O Notation Examples

O Notation Categories

- $O(1)$ – *constant* time
  - The time is independent of n
- $O(\log n)$ – *logarithmic* time
  - Usually the log is to the base 2
- $O(n)$ – *linear* time
- $O(n*\log n)$
- $O(n^2)$ – *quadratic* time
- $O(n^k)$ – *polynomial* (where k is some constant)
- $O(2^n)$ – *exponential* time

Maximum Value in an Array

```
// PRE: arr is sorted
int maxSorted(int arr[], int n){
    return arr[n-1];
}
```

$O(1)$

Maximum Value in an Array

```
int max(int arr[], int n){
    int maximum = arr[0];
    for (int i=0; i < n; ++i){
        if arr[i] > maximum {
            maximum = arr[i];
        }
    }
    return maximum;
}
```

$O(n)$

Loops

- What is the difference between the two *max* functions?
  - The first always looks at the last element of the array
    - Arrays support random access so the time it takes to retrieve this value is not dependent on the array size
  - The second contains a for loop
- The for loop in the *max* function iterates n times
  - The loop control variable starts at 0, goes up by 1 for each loop iteration and the loop ends when it reaches n
- If a function contains a for loop is it always $O(n)$?
  - Not necessarily

Mean By Sampling

```
float approximateMean(int arr[], int n){
    float sum = 0;
    for (int i=0; i < n; i+=10){
        sum += arr[i];
    }
    return sum / (n / 10.0);
}
```

$T_A = 0.3n + 3$

$O(n)$

Binary Search

```
bool search(int arr[], int n, int x){
    int low = 0;
    int high = n - 1;
    int mid = 0;
    while (low <= high){
        mid = (low + high) / 2;
        if(x > arr[mid]){
            low = mid + 1;
        } else if(x < arr[mid]) {
            high = mid - 1;
        }
        else { // x == arr[mid]
            return true;
        }
    } //while
    return false;
}
```

$O(\log(n))$

Average and worst case

Analyzing Loops

- It is important to analyze how many times a loop iterates
  - By considering how the loop control variable changes through each iteration
- Be careful to ignore constants
  - Consider how the running time would change if the input doubled
  - In an $O(n)$ algorithm the running time will double
  - In a $O(\log(n))$ algorithm it increases by 1

Mean

```
float mean(int arr[], int n){
    float sum = 0;
    for (int i=0; i < n; ++i){
        sum += arr[i];
    }
    return sum / n;
}
```

$O(n)$

Variance

```
int stupidVariance(int arr[], int n)
{
    float result = 0;
    float sqDiff = 0;
    for (int i=0; i < n; ++i){
        sqDiff = arr[i] - mean(arr, n);
        sqDiff *= sqDiff;
        result += sqDiff;
    }
    return result;
}
```

$O(n^2)$

How could this be improved?

Less Stupid Variance

```
float variance(int arr[], int n)
{
	float result = 0;
	float avg = mean(arr, n);
	for (int i=0; i < n; ++i){
		float sqDiff = arr[i] - avg;
		sqDiff *= sqDiff;
		result += sqDiff;
	}
	return result;
}
```

$$T_A = T_{mean} + 5n + 4$$

$$O(n)$$

Bubble

```
void bubble(int arr[], int n)
{
        bool swapped = true;
        while(swapped){
                swapped = false;
                for (int i=0; i < n-1; ++i){
                        if(arr[i] > arr[i+1]){
                                int temp = arr[i];
                                arr[i] = arr[i+1];
                                arr[i+1] = temp;
                                swapped = true;
                        }
                }
        }
}
```

Average and worst case

$O(n^2)$

Best case?

Duplicates

```
bool duplicates(int arr[], int n)
{
    for(int i=0; i < n; ++i){
        for (int j=0; j < n; ++j){
            if(i != j){
                if (arr[i] == arr[j])
                    return true;
            }
        }
    }
    return false;
}
```

In worst case

$O(n^2)$

Best case?

