Announcement

Programming assignment #4 has been posted in Blackboard and course website

Due at 11:59pm, Sunday, April 24th
Announcement

According to the UofSC final exam schedule [Final Exam Schedule Spring 2022 - University Registrar | University of South Carolina](https://example.com)

Final exam: **May 3, Tuesday, 9:00am – 11:30 am**

Cover all materials in our lectures

Closed-book and closed-notes.

A double-sided letter-size cheat sheet is allowed
## Huffman Coding Example

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td>0.2</td>
<td>0.15</td>
<td>0.05</td>
<td>0.1</td>
<td>0.15</td>
<td>0.05</td>
<td>0.1</td>
<td>0.15</td>
<td>0.05</td>
</tr>
</tbody>
</table>
Average number of bits per character (code length):

\[
0.2 \times 2 + 0.15 \times 3 + 0.05 \times 5 + 0.1 \times 3 + 0.15 \times 3 + 0.05 \times 5 + 0.1 \times 3 + 0.15 \times 3 + 0.05 \times 4 = 3.05
\]

Note:
The resulting Huffman tree varies according to your choices, e.g., assigning 0/1 to left/right
But the average code length is the same
Reading Assignments

Read Chapter 10. Iterative Improvement
# Chapter 11: Limitations of Algorithm Power

Basic Asymptotic Efficiency Classes (big $O$, big $\Theta$, and big $\Omega$)

<table>
<thead>
<tr>
<th>Class</th>
<th>n=10</th>
<th>n=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</td>
<td>1 (with base 10)</td>
</tr>
<tr>
<td>$n$</td>
<td>linear</td>
<td>10</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>10 (with base 10)</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
<td>100</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
<td>1000</td>
</tr>
<tr>
<td>$2^n$</td>
<td>exponential</td>
<td>1024</td>
</tr>
<tr>
<td>$n!$</td>
<td>factorial</td>
<td>3,628,800</td>
</tr>
</tbody>
</table>
Polynomial-Time Complexity

Polynomial-time complexity: the complexity of an algorithm is

$$a_b n^b + a_{b-1} n^{b-1} + \ldots + a_1 n + a_0 \Rightarrow \Theta(n^b)$$

with a fixed degree $b>0$. Usually $b<=3$

If a problem can be solved in polynomial time, it is usually considered to be theoretically \textbf{tractable} in current computers.

When an algorithm’s complexity is larger than polynomial, i.e., exponential, theoretically it is considered to be too expensive to be useful – \textbf{intractable}
# Polynomial-Time Complexity

## Polynomial time complexity

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>( \log n )</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( n )</td>
<td>linear</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>quadratic</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>cubic</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>exponential</td>
</tr>
<tr>
<td>( n! )</td>
<td>factorial</td>
</tr>
</tbody>
</table>
## List of Problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorting</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Searching</strong></td>
<td>$O(n)$</td>
</tr>
<tr>
<td>All shortest paths in a graph</td>
<td>$O(</td>
</tr>
<tr>
<td><strong>Minimum spanning tree</strong></td>
<td>$O(</td>
</tr>
<tr>
<td>Assignment problem</td>
<td>$O(n!) \sim O(n^3)$</td>
</tr>
<tr>
<td><strong>Towers of Hanoi</strong></td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td><strong>Knapsack problem</strong></td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td><strong>Traveling salesman problem</strong></td>
<td>$O(n!) \sim O(n^2 2^n)$ – Current record 85,900 cities (Applegate et al. 2006)</td>
</tr>
</tbody>
</table>

Problem A can be solved by algorithms $a_1, a_2, \ldots, a_p$

Problem B can be solved by algorithms $b_1, b_2, \ldots, b_q$

We may ask

• Which algorithm is more efficient? This makes more sense when the compared algorithms solve the same problem
  – It’s not fair to compare selection sorting with Warshall’s algorithm

• Which problem is more complex? We may compare the complexity of the best algorithm for $A$ and the best algorithm for $B$

For each problem, we want to know the **lower bound**: the best possible algorithm’s efficiency for a problem $\rightarrow \Omega(.)$

**Tight** lower bound: we have found an algorithm in the this lower-bound efficiency class $\Theta(.)$. The constant factor makes the difference.
Many problems need to ‘read’ all the necessary items and write the ‘output’

→ Their sizes provide a trivial lower bound

Example:

