Today

- Announcements
  - Final exam: self-scheduled; 1 sheet of notes allowed

- Computational Complexity
  - Big-O notation to describe # of operations
  - Example: Elementary sorting methods are $O(n^2)$
  - Divide-and-Conquer sorting methods are $O(n \log n)$

- Computability
  - Turing Machines
Binary Search

Fast search of a sorted list

Compare middle element to the target, then refine search to one half of list

| 2 | 5 | 8 | 11 | 15 | 16 | 21 | 24 | 29 | 41 | 45 | 58 | 71 | 85 | 92 | 95 |

Number of operations for a list of $n$ elements: $O(\log_2 n)$ or $O(\log n)$
Binary Search: $O(\log n)$

- Time required to execute the algorithm
- Time increasing by decreasing increments
- Length increasing by uniform increments
- Length of list
How to sort n values into increasing order?
Elementary Sorting Algorithms

Selection Sort $\rightarrow O(n^2)$

Find smallest value and swap into first position. Repeatedly select the smallest n-1 times.

Insertion Sort $\rightarrow O(n^2)$

Examine each element in turn, inserting it into its proper position among the already sorted values to the left.

Bubble Sort $\rightarrow O(n^2)$

Swap neighboring values if out of order (largest bubbles to end). Do this n-1 times.
Elementary sort algs are $O(n^2)$
Merge Sort

If a list has only 1 item, then we’re done

Else:
  1. Split the list in half
  2. Recursive sort each half
  3. Merge the two sorted halves together
Merge Sort Complexity

- How many **levels of splits**?
- How many **basic operations at each level**?
  - Hint: how many basic operations to merge two lists of size $n/2$?
  - Hint: how many basic operations to merge four lists of size $n/4$?
- Merge sort complexity is $O(\log n) \times O(n) = O(n \log n)$
Exponentiation

\[ a^b = a \times a \times \ldots \times a \text{ (b times)} \]

How can we define \( a^b \) recursively?

\[ a^b = \begin{cases} 
1 & \text{if } b = 0 \\
 a \times a^{b-1} & \text{if } b > 0 
\end{cases} \]

def expt(a, b):
    if b == 0:
        return 1
    else:
        return a * expt(a, b-1)

How do \( a \) and \( b \) affect the running time?
Fast Exponentiation

Is there a “better” recursive definition?

We can use the fact that $a^b = (a^{b/2})^2$

$$a^b = \begin{cases} 
1 & \text{if } b = 0 \\
(a^{b/2})^2 & \text{if } b \text{ even} \\
a \times (a^{b-1}) & \text{if } b \text{ odd}
\end{cases}$$

```python
def fastExpt(a, b):
    if b == 0:
        return 1
    elif b % 2 == 0:
        return square(fastExpt(a, b/2))
    else:
        return a * fastExpt(a, b-1)
```

Now how do $a$ and $b$ affect the running time?
Complexity Comparison

- How does the number of calls to \texttt{expt} or \texttt{fastExpt} increase as \( b \) increases?

- Exponentiation
  - Number of calls grows in linear proportion to \( b \)
  - Thus, \texttt{expt} is an \( O(b) \) algorithm

- Fast-Exponentiation
  - Number of calls grows incrementally as \( b \) doubles
  - Thus, \texttt{fastExpt} is an \( O(\log b) \) algorithm
Summary

- **Computational complexity** describes how the time needed to execute an algorithm increases as its input size increases.

- **Big-O notation** summarizes an algorithm’s complexity in terms of its input size:
  - Linear search: $O(n)$
  - Binary search: $O(\log n)$
  - Selection sort: $O(n^2)$
  - Insertion sort: $O(n^2)$
  - Bubble sort: $O(n^2)$
  - Merge sort: $O(n \log n)$
  - Fast exponentiation to power $n$: $O(\log n)$
NP-Completeness

Some problems are solvable but \textit{intractable}

Examples:

- Traveling Salesperson: find minimum length tour
- Partition: split values of a set into two subsets with equal sums
- Hamiltonian circuit: visit each vertex of a graph once
- Satisfiability: find values to satisfy a boolean formula
NP-Completeness

\[ P = \{ \text{problems solvable in deterministic polynomial time} \} \]

\[ \text{NP} = \{ \text{problems solvable in non-deterministic polynomial time} \} \]

Open problem: is \( P = \text{NP} \)?

A problem is in NP if we can guess a solution and verify it in polynomial time.
What is computable?

Are there non-computable functions?

Could there exist a machine that would correctly determine whether any theorem is true or false?

David Hilbert, 1928: Entscheidungsproblem (Decision problem): Might there exist a process for determining whether any given mathematical assertion is provable?

Language description → Decision process → True / False

Theorem in language →
Alan Turing (1912-1954)

- British mathematician, founder of field of computer science
- 1936 paper describes a machine that captures notion of what is “computable”
- 1939-1945 worked at British Foreign Office cracking German Enigma codes, critical to Allied win of WWII
- 1950 proposed Turing test for machine intelligence
- 1952 arrested for homosexuality
- 1954 death by apparent suicide

ACM awards annual Turing award for contributions to computing field
The Turing Test

Test for machine intelligence

- Judge has conversation via terminal with a human and a computer
- Loebner contest held annually since 1990
- Turing (wrongly) predicted that computers would pass the Turing test by 2000
Turing Machines

Proposed by Alan Turing in 1936

Mathematical model of a computer
- Finite number of states
- Rules for transitions between states
- Infinite tape for reading / writing
A Turing Machine Example

Determine if an input string is a palindrome

a'  b'  a'  a'  b'  b'  b'  b'  a''  a''  b'  a''

states
The Halting Problem

Given a computer program (ie, a Turing Machine and an input) will it ever halt?

Turing showed that halting is undecidable

There is no effective method to determine if a program will halt or run forever