Average case?

Nested Loops

- The (stupid) *variance*, *bubble* and *duplicates* functions contain nested loops
  - Both the inner loops perform O(n) iterations
    - In *variance* the inner loop is contained in a function
  - And the outer loops also perform O(n) iterations
- The functions are therefore O(n^2)
  - Make sure that you check to see how many times both loops iterate

Another Nested Loop

```
int foo(int arr[], int n){
      int result = 0;
      int i = 0;
      while (i < n / 2){
             result += arr[i];
             i += 1;
             while (i >= n / 2 && i < n){
                    result += arr[i];
                    i += 1;
             }
      }
   return result;
}
```

$O(n)$

Alphabetical Order

```
bool alphaOrder(string s){
    int end = s.size() - 1;
    for (int i = 0; i < end; ++i){
        if (s[i] > s[i+1]){
            return false;
        }
    }
    return true;
}
```

Best case - $O(1)$

Average case - ?

Worst case - $O(n)$

Best, Average and Worst Case

- Best case and worst case analysis are often relatively straightforward
  - Although they require a solid understanding of the algorithm's behaviour
- Average case analysis can be more difficult
  - It may involve a more complex mathematical analysis of the function's behaviour
  - But can sometimes be achieved by considering whether it is closer to the worst or best case

Recursive Sum

```
int sum(int arr[], int n, int i){
    if (i == n - 1){
        return arr[i];
    }
    else{
        return arr[i] + sum(arr, n, i + 1);
    }
}
```

Assume there is a calling function that calls *sum*(*arr*, *size*, 0)

