Final exam timing

Time-limited take-home exam

Can take whenever you want and you’ll have 3 hours to take it

Will be available Monday, May 1

Must be submitted by
- Seniors: 12pm (noon) on Thursday, May 4
- Everyone else: 11:59pm on Wednesday, May 10

Run-time analysis

We’ve spent a lot of time in this class putting algorithms into specific run-time categories:
- $\Theta(\log n)$
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(n \log \log n)$
- $\Theta(n^{1.67})$
- ...

When I say an algorithm is $\Theta(f(n))$, what does that mean?
Tractable vs. intractable problems

**Tractable**

adj. 1. Easily managed or controlled; governable. 2. Easily handled or worked; malleable.

What is a “tractable” problem?

Tractable problems can be solved in $O(f(n))$ where $f(n)$ is a polynomial.

We’ll call $P$, the set of tractable problems.

What about...

$O(n^{100})$?

$O(n^{\log \log \log n})$?

Technically $O(n^{100})$ is tractable by our definition.

Why don’t we worry about problems like this?
Tractable vs. intractable problems

- **tractable**
  - $O(n^{100})$ is tractable by our definition
  - Few practical problems result in solutions like this
  - Once a polynomial time algorithm exists, more efficient algorithms are usually found
  - Polynomial algorithms are amenable to parallel computation

Solvable vs. unsolvable problems

- **solvable**
  - Possible to solve: solvable problems; a solvable riddle.

**What is a “solvable” problem?**

**Solvable** vs. **unsolvable** problems

Given $n$ integers, sort them from smallest to largest.

- **Tractable/intractable?**
- **Solvable/unsolvable?**
<table>
<thead>
<tr>
<th>Slide</th>
<th>Title</th>
<th>Text</th>
</tr>
</thead>
</table>
| 13    | Sorting | Given n integers, sort them from smallest to largest.  
Solvable and tractable:  
Mergesort: $\Theta(n \log n)$ |
| 14    | Enumerating all subsets | Given a set of n items, enumerate all possible subsets.  
Tractable/intractable?  
Solvable/unsolvable? |
| 15    | Enumerating all subsets | Given a set of n items, enumerate all possible subsets.  
Solvable, but intractable: $\Theta(2^n)$ subsets  
For large n this will take a very, very long time |
| 16    | Halting problem | Given an arbitrary algorithm/program and a particular input, will the program terminate?  
Tractable/intractable?  
Solvable/unsolvable? |
**Halting problem**

Given an arbitrary algorithm/program and a particular input, will the program terminate?

Unsolvable 😞

---

**Integer solution?**

Given a polynomial equation, are there integer values of the variables such that the equation is true?

\[ x^3yz + 2y^4z^2 - 7xy^3z = 6 \]

Tractable/intractable?

Solvable/unsolvable?

---

**Integer solution?**

Given a polynomial equation, are there integer values of the variables such that the equation is true?

\[ x^3yz + 2y^4z^2 - 7xy^3z = 6 \]

Unsolvable 😞

---

**Hamiltonian cycle**

Given an undirected graph \( G = (V, E) \), a hamiltonian cycle is a cycle that visits every vertex \( V \) exactly once.
Hamiltonian cycle

Given an undirected graph $G=(V, E)$, a hamiltonian cycle is a cycle that visits every vertex $V$ exactly once.

Tractable/intractable?

Solvable/unsolvable?
Hamiltonian cycle

Given an undirected graph, does it contain a hamiltonian cycle?

**Solvable:** Enumerate all possible paths (i.e. include an edge or don’t) check if it’s a hamiltonian cycle

How would we do this check exactly, specifically given a graph and a path?

Checking hamiltonian cycles

```
HAM-CYCLE-VISIT(G, p)
1. for i = 1 to |V|
2. visited[i] = false
3. n = length(p)
4. if p_i \neq p_{i+1} or n \neq |V| + 1
5. return false
6. visited[p_i] = true
7. for i = 1 to n - 1
8. if visited[p_i]
9. return false
10. if \{p_i, p_{i+1}\} \notin E
11. return false
12. visited[p_{i+1}] = true
13. for i = 1 to |V|
14. if visited[i]
15. return false
16. return true
```

NP problems

**NP** is the set of problems that can be verified in polynomial time

A problem can be verified in polynomial time if you can check that a given solution is correct in polynomial time

(NP is an abbreviation for non-deterministic polynomial time)
Checking hamiltonian cycles

```
HAM-CYCLE-VERIFY(G, p)
1. for i ← 1 to |V|
2. visited[i] ← false
3. n ← length(p)
4. if p_i ≠ p_{i+1} or i ≠ |V| + 1
5. return false
6. visited[p_i] ← true
7. for i ← 1 to n − 1
8. if visited[p_i]
9. return false
10. if (p_i, p_{i+1}) ∉ E
11. return false
12. visited[p_i] ← true
13. for i ← 1 to |V|
14. if visited[i]
15. return false
16. return true
```

Running time?

