EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Permanent URL to TA office hours survey

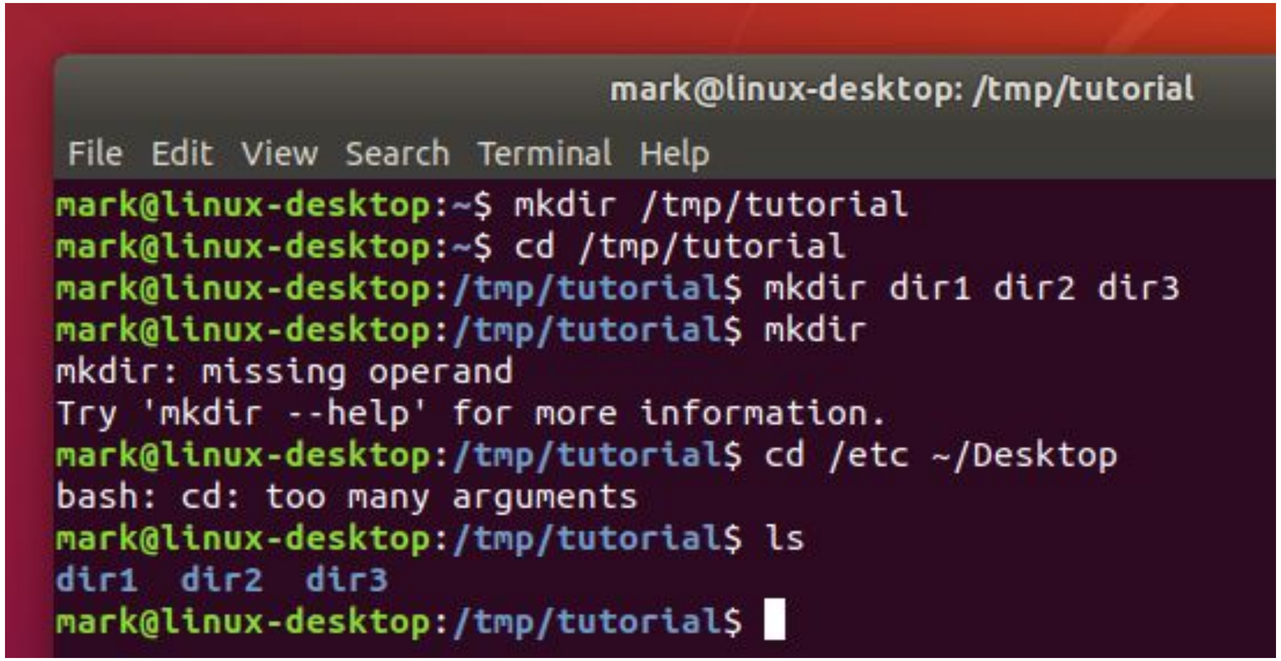
- https://forms.gle/3xWdi4VouxGTaaty9
  - Also on Canvas
- Please respond **each week** by **Thursday 5:00 p** when you plan to attend TA office hours
- Qiangeng will announce the time Thursday night/Friday morning every week
- Location: NEB 401

Linux Command Line & ECE Server

Readings:

- Bare basics
- Detailed introduction

Linux/Unix Command Line *Aka "terminal" or "shell"*



- A key tool for textual interaction with computing systems
- Any Linux distribution (*Ubuntu, Debian, Red Hat,* …) and MacOS has native access to the terminal (typically bash or zsh interpreters)
- Windows has its own command line, needs a program to emulate
  - Cygwin, Windows Subsystem for Linux (WSL), …

Commonly used terminal commands

- cd – change directory
- ls – list directory contents
- mkdir – make directory
- clear – clear the terminal window
- history – show a history of previous used commands
- exit – log out of the terminal
- make – runs commands based on a Makefile (advanced)
- man – shows the manual page of a command
- pwd – shows the path of the current directory
- cp – copy files
- mv – move/rename files

Software for network connection

- **SSH client**
  - Stands for <u>S</u>ecure <u>SH</u>ell
  - Creates a terminal interface to login to a host
  - Use ssh on Linux/Mac
  - Ex. PuTTY – the commonplace Windows client
  - Ex. MobaXTerm – a richer Windows client
- **FTP client**
  - Stands for <u>F</u>ile <u>T</u>ransfer <u>P</u>rotocol
  - Transfers files between two machines
  - Use scp on Linux/Mac
  - Ex. PuTTY or FileZilla

