Outline

Today:

- P, NP, NP-Complete, NP-hard, PSPACE definitions
- Graph Coloring Problem
- Convex Hull
- Dynamic Programming: All-pair shortest path
P, NP, and NP-Complete

If there’s an algorithm to solve a problem that runs in polynomial time, the problem is said to be in the set $P$.

If the outcome of an algorithm to solve a problem can be verified in polynomial time, the problem is said to be in the set $NP$ (non-deterministic polynomial, the “non-determinism” refers to the outcome of the algorithm, not the verification).

There is a set of problems in NP for which if there’s a polynomial solution to one there will be a polynomial solution to all. The set is called $NP$-Complete.
NP-Complete, NP-Hard

If you can show that a problem is equivalent (can be reduced) to a known NP-Complete problem, you may as well not try to find an efficient solution for it (unless you’re convinced you’re a genius)

If such a polynomial solution exists, \( P = NP \)

It is not known whether \( P \subset NP \) or \( P = NP \)

**NP-hard** problems are at least as hard as an NP-complete problem, but NP-complete technically refers only to decision problems, whereas NP-hard is used to refer to optimization problems
PSPACE

If a problem can be solved by an algorithm that uses an amount of space polynomial in the size of its input, the problem is said to be in the set PSPACE.

It is known that $P \subseteq \text{PSPACE}$ and $\text{NP} \subseteq \text{PSPACE}$, but not whether $P \neq \text{PSPACE}$.
Examples of NP-Complete Problems

Hamiltonian Cycle Problem

Traveling Salesman Problem

0/1 Knapsack Problem

Graph Coloring Problem: can you color a graph using $k \geq 3$ colors such that no adjacent vertices have the same color?
Graph Coloring Problem

Brute-force BnB algorithm:
• branch:
• bound:

Running time:

Theorem: any planar graph can be colored using 4 colors

Planar graph: a graph that can be drawn on a plane such that no two edges cross each other
Application of Graph Coloring Problem

Register allocation:
- want local variables in registers
- each local variable represented as a node
- if the lifetime/scope of 2 variables overlap, draw an edge between the two nodes
- can the variables be stored in the $k$ available registers?