1. Generate all permutations of \( n \) distinct items \( \rightarrow \Omega(n!) \)
   Why?
   Is this a tight lower bound? \textbf{Yes.}

2. Evaluate the polynomial at a given \( x \)
   \[
   a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0
   \]

   \( \rightarrow \Omega(n) \), Is this tight? \textbf{Yes.}
Notes on Trivial Lower Bound

Multiplying two $nxn$ matrices $\rightarrow \Omega(n^2)$
- because we need to process $2n^2$ elements and output $n^2$ elements
- We do not know whether this is tight – a lower bound of $\Omega(n^2 \log n)$ has been proven Raz 2002

Many trivial lower bounds are too low to be useful
- TSP $\rightarrow \Omega(n^2)$ because its input is $n(n-1)/2$ intercity distance and output is $n+1$ city in sequence
- There is no known polynomial-time algorithm to solve it

Trivial lower bound sometime have problems
- We do not need to process all the input elements
- For example: searching an element in a sorted array. What is its complexity?
For each problem, we want to know the **lower bound**: the best possible algorithm’s efficiency for a problem $\rightarrow \Omega(.)$.

**Tight** lower bound: we have found an algorithm in the this lower-bound efficiency class $\Theta(.)$.

**Trivial** lower bound: the problem’s input/output size
- Too low
- Too high
Information-Theoretic Arguments

This approach seeks to establish a lower bound based on the amount of information it has to produce – an information-theoretic lower bound.

Recall the problem of guessing the number from 1...n by asking ‘yes/no’ questions.

Fundamentally, it is a coding problem. If the input number can be encoded into m bits, each ‘yes/no’ question just resolve one bits and therefore, the lower bound is m.

We know that \( m = \log_2 n \).

**Solution:** a decision tree. We will apply the decision tree to find the lower bound for several problems.
Find the Smallest from three numbers using comparison

Leaves in the decision tree is the possible output. The output size is at least 3 (maybe larger than 3) here

Complexity for the worst case: the height of this decision tree

Given $L$ leaves, the height of the binary tree is at least $\left\lceil \log_2 L \right\rceil$
### Decision Tree for Sorting Algorithms

The number of leaves: $n!$

The lower bound for **worst case**:

$$\left\lfloor \log_2 n! \right\rfloor \approx \log_2 \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \approx n \log_2 n$$

Is this tight?

**Selection Sort**

```
<table>
<thead>
<tr>
<th>a &lt; b</th>
<th>abc</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>abc</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>b &lt; c</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>a &lt; c</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>b &lt; c</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>b &lt; a</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>a &lt; c</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>b &lt; a</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>c &lt; b</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>c &lt; a</td>
</tr>
</tbody>
</table>
```

*Average number of comparisons:* $(3+3+3+3+3+3)/6 = 3 = \lfloor \log_2 6 \rfloor$
Example: Decision Tree for Insertion Sort

Average number of comparisons: \( (2+3+3+2+3+3)/6 \approx 2.666 > \log_2 6 \)

Worst case: 3 comparisons = \([\log_2 6]\)
Decision Tree for Searching a Sorted Array

Decision tree for binary search in a four-element array

# of leaves for n elements \( \rightarrow n+n+1 \)

\( \rightarrow \) lower bound for the worst case: \( \left\lceil \log_3 (2n + 1) \right\rceil = \log_3 9 = 3 \)

\( \rightarrow \Omega(\log n) \)
Binary Search → Binary Decision Tree

Lower bound is then $\left\lceil \log_2(n + 1) \right\rceil$ → A tight lower bound

Leaves → unsuccessful search

Parent nodes → successful search
As we discussed, problems that can be solved in polynomial time are usually called **tractable** and the problems that cannot be solved in polynomial time are called **intractable**, now

*Is there a polynomial-time algorithm that solves the problem?*

Possible answers:

- yes
- no
  - because it can be proved that all algorithms take exponential time
  - because it can be proved that no algorithm exists at all to solve this problem
- don’t know
- don’t know, but if such algorithm were to be found, then it would provide a means of solving many other problems in polynomial time
Types of Problems

Two types of problems:

- **Optimization problem**: construct a solution that maximizes or minimizes some objective function
  - MST, all shortest paths, single source shortest paths, …

- **Decision problem**: answer yes/no to a question
  - Selection, searching, …

Many problems have BOTH decision and optimization versions.

