Recursion

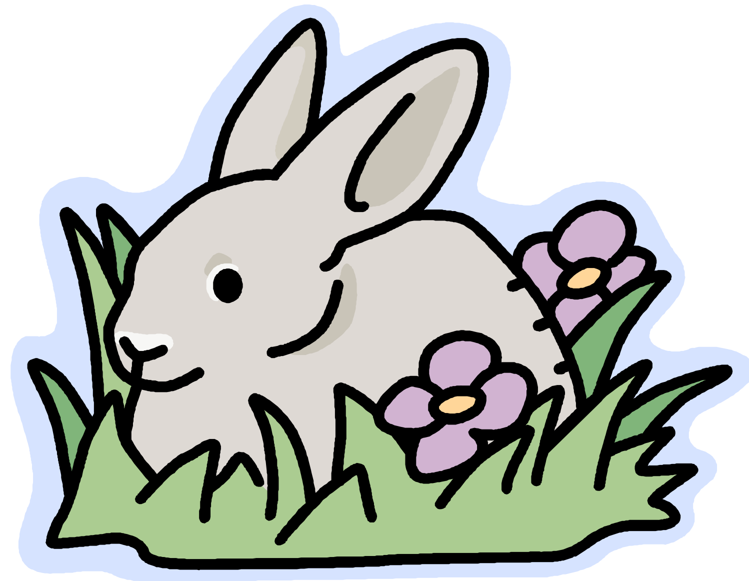
CMPT 225

Objectives

- Understand how the Fibonacci series is generated
- Recursive Algorithms
  - Write simple recursive algorithms
  - Analyze simple recursive algorithms
  - Understand the drawbacks of recursion
- Name other recursive algorithms and data structures

Bunnies

- What happens if you put a pair of rabbits in a field?
  - More rabbits!
- Assume that rabbits take one month to reach maturity and that
- Each pair of rabbits produces another pair of rabbits one month after mating.



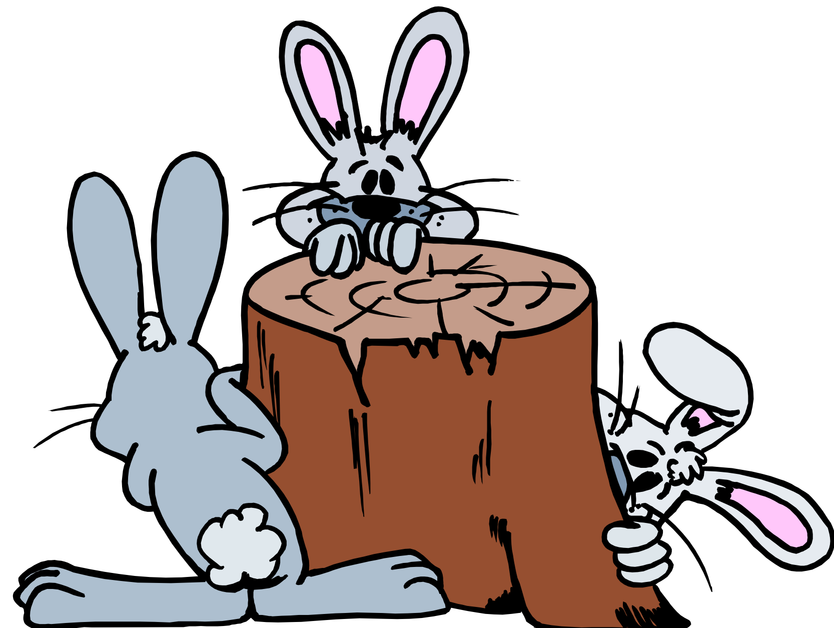
… and more Bunnies

- How many pairs of rabbits are there after 5 months?
  - Month 1: start – **1**
  - Month 2: the rabbits are now mature and can mate – **1**
  - Month 3: – the first pair give birth to two babies – **2**
  - Month 4: the first pair give birth to 2 babies, the pair born in month 3 are now mature – **3**
  - Month 5: the 3 pairs from month 4, and 2 new pairs – **5**

… and even more Bunnies

- After 5 months there are 5 pairs of rabbits
  - i.e. the number of pairs at 4 months (3) plus the number of pairs at 3 months (2)
  - Why?
- While there are 3 pairs of bunnies in month 4 only 2 of them are able to mate
  - the ones alive in month 3
- This series of numbers is called the Fibonacci series

month:	1	2	3	4	5	6
pairs:	1	1	2	3	5	8



Fibonacci Series

- The **n**th number in the Fibonacci series, **fib**(**n**), is:
  - 0 if **n** = 0, and 1 if **n** = 1
  - **fib**(**n** – 1) + **fib**(**n** – 2) for any **n** > 1
- e.g. what is **fib**(23)
  - Easy if we only knew **fib**(22) and **fib**(21)
  - The answer is **fib**(22) + **fib**(21)
  - What happens if we actually write a function to calculate Fibonacci numbers like this?

Calculating the Fibonacci Series

C++

- Let's write a function just like the formula
  - fib(n) = 0 if n = 0, 1 if n = 1,
  - otherwise fib(n) = fib(n – 1) + fib(n – 2)

