LECTURE 9

ANALYSIS OF ALGORITHMS

ANALYSIS OF ALGORITHMS

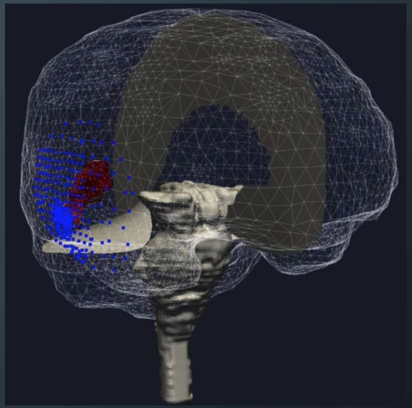
- Algorithm Efficiency
- Algorithm Growth Rates
- Properties of growth rates
- Best, Worst, Average cases
- Tractable algorithms and class P
- Class NP problems
- Examples

- When comparing algorithm *B* to algorithm *G*, how can we compare them?

PARAMETERS FOR ALGORITHM EFFICIENCY

1. <u>Time</u>- how fast can *B* and *G* finish on a given platform?
   - Real-time systems (Real-time rendering (VR), Real-time brain-surgery algorithm which provides image-processing-based corrections for tissue deformation)
   - Finishing a simulation in seconds instead of days
2. <u>Space</u>- how much memory and storage do *B* and *G* require?
   - Space-constrained systems (FPGAs, embedded devices)
   - Fitting in main memory avoids unnecessary page swaps
3. <u>Energy</u>- how much energy do *B* and *G* require to run?
   - Energy constrained systems (embedded systems, satellites, cell phones)
   - Satellite or remote sensor lasting years instead of months before replacement/retirement

- Others important parameters <u>not covered in this course</u>
  - <u>Security</u>- does your algorithm have security holes?
  - <u>Bandwidth</u>- how much and often does your algorithm need to communicate over a restricted-bandwidth connection (I.e., space station to earth)



Real-time corrections for soft-tissue deformation
M. Luo et al.

STP-H5 on ISS

SIMPLE TOY EXAMPLE

- Assume no compiler optimizations (aka, the code runs on the processor as written)
- Which version is faster?

```
//Version 1:
int halve(int n)
{
    return n/2;
}

//Version 2:
int halve(int n)
{
    return n>>1;
}
```

EXAMPLE 2

- What requires more operations on average:
  - Insertion into linked list
  - Insertion into an array

87	**68**	**99**	**71**	**66**	**59**	**60**

```cpp
void ArrayList<T>::insert(size_t position, const T& item)
{
    if(size == capacity){
    // need to reallocate
    } // capacity > size and size >= 1 now
    for(std::size_t i = size-1; i > position; --i){
        data[i+1] = data[i];
    }
    data[position-1] = item;
    ++size;
}
```

```cpp
void LinkedList<T>::insert(size_t position, const T& item)
{
    if(position == 1){
        Node<T> *temp = new Node<T>(item);
        temp->next = head;
        head = temp;
    }
    else{
        Node<T> *loc = find(position);
        Node<T> *temp = new Node<T>(item);
        temp->next = loc->next;
        loc->next = temp;
    }
    size += 1;
}
```

