# Recursion

CS211
Fall 2000

---

## Recursive Functions

- A recursive function is a function that is defined in terms of itself
- In practice:
  - The result for “big” inputs is defined in terms of results for smaller inputs
  - The results for the smallest inputs are defined independently — the Base Case(s)
- Trivial example:
  - n! = n times (n-1)!
  - Base case: 0! = 1

---

## Simple Recursive Programs

- Most programming languages allow recursion (Algol 60 was one of the first)
- Note the base cases
  - Without a base case, a recursive program runs forever

```
int factorial (int n) {
    if (n == 0) return 1;          // Base Case
    return n*factorial(n-1);
    }
```

```
int fibonacci (int n) {
    if (n <= 1) return 1;          // Base Case
    return fibonacci(n-1)+fibonacci(n-2);
    }
```

---

## Example: Squaring without Multiply

- $n^2 = (n-1)^2 + 2n - 1$

```
int square (int n) {
    if (n == 0) return 0;
    return square(n-1) + n + n – 1;
    }
```

- All the examples so far are trivial
- They can be done more efficiently by using a loop
- Any recursive program can be converted to an iterative program that performs the same computation
- But, many program are easier to write (and maintain) if they are written recursively

---

## Two Major Uses of Recursion

Divide & Conquer Algorithms

- Divide & Conquer is an algorithm-design technique
  - It uses recursion
- The resulting algorithms are
  - relatively easy to design and
  - relatively easy to analyze

Recursive Descent Parsing

- Parse: to divide language into small components that can be analyzed
- In Computer Science
  - Compilers must parse source code to be able to translate it into object code
  - Any application that processes commands must be able to parse the commands

---

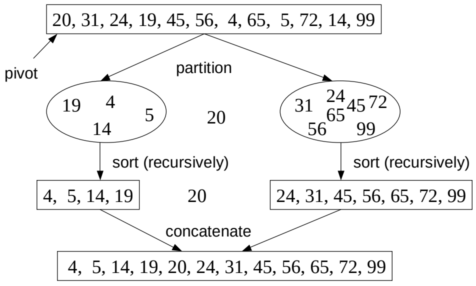
## QuickSort: Recursion on Arrays

QuickSort is based on Divide & Conquer

Intuitive idea:

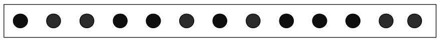
- Given an array A and a *pivot value* p
- Partition A into two subarrays L and R
  - L contains only elements less than or equal to p
  - R contains only elements greater than p
- Sort L and R separately ← Recursion!!!
- Concatenate L and R to produce a sorted result

## QuickSort Example



## QuickSort Advantages

- Fast (fastest known method for most sorting situations)
- Sorting can be done *in-place* (no extra arrays needed)
  - But how can we partition in-place?
  - Simplified version: How do we efficiently get the blue balls all on the left?

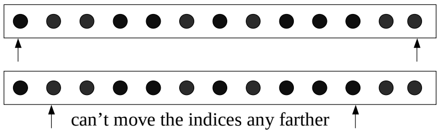


## In-Place Partitioning

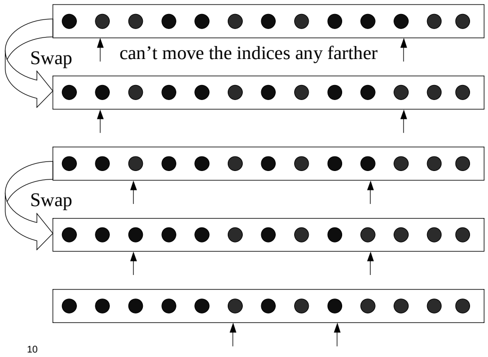
- Keep two indices *i* and *j*
  - Invariant:

    all balls to left of *i* are blue

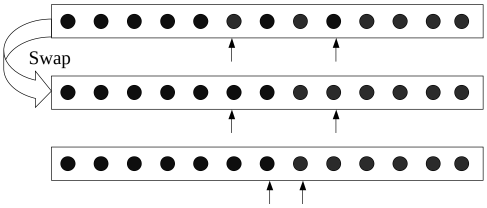
    all balls to right of *j* are red
  - Move the two indices as long as invariant holds



## Swapping to Maintain Invariant



## Done When Indices Cross



- Once indices cross, partitioning is done
- For QuickSort
  - blue = less than or equal to pivot
  - red = greater than or equal to pivot

## What Should We Use for the Pivot?

- Ideal: Choose the median
  - Splits set exactly in half
  - But *way* too hard to find
- Poor strategies
  - Choose leftmost element or choose rightmost element
  - Either works well for arrays in random order
  - But both work *very* badly for sorted arrays
- Effective strategies
  - Choose middle element
  - Choose a random element
  - Choose median of leftmost, middle, and rightmost elements