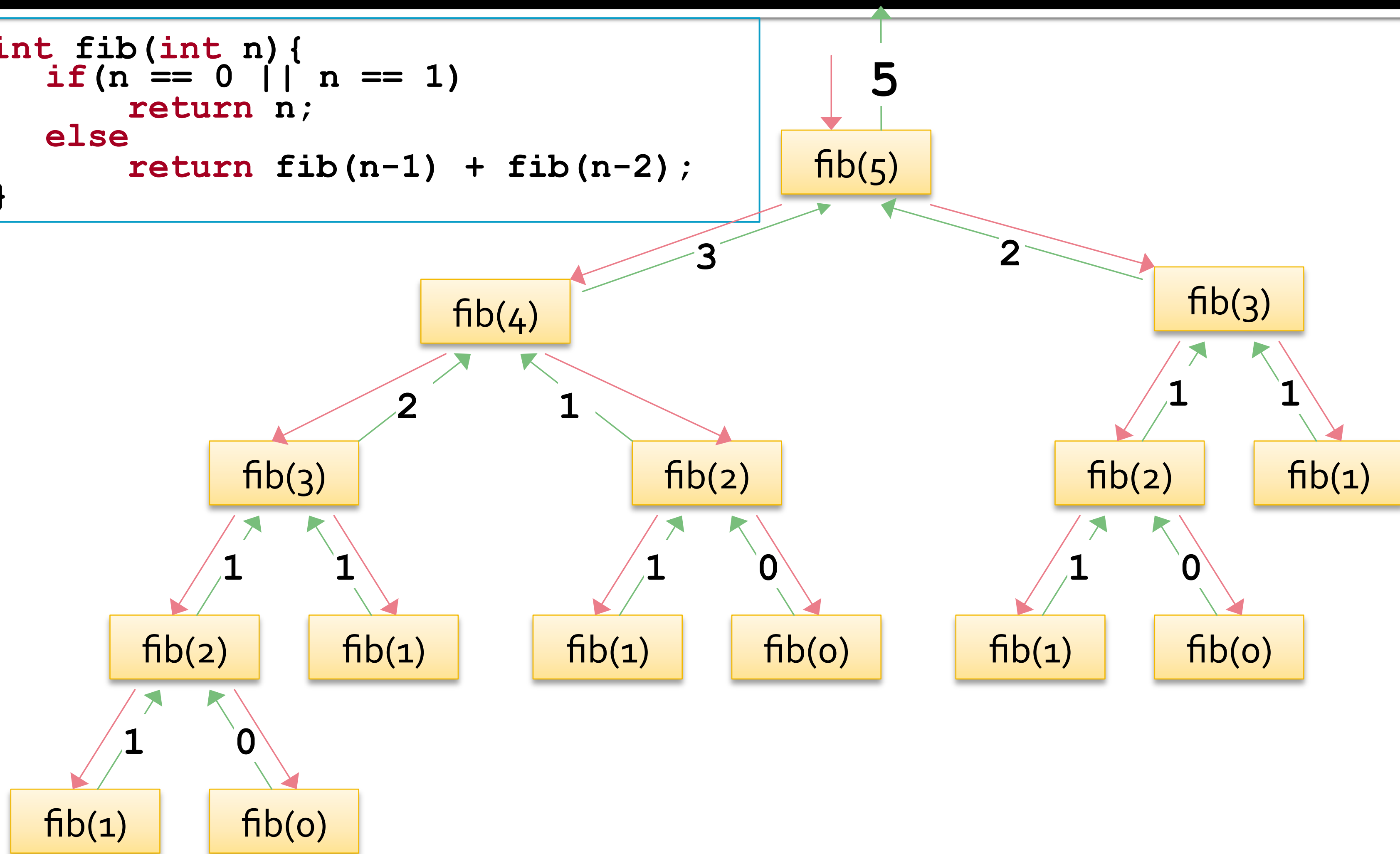
```
int fib(int n){
     if(n == 0 || n == 1){
          return n;
     }else{
          return fib(n-1) + fib(n-2);
     }
}
```

The function calls itself

Recursive Functions

- The Fibonacci function is **recursive**
  - A recursive function calls itself
  - Each call to a recursive method results in a *separate* call to the method, with its own input
- Recursive functions are just like other functions
  - The invocation is pushed onto the call stack
  - And removed from the call stack when the end of a method or a return statement is reached
  - Execution returns to the previous method call

Analysis of fib(5)



Recursive Functions – the Stack

- When a function is called it is pushed onto the call stack
  - This applies to each invocation of that function
- When a recursive call is made execution switches to **that** method call
  - The call stack records the line number of the previous method where the call was made from
  - Once a method call execution finishes, returns to the previous invocation

Recursion for Problem Solving

Recursive Function Anatomy

- Recursive functions do not use loops to repeat instructions
  - But use recursive calls, in if statements
- Recursive functions consist of two or more cases, there must be at least one
  - Base case, and one
  - Recursive case

Base Case

- The base case is a smaller problem with a simpler solution
  - This problem's solution must **not** be recursive
    - Otherwise the function may never terminate
- There can be more than one base case

Recursive Case

- The recursive case is the same problem with smaller input
  - The recursive case must include a recursive function call
  - There can be more than one recursive case

Finding Recursive Solutions

- Define the problem in terms of a smaller problem of the same type
  - The recursive part
  - e.g. `return fib(n-1) + fib(n-2);`
- And the base case where the solution can be easily calculated
  - This solution should not be recursive
  - e.g. `if (n == 0 || n == 1) return n;`

Steps Leading to Recursive Solutions

- How can the problem be defined in terms of smaller problems of the same type?
  - By how much does each recursive call reduce the problem size?
  - By 1, by half, …?
- What are the base cases that can be solved without recursion?
  - Will a base case be reached as the problem size is reduced?

Recursive Searching

Recursive Searching

- Linear Search
- Binary Search
  - Assume sorted array

Linear Search Algorithm

C++

```
int linSearch(int *arr, int n, int x){
    for (int i=0; i < n; i++){
        if(x == arr[i]){
            return i;
        }
    } //for
    return -1; //target not found
}
```

The algorithm searches the array one element at a time using a for loop

Recursive Linear Search

- Base cases
  - Target is found at first position in array
  - The end of the array is reached
- Recursive case
  - Target not found at first position
    - Search again, discarding the first element of the array

Recursive Linear Search

C++

```
int linSearch(int *arr, int n, int x){
    return recLinSearch(arr,n,0,x);
}
int recLinSearch(int *arr, int n, int i, int x){
    if (i >= n){
        return -1;
    } else if (x == arr[i]){
        return i;
    } else
        return recLinSearch(arr, n, i + 1, x);
    }
}
```

Binary Search

- Of course, if it's a sorted array we wouldn't do linear search

Thinking About Binary Search

- Each sub-problem searches a subarray
  - Differs only in the upper and lower array indices that define the subarray
  - Each sub-problem is smaller than the last one
  - In the case of binary search, half the size
- There are two base cases
  - When the target item is found and
  - When the problem space consists of one item
    - Make sure that this last item is checked

Recursive Binary Search

C++

```
int binSearch(int *arr, int lower, int upper,
       int x){

    int mid = (lower + upper) / 2;
    if (lower > upper){
        return - 1; //base case
    } else if(arr[mid] == x){
        return mid; //second base case
    } else if(arr[mid] < x){
         return binSearch(arr, mid + 1, upper, x);
    } else { //arr[mid] > target
         return binSearch(arr, lower, mid - 1, x);
    }
}
```