The ECE Linux Server

- Running the Red Hat Linux
- Accessible in two ways:
  - From **NEB 288**
  - Via SSH to linux.ece.ufl.edu
- Follow the detailed instructions (ECE Student Computing Access)
  - Supports remote GUI connection via VNC
- The server compiler version (g++ 4.8.5) and C++'11 will be **our standard C++ baseline**
  - If your code compiles and runs on the ECE server, you're good to submit (if grading issues, *complain to the TA*)
  - You'll need to execute: `source /opt/rh/devtoolset-12/enable`
- I uploaded **demo videos** to *Canvas->Files->Misc*

Basic Sorting:
Bubblesort, Insertion Sort

Readings:

- Cormen 2.1
- Weiss 7.1–7.3 (if familiar with STL; if not, Deitel chapters 7 and 15 explain it)

Sorting Algorithms

Permute elements of an array such that they are ordered

1. Bubblesort
2. Insertion sort
3. Mergesort
4. Quicksort
5. Heapsort
6. Radix sort
7. …

Comparison-based sorts, uses only comparisons (<, ≤, =, ≥, >)

Bogosort

Sorts a list of values by repetitively shuffling them and checking if they are sorted

- Scan the list, seeing if it is sorted
- If not, randomly permute the entries in the array and repeat

What is the worst-case running time of Bogosort?
What is the best-case running time of Bogosort?

Bubblesort

- Compare neighboring elements
- Swap if out of order

6 5 3 1 8 7 2 4

```
for (i=0; i<n-1; i++) {
    for (j=0; j<n-1-i; j++)
     if (a[j+1] < a[j]) {   /* compare neighbors    */
        tmp = a[j];          /* swap a[j] and a[j+1] */
        a[j] = a[j+1];
        a[j+1] = tmp;
  }
}
```

2, 3, 1, 15

2, 1, 3, 15 // after one loop

1, 2, 3, 15 // after second loop

1, 2, 3, 15 // after third loop

What is the worst-case time complexity?
What is the extra space complexity?

Variations of Bubblesort

- Alternate passes start->end and end->start (*Cocktail shaker sort*)
- Stop if any iteration does not cause any swap

C++ Templates

Readings:

- Weiss 1.6
- Stroustrup 23, 24

A Digression: C++ Templates

```
int max (int x, int y)
{
        return (x > y? x : y);
}
```

It only works for the `int` type of variables!

```
template <class T>
T max (T x, T y)
{
        return (x > y? x : y);
}
```

Use function overloading!

```
float max (float x, float y)
{
        return (x > y? x : y);
}

double max (double x, double y)
{
        return (x > y? x : y);
}

char max (char x, char y)
{
        return (x > y? x : y);
}

......
```

(See Deitel 10 for details)

Bubblesort with Templates

```cpp
// CPP code for bubble sort
// using template function
#include <iostream>
using namespace std;

// A template function to implement bubble sort.
// We can use this for any data type that supports
// comparison operator < and swap works for it.
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
              swap(a[j], a[j - 1]);
}
```

```cpp
int main() {
    int a[5] = {10, 50, 30, 40, 20};
    int n = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort(a, 5);

    cout << " Sorted array : ";
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;

  return 0;
}
```

Class Templates

```
template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}
```

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Templates with Multiple Arguments

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

template<class T, class U>
class A  {
    T x;
    U y;
public:
    A() {    cout<<"Constructor Called"<<endl;    }
};

int main()  {
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

Templates with Default Arguments

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

template<class T, class U = char>
class A  {
public:
     T x;
     U y;
     A() {    cout<<"Constructor Called"<<endl;    }
};