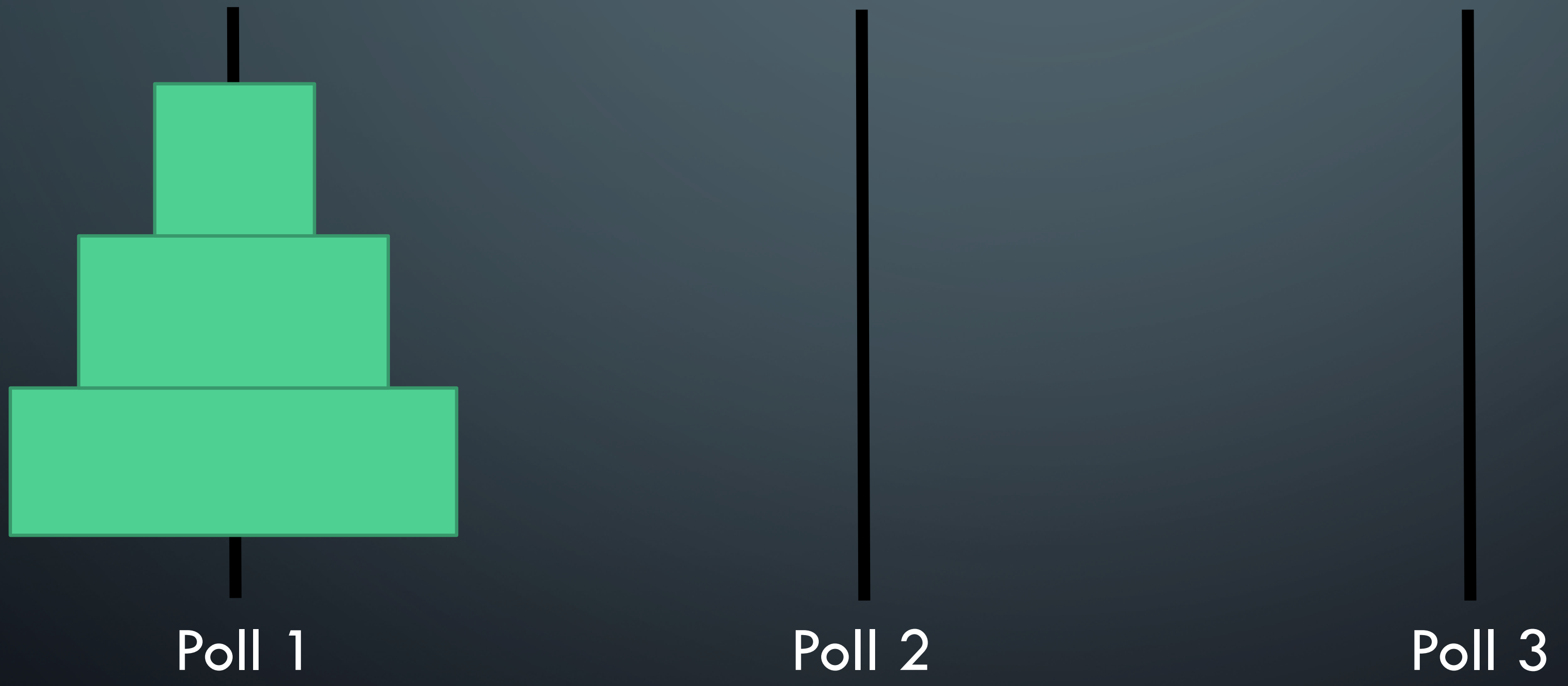
A BETTER WAY TO COMPARE ALGORITHM ANALYSIS

- Just comparing on implementation has many limitations:
  - Machine used to test affects results
  - Date/time of test (and other programs running) affect results
  - Language chosen affects results
  - Data/benchmark used as input to the program affect results

- Instead, algorithm analysis quantifies complexity on number of basic ops
  - Additions, subtractions, multiply, divide
  - Comparisons
  - Assignments and/or function calls

EXAMPLE: TOWERS OF HANOI

- Rules: Move tower from poll 1 to poll 3. A bigger block can never be on top of a smaller block. Use 3 polls.



TOWERS OF HANOI PSEUDOCODE

- Works for n blocks, 3 polls/pegs
- How does the number of operations depend on n and Hanoi(n-1)?

```
function Hanoi(int current_peg, int aux_peg, int dest_peg, int n)
{
  if (n==1)
  {
     //move_disk moves disk n from current_peg to dest_peg
     move_disk(n, current_peg, dest_peg);
  } else{
    Hanoi(current_peg, dest_peg, aux_peg, n-1);
    //move_disk moves disk n from current_peg to dest_peg
    move_disk(n, current_peg, dest_peg);
    Hanoi(aux_peg, current_peg, dest_peg, n-1);
  }
}
```

TOWERS OF HANOI

- Let Operations(Hanoi(n)) = T(n)
- $T(n) = 2*T(n-1) + 1$
- Can prove from this relation that $T(n) = 2^n-1$
  - Verify (induction):
  - Assume $T(k) = 2*T(k-1)+1 = 2^k-1$
  - Then $T(k+1) = 2*T(k)+1$ (from recurrence relation)
  - $= 2(2^k-1)+1$ (from assumption)
  - $= 2^{k+1}-2+1$ (expansion)
  - $= 2^{k+1}-1$ (simplify)

RECURRENCE TO COMPLEXITY

- It is convenient when analyzing algorithms to have a closed form solution to recurrence relations.
- So, for n disks $T(n) = 2^n-1$ for Towers of Hanoi
- Question: How fast does this grow as a function of the number of disks?
  - The number of moves is exponential in the size of the problem (n disks)
- How does the number of operations for the Towers of Hanoi grow as n grows?
  - We say that the growth of the operations is *exponential* in n
- As we move forward with algorithms, we will explore this as the *complexity* of the algorithm
  - This is an estimate of the runtime of the algorithm on a traditional computing engine

GROWTH RATES

- Compare these hypothetical algorithms:
  - Algorithm A takes $2n^2$ operations
  - Algorithm B takes $12n^2$ operations
  - Algorithm C takes 1,000n operations
  - Algorithm D takes 2^n operations

- How do they compare as n grows?