$O(n)$

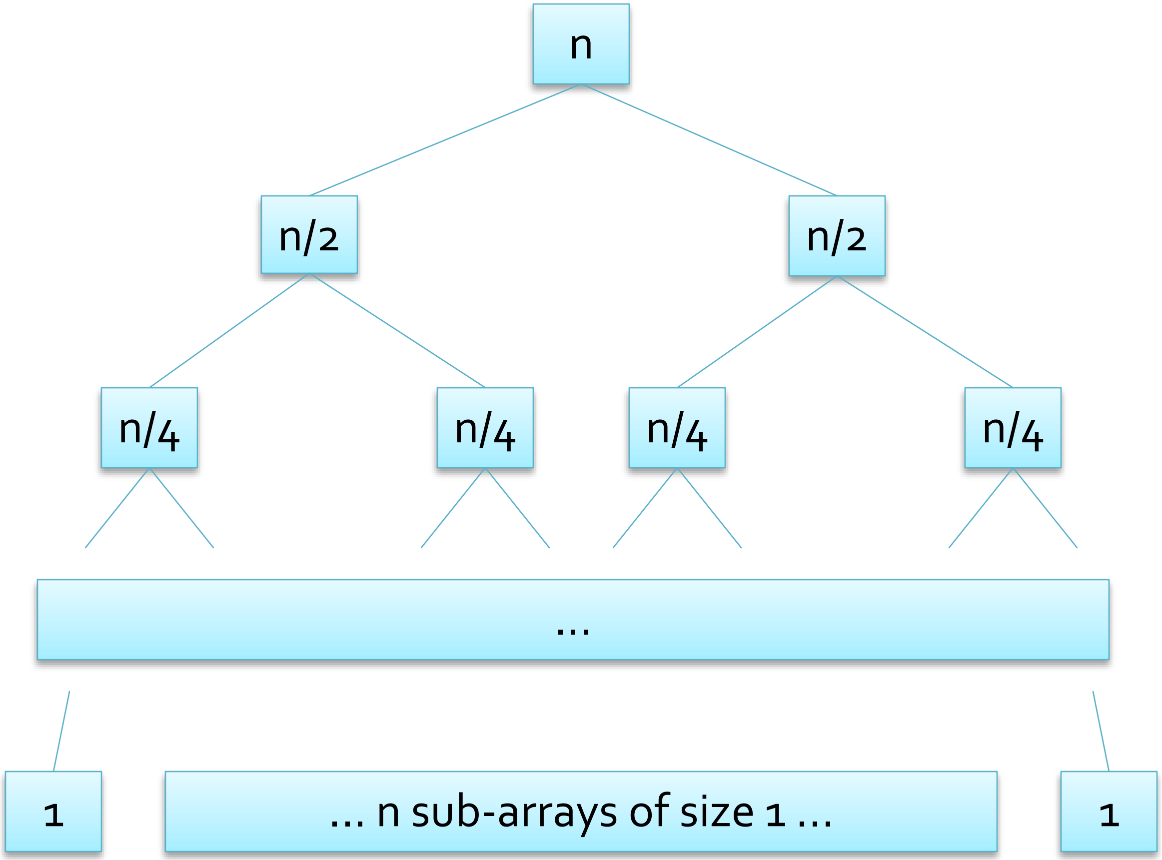
Recursive Functions

- The analysis of a recursive function revolves around the number of recursive calls made
  - And the running time of a single recursive call
- In the *sum* example the amount of a single function call is constant
  - It is not dependent on the size of the array
  - One recursive call is made for each element of the array

Quicksort Analysis

- One way of analyzing a recursive algorithm is to draw a tree of the recursive calls
  - Determine the depth of the tree
  - And the running time of each level of the tree
- In Quicksort the partition algorithm is responsible for partitioning sub-arrays
  - That at any level of the recursion tree make up the entire array when aggregated
  - Therefore each level of the tree entails O(*n*) work

Quicksort Best Case



At each level the partition process performs roughly *n* operations, how many levels are there?

At each level the sub-array size is half the size of the previous level

O(log(n)) levels

Multiply the work at each level by number of levels

O(n * log(n))

Quicksort Worst Case

At each level the partition process performs roughly *n* operations, how many levels are there?

