(n) = d_0$ for $n = 0$

ANALYSIS

- **What is the recurrence we produced?**
  - $T(n) = d_0$ for $n = 0$
  - $T(n) = c_0 + T(n/2)$ for $n > 0$

ANALYSIS

- **What is the recurrence we produced?**
  - $T(n) = d_0$ for $n = 0$
  - $T(n) = c_0 + T(n/2)$ for $n > 0$
- **Important to note**

ANALYSIS

- **What is the recurrence we produced?**
  - $T(n) = d_0$ for $n = 0$
  - $T(n) = c_0 + T(n/2)$ for $n > 0$
- **Important to note**
  - How many times can we divide n by 2 until we get 1?

ANALYSIS

- **What is the recurrence we produced?**
  - $T(n) = d_0$ for $n = 0$
  - $T(n) = c_0 + T(n/2)$ for $n > 0$
- **Important to note**
  - How many times can we divide n by 2 until we get 1?
  - $\mathrm{Log}_2\ n$

ANALYSIS

- **Let's consider a recursive function which counts the number of instances of an element in an array.**

```
int countNumber(String[] array, String toFind, int lo, int hi){
        if(lo == hi) return array[lo]==toFind?0:1
        else
                int mid = (lo+hi)/2
                return countNumber(array,toFind,lo,mid) +
                        countNumber(array,toFind,mid,hi)
```

ANALYSIS

- **Let's consider a recursive function which counts the number of instances of an element in an array.**

```
int countNumber(String[] array, String toFind, int lo, int hi){
        if(lo == hi) return array[lo]==toFind?0:1
        else
                int mid = (lo+hi)/2
                return countNumber(array,toFind,lo,mid) +
                        countNumber(array,toFind,mid,hi)
```

What is the recurrence here?

ANALYSIS

- **Let's consider a recursive function which counts the number of instances of an element in an array.**

```
int countNumber(String[] array, String toFind, int lo, int hi){
        if(lo == hi) return array[lo]==toFind?0:1
        else
                int mid = (lo+hi)/2
                return countNumber(array,toFind,lo,mid) +
                        countNumber(array,toFind,mid,hi)
```

What is the recurrence here? $T(n) = O(1) + 2\,T(n/2)$

ANALYSIS

- Graphically count the operations using what is called a recurrence tree

ANALYSIS

- Graphically count the operations using what is called a recurrence tree
- Each “node” is the work done, and each of the children are their own nodes
- Calculate the work going throughout.