Recursive Sorting

Recursive Searching

- Merge Sort
- Quicksort

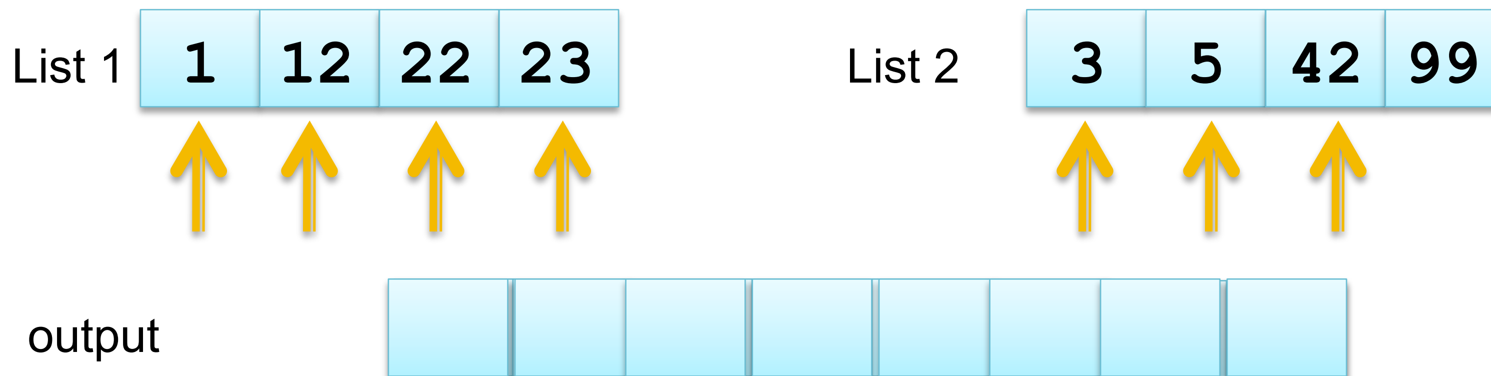
Merge Sort

Merge Sort

- What's the easiest list to sort?
  - A list of 1 number

Merging sorted lists

- Let's say I have 2 sorted lists of numbers
  - How can I merge them into 1 sorted list?



Merge Sort

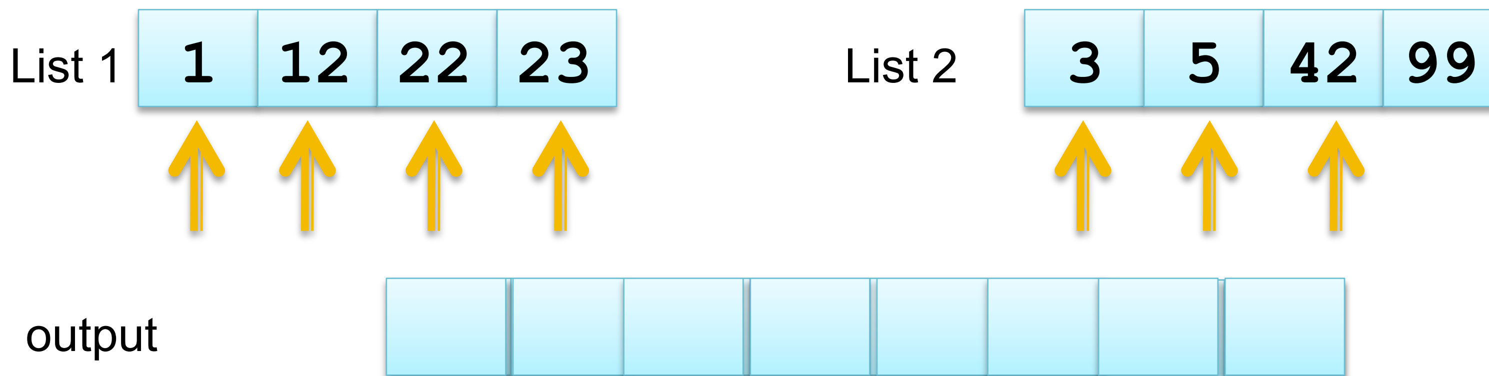
- If I have a list of n numbers, how should I sort them?
- I know two things
  - How to sort a list of 1 number
  - How to merge 2 sorted lists of numbers into 1 sorted list
- Smells like recursion

Merge Sort Pseudocode

```
mergeSort (array)
    if (array is length 1)
        // base case, one element
        return the array
    else
        arr1 = mergeSort(first half of array)
        arr2 = mergeSort(second half of array)
        return merge(arr1,arr2)
```

Merge Sort Analysis

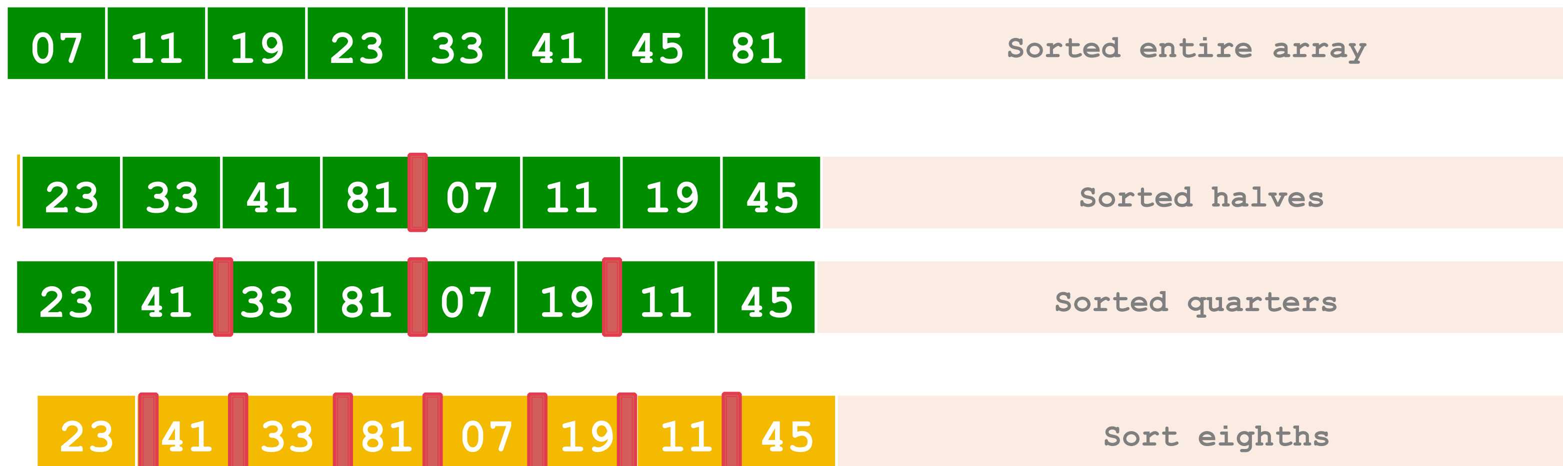
- What is the time complexity of a merge?