GROWTH RATES

- Let's say n is a data set of all the English pages on Wikipedia (currently 29 million)
  - Algorithm A takes [$2n^2$] $2(29{,}000{,}000)^2$ = 1.682 quadrillion = $1.682*10^{15}$ operations
  - Algorithm B takes [$12n^2$] $12(29{,}000{,}000)^2$ = 10.092 quadrillion = $1.0092*10^{16}$ operations
  - Algorithm C takes [1,000n] 1,000*(29,000,000) = 29 trillion = $2.9*10^{13}$ operations
  - Algorithm D takes [2^n] 2^(29,000,000) = $7.486*10^{8729869}$ operations

- Assume we can do ~100 operations per nanosecond [$1*10^{11}$] (cluster of computers)
  - Algorithm A takes 16820 seconds (4.67 hours)
  - Algorithm B takes 100920 seconds (28.03 hours)
  - Algorithm C takes 290 seconds (0.8 hours)
  - Algorithm D takes $2.373*10^{8729851}$ *years*
    - *$1.648*10^{8729837}$* times longer than the age of the universe

- A, B proportional to n^2, C proportional to n, D proportional to 2^n
  - These are the *growth rates* of these functions

SO DO WE HAVE A CLUSTER OF COMPUTERS?

- What we are really interested in is *general trends*
    - This is a computing system agnostic way of expressing runtimes
- Algorithm A and B are proportional to n^2
    - Constant multipliers don't make a huge difference
- Algorithm C is proportional to n
    - Again, constant multipliers don't make a huge difference
- Algorithm D is exponential in n
    - This is sort of bad, right?

We have these classifications for algorithms for ok and bad more precisely,
stay tuned

HOW CAN WE EXPRESS GROWTH RATES?

- Algorithm A is said to be of **order** f(n) where f(n) is proportional to the *growth rate function*
  - Also denoted as **O(f(n))**
  - Also referred to as **Big O notation**
- Algorithms A and B are $O(n^2)$
- Algorithm C is $O(n)$
- Algorithm D is $O(2^n)$
- This trims the fat of all the superfluous information and cuts to the heart of the issue.

Now we can reason about algorithm choices before detailed implementation

PROPERTIES OF GROWTH RATES

- Ok so let's formalize this a little bit more
- Definition: an algorithm $f(n)$ is of Order $g(n)$ (aka, O(g(n)) if for some k and all n:
  - $k*g(n) > f(n)$
  - Where k and n are positive integers
- Therefore, we can *ignore lower order terms*
- Example: Algorithm A has complexity $2n^2+3n+4$
  - We say algorithm A has complexity of order $O(n^2)$
    - Verify: if $k=10$, $10n^2 > 2n^2+3n+4$ for all values of $n>=1$

PROPERTIES OF GROWTH RATES

- Combining orders of growth: $O(O(f(n)) + O(g(n))) = O(f(n) + g(n))$
- Example:
  - Algorithm E is $O(n^2)$
  - Algorithm F is $O(n)$
  - Algorithm G uses algorithm E, then algorithm F
- Therefore algorithm G is
  A. $O(n^2+n)$
  B. $O(n^2)$
  C. $O(n^3)$
  D. $O(n)$

WHAT ABOUT MULTIPLE DATASETS?

