BMI 203
Biocomputing Algorithms

Lecture 1: Introduction, Complexity Theory and Sorting

Instructor: Ryan Hernandez <ryan.hernandez@ucsf.edu>
TA_1: Tamas Nagy <tamas@tamasnagy.com>
TA_2: Seth Axen <tamas@tamasnagy.com>

Dropbox: http://tinyurl.com/BMI203-18

Course email list

- Email me: ryan.hernandez@ucsf.edu
- Include:
  - Subject: “BMI 203 email list”
  - Body:
    - Name
    - Program
    - Favorite programing language
    - Audit, Grade, or Pass/Fail
    - BMI students: mini-qual?
    - Auditing and want to present papers?

Important stuff…

- This course meets 3 times per week, all are required
- **Mondays 2:30-3:30**
    - Typically a TA discussion session
        - additional material and homework
- **Wednesdays 10:30-12**
    - Lecture
- **Fridays 10:30-12**
    - Paper discussion/labs

Paper discussions

- Slides or chalk talk is fine
    - Probably best to do one of each
    - Different opportunities in each format
- What should you focus on?
    - What did the authors try to show?
    - How does the algorithm work?
    - Are there tradeoffs in algorithm construction that yield lower complexity for potentially lower performance? What are the cases where this will manifest?
    - Did the authors manage to convince you? Did they miss obvious controls? Did they make appropriate use of statistics?

Textbooks

- No single book covers all the topics in this course…
- **Introduction to Algorithms,** Cormen, Leiserson, and Rivest.
- **Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids,** Durbin, Eddy, Krogh, and Mitchison.
- **Numerical Recipes in C: The Art of Scientific Computing,** Press, Teukolsky, Vetterling, and Flannery.
  - www.nr.com
- **You should probably buy the first book. You will be looking at it 20 years from now if you continue research in bioinformatics and algorithms.**

Biocomputing Algorithms

- Computational issues and methods in bioinformatics and computational biology
  - Analytical thinking
  - Problem decomposition
  - Algorithm understanding, design, and implementation
- This course is **not** about:
  - Expert use of existing tools
  - Learning how to program (if you can’t program in C, Python, or a similar language, you should take the course after you have become somewhat proficient)

Programming Languages

- We are formally agnostic about programming languages
- However, for some assignments, it will be easier to use Python, since you will be provided some code, and we can provide more help with Python than other languages.
- Languages that are OK: Python, C, C++, Julia, Fortran, Java, …

Course Information

- Three homeworks represent 40% of your grade
- Classroom participation, including paper presentations, represent 30%
- Your final projects represent 30%
- Course Lecturers: Mike Keiser, Scott Pegg, and Jimmy Ye
- TA1: Tamas Nagy <tamas@tamasnagy.com>
- TA2: Seth Axen <tamas@tamasnagy.com>
- We encourage you to work individually at first on all homework. However, making use of the overlapping prior training among you is OK. If you are a weaker hacker, find someone who is not! Lend a hand (*<u>not solutions</u>*) if you are!

Computer Resources

- We expect you to have access to your own computer!
- You can download Python: www.python.org
- You can download Cygwin (gnu c): www.cygwin.com
- You can run clang to compile on Mac (OS X ships with developer tools, but you have to install them)

Reproducibility

- Research must be reproducible.
- Computational tools/methods SHOULD be open and accessible.
- Github is one resource for maintaining a code repository.
  - More on this Wednesday!

Outline

- **Complexity Theory** (See Cormen, Chapters 1 and 2)
  - Every computer algorithm has execution time and space and/or bandwidth requirements dependent on the size or complexity of its input
  - Design of useful algorithms is largely dominated by complexity considerations
  - We will cover very basic notational conventions (no proofs)
- **Sorting** (see Cormen, Part II, particularly Chapter 8)
  - Sorting is the classic algorithms problem space in which complexity issues are taught
    - Bubble sort
    - Quicksort
- Reference: Introduction to Algorithms, Second Edition by Thomas H. Cormen (Editor), Charles E. Leiserson, Ronald L. Rivest

Computational Complexity Theory

- **What is an algorithm?**
  - Given a *precise* problem description
    - Sort a list of N real numbers from lowest to highest
  - An algorithm is a *precise* method for accomplishing the result
- A critical concern is efficiency: how do we characterize it?