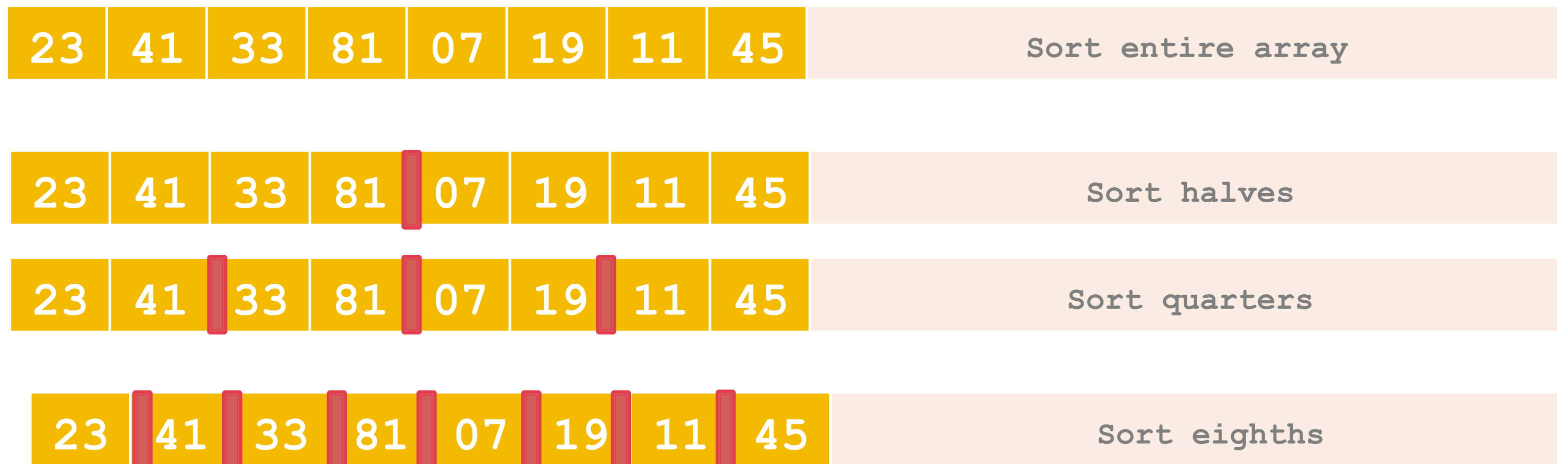
Merge Sort Analysis

- How many recursive steps are there?
- How large are the merges at each recursive step?
  - Merge takes O(n) time for n elements

Merge Sort Recursion

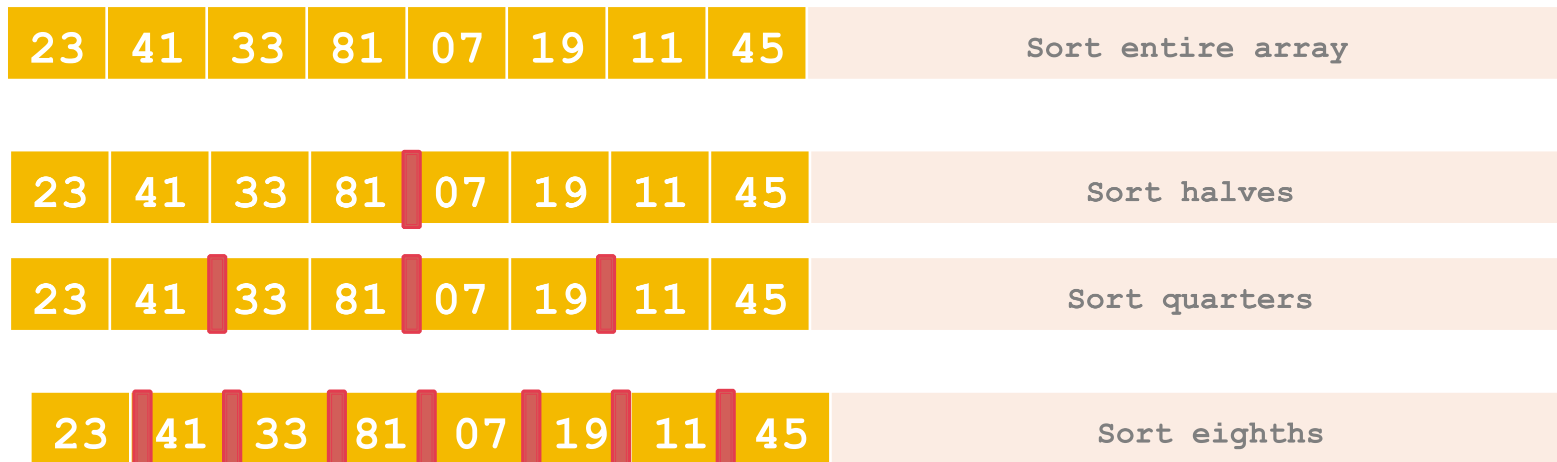


Merge Sort Recursion



- How many recursive steps are there?
- How large are the merges at each recursive step?
  - Merge takes O(n) time for n elements

Merge Sort Recursion



- How many recursive steps are there?
  - O(log n) steps: split array in half each time
- How large are the merges at each recursive step?
  - In total, merge n elements each step
- Time complexity is O(n log n)

O Notation Running Times

- Mergesort
  - Best case: $O(n(\log_2 n))$
  - Average case: $O(n(\log_2 n))$
  - Worst case: $O(n(\log_2 n))$

Introduction to QuickSort

QuickSort Introduction

- Quicksort is a more efficient sorting algorithm than either selection or insertion sort
  - It sorts an array by repeatedly **partitioning** it
- We will go over the basic idea of quicksort and an example of it
  - See text / on-line resources for details

Partitioning

- Partitioning is the process of dividing an array into sections (partitions), based on some criteria
  - "Big" and "small" values
  - Negative and positive numbers
  - Names that begin with **a-m**, names that begin with **n-z**
  - Darker and lighter pixels
- Quicksort uses repeated partitioning to sort an array