Eg: Traveling Salesman Problem

- *optimization*: find Hamiltonian cycle of minimum weight
- *decision*: Is there a Hamiltonian cycle of weight $< k$

**Hamiltonian Circuit**: a closed path in a graph that visits every node in the graph exactly once
Deterministic VS Nondeterministic Algorithm

A deterministic algorithm is the algorithm we discussed before
• A math function: given a specific input, generate the same and unique output in different runs

A nondeterministic algorithm is the counterpart
• May have different outputs in different runs
• It is a two-stage process:
  – Guessing stage: generate a random string $S$ as a candidate solution
  – Verification stage: using a deterministic algorithm which takes the original input $I$ and $S$ as input and determine if $S$ is a solution to $I$

Why becomes nondeterministic?
• System noise
• random number generator
Example: Conjunctive Normal Form (CNF) Satisfiability

**Problem:** Is a Boolean expression in its conjunctive normal form (CNF), i.e., are there “true” or “false” assignments of these variables that makes the Boolean expression true?

This problem is in \( NP \).

**Nondeterministic algorithm:**

- Guess truth assignment
- Check assignment to see if it satisfies CNF formula

Example: (Boolean operation)

\[
(a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor b) \land (\overline{a} \lor \overline{b} \lor \overline{c})
\]

\( \lor \) is logic “or” \( \land \) is logic “and” or “logical conjunction”

**Truth assignments:** \( a = true, b = true, c = false \) \( \Rightarrow \)

the entire expression = true

Checking phase: \( \Theta(n) \)
Deterministic VS Nondeterministic Algorithm

A problem can have BOTH *deterministic* and *nondeterministic* algorithms

Example:

**Shortest path problem:** find the shortest path from \( a \) to \( b \) in a weighted graph

- **Deterministic algorithm:** searching the shortest path (brute force enumerating)

- **Nondeterministic algorithm:** generate a path \( P \) and decide whether \( P \) is a simple path (all vertices on the path are distinct) from \( a \) to \( b \) of length\( \leq \) Threshold.
The Class P & NP

*P*: the class of decision problems that are solvable by deterministic algorithms in \( O(p(n)) \), where \( p(n) \) is a polynomial on \( n \)

*NP*: the class of decision problems that are solvable in polynomial time by nondeterministic algorithms

Thus *NP* can also be thought of as the class of problems

- whose solutions can be verified in polynomial time; or
- that can be solved in polynomial time on a machine that can pursue infinitely many paths of the computation in parallel

Note that *NP* stands for “Nondeterministic Polynomial-time”

All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

\[ P \subseteq NP \]
**NP-Complete problems**

A decision problem $D$ is **NP-complete** iff

1. $D \in \text{NP}$
2. every problem in $\text{NP}$ is polynomial-time reducible to $D$

The class of **NP-complete** problems is denoted **NPC**
Polynomial Reductions

A decision problem $D_1$ is said to be polynomial reducible to a decision problem $D_2$ if there exists a function $f$ that transforms instances of $D_1$ to instances of $D_2$ such that

1. $f$ maps all “yes” instances of $D_1$ to “yes” instances of $D_2$ and all “no” instances of $D_1$ to “no” instances of $D_2$

2. $f$ is computable by a polynomial-time algorithm

If $D_2$ can be solved in polynomial time $\rightarrow D_1$ can be solved in polynomial time
Polynomial Reductions

Example: Polynomial-time reduction of Hamiltonian Circuit to decision version of Traveling Salesman Problem (Is there a solution of TSP with total distance no larger than \( k=n \)) given integer distance

Hamiltonian Circuit: a closed path in a graph that visits every node in the graph exactly once

Traveling Salesman: find the shortest path that visits every city exactly once and returns to the origin
Polynomial Reductions

- If $G$ has a Hamiltonian cycle, $G'$ has a cycle w/ weight $n$

What does this prove?

- If HC is NPC $\rightarrow$ TSP(D) is NPC? or
- If TSP(D) is NPC $\rightarrow$ HC is NPC?
To Prove a Decision Problem is in NPC

1. Prove it is in $NP$ (verification takes polynomial time)

2. Prove that all problems in $NP$ is reducible to this problem

3. Or Prove that a known $NPC$ problem is reducible to this problem

BIG problem: If we can prove any given $NPC$ problem can be solve in polynomial time $\Rightarrow P=NP$