Computational Complexity Theory

- **O notation**: informally
  - Want to capture how fast or how much space an algorithm requires
  - We ignore constant factors (even if very large)
  - O(N) indicates that an algorithm is linear in the size of its input
  - Example: sum of N numbers is O(N)

Notes on Notation

- We will define the complexity of algorithms based on describing a function that provides a boundary on time or space
- Formally, we will describe complexity in terms of the membership of the function in a larger set of functions
- Notation
  - $N = \{0,1,2,3, \ldots\}$ "Natural numbers"
  - $N^+ = \{1,2,3,4, \ldots\}$ "Positive natural numbers"
  - R = Set of Reals
  - R^+ = Set of Positive Reals
  - $R^* = R^+ \cup \{0\}$

Big O notation

- Big O describes the complexity of the worst-case scenario of an algorithm.
    - Either in terms of runtime or memory.
    - Usually described as a function of the number of input elements (N).
    - Always excludes constants!

https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Big O notation examples

- **O(1)**: Complexity is independent of the size of the input data.
  - Check the sign of the first element of a list.
- **O(N)**: Complexity is linear in the input size.
  - ```
    s=0;
    for (i=0; i<n; i++) { s += i; }
    ```
  - O(1) implementation?
    ```

Big O notation examples

- **O(N^2)**: Complexity is proportional to the square of the input data size.
  - ```
    s=0;
    for (i=0;i<n;i++) {
       for (j=0;j<n;j++) {
          s += i+j;
       }
    }
    ```
  - Nested loops are sometimes convenient, but will become VERY PAINFUL. Avoid if at all possible.
    ```

Big O notation examples

- **O(log N)**: Less intuitive.
  - Often arises in a binary search for a particular element in a sorted list.
  - A common algorithm is a multiplicative index jump:
    - ```
      h = 1;
      while (h < n){
         s;
         h = 2*h;
      }
      ```
      ```

Big O notation examples

- $O(2^N)$: dire situation… runtime doubles with each additional element. Exponential growth!!
  - An example is a recursive calculation of Fibonacci numbers:
  - 1,2,3,5,8,13, …, F(i-2)+F(i-1), …
  - ```
    int Fibonacci(int number){
      if (number <= 1) return number;
      return Fibonacci(number - 2) + Fibonacci(number - 1);
    }
    ```
    ```

Comparing $f(n)$ and $g(n)$

- Let **f** be a function from **N** to **R**.
- **O(f)** (Big **O** of **f**) is the set of all functions **g** from **N** to R such that:
  - There exists a real number $c>0$
  - AND there exists an $\mathbf{n_0}$ in **N**
- Such that: $\mathbf{g(n) \leq cf(n)}$ whenever $n \geq n_0$
- In English: **g** grows no faster than **f**.

Notation and pronunciation

- Proper Notation: $g \in O(f)$
- Also Seen: $g = O(f)$
  - **“g is oh of f”**
- **g** is essentially bounded above by **f**
- “My function is no worse than linear in input size”

Big Omega

- Let **f** be a function from **N** to **R**.
- **Ω(f)** (Big Ω of f) is the set of all functions **g** from **N** to **R** such that:
  - There exists a real number **c**>0
  - AND there exists an n_0 in **N**
- Such that: $g(n) \geq cf(n)$ whenever $n \geq n_0$
- **g** is essentially bounded below by **f**
- "My function is worse than linear in input size"

Big Theta

- **$\Theta(f) = O(f) \cap \Omega(f)$**
- **$g \in \Theta(f)$**
- **"g is of Order f"**

English Interpretations

- **$O(f)$** - Functions that grow no faster than **f**
- **$\Omega(f)$** - Functions that grow no slower than **f**
- **$\Theta(f)$** - Functions that grow at the same rate as **f**

Properties

- Constant factors may be ignored
  - For all **k**>0, **kf** is **O(f)**
- Higher powers of **n** grow faster than lower powers
- The growth rate of a sum of terms is the growth rate of its **fastest term**
  - So, if you have a linear element of an algorithm and a element that is $\mathbf{n^2}$, then the algorithm will be $\mathbf{O(n^2)}$
- Transitivity: If **f** grows faster than **g** which grows faster than **h**, then **f** grows faster than **h**.
- Exponential functions grow faster than powers
- Logarithms grow more slowly than powers (and all logarithms grow at the same rate, irrespective of their base)

When complexity gets bad

- **Polynomial time algorithms**
  - All algorithms such that there exists an integer **d** where the algorithms is **O(nd)**
- **Intractable algorithms**
  - The class of problems that cannot be solved in polynomial time
  - Particularly interesting class of intractable problems: NP-complete
- When this happens, we often care about approximate solutions
  - Traveling Salesman Problem: Given **N** cities, find the route that goes to each city exactly once that minimizes the total distance traveled
  - **N!** ways of ordering **N** cities
  - NP-complete: if you can solve this problem in polynomial time, you can solve all NP-complete problems in polynomial time
  - ε-approximate solutions abound
    - You can find a city ordering such that for any ε, your solution is within ε of the optimal solution
    - You can do this in polynomial time

Sorting algorithms

- Input: a sequence of **n** numbers (a_1, a_2, ... , a_n)
- Output: a permutation (b_1, b_2, ... , b_n)
- Such that $a[b_1] \leq a[b_2] \leq \ldots \leq a[b_n]$
- Example: **Insertion Sort** (order a hand of cards)
  - Create an empty array of size n
  - Find the smallest value in input array
    - Put it in the new array in the first unfilled position
    - Mark the input array value as done
  - Repeat until the new array has n values

Bubble sort: In English

- Go through your list of **n** numbers, checking for misordered adjacent pairs
- Swap any adjacent pairs where the second value is smaller than the first
- Repeat this procedure a total of **n** times
- Your final list will be sorted low to high

Bubble sort: In C