Partitioning an Array

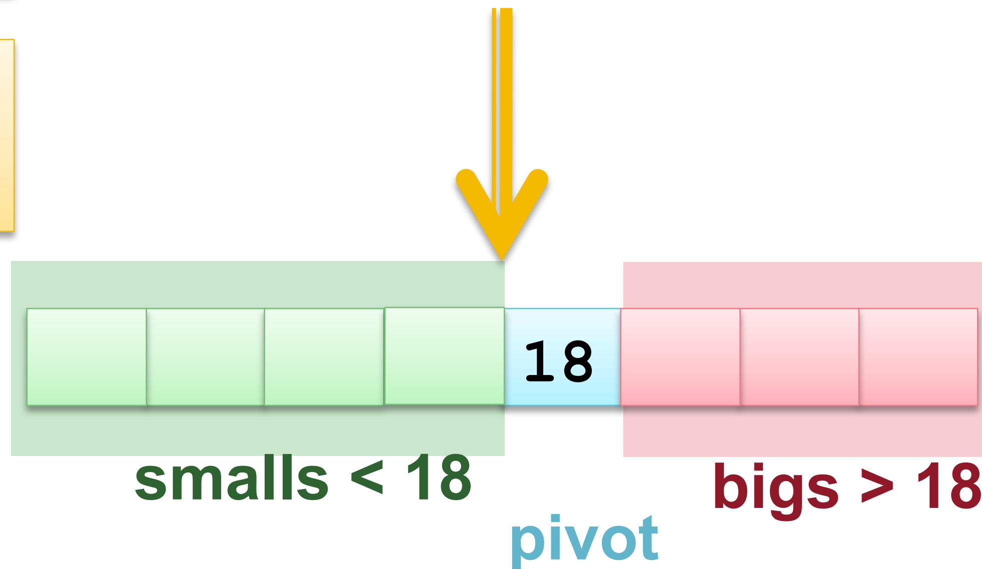
Partition this array into **small** and **big** values using a partitioning algorithm

31	12	07	23	93	02	11	18

Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

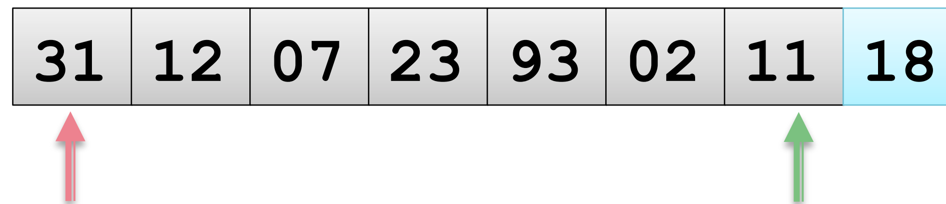


Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high



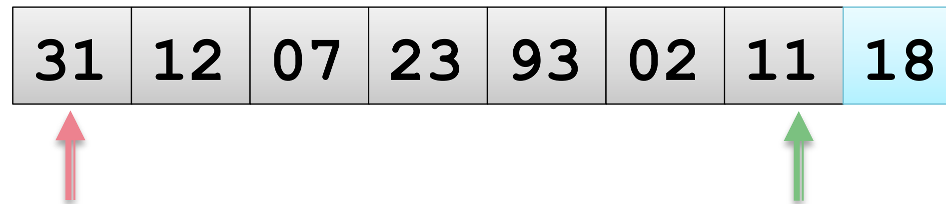
arr[low] is greater than the pivot and should be on the right, we need to swap it with something

Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high



arr[low] (31) is greater than the pivot and should be on the right, we need to swap it with something

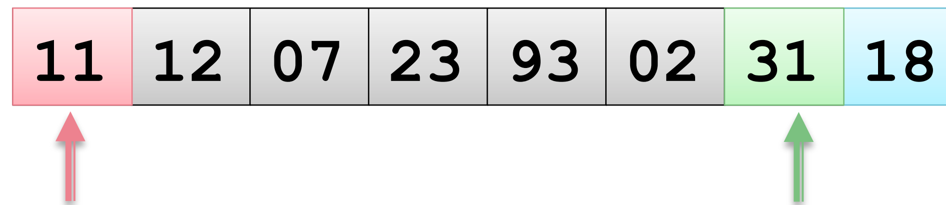
arr[high] (11) is less than the pivot so swap with arr[low]

Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high

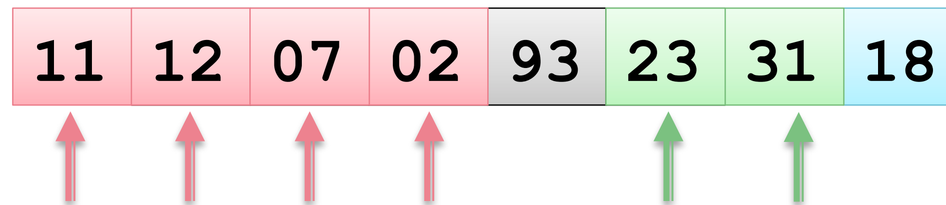


Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high



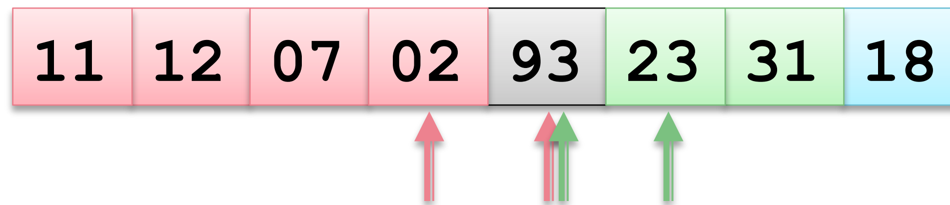
repeat this process until:

Partitioning Algorithm

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high



repeat this process until:

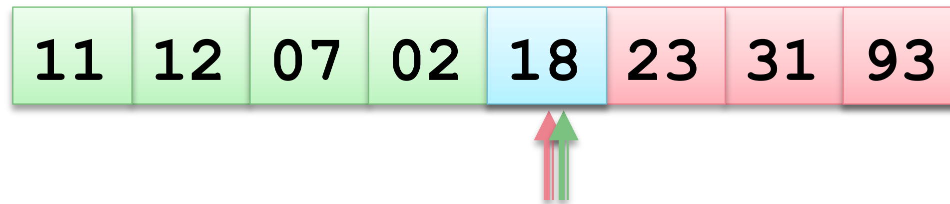
high and low are the same

Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them **low** and **high**



repeat this process until:

high and low are the same

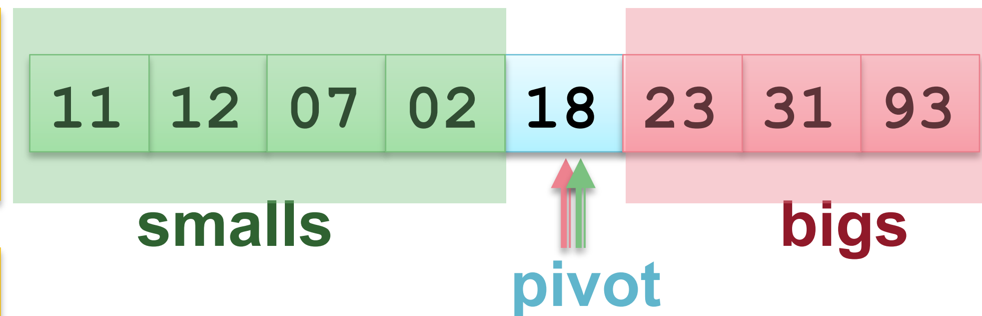
We'd like the pivot value to be in the centre of the array, so we will swap it with the first item greater than it