Example: for a 4-register CPU, can we store all local variables inside the for loop in the registers?

```c
f (int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        int u, v;
        // . . . statements involving i, u, v
        j = u + v;
    }
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Graph Exercises

1. Strongly connected graph
   (a) Given the graph in Fig. 1, how many minimal number of additional edges are needed to make the graph strongly connected?
   (b) Draw the graph with the additional edges.

2. MST and SPF
   (a) Given the graph in Fig. 2, assuming one builds an MST and an SPF starting from node $A$, assign minimal, non-negative, integer weights to the edges such that the MST is different from the SPF.
   (b) Draw the resulting MST and SPF.
2D Graphics Primitives

**Point:** \((x,y)\)

**Line:**
- two end points
- line formed by drawing all points in between the two end points

**Polygon:**
- defined by vertices
- closed: all lines connected
- draw one line at a time
- color (shading)
Lines

Two points \((x_1, y_1), (x_2, y_2)\) form a line:

\[
\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}
\]

\[
y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1
\]

\[
y = mx + b
\]

where \(m = \frac{y_2 - y_1}{x_2 - x_1}\), and \(b = y_1 - mx_1\)

Careful that we are usually only dealing with a line segment

\(y_3 = mx_3 + b\) is on the above line segment iff:

\[
MIN(x_1, x_2) \leq x_3 \leq MAX(x_1, x_2)
\]

and

\[
MIN(y_1, y_2) \leq y_3 \leq MAX(y_1, y_2)
\]
Line Intersection and Relative Position

If \( y = m_1 x + b_1 \) intersects \( y = m_2 x + b_2 \) at \((x_i, y_i)\),
\[
x_i = \frac{b_2 - b_1}{m_1 - m_2} \quad \text{and} \quad y_i = \frac{b_2 m_1 - b_1 m_2}{m_1 - m_2}
\]

Given a line segment between \((x_1, y_1)\) and \((x_2, y_2)\),
a point at \((x_3, y_3)\) is to the right of the line segment if
\[
(y_3 - y_1)(x_2 - x_1) - (x_3 - x_1)(y_2 - y_1) < 0,
\]
that is, the slope \(\frac{y_3 - y_1}{x_3 - x_1}\) is smaller than the slope \(\frac{y_2 - y_1}{x_2 - x_1}\)
Orientation of Three Points

Given an ordered triplet \((p, q, r)\) of points, if going from \(p\) to \(q\) to \(r\)

- the angle that stays on the left hand side is \(< \pi\), they are said to make a **left turn** or is oriented **counterclockwise**
- the angle that stays on the right hand side is \(< \pi\), they are said to make a **right turn** or is oriented **clockwise**
- the angle is \(\pi\), they are **collinear**

\(p_1(x_1, y_1), p_2(x_2, y_2), \text{ and } p_3(x_3, y_3)\) make a left turn if

\[
\frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1}
\]

Line intersection can also be determined by checking the orientations of three of the four end points
Convex Hull

A polygon is **simple** if all its edges intersect only at its vertices

A polygon is **convex** if it is simple and all its internal angles are $< \pi$

The **convex hull** of a set of points is the **boundary** of the smallest convex region that contains all the points in the region or on the boundary

Think of tying a rubber band around a set of pegs nailed on a plank

Useful for collision detection and path planning by robots (including software robots in games), for example
Jarvis’s March or Gift Wrapping Algorithm

Algorithm:

• identify \( a \) the anchor point of the convex hull with minimum \( y \)-coordinate (and minimum \( x \)-coordinate if there are ties)
• the next convex hull vertex \( (b) \) is the point with the smallest polar angle with respect to \( a \) (in case of ties, pick the point with the largest \( x \)-coordinate)
• similarly the next vertex \( c \) has the smallest polar angle with respect to \( b \), etc.

What is the running time complexity of the algorithm?
Gift Wrapping Time Complexity

Finding the anchor point takes $O(n)$ time

**Radial comparator**: compare the polar angles of two points with respect to a third by checking the orientation of the three points; the next vertex on the convex hull will have a left turn to all other vertices with respect to the current vertex; running time $O(1)$

For each vertex, it takes $O(n)$ comparisons to find the smallest polar angle

There are $h$ vertices to the convex hull, so the algorithm runs in $O(hn)$ time

Worst case, $h = n$ and running time is $O(n^2)$

Such algorithms are said to be **output sensitive**
Graham Scan Algorithm

Not output sensitive

Algorithm:

1. identify $a$ the anchor point of the convex hull with minimum $y$-coordinate (and minimum $x$-coordinate if there are ties)
2. sort the remaining points using the radial comparator with respect to $a$
3. let $H$ be the convex hull, initially $H = \{a\}$
4. consider the points in sorted order, for each new point $p$:
   - if $p$ forms a left turn with the last two points in $H$, or if $H$ contains less than 2 points, add $p$ to $H$
   - else remove that last point in $H$ and repeat the test for $p$
5. stop when all points have been considered; $H$ contains the convex hull
Running Time of Graham Scan

Finding the anchor point takes $O(n)$ time

Sorting the points takes $O(n \log n)$ (using heap-sort for example)

Adding a new point takes at most $2n$ times

Total time is $O(n \log n)$

If $h = |H| < \log n$, Gift Wrapping algorithm is faster
Computing the Fibonacci Sequence

What is a Fibonacci sequence?

How would you generate up to the $n$-th Fibonacci numbers?
Computing the Fibonacci Sequence (contd)

What is a Fibonacci sequence?
\[ f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n-2}, n \geq 2 \]

Recursive implementation:

```c
int rfib(int n)
{ // assume n >= 0
    return (n <= 1 ? n : rfib(n-1)+rfib(n-2));
}
```

Running time: \( \Omega((\frac{3}{2})^n) \)
[Preiss 3.4.3]

Iterative version:

```c
int ifib(int n)
{ // assume n >= 2
    int i, f[n];
    f[0] = 0; f[1] = 1;
    for (i = 2 to n) {
        f[i] = f[i-1]+f[i-2];
    }
    return f[n];
}
```

Running time: \( \Theta(n) \)
[Preiss 14.3.2, 14.4.1]

Sugih Jamin (jamin@eecs.umich.edu)
Computing the Fibonacci Sequence (contd)

Why is the recursive version so slow?

Why is the iterative version so fast?
Computing the Fibonacci Sequence (contd)

Why is the recursive version so slow?
The number of computations grows exponentially!
Each $rfib(i), i < n - 1$ computed more than once
Tree size grows almost $2^n$
Actually the number of base case computations in computing $f_n$ is $f_n$
Since $f_n > (\frac{3}{2})^{n-1}$ (see Preiss Thm. 3.9), complexity is $\Omega((\frac{3}{2})^n)$

Why is the iterative version so fast?
Instead of recomputing duplicated subproblems, it saves their results in an array and simply looks them up as needed

Can we design a recursive algorithm that similary look up results of duplicated subproblem?
Memoized Fibonacci Computation

```c
int fib_memo[n] = {0, 1, -1, . . . , -1};
int
mfib(int n, *fib_memo)
{ // assume n >= 0 and left to right evaluation
  if (fib_memo[n] < 0)
    fib_memo[n] = mfib(n-2, fib_memo) + mfib(n-1, fib_memo);
  return fib_memo[n];
}
```

Memoization (or tabulation): use a result table with an otherwise inefficient recursive algorithm
Record in table values that have been previously computed

Memoize only the last two terms:

```c
int
rfib2(int fn2, fn1, n)
{ // assume n >= 0
  return (n <= 1 ? n:
           rfib2(fn1, fn2+fn1, --n);
}
main() { return rfib2(0, 1, n); }
Devide et impera

Divide-and-conquer:

• for base case(s), solve problem directly
• do recursively until base case(s) reached:
  o divide problem into 2 or more subproblems
  o solve each subproblem independently
• solutions to subproblems combined into a solution
to the original problem

Works fine when subproblems are non-overlapping,
otherwise overlapping subproblems must be solved more than once
(as with the Fibonacci sequence)
Dynamic Programming

- used when a problem can be divided into subproblems that **overlap**
- solve each subproblem once and store the solution in a table
- if run across the subproblem again, simply look up its solution in the table
- reconstruct the solution to the original problem from the solutions to the subproblems
- the more overlap the better, as this reduces the number of subproblems

Origin of name (Bellman 1957):

**programming**: planning, decision making by a tabular method
**dynamic**: multi-stage, time-varying process
Dynamic Programming and Optimization Problem

DP used primarily to solve **optimization** problem, e.g., find the shortest, longest, “best” way of doing something

**Requirement**: an optimal solution to the problem *must* be a composition of optimal solutions to all subproblems
In other words, there must not be an optimal solution that contains suboptimal solution to a subproblem
All-Pairs Shortest Path

All-pairs shortest path problem:
Given a weighted, connected, directed graph \( G = (V, E) \), for all pairs of vertices in \( V \), find the shortest (smallest weighted) path length between the two vertices

First solution: run Dijkstra’s SPF algorithm \(|V|\) times
Runtime complexity of Dijkstra’s SPF: \( O(|E| \log |V|) = O(|V|^2 \log |V|) \)
Solution’s runtime: \( O(|V|^3 \log |V|) \)
Floyd’s Algorithm

A dynamic programming method for solving the all-pairs shortest path problem (on a dense graph)

Floyd’s algorithm:
• uses an adjacency matrix
• initially:
  o all nodes have distance 0 to itself, $D_0(v, v) = 0$
  o distance between directly connected nodes is the weight of the connecting edge, $D_0(u, v) = C(u, v)$
  o all other distances are set to $\infty$, $D_0(v, w) = \infty$