```
/* Bubble sort for integers */
void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
{
  int i, j, t;
  /* Make n passes through the array */
  for(i=0; i<n; i++) { Outer loop
    /* From the first element to
       the end of the unsorted section */
    for(j=1; j<(n-i); j++) { Inner loop
      /* If adjacent items are out of order, swap */
      if( a[j-1]>a[j] ) {  One conditional
        t = a[j];
        a[j] = a[j-1];   Three assignments
        a[j-1] = t;
      }
    }
  }
}
```

Bubble sort complexity: Worst case

- We make (n-1) passes through the data
  - When i=(n-1), (n-i) is (n-(n-1))=1
  - So, on the last outer loop pass, we don't do the inner loop
- How many operations do we do in each pass (at worst)?
  - On the last pass, we do one conditional and three assignments
  - On the second to last pass, we do 2 and 6
  - Etc...
- So
  - (1*(1+2+ ... + (n-1))) compares
  - (3*(1+2+ ... + (n-1))) assignments
  - Recall that sum (1...k) is k(k+1)/2
  - We have n(n-1)/2 compares and 3n(n-1)/2 assignments
- Since we don't care about constant factors and higher-order polynomials dominate, BubbleSort is $O(n^2)$.

QuickSort: In English

- Quicksort is a divide and conquer algorithm
- It was invented by C. A. R. Hoare
- **Divide**: The array A[p...r] is partitioned into two nonempty subarrays A[p...q] and A[q+1...r] (q is pivot element)
- **Conquer**: The two subarrays A[p...q-1] and A[q+1...r] are themselves subjected to Quicksort (by recurrence)
- **Combine**: The results of the recursion don’t need combining, since the subarrays are sorted in place
- The final A[p...r] is now sorted

Quicksort: Complexity

- The average case for Quicksort is **O(n log(n))** with smallish constant factors for good implementations
  - The partitioning algorithm requires O(n) time to rearrange the array (it examines every element once)
  - The partitioning is done around a pivot, which is chosen with no knowledge (in the simplest case); elements are partitioned to be less than or greater than the pivot
  - We expect that randomly chosen pivots will tend to partition an array into roughly two halves
  - So, we end up doing O(log(n)) partitions, and O(n log(n)) overall
- In practice, this is one of the fastest sorting methods known
- However, its worst case behavior is **O(n^2)**: poor luck with the pivot choices can lead to n partitions!

Quicksort: In C

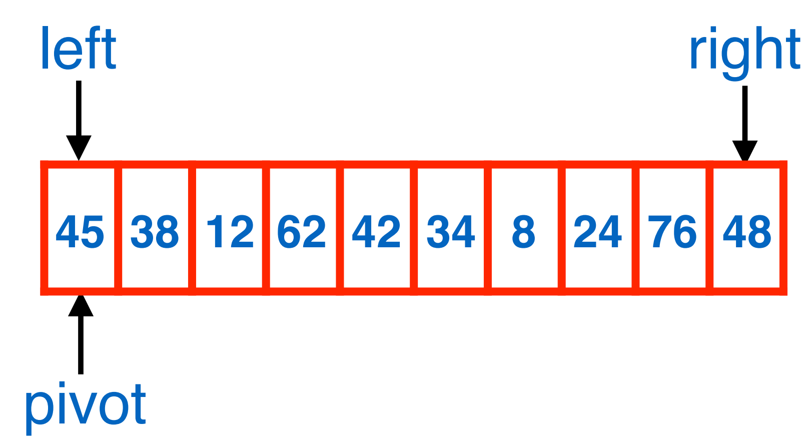
- ```
  /* We would call quicksort(a, 0, n-1) */
  quicksort( void *a, int p, int r ) {
    int pivot;
    /* Termination condition! */
    if ( r > p ) {
      pivot = partition( a, p, r );
      quicksort( a, p, pivot-1 );
      quicksort( a, pivot+1, r );
    }
  }
  ```
- The partition function does all of the work
- It selects the pivot element
- It partitions the subarray
- The quicksort function just does bookkeeping
- Note: in C, arrays are passed by reference, so the operations are occurring on the same array
  ```