- O(V) adjacency matrix
- O(V+E) adjacency list

What does that say about the hamiltonian cycle problem?

It belongs to NP

NP problems

Why might we care about NP problems?

- If we can’t verify the solution in polynomial time then an algorithm cannot exist that determines the solution in this time (why not?)
- All algorithms with polynomial time solutions are in NP

The NP problems that are currently not solvable in polynomial time could in theory be solved in polynomial time

P and NP

Big-O allowed us to group algorithms by run-time

Today, we’re talking about sets of problems grouped by how easy they are to solve

Reduction function

Given two problems P_1 and P_2 a reduction function, f(x), is a function that transforms a problem instance x of type P_1 to a problem instance of type P_2 such that: a solution to x exists for P_1 iff a solution for f(x) exists for P_2

```
x -- f -- f(x)
P_i instance
```


Reduction function

Where have we seen reductions before?
- Bipartite matching reduced to flow problem
- All pairs shortest path through a particular vertex reduced to single source shortest path

Why are they useful?

Reduction function: Example

Reduction function (f): Given any bipartite matching problem turn it into a network flow problem

P1 = Bipartite matching
P2 = Network flow

What is f and what is f'?
Reduction function: Example

A problem is NP-complete if:
1. it can be verified in polynomial time (i.e. in NP)
2. any NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

\[ A \leq_p B ; A \text{ is polynomial time reducible to } B \]
A problem is **NP-complete** if:
1. it can be verified in polynomial time (i.e. in NP)
2. any NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

The hamiltonian cycle problem is NP-complete

What are the implications of this?
What does this say about how hard the hamiltonian cycle problem is compared to other NP-complete problems?

If a polynomial-time solution to the hamiltonian cycle problem is found, we would have a polynomial time solution to any NP-complete problem:
- Take the input of the problem
- Convert it to the hamiltonian cycle problem (by definition, we know we can do this in polynomial time)
- Solve it
- If yes output yes, if no, output no

If I found a polynomial-time solution to the hamiltonian cycle problem, what would this mean for the other NP-complete problems?
Similarly, if we found a polynomial time solution to any NP-complete problem we’d have a solution to all NP-complete problems.

**NP-complete problems**

3D matching
- Bipartite matching: given two sets of things and pair constraints, find a matching between the sets
- 3D matching: given three sets of things and triplet constraints, find a matching between the sets of size at least K

**NP-complete problems**

Longest path
Given a graph G with nonnegative edge weights does a simple path exist from s to t with weight at least g?

**P vs. NP**

<table>
<thead>
<tr>
<th>Polynomial time solutions exist</th>
<th>NP-complete (and no polynomial time solution currently exists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest path</td>
</tr>
<tr>
<td>Bipartite matching</td>
<td>3D matching</td>
</tr>
<tr>
<td>Linear programming</td>
<td>Integer linear programming</td>
</tr>
<tr>
<td>Minimum cut</td>
<td>Balanced cut</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Proving NP-completeness

A problem is **NP-complete** if:

1. it can be verified in polynomial time (i.e. in NP)
2. any NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

**Ideas?**

---

Proving NP-completeness

Given a problem NEW to show it is NP-Complete

1. Show that NEW is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time
2. Show that NEW is NP-Hard (i.e., all NP-complete problems are reducible to NEW in polynomial time)
   a. Describe a reduction function \( f \) from a known NP-Complete problem to NEW
   b. Show that \( f \) runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists to the NEW problem generated by \( f \)

---

Proving NP-completeness

Show that a solution exists to the NP-Complete problem IFF a solution exists to the NEW problem generated by \( f \)

- Assume we have an NP-Complete problem instance that has a solution, show that the NEW problem instance generated by \( f \) has a solution
- Assume we have a problem instance of NEW generated by \( f \) that has a solution, show that we can derive a solution to the NP-Complete problem instance

Other ways of proving the IFF, but this is often the easiest

---

Proving NP-completeness

Show that all NP-complete problems are reducible to NEW in polynomial time

Why is it sufficient to show that one NP-complete problem reduces to the NEW problem?
Proving NP-completeness

Show that all NP-complete problems are reducible to NEW in polynomial time

All others can be reduced to NEW by first reducing to the one problem, then reducing to NEW. Two polynomial time reductions is still polynomial time!

