What makes a problem hard?

Matvey Soloviev (Cornell University)

CS 4820, Summer 2020

How fast can we solve a problem?

For any computational problem like sorting or maxflow, there are multiple algorithms.

How fast can we solve a problem?

For any computational problem like sorting or maxflow, there are multiple algorithms.

People keep coming up with new ones. (Simpler, more interesting, and in particular faster.)

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m) \log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m) \log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...
- Maxflow:

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m)\log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...
- Maxflow:
  - Ford-Fulkerson in $\Theta(m|f^*|)$.

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m)\log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...
- Maxflow:
  - Ford-Fulkerson in $\Theta(m|f^*|)$.
  - Edmonds-Karp in $\Theta(nm^2)$.

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m)\log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...
- Maxflow:
  - Ford-Fulkerson in $\Theta(m|f^*|)$.
  - Edmonds-Karp in $\Theta(nm^2)$.
  - Dinic's in $\Theta(n^2 m)$. (You'll see this in CS 6820.)

Examples of different algorithms

- Sorting: Bubblesort in $\Theta(n^2)$, mergesort in $\Theta(n \log n)$, and many others.
- Shortest paths: Dijkstra ('56) in $\Theta((n+m)\log n)$, Fredman-Tarjan ('84) in $\Theta(m + n \log n)$, ...
- Maxflow:
  - Ford-Fulkerson in $\Theta(m|f^*|)$.
  - Edmonds-Karp in $\Theta(nm^2)$.
  - Dinic's in $\Theta(n^2 m)$. (You'll see this in CS 6820.)
  - State of the art is $O(nm)$ (Orlin 2013(!)+King-Rao-Tarjan '94)

When to give up?

Is there some point at which we can be confident that we have found the fastest algorithm there is for a problem, and there is no point in looking for a better one?

When to give up? (2)

If yes, it seems reasonable to say that we'd have bumped into something like the intrinsic hardness of the problem.

When to give up? (2)

If yes, it seems reasonable to say that we'd have bumped into something like the intrinsic hardness of the problem.

"You can't solve maxflow faster than in $\Theta(nm)$ time because $\Theta(nm)$ is just a measure of how hard maxflow actually is."

When to give up? (2)

If yes, it seems reasonable to say that we'd have bumped into something like the intrinsic hardness of the problem.

"You can't solve maxflow faster than in $\Theta(nm)$ time because $\Theta(nm)$ is just a measure of how hard maxflow actually is."

"No matter *how* you go about solving your maxflow instance, at some point you have to put in $\Theta(nm)$ work."

An almost physical statement: compare "if you want to get a satellite into orbit..."

Tight lower bounds are rare

Very few instances where we know that an algorithm we know for a problem is optimal.

Tight lower bounds are rare

Very few instances where we know that an algorithm we know for a problem is optimal.

(Sorting is one of those, for a cool reason: Stirling's formula says $\log(n!) = \Theta(n \log n)$.)

Tight lower bounds are rare

Very few instances where we know that an algorithm we know for a problem is optimal.

(Sorting is one of those, for a cool reason: Stirling's formula says $\log(n!) = \Theta(n \log n)$.)

How would we go about learning anything about the hardness of a general problem?

Simple fact about lower bounds

Well, for starters: if we do have an algorithm solving a problem in $O(f)$ time, we know that the problem can't be any harder than that.

Simple fact about lower bounds

Well, for starters: if we do have an algorithm solving a problem in $O(f)$ time, we know that the problem can't be any harder than that.

Edmonds-Karp solves maxflow in $\Theta(nm^2)$. Therefore, a statement like "maxflow takes at least $\Theta(n^2m^2)$ time to solve" is plainly wrong.

Algorithms calling algorithms

Often, an algorithm we write to solve one problem will, in the course of it execution, create an instance of another problem, call out to an algorithm (a subroutine) to solve that subproblem, and do something with the result.

Algorithms calling algorithms

Often, an algorithm we write to solve one problem will, in the course of it execution, create an instance of another problem, call out to an algorithm (a subroutine) to solve that subproblem, and do something with the result.

- Sort some array before doing something with it.
- For some scheduling problem, construct a flow graph and call a maxflow algorithm. Compute a schedule from the maximum flow and return it.

Algorithms calling algorithms: Example

Imagine we have to solve the problem of scheduling widgets, and the following algorithm is provably correct:

```
// Input: n widgets to schedule

sort(widgets); // O(n log n)

for(int i=0;i<n;++i) {
    G = createFlowNetwork(widgets, i); // O(n^2)
    // G has n vertices and n^2 edges
    int f = maxflow(G); // O(?)

    maxima[i]=f; // O(1)
}

schedule = scheduleFromMaxima(maxima); // O(n)
// Output: a schedule for the n widgets
```

Analysing the example (1)

What's the time complexity of this algorithm? And, what does it say about the hardness of Widget Scheduling?

Analysing the example (1)

What's the time complexity of this algorithm? And, what does it say about the hardness of Widget Scheduling?