Partitioning an Array

Partition this array into **small** and **big** values using a partitioning algorithm

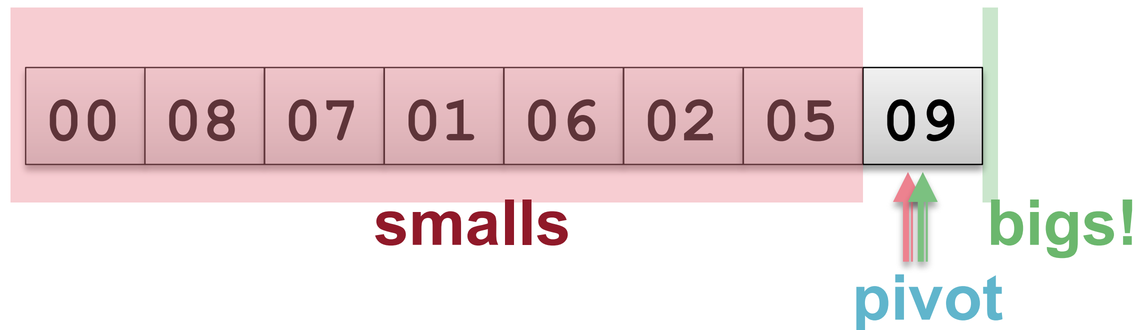
We will partition the array around the last value (18), we'll call this value the **pivot**

Use two indices, one at each end of the array, call them low and high



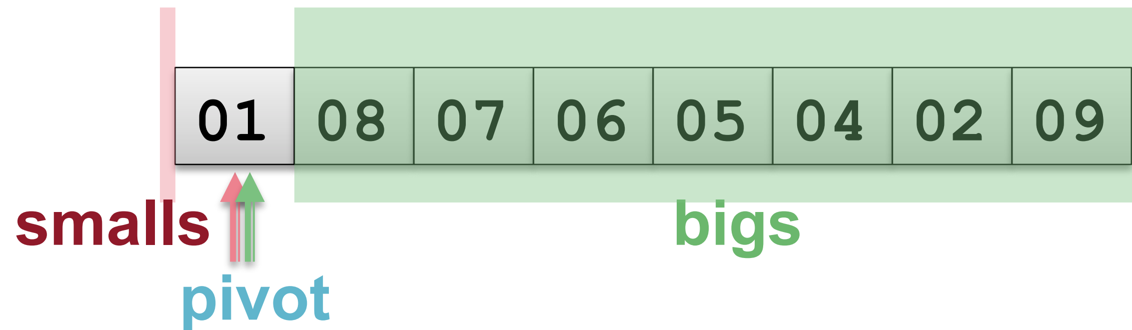
Partitioning Question

Use the same algorithm to partition this array into small and big values



Partitioning Question

Or this one:



Quicksort

- The quicksort algorithm works by *repeatedly partitioning* an array
- Each time a subarray is partitioned there is
  - A sequence of **small** values,
  - A sequence of **big** values, and
  - A **pivot** value **which is in the correct position**
- Partition the small values, and the big values
  - Repeat the process until each subarray being partitioned consists of just one element

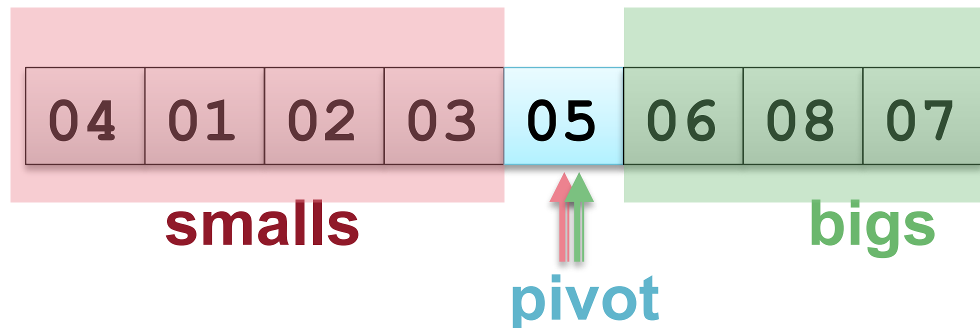
Quicksort Analysis

- How long does quicksort take to run?
  - Let's consider the best and the worst case
  - These differ because the partitioning algorithm may not always do a good job
- Let's look at the best case first
  - Each time a subarray is partitioned the pivot is the exact midpoint of the slice (or as close as it can get)
    - So it is divided in half
  - What is the running time?