int main()   {
    A<char> a;  // This will call A<char, char>
    return 0;
}
```

Insertion sort

Orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- Assume that array a[0..i-1] is sorted
- Insert a[i] to that array so that the “sortedness” is preserved

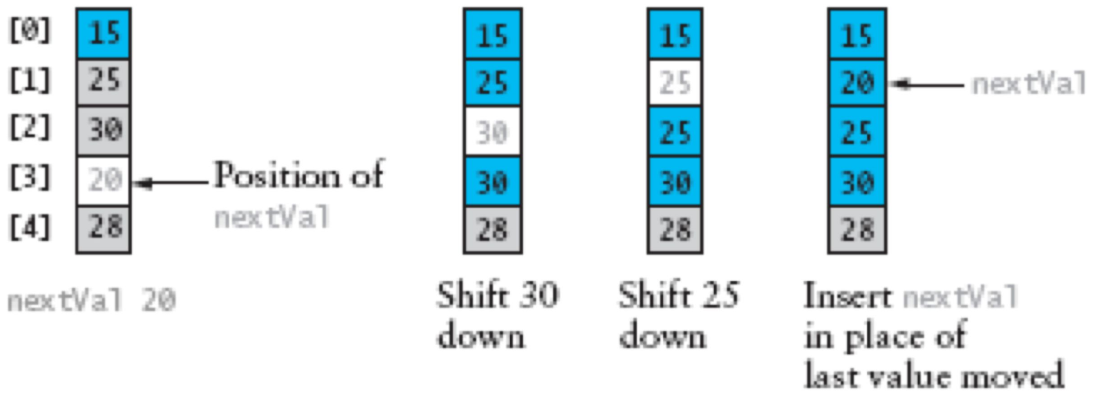
```
for (int i = 1; i < n; i++) {
   nextVal = a[i];
   // slide elements down to make room for nextVal
   int j = i;
   while (j > 0 && a[j - 1] > nextVal) {
       a[j] = a[j - 1];
       j--;
   }

   a[j] = nextVal; // put a[i] into the open spot
}
```

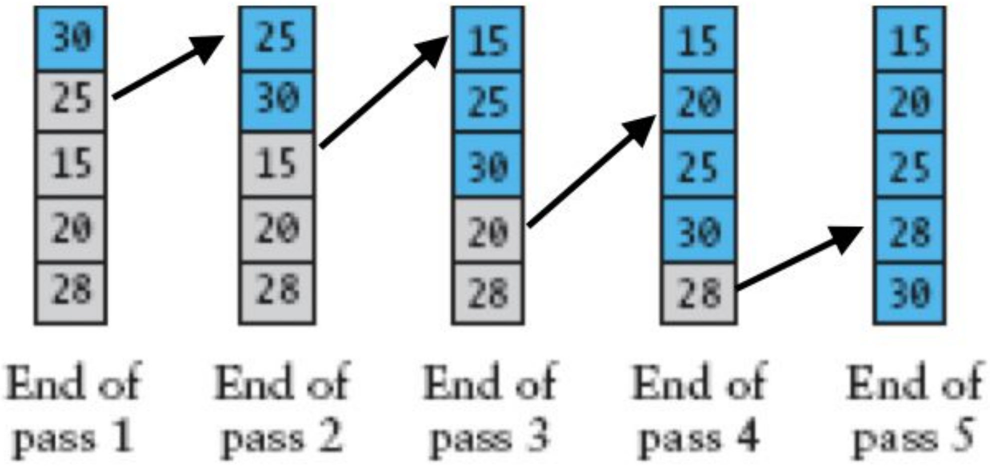
What is the time complexity?
What is the space complexity?

6 5 3 1 8 7 2 4

Insert an element into an array



Insertion Sort



Recursion

Readings:
- Weiss 1.3
- Horowitz 1.2.2
- Deitel 6

Recursion

Recursion: A way of defining a concept where the definition refers to the concept that is being defined ("define through itself")

- Sometimes, the best way to solve a problem is by solving a <u>smaller version</u> of the exact same problem first
- *Recursion* is a technique that connects the solution of a <u>smaller problem</u> to solving a <u>larger problem</u>. The problems have to be of the same type.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n\text{-}1)*n & \text{if } n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n\text{-}1)!*n & \text{if } n > 0 \end{cases}$$