At each level the sub-array size is one less than the size of the previous level

O(n) levels

Multiply the work at each level by levels

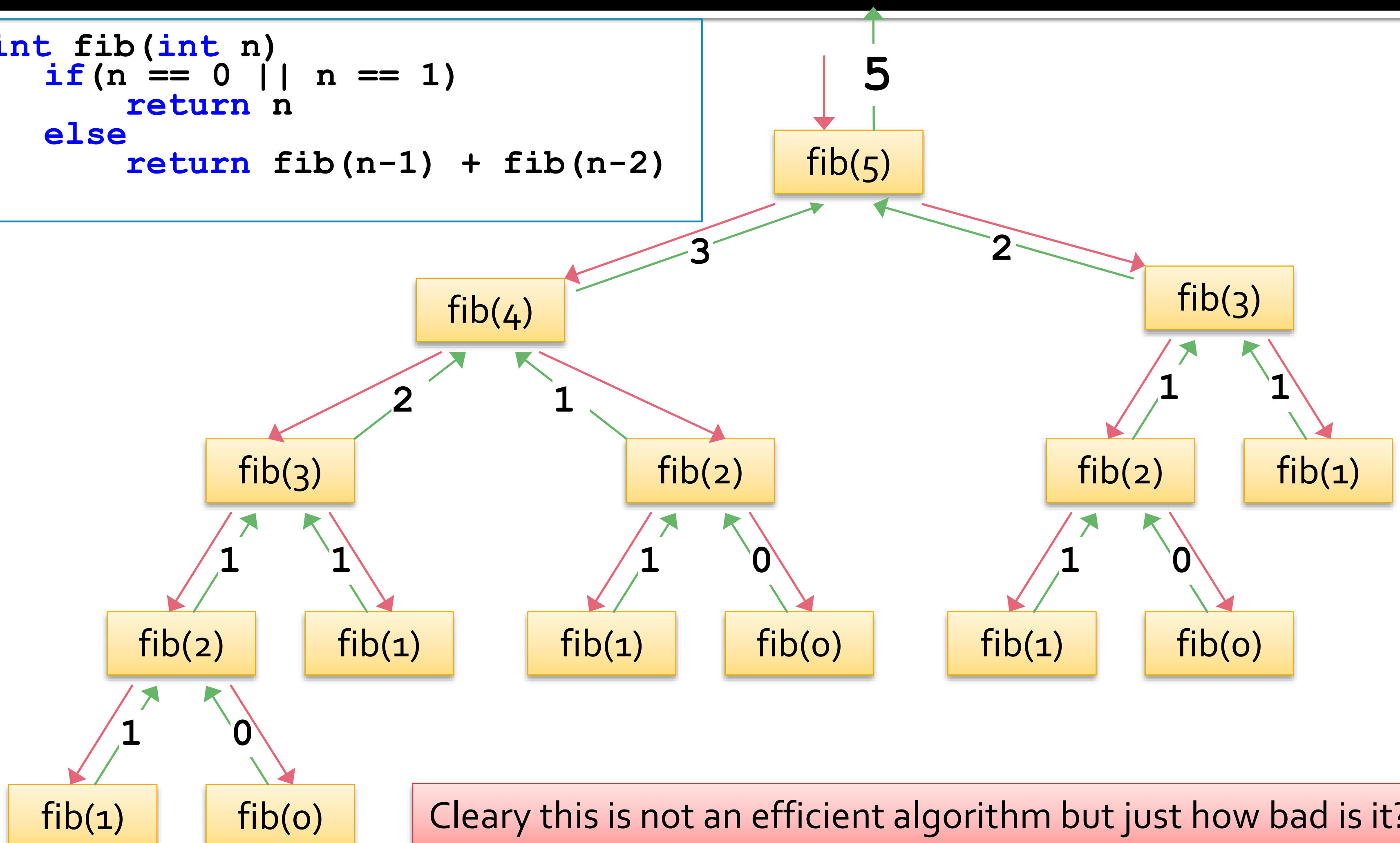
$O(n^2)$

Recursive Fibonacci Analysis

- The running time of the recursive Fibonacci function we looked at was painfully slow
  - But just how bad was it?
  - Let's consider a couple of possible running times
    - $O(n^2)$
    - $O(2^n)$
- We will use another tool to reason about the running time
  - Induction

Analysis of fib(5)

```
int fib(int n)
   if(n == 0 || n == 1)
       return n
   else
       return fib(n-1) + fib(n-2)
```



Cleary this is not an efficient algorithm but just how bad is it?

Fibonacci Analysis - 1

- Let's assume that it is $O(n^2)$
  - Although this isn't supported by the recursion tree
- Base case – $T(n \leq 1) = O(1)$
  - True, since only 2 operations are performed
- Inductive hypothesis: $T(n\text{-}1) = (n\text{-}1)^2$
- Inductive proof – prove that $T(n) = n^2$ given hypothesis
  - we claim that: $n^2 \geq (n\text{-}1)^2 + (n\text{-}2)^2$
  - $n^2 \geq (n^2 - 2n + 2) + (n^2 - 4n + 4)$
  - $n^2 \geq 2n^2 - 6n + 6$
  - But $2n^2 - 6n + 6 > n^2$, the inductive hypothesis is **not** proven

Fibonacci Analysis - 2

- Let's assume that it is $O(2^n)$
- Base case – $T(n \leq 1) = O(1)$
  - True, since only 2 operations are performed
- Inductive hypothesis: $T(n\text{-}1) = 2^{n-1}$
- Inductive proof – prove that $T(n) = 2^n$
  - $2^n \geq 2^{n-1} + 2^{n-2}$
  - Since $2^n = 2^{n-1} + 2^{n-1}$, 2^n *is* greater than $2^{n-1} + 2^{n-2}$
  - The inductive hypothesis is proven