Quicksort partitioning

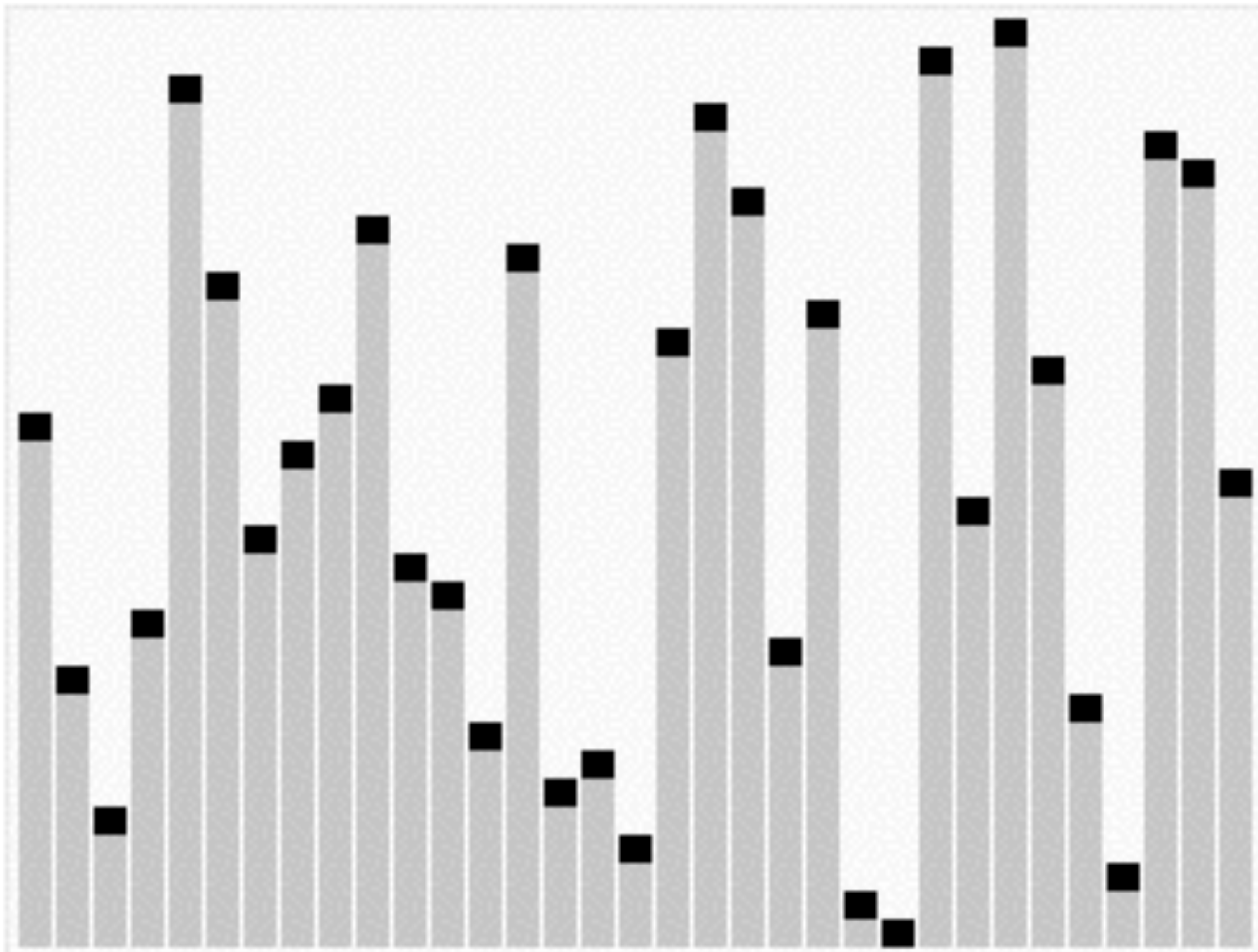
- There is a straightforward way to partition
    - Pick any element as the pivot (say the first)
    - Create a new array of the same size as input
    - For each element in the old array, put it at the beginning if it is less than the pivot element
    - Else, put it at the end
    - [Keep track of the “beginning” and “end”, which move]
    - Copy the new array back into the original one
    - Return the value of the pivot index
- Problem: requires additional space (allocate and free) and an additional n assignments in the end

Quicksort: Partition in place

```
• int partition( void *a, int p, int r ) {
    int left, right;
    void *pivot_item;
    pivot_item = a[p];
    pivot = left = p;
    right = r;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] > pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
    }
    /* right is final position for the pivot */
    a[p] = a[right];
    a[right] = pivot_item;
    return right;
  }
```



https://en.wikipedia.org/wiki/Quicksort



Wednesday

- Tamas/Seth will give an overview of computational aspects of the course
- Homework 1 will be distributed
- **No class on Friday**