```
int Factorial(int n)
{
  if (n==0)   // base case
    return 1;
  else        // general case
    return n * Factorial(n-1);
}
```

Visualizing Recursion

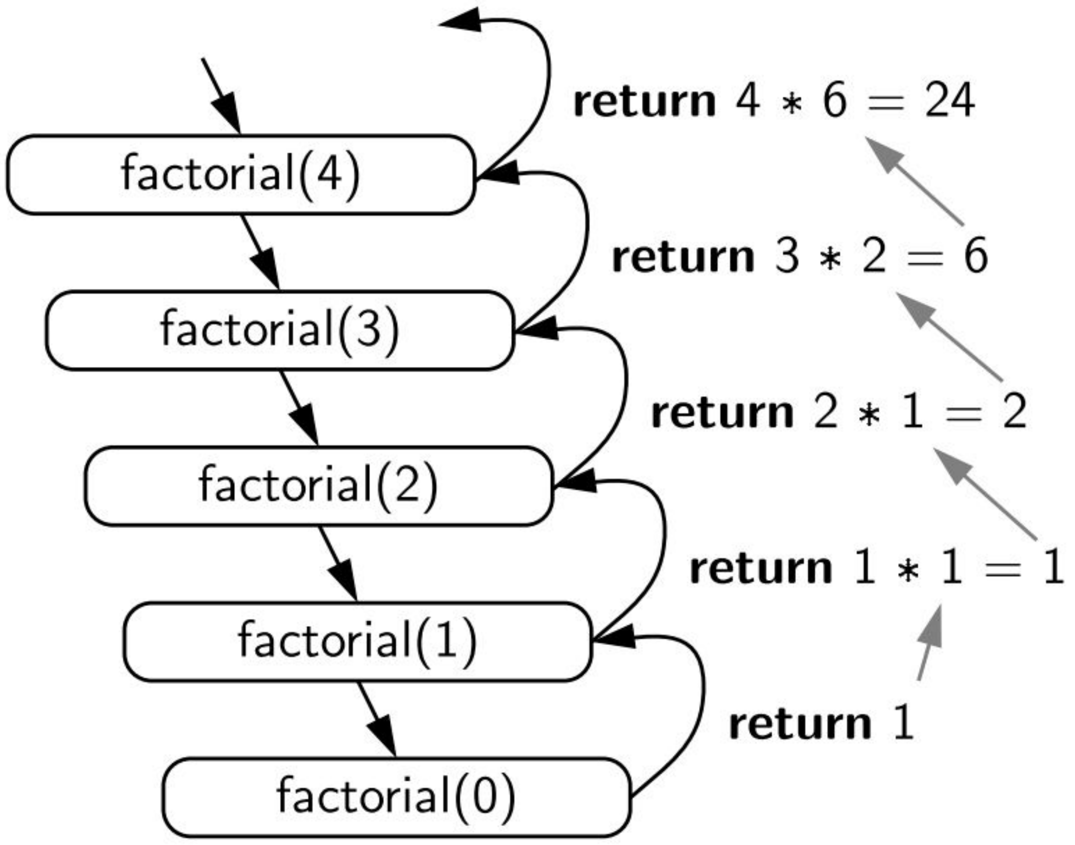
Recursion trace

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

```
int Factorial(int n){
   if (n==0)   // base case
      return 1;
   else         // general case
      return n * Factorial(n-1);
}
```

What is the time complexity?

Example recursion trace:



Reversing an Array

```
void reverse(int arr[], int start, int end) {
  int temp;

  if(start < end) {
     // Swap the elements between arr[start] and arr[end]
     temp = arr[start];
     arr[start] = arr[end];
     arr[end] = temp;

     // recursive function call
     reverse(arr, start+1, end-1);
  }

  return;
}
```

What is the time complexity?

Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

How to Write Recursive Programs

- Determine the size factor
- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (use the "Three-Question-Method")

Three-Question Method

The Base-Case Question:

Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?

The Smaller-Caller Question:

Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

The General-Case Question:

Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Binary Search Revisited

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

```
int binarySearch(int arr[], int left, int right, int x) {
   while (left <= right) {
       int mid = left + (right - left) / 2;
       if (arr[mid] == x)  return mid;
       else if (arr[mid] < x) left = mid + 1;
       else right = mid - 1;
   }
   return -1;
}
```

What is the *size factor*?

The number of elements in ($A[l]$ ... $A[r]$)

What is the *base case(s)*?

(1) If $l > r$, return -1

(2) If $x == A[m]$, return m

What is the *general case*?

if $x < A[m]$ search the first half

if $x > A[m]$, search the second half

Let's walk through a recursive implementation of binary search

Binary Search Revisited

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

```
int binarySearch(int arr[], int left, int right, int x) {

  // base case: x not found
  if (left > right) return -1;
  int mid = left + (right - left) / 2;

  // base case: x found at arr[mid]
  if (arr[mid] == x)
    return mid;

  // recursive cases
  if (x < arr[mid])
    return binarySearch(arr, left, mid-1, x);
  else
    return binarySearch(arr, mid+1, right, x);