Quicksort Best Case

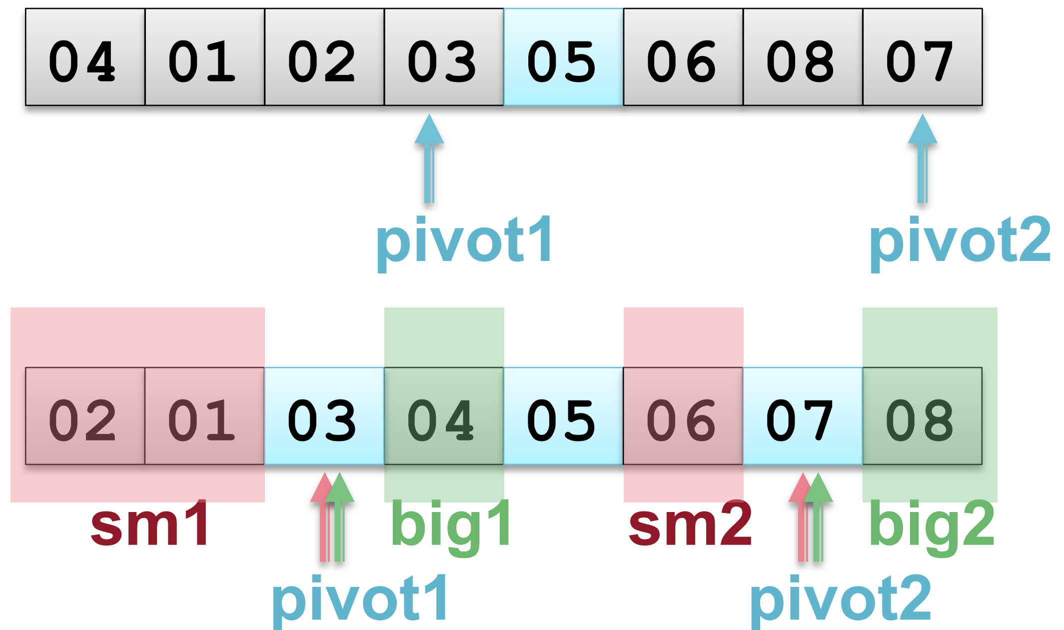
08	01	02	07	03	06	04	05

First partition



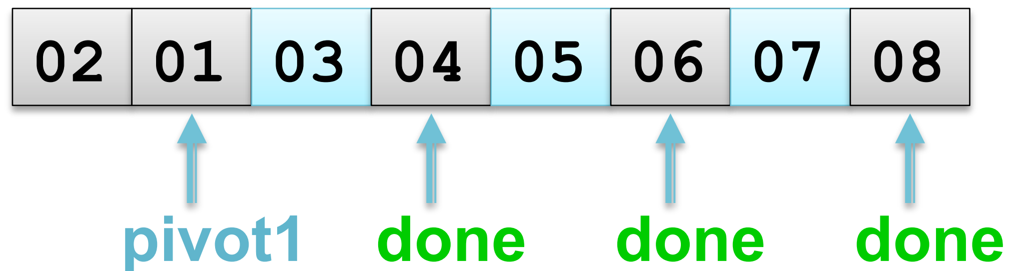
Quicksort Best Case

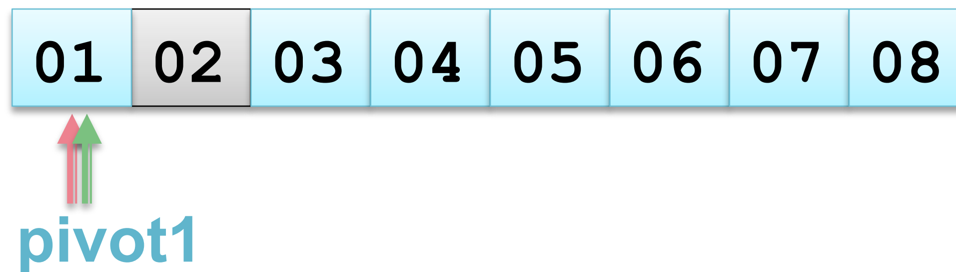
Second partition



Quicksort Best Case

Third partition





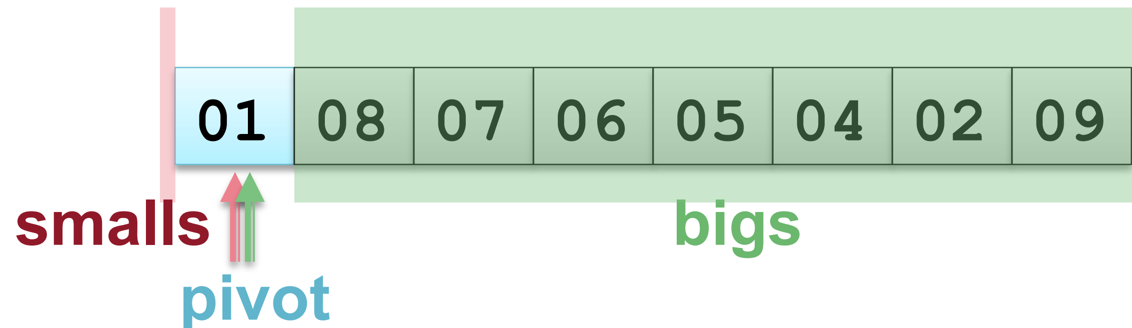
Quicksort Best Case

- Each subarray is divided exactly in half in each set of partitions
  - Each time a series of subarrays are partitioned around **n** comparisons are made
  - The process ends once all the subarrays left to be partitioned are of size 1
- How many times does **n** have to be divided in half before the result is 1?
  - $\log_2(n)$ times
  - Quicksort performs around $n * \log_2(n)$ operations in the best case