• add nodes to $V$ one at a time, for each node $v_i$ added, compare all distances with and without using this node:
  $D_i(v, w) = \text{MIN}(D_{i-1}(v, w), D_{i-1}(v, v_i) + D_{i-1}(v_i, w))$

Runtime: $O(|V|^3)$
Floyd’s Algorithm

floyd(G)
{
    D[*][*] = INFINITY; D[v][v] = 0;
    forall ((v,w) in E) D[v][w] = C(v,w);

    for (i = 0; i < n; i++)
        for (v = 0; v < n; v++)
            for (w = 0; w < n; w++)
                D[v][w] = MIN(D[v][w], D[v][i]+D[i][w]);
}
Floyd's Example (init $V = \emptyset$)

$V = \{\}$

\[\begin{array}{c|cccc}
   & a & b & c & d \\
\hline
  a & 0 & 5 & 1 & \infty \\
  b & 1 & 0 & \infty & \infty \\
  c & \infty & 3 & 0 & 1 \\
  d & \infty & 1 & \infty & 0 \\
\end{array}\]
Floyd’s Example \((V \cup \{a\})\)

\[ V = \{ \} \]

\[
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & 0 & 5 & 1 & \infty \\
b & 1 & 0 & \infty & \infty \\
c & \infty & 3 & 0 & 1 \\
d & \infty & 1 & \infty & 0 \\
\end{array}
\]

\[ V = \{a\} \]

\[
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & 0 & 5 & 1 & \infty \\
b & 1 & 0 & 2 & \infty \\
c & \infty & 3 & 0 & 1 \\
d & \infty & 1 & \infty & 0 \\
\end{array}
\]

Sugih Jamin (jamin@eecs.umich.edu)
Floyd’s Example \((V \cup \{b\})\)

\[ V = \{a\} \]

\[
\begin{array}{cccc}
  & a & b & c & d \\
 a & 0 & 5 & 1 & \infty \\
b & 1 & 0 & 2 & \infty \\
c & \infty & 3 & 0 & 1 \\
d & \infty & 1 & \infty & 0 \\
\end{array}
\]

\[ V = \{a, b\} \]

\[
\begin{array}{cccc}
  & a & b & c & d \\
 a & 0 & 5 & 1 & \infty \\
b & 1 & 0 & 2 & \infty \\
c & 4 & 3 & 0 & 1 \\
d & 2 & 1 & 3 & 0 \\
\end{array}
\]
Floyd’s Example \((V \cup \{c\})\)

\[
V = \{a, b\}
\]

\[
V = \{a, b, c\}
\]

\[
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & 0 & 5 & 1 & \infty \\
b & 1 & 0 & 2 & \infty \\
c & 4 & 3 & 0 & 1 \\
d & 2 & 1 & 3 & 0
\end{array}
\]

\[
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & 0 & 4 & 1 & 2 \\
b & 1 & 0 & 2 & 3 \\
c & 4 & 3 & 0 & 1 \\
d & 2 & 1 & 3 & 0
\end{array}
\]

Sugih Jamin (jamin@eecs.umich.edu)
Floyd’s Example \((V \cup \{d\})\)

\[ V = \{a, b, c\} \]

\[
\begin{array}{cccc}
  & a & b & c & d \\
\hline
  a & 0 & 4 & 1 & 2 \\
b & 1 & 0 & 2 & 3 \\
c & 4 & 3 & 0 & 1 \\
d & 2 & 1 & 3 & 0 \\
\end{array}
\]

\[ V = \{a, b, c, d\} \]

\[
\begin{array}{cccc}
  & a & b & c & d \\
\hline
  a & 0 & 3 & 1 & 2 \\
b & 1 & 0 & 2 & 3 \\
c & 3 & 2 & 0 & 1 \\
d & 2 & 1 & 3 & 0 \\
\end{array}
\]