}
```

What is the *size factor*?

The number of elements in ($A[l]$ ... $A[r]$)

What is the *base case(s)?*

(1) If $l > r$, return -1

(2) If $x == A[m]$, return m

What is the *general case*?

if $x < A[m]$ search the first half

if $x > A[m]$, search the second half

What is the time complexity?

Pros/Cons of Recursion

Advantages of Recursion in C++

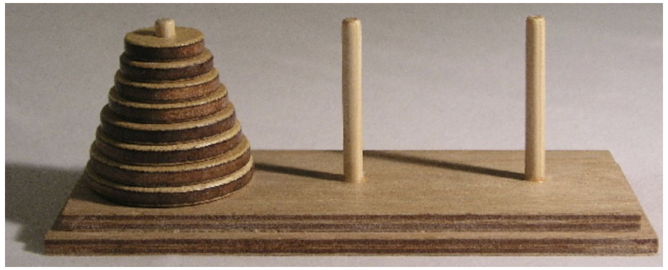
- Shorter code
- Often cleaner, more elegant code
- Good for branching problems that would require tedious record-keeping for iterative code
- Particularly convenient in trees and graphs

Disadvantages of Recursion in C++

- Takes up a lot of program stack space compared to an iterative program
- Uses more CPU time
- May be more difficult to debug and understand

Some Practice Exercises

1. Write a recursive procedure to insert an element x into a sorted array $A[0..i]$ such that the resulting array is $A[0..i+1]$ and is sorted. Then write recursive insertion sort.
2. Write a recursive procedure to search an element x in an unsorted array A.
3. Write a recursive procedure to find the number of occurrences of an element e in an array A.
4. Solve the Towers of Hanoi Problem:



Tower of Hanoi consists of three pegs or towers with n disks placed one over the other.

The objective of the puzzle is to move the stack to another peg following these simple rules.

1. Only one disk can be moved at a time.
2. No disk can be placed on top of the smaller disk.

Divide-and-Conquer Sorting: Mergesort, Quicksort

Readings:

- Horowitz 3.1, 3.4, 3.5
- Cormen 7, 8
- Weiss 7.6, 7.7 (STL warning)

Sorting Algorithms

Permute elements of an array such that they are ordered

1. Bubblesort
2. Insertion sort
3. Mergesort
4. Quicksort
5. Heapsort
6. Radix sort
7. …

Comparison-based sorts, uses only comparisons (<, ≤, =, ≥, >)

Divide-and-Conquer Sorting

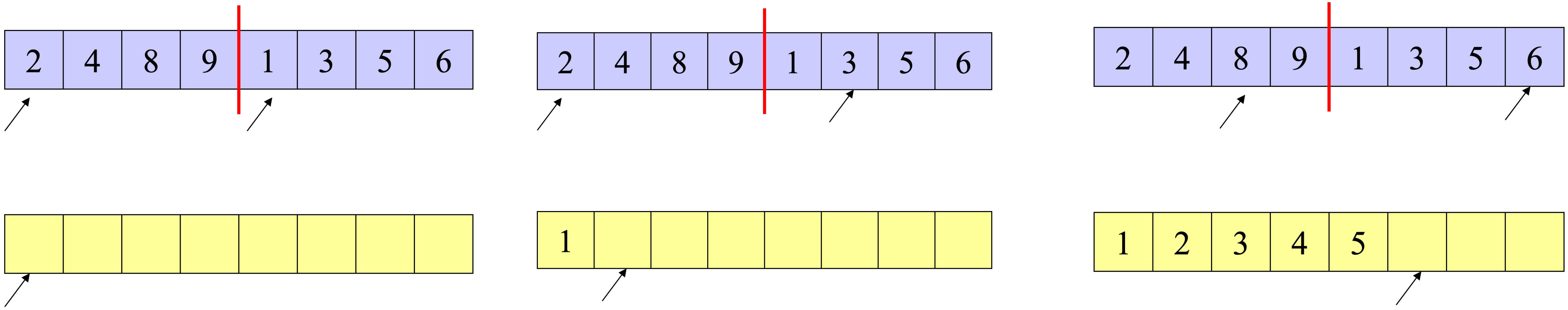
- Very important **strategy** in algorithm design:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get the overall solution
  - **Where have we seen this strategy already?**

- **Approach 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves -> Mergesort
- **Approach 2:** *Partition* array into items that are "small" and items that are "large", then *recursively* sort the two sets -> Quicksort

Mergesort

- If n == 1: terminate (every one-element list is already sorted)
- If n > 1:
    - Divide the array into two halves
    - **Recursively sort each half**
    - **Merge the two sorted halves**

Merging



Time complexity: $O(n)$