- Dependent on two separate, independent input vectors [articles in Wikipedia (n) and people who use Wikipedia in Pennsylvania (m)]
  - Algorithm H is $O(n^2+m)$
  - Algorithm J is $O(n+m^2)$
  - Algorithm K uses H, then J
  - Therefore algorithm K is
    A. $O((n^2+m)+(n+m^2))$
    B. $O(n^2+m^2)$
    C. $O(n^3+m^3)$
    D. $O(n^3)$ because $n > m$

BEST, WORST, AND AVERAGE CASE COMPLEXITY

- An algorithm might not be uniform in all cases
- Example: linked list with one head pointer get(index) function
  - Best case: index = 0 -> O(1)
  - Worst case: index = n-1 -> O(n)
  - Average case: index = n/2 -> O(n)

TRACTABLE ALGORITHMS

- Algorithm is *tractable* if worst case complexity is polynomial (P)
  - Polynomial if for some value x and all values of n:
    - $n^x > O(f(n))$
- If a problem can be solved with a tractable algorithm, it is in class *P*
- Is Get(index) from a linked-list is in class P?
  - What's the complexity?
  - O(n)
  - Is there an x such that $n^x > O(n)$
  - Yes $x = 2$
- Is Towers of Hanoi in class P?
  - What's the complexity?
  - $O(2^n)$
  - Is there an x such that $n^x > O(2^n)$
  - No

WHAT IS THE ROLE OF THE INPUT SET?

- Consider the problem of testing whether a given integer is prime
  - This is an example of a decision problem
- Complexity appears to be $O(\sqrt{n})$
- There is an algorithm based on the number of digits (m) with respect to a base (b) [$b \geq 2$]
- Complexity is $O(b^{m/2})$
- Prime testing is a rich algorithmic area, beyond the scope of what we will discuss here.
  - It is not known if prime testing is in P

CLASS NP PROBLEMS

- NP = **nondeterministic** polynomial time
- If a guess and a verification of the answer can be completed in polynomial time, the problem is NP (P is a subset of NP)
- NP is for decision problems (like the prime number determination)

NP: SUM OF SUBSETS

- Given a set of input integers {s1, s2, …, sn} and a target sum C
  - Is there a subset of integers whose sum is C?
- Is there any thing we can do but guess and check? (Not really)
- So, we randomly select a subset…
- Verification is just to compute the sum O(n)
- The naïve algorithm is to pick every possible combination of integers and check them against the target sum until the answer is found
  - If no answer is found, then the answer is no.
- Generally – If the combination of this guess and check (where the check is in P) gives us the answer, then the problem is in NP. *(simple, right?!)*

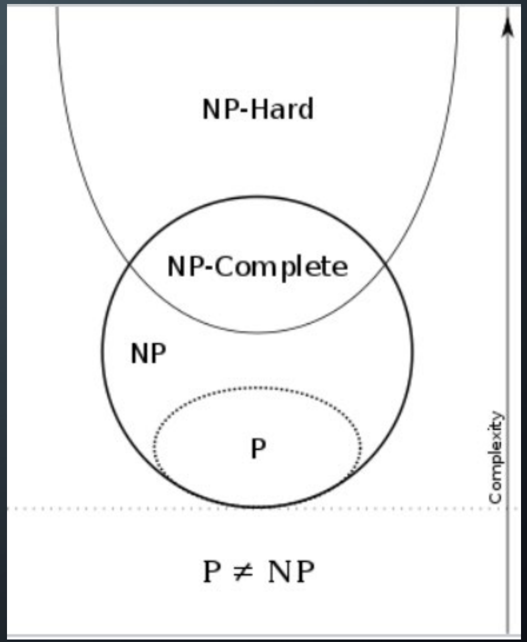
SO WHAT ABOUT TOWERS OF HANOI?

- Is Towers of Hanoi hard?
- Well it has a complexity of $O(2^n)$, that's not in P, right?
- Is Towers of Hanoi a decision problem?
- I can't think of a good way to describe it as a decision problem.
- Is there a P method to verify the solution is correct?
- Verifying the solution seems to be as difficult as solving the problem.