## QuickSort Code

```
static int partition
          (int[ ] A, int low, int high) {
    int pivot = A[(low+high)/2];
    int i = low;
    int j = high;
    while (true) {
          while (i < high && A[i] < pivot)
                    i++;
          while (j > low && A[j] > pivot)
                    j – –;
          if (i < j) swap(A, i++, j – –);
          else break;
          }
    return i;
    }
```

```
static void swap (int[ ] A, int i, int j) {
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
    }

private static void quickSort
          (int[ ] A, int low, int high) {
    if (low < high) {
          int p = partition(A,low,high);
          quickSort(A,low,p – 1);
          quickSort(A,p,high);
          }
    }

public static void quickSort (int[ ] A)
    {quickSort(A,0,A.length – 1);}
```

## What About Equal Elements?

```
static int partition
          (int[ ] A, int low, int high) {
    int pivot = A[(low+high)/2];
    int i = low;
    int j = high;
    while (true) {
          while (i < high && A[i] < pivot)
                    i++;
          while (j > low && A[j] > pivot)
                    j – –;
          if (i < j) swap(A, i++, j – –);
          else break;
          }
    return i;
    }
```

- Elements equal to the pivot element are placed on both sides
- We even swap equal elements
  - Why?
  - What happens if we sort an array with all equal elements?
  - Why would we want to?

## Using *Sentinels* to Improve Partition

```
static int partition
          (int[ ] A, int low, int high) {
    int pivot = A[(low+high)/2];
    if (A[low] > A[high]) swap(A,low,high);
    if (pivot < A[low]) pivot = A[low];
    else if (A[high] < pivot) pivot = A[high];
    // A[low] is <= pivot
    // A[high] is >= pivot
    int i = low; int j = high;
    while (true) {
          do {i++} while (A[i] < pivot);
          do {j – –} while (A[j] > pivot) ;
          if (i < j) swap(A,i,j);
          else break;
          }
    return i;
    }
```

- We can't run off the right end because A[high] is greater than or equal to the pivot (so i has to stop)
- We can't run off the left end because A[low] is less than or equal to the pivot (so j has to stop)
- In general: Anything that speeds up the partition loop speeds up the algorithm

## Improvements to QuickSort

- Can rewrite using Comparable instead of int
  - Allows sorting of any array containing objects that implement the Comparable interface
  - But this QuickSort can't sort an int-array
- It pays to stop the recursion when low and high are "pretty close"
  - If this is done, each element is near its final position
  - It's faster to then sort the *entire array* in a postprocessing step using a simpler sorting method (InsertionSort)

## Some Comments on Recursion

- Tail Recursion
  - Occurs when the *only* recursive call appears just before the return
  - Tail recursion can be easily converted to a loop
  - Some compilers and interpreters do this automatically
- A common error when using recursion is to neglect to establish a base case
- Recursion and Induction are closely related
  - An inductive proof is used to show a recursive algorithm is correct

## A Simple Inductive Proof

<u>Theorem</u> The sum of the first n integers is n(n+1)/2

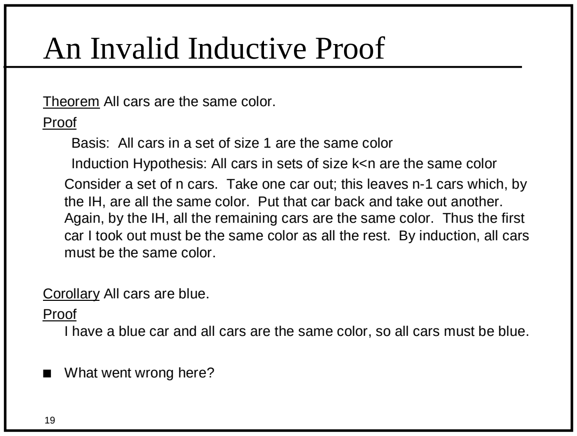
<u>Proof</u>

Basis: The sum of the first 1 integer is 1(1+1)/2.

Induction Hypothesis: The sum of the first k integers is k(k+1)/2 for k<n.

The sum of the first n integers can be written as
[1+…+ n –1] + n.

By the IH for k=n–1, this is the same as
[(n –1)(n)/2] + n
which is equal to n(n+1)/2.



# An Invalid Inductive Proof

Theorem All cars are the same color.

Proof

Basis: All cars in a set of size 1 are the same color

Induction Hypothesis: All cars in sets of size k<n are the same color

Consider a set of n cars. Take one car out; this leaves n-1 cars which, by the IH, are all the same color. Put that car back and take out another. Again, by the IH, all the remaining cars are the same color. Thus the first car I took out must be the same color as all the rest. By induction, all cars must be the same color.

Corollary All cars are blue.

Proof

I have a blue car and all cars are the same color, so all cars must be blue.

- What went wrong here?