NP-complete: 3-SAT

A boolean formula is in n-conjunctive normal form (n-CNF) if:
- It is expressed as an AND of clauses
- Where each clause is an OR of no more than n variables

\[(a \lor \neg a \lor \neg b) \land (c \lor b \lor d) \land (\neg a \lor \neg c \lor \neg d)\]

3-SAT: Given a 3-CNF boolean formula, is it satisfiable?

3-SAT is an NP-complete problem

NP-complete: SAT

Given a boolean formula of n boolean variables joined by m connectives (AND, OR or NOT) is there a setting of the variables such that the boolean formula evaluate to true?

\[(a \land b) \lor (\neg a \land \neg b)\]

\[((\neg (b \lor \neg c)) \land a) \lor (a \land b \land c) \land c \land \neg b\]

Is SAT an NP-complete problem?
**NP-complete: SAT**

Given a boolean formula of \( n \) boolean variables joined by \( m \) connectives (AND, OR or NOT) is there a setting of the variables such that the boolean formula evaluate to true?

\[
((\neg b \lor \neg c) \land \neg a) \lor (a \land \neg b \land c) \lor \neg a \land \neg b
\]

1. Show that SAT is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time
2. Show that NEW is NP-Hard (i.e., all NP-complete problems are reducible to NEW in polynomial time)
   a. Describe a reduction function \( f \) from a known NP-Complete problem to SAT
   b. Show that \( f \) runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists to the SAT problem generated by \( f \)

**NP-Complete: SAT**

Verifier: A solution consists of an assignment of the variables
- If clause is a single variable:
  - return the value of the variable
- otherwise
  - for each clause:
    - call the verifier recursively
    - compute a running solution

- at most a linear number of recursive calls (each call makes the problem smaller and no overlap)
- overall polynomial time
NP-Complete: SAT

- Assume we have a 3-SAT problem with a solution:
  - Because 3-SAT problems are a subset of SAT problems, then the SAT problem will also have a solution
- Assume we have a problem instance generated by our reduction with a solution:
  - Our reduction function simply does a copy, so it is already a 3-SAT problem
  - Therefore the variable assignment found by our SAT-solver will also be a solution to the original 3-SAT problem

Show that a solution exists to the NP-Complete problem IFF a solution exists to the NEW problem generated by f

- Assume we have an NP-Complete problem instance that has a solution, show that the NEW problem instance generated by f has a solution
- Assume we have a problem instance of NEW generated by f that has a solution, show that we can derive a solution to the NP-Complete problem instance

NP-Complete problems

Why do we care about showing that a problem is NP-Complete?

- We know that the problem is hard (and we probably won’t find a polynomial time exact solver)
- We may need to compromise:
  - reformulate the problem
  - settle for an approximate solution
- Down the road, if a solution is found for an NP-complete problem, then we’d have one too…”

CLIQUE

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices that are fully connected, i.e. every vertex in $V'$ is connected to every other vertex in $V'$

CLIQUE problem: Does $G$ contain a clique of size $k$?

Is there a clique of size 4 in this graph?

CLIQUE is an NP-Complete problem

CLIQUE

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices that are fully connected, i.e. every vertex in $V'$ is connected to every other vertex in $V'$

CLIQUE problem: Does $G$ contain a clique of size $k$?
HALF-CLIQUE

Given a graph $G$, does the graph contain a clique containing exactly half the vertices?

Is HALF-CLIQUE an NP-complete problem?

Is Half-Clique NP-Complete?

1. Show that Half-Clique is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time
2. Show that Half-Clique is NP-Hard (i.e., all NP-complete problems are reducible to Half-Clique in polynomial time)
   a. Describe a reduction function $f$ from a known NP-Complete problem to Half-Clique
   b. Show that $f$ runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists to the Half-Clique problem generated by $f$