Seems like Tower of Hanoi is hard but not in NP

NP-HARD AND NP-COMPLETE PROBLEMS

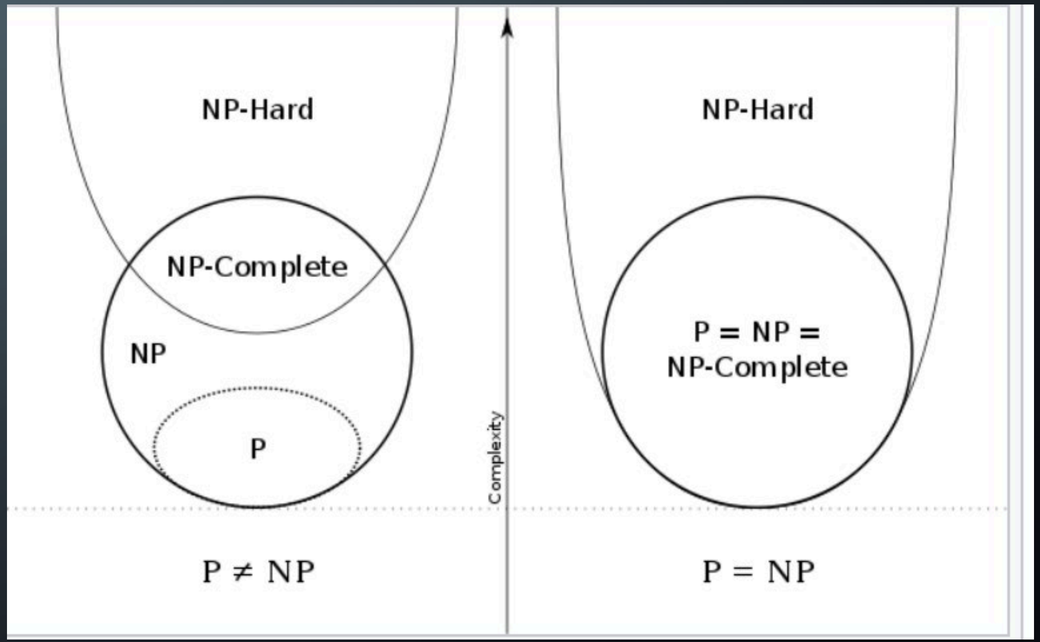
- NP-Complete
  - It is in the class NP
  - If Problems in NP-Complete can be solved in polynomial time, then all problems in NP can be solved in polynomial time
    - These problems are “complete” because all other NP problems can be reduced to them in polynomial time
- NP-Hard: Problems which are at least as hard as the hardest problems in NP.
  - Does not have to be in NP, does not have to be decision problems
  - NP-Hard because all problems in NP can be reduced to them in polynomial time (if there is a polynomial time solution for an NP-hard problem, there is a polynomial time solution for all NP problems)



- Which classification fits for Towers of Hanoi?

P=NP?

- it has not been proven that NP cannot be reduced to P (P != NP)
- If P=NP, then all decision problems which can be guessed and verified in polynomial time also have a polynomial time algorithm
- Consequences:
    - RSA encryption would be broken in polynomial time
    - AES encryption would be broken in polynomial time
    - Polynomial time solving of mathematical formal proofs
- One of the seven Millennium Prize Problems by the Clay Mathematics Institute, each of which carries a US$1,000,000 prize for the first correct solution.
- This stuff is SUPER IMPORTANT for CoEs
    - This is the extent to which we'll cover it in the course.
    - ***Strongly encourage you to take an algorithms course***



SORTING ALGORITHMS

- We will now examine sorting algorithms in the context of complexity
- Searching (binary search) and other algorithms greatly benefit from sorting

- Will cover:
  - Selection sort
  - Bubble sort (next time)
  - Insertion sort (next time)
  - Merge sort (next time)
  - Quick sort (next time)

SORTING TYPES

- Internal vs External
  - Internal: directly operates on the data structure, do not require additional storage
  - External: require additional data structures (required memory is > O(1))

SELECTION SORT

- Locate the smallest (or largest) item in the list
- Swap that key and the last key
- "Remove" the last key from the working list
- Repeat until list size is 1

Sort from smallest to largest

Initial List	87	68	99	71	66	59	60

swap

	87	68	60	71	66	59	99

swap

	59	68	60	71	66	87	99

swap

	59	68	60	66	71	87	99

swap

Final Sorted List	59	60	66	68	71	87	99

SELECTION SORT PSUEDOCODE

- How many compares, worst case?
  - n*(n-1)/2
- How many moves, worst case?
  - 3*(n-1)
- How many moves + compares?
  - $O(n^2)$
- What is different in the best case?

```
function SelectionSort( list )
{
  n = list.size() - 1
  for (last =n; n>=0; n--)
  {
    // find the largest entry
    max = 0
    for (i = 1; i<=last; i++)
    {
      if (list[i] > list[max])
        max = i
    }
    // swap the entries
    temp = list[last]
    list[last] = list[max]
    list[max] = temp
  }
}
```

Algorithm	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort		
Insertion Sort		
Merge Sort		
Quick Sort		

ASSIGNMENT/HOMEWORK

- Read Carrano Chapter 325 - 349
- ICE 5 due today
- HW4 released today: Carrano Chapter 10 Exercises 1, 3, 6, 9