```c
// Merges sorted P ← A[l..m] and Q ← A[m+1..r] into A[] starting from l
void merge(int A[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int P[n1], Q[n2];
    for (int i = 0; i < n1; i++) P[i] = A[l + i];
    for (int j = 0; j < n2; j++) Q[j] = A[m + 1 + j];
    //  Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0; j = 0; k = l; // k equals “el”
    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[l..r]
    while (i < n1 && j < n2) {
       if (P[i] <= Q[j]) {
          A[k] = P[i]; i++;
       } else {
         A[k] = Q[j]; j++;
       }
       k++;
     }
    // When we run out of elements in either P or Q, flush the rest
    while (i < n1) {
      A[k] = P[i]; i++; k++;
    }
    while (j < n2) {
     A[k] = Q[j]; j++; k++;
    }
}
```

Left/middle/right indices:
l = 0, m = 3, r = 5

Counters: i in P, j in Q, k in A

Mergesort Algorithm

6 5 3 1 8 7 2 4

```
void mergeSort(int A[], int l, int r) {
    if (l < r) {
        // m is the point where the array is divided into two subarrays
        int m = l + (r - l) / 2;
        mergeSort(A, l, m);
        mergeSort(A, m + 1, r);
        // Merge the sorted subarrays
        merge(A, l, m, r);
    }
}
```

Base case:

```
mergesort(A, 0, 0);
// does not enter the if ->
// does nothing
```

Example run:

```
int A[8] = …
mergesort(A, 0, 7);
```

```
    if (0 < 7) {
        int m = 3; // does nothing
        mergeSort(A, 0, 3);
        mergeSort(A, 4, 7);
        // Merge the sorted subarrays
        merge(A, 0, 3, 7);
    }
```

Mergesort Algorithm

6 5 3 1 8 7 2 4

```
void mergeSort(int A[], int l, int r) {
    if (l < r) {
        // m is the point where the array is divided into two subarrays
        int m = l + (r - l) / 2;
        mergeSort(A, l, m);
        mergeSort(A, m + 1, r);
        // Merge the sorted subarrays
        merge(A, l, m, r);
    }
}
```

Time complexity: O(n*log(n))

What is the space complexity?

$T(n) = 2T(n/2) + cn$

$= 2(2T(n/4)+c(n/2)) + cn = 2^2T(n/2^2) + cn + cn = 2^2T(n/2^2) + 2cn$

$= \ldots = 2^kT(n/2^k) + k*cn = \ldots$

This completes when $n/2^k <= 1$, i.e., $k = \log n$

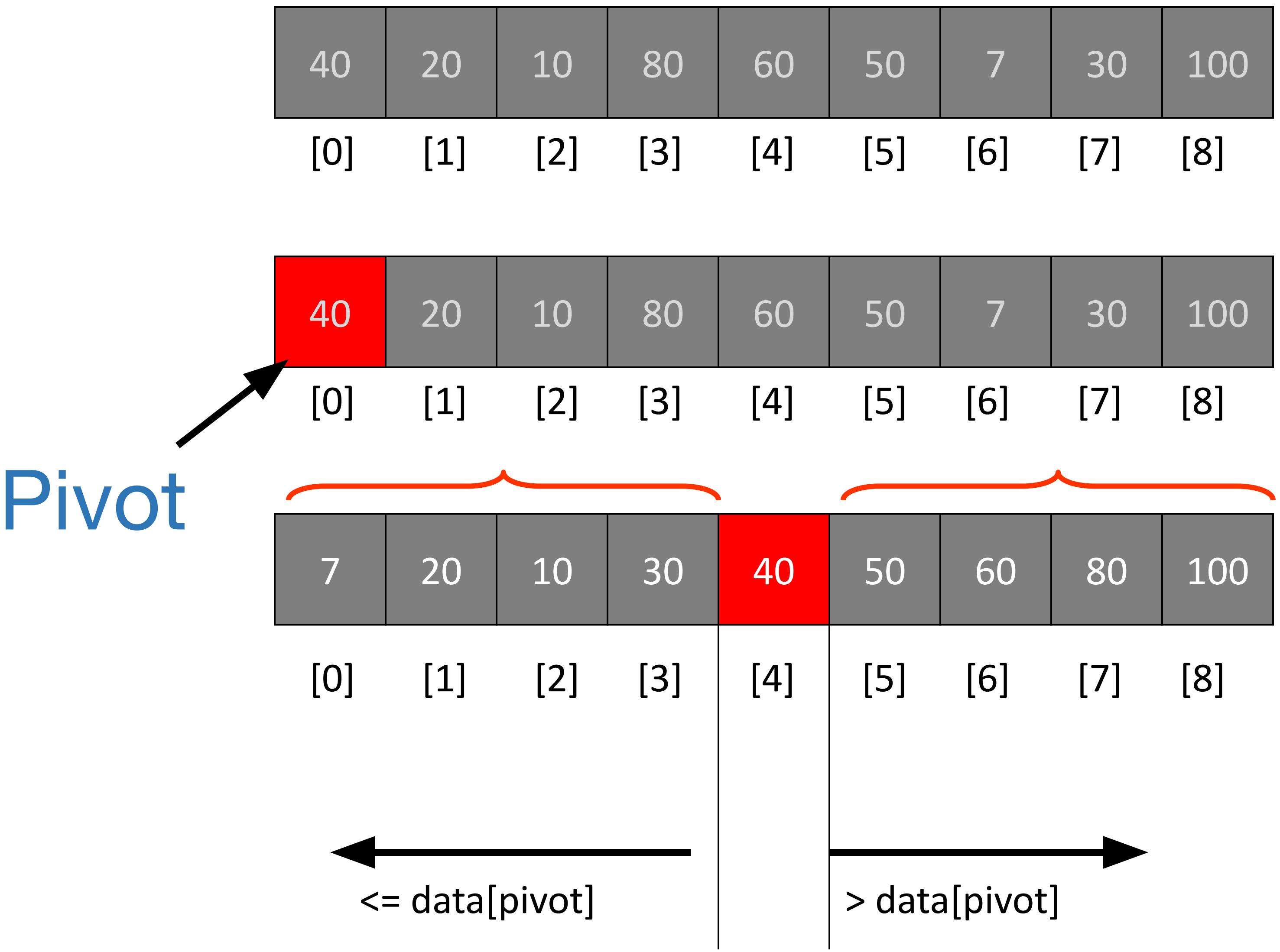
$= n + cn*\log n$, so we get $O(n*\log n)$

Quicksort

Given an array of n comparable elements (e.g., integers):

- If array only contains one element, return
- Else
  - Pick one element to use as *pivot*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Example


40 20 10 80 60 50 7 30 100
[0] [1] [2] [3] [4] [5] [6] [7] [8]
40 20 10 80 60 50 7 30 100
[0] [1] [2] [3] [4] [5] [6] [7] [8]
Pivot
7 20 10 30 40 50 60 80 100
[0] [1] [2] [3] [4] [5] [6] [7] [8]
<= data[pivot]
> data[pivot]

Quicksort