Quicksort Worst Case

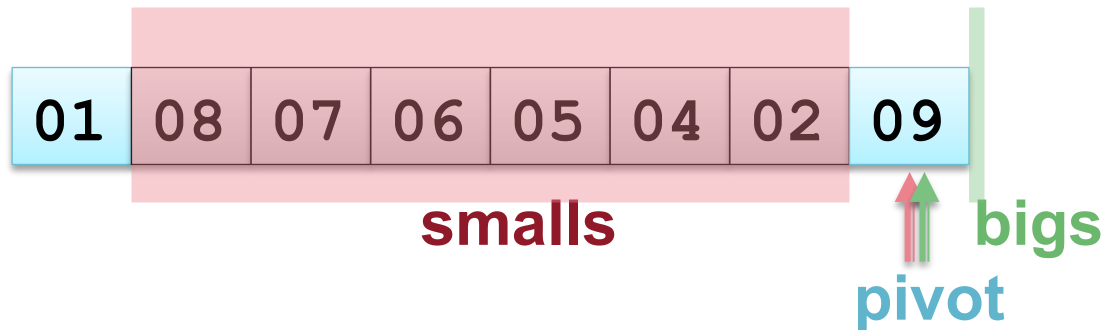
09	08	07	06	05	04	02	01

First partition



Quicksort Worst Case

Second partition

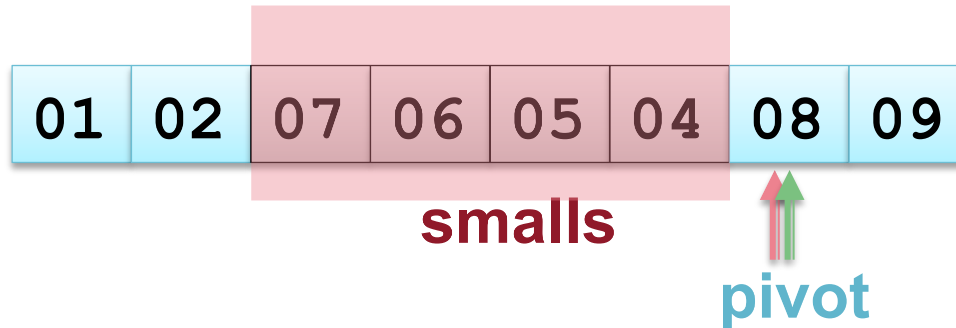


Quicksort Worst Case

Third partition

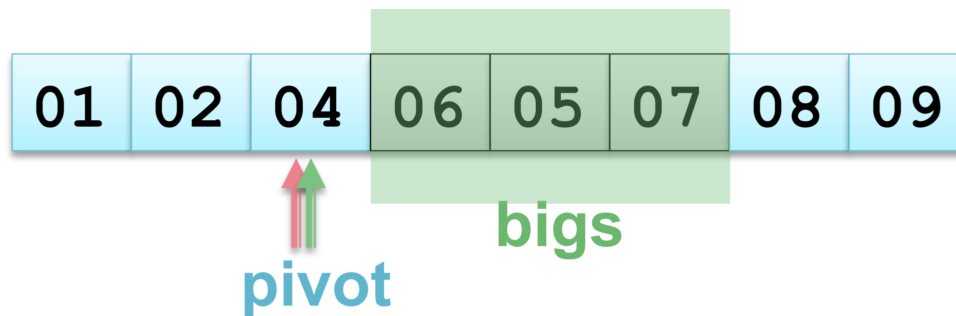
Quicksort Worst Case

Fourth partition



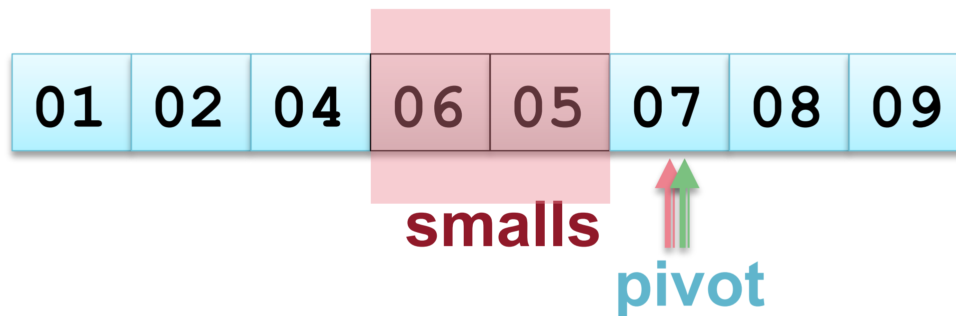
Quicksort Worst Case

Fifth partition



Quicksort Worst Case

Sixth partition



Quicksort Worst Case

Seventh(!) partition

pivot

Quicksort Worst Case

- Every partition step results in just one partition on one side of the pivot
  - The array has to be partitioned **n** times, not $\log_2(\mathbf{n})$ times
  - So in the worst case quicksort performs around $\mathbf{n}^2$ operations
- The worst case usually occurs when the array is nearly sorted (in either direction)

Quicksort Average Case

- With a large array we would have to be very, very unlucky to get the worst case
  - Unless there was some reason for the array to already be partially sorted
    - In which case first randomize the position of the array elements!
  - The average case is much more like the best case than the worst case

Recursion Pitfalls

Recursion Caveat

- Recursive algorithms have more overhead than similar iterative algorithms
  - Because of the repeated method calls
  - This may cause a **stack overflow** when the call stack gets full
- It is often useful to derive a solution using recursion and implement it iteratively
  - Sometimes this can be quite challenging!

Another Recursion Caveat

- Some recursive algorithms are inherently inefficient
  - e.g. the recursive Fibonacci algorithm which repeats the same calculation again and again
  - Look at the number of times `fib(2)` is called
- Such algorithms should be **implemented** iteratively
  - Even if the solution was **determined** using recursion

Analyzing Recursive Functions

- It is useful to trace through the sequence of recursive calls
  - This can be done using a recursion tree
- Recursion trees can be used to determine the running time of algorithms
  - Annotate the tree to indicate how much work is performed at each level of the tree
  - And then determine how many levels of the tree there are

Recursion and Induction

Recursion and Induction

- Recursion is similar to induction
- Recursion *solves* a problem by
  - Specifying a solution for the base case and
  - Using a recursive case to derive solutions of any size from solutions to smaller problems
- Induction *proves* a property by
  - Proving it is true for a base case and
  - Proving that it is true for some number, **n**, if it is true for all numbers less than **n**

Recursive Factorial

C++

```
int fact (int x){
    if (x == 0){
        return 1;
    } else
        return n * fact(n – 1);
    }
}
```

- Prove, using induction that the algorithm returns the values
  - **fact**(0) = 0! =1
  - **fact**(n) = **n**! = **n** * (**n** – 1) * … * 1 if **n** > 0

Proof by Induction

- **Basis:** Show that the property is true for **n** = 0, i.e. that **fact**(0) returns 1
  - This is true by definition as **fact**(0) is the base case of the algorithm and returns 1
- Establish that the property is true for an arbitrary **k** implies that it is also true for **k** + 1
- **Inductive hypothesis:** Assume that the property is true for **n** = **k**, that is assume that
  - **fact**(**k**) = **k** * (**k** – 1) * (**k** – 2) * … * 2 * 1

Proof by Induction

- **Inductive conclusion:** Show that the property is true for **n** = **k** + 1, i.e., that **fact**(**k** + 1) returns
  - (**k** + 1) * **k** * (**k** – 1) * (**k** – 2) * … * 2 * 1
- By definition of the function: **fact**(**k** + 1) returns
  - (**k** + 1) * **fact**(**k**) – the recursive case
- And by the inductive hypothesis: **fact**(**k**) returns
  - **k** * (**k** – 1) * (**k** – 2) * … * 2 * 1
- Therefore **fact**(**k** + 1) *must* return
  - (**k** + 1) * **k** * (**k** – 1) * (**k** – 2) * … * 2 * 1
- Which completes the inductive proof

More Recursive Algorithms

- Recursive sum
- Towers of Hanoi – see text
- Eight Queens problem – see text
- Sorting
  - Mergesort
  - Quicksort

Recursive Data Structures

- Linked Lists are recursive data structures
  - They are defined in terms of themselves
- There are recursive solutions to many list methods
  - List traversal can be performed recursively
  - Recursion allows elegant solutions of problems that are hard to implement iteratively
    - Such as printing a list backwards

Summary

Summary

- Recursion as a problem-solving tool
  - Identify base case where solution is simple
  - Formulate other cases in terms of smaller case(s)
- Recursion is not always a good implementation strategy
  - Solve the same problem many times
  - Function call overhead
- Recursion and induction
  - Induction proves properties in a form similar to how recursion solves problems

Readings

- Carrano Ch. 2, 5