```
void quickSort(int arr[], int start, int end)
{
    // Base case
    if (start >= end)
        return;

    // Partitioning the array, returns the pivot index
    int p = partition(arr, start, end);

    // Sorting the left part
    quickSort(arr, start, p - 1);

    // Sorting the right part
    quickSort(arr, p + 1, end);
}
```

Partitioning

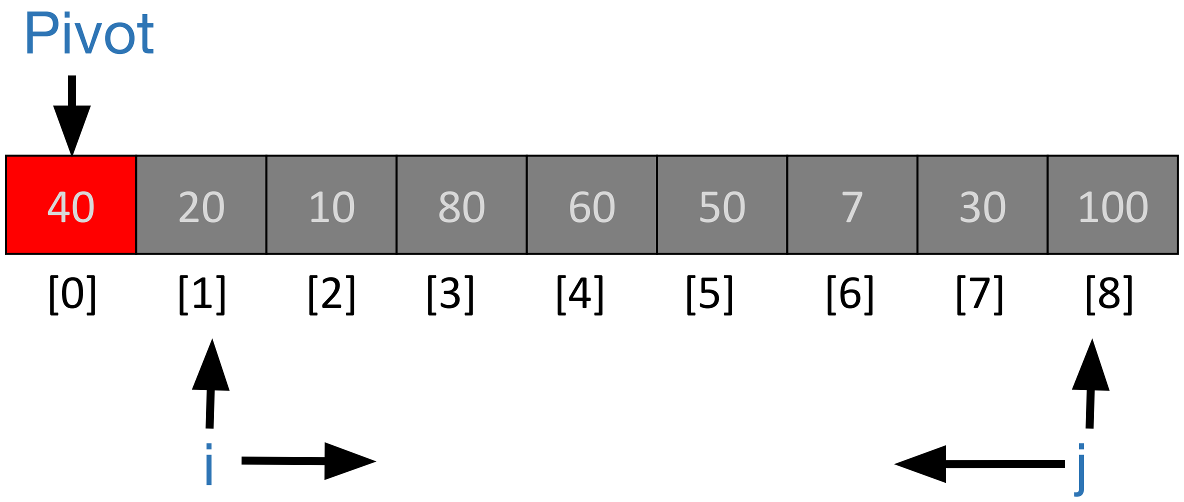
- **Need to partition the array into left and right sub-arrays**
  - The elements in left sub-array are ≤ pivot
  - Elements in right sub-array are > pivot
- **How do the elements get to the correct partition?**
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning

```
int partition (int arr[],int start,int last) {
    int i=start+1, j=last, temp;
    // We use arr[start] as the pivot element
    while(i <= j) {
        if (arr[i] <= arr[start]) i++;
        if (arr[j] > arr[start]) j--;
        if(i <= j) {
          temp = arr[i];
          arr[i]= arr[j];
          arr[j] = temp;
        }
    }
    // Swap the arr[start] and arr[j]
    temp = arr[start];
    arr[start] = arr[j];
    arr[j] = temp;

    return j;
}
```

Partitioning takes O(N) time



Other pivot choices:

last, random, median of first 3

Quicksort Animation

6 5 3 1 8 7 2 4

Solid black: pivot
Red frames: iterators i, j for partitioning
Black frame: already sorted

Performance of Quicksort

Partitioning takes O(N) time

Best pivot choice:

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - › $T(0) = T(1) = O(1)$
    - constant time if 0 or 1 element
  - › For $N > 1$, 2 recursive calls plus linear time for partitioning
  - › $T(N) = 2T(N/2) + O(N)$
    - Same recurrence relation as Mergesort
  - › $T(N) =$ O(N log N)

Worst pivot choice:

(for when arrays of size <= C are sorted with another algorithm in time a)

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - › $T(N) \leq T(N-1) + bN$
  - › $\leq T(N-2) + b(N-1) + bN$
  - › $\leq T(C) + b(C+1)+ \ldots + bN$
  - › $\leq a + b(C + (C+1) + (C+2) + \ldots + N)$
  - › $T(N) = O(N^2)$

Sorting Algorithms

Permute elements of an array such that they are ordered.

1. Bubblesort
2. Insertion sort
3. Mergesort
4. Quicksort
5. Heapsort
6. Radix sort
7. …

Comparison-based sorts, uses only comparisons (<, ≤, =, ≥, >)

Sorting in “Linear” Time: Binsort and Radix Sort

Binsort/counting sort: for keys from a fixed (small) domain

- K=5. list=(5,1,3,4,3,2,1,1,5,4,5)

Bins in array	
key = 1	1,1,1
key = 2	2
key = 3	3,3
key = 4	4,4
key = 5	5,5,5

Sorted list:
1,1,1,2,3,3,4,4,5,5,5

Time complexity: O(n)
assuming k is constant or O(n)

Radix Sort

- Radix = “The base of number system”
- **History:** used in 1890 U.S. census by Hollerith*
- **Idea:** BinSort on each digit, bottom up

- **Input list:**
  126, 328, 636, 341, 416, 131, 328
- **BinSort on lower digit:**
  341, 131, 126, 636, 416, 328, 328
- BinSort result on next-higher digit:
  416, 126, 328, 328, 131, 636, 341
- BinSort that result on highest digit:
  126, 131, 328, 328, 341, 416, 636

Sorting Stability

A **stable** sorting algorithm preserves the relative order of “equal” items

E.g., think of duplicate numbers having different colors: [6, 3, 7, 8, 1, 7]
- Stable sorting: [1, 3, 6, 7, 7, 8]
- Unstable sorting: [1, 3, 6, 7, 7, 8]

Which of these sorting algorithms are stable: bozosort, bubblesort, insertion sort, mergesort, quicksort?

Which Sorting Algorithm to Use?

- **Data size**
  - For small counts (<= 100), $O(n^2)$ are fine
  - For larger datasets, use $O(n \log n)$ algorithms
- **Duplicates in data:**
  - How are the ties broken? Is there a secondary key?
  - Is sorting stability required?
- **Prior knowledge about the data**
  - Is the data partially sorted? Insertion sort works well
  - Is the distribution of keys known? Clumping is bad for binsort
  - Are the keys very large and hard to compare? Use radix sort
- **Memory constraints**
  - Quicksort and insertion sort do not require extra memory, unlike mergesort
- **Programmer efficiency**
  - How much time do you have to debug? Use a less complicated one

Extra: cool controllable visualizations of sorting etc (thanks to Ryan)