Well, we do $\Theta(n \log n + n)$ work outside the loops, $\Theta(n^2)$ work to create a flow network in each of the n iterations of the loop, and we also call an algorithm for maxflow (on n vertices and n^2 edges) n times.

Analysing the example (2)

So the complexity depends on the complexity of our maxflow algorithm: in general, it's $\Theta(n^3 + n \cdot f(n, n^2))$, where $f(n, m)$ is the complexity of the maxflow algorithm.

Analysing the example (2)

So the complexity depends on the complexity of our maxflow algorithm: in general, it's $\Theta(n^3 + n \cdot f(n, n^2))$, where $f(n, m)$ is the complexity of the maxflow algorithm.

- Plug in Edmonds-Karp (where $f(n, m) = \Theta(nm^2)$), and we get $\Theta(n^6)$.

Analysing the example (2)

So the complexity depends on the complexity of our maxflow algorithm: in general, it's $\Theta(n^3 + n \cdot f(n, n^2))$, where $f(n, m)$ is the complexity of the maxflow algorithm.

- Plug in Edmonds-Karp (where $f(n, m) = \Theta(nm^2)$), and we get $\Theta(n^6)$.
- Plug in Dinic's ($f(n, m) = \Theta(n^2 m)$), and we get $\Theta(n^5)$.

Analysing the example (2)

So the complexity depends on the complexity of our maxflow algorithm: in general, it's $\Theta(n^3 + n \cdot f(n, n^2))$, where $f(n, m)$ is the complexity of the maxflow algorithm.

- Plug in Edmonds-Karp (where $f(n, m) = \Theta(nm^2)$), and we get $\Theta(n^6)$.
- Plug in Dinic's ($f(n, m) = \Theta(n^2 m)$), and we get $\Theta(n^5)$.
- The state-of-the-art algorithm gives $\Theta(n^4)$.

Intermission

Let's take a short break. Some questions:

- If someone discovers an $O(n \log m)$ algorithm for maxflow tomorrow, what will be the complexity of our Widget Scheduling algorithm?
- What if the graph G created by `createFlowNetwork` has n^2 vertices instead?

Example: Implications for Hardness (1)

What can we conclude from this algorithm about the hardness of Widget Scheduling?

Example: Implications for Hardness (1)

What can we conclude from this algorithm about the hardness of Widget Scheduling?

Whatever algorithm we plug in to solve the subproblem (maxflow), the correctness of our algorithm for Widget Scheduling implies that it will run in time $\Theta(n^3 + n \cdot f(n, n^2))$, where $f(n, m)$ is the runtime of the maxflow algorithm.

Example: Implications for Hardness (2)

The hardness of maxflow, as we defined it, is the runtime of the fastest algorithm that solves it.

As we observed before, the hardness of Widget Scheduling is bounded above by the runtime of any algorithm that solves it,

Example: Implications for Hardness (2)

The hardness of maxflow, as we defined it, is the runtime of the fastest algorithm that solves it.

As we observed before, the hardness of Widget Scheduling is bounded above by the runtime of any algorithm that solves it, including the one where we plugged in the fastest possible maxflow algorithm.

Therefore, we have established a relationship between the hardness of Widget Scheduling and Maxflow:

$$H(\textsc{Widget Scheduling}) \lessapprox n^3 + n \cdot H(\textsc{Maxflow})).$$

(H is not standard notation. $f \lessapprox g$ means $\Theta(f) \leq \Theta(g)$.)

Reductions

When we solve a problem by invoking an algorithm for another problem like this, we call this a reduction.

In this case, we have reduced Widget Scheduling to Maxflow.

Reductions

When we solve a problem by invoking an algorithm for another problem like this, we call this a reduction.

In this case, we have reduced Widget Scheduling to Maxflow.

Why the term? Intuition: We had a "big" problem (write an algorithm to schedule widgets). We filled in part of the solution (the beginning, loop and end). In the left, we are left with a "smaller" gap that we still have to fill in, namely the part where we solve maxflow.

We had a big problem, and wound up with a smaller problem: i.e. we have reduced the size of the problem we have to deal with.

Reductions: formal definition

Definition

A time-$h(n)$ reduction from P to ($\Theta(g(n))$ size-$\Theta(f(n))$ instances of) Q is an algorithm that solves a problem P on inputs of length n and calls an unspecified algorithm to solve a problem Q of size $\Theta(f(n))$ $\Theta(g(n))$ times, while also doing $h(n)$ work outside of the Q calls.

If all of f, g and h are polynomials in n, then we call P a polynomial-time reduction from P to Q.

Flipping the Inequality

In the inequality we saw before, we bounded the hardness of the "superproblem" (the one that we reduced from) from above:

$$H(P)(n) \lessapprox h(n) + g(n) \cdot H(Q)(f(n)).$$

Shorter notation, allowing more parameters:

$$H(P) \lessapprox h + g \cdot H(Q) \circ f.$$

Flipping the Inequality

In the inequality we saw before, we bounded the hardness of the "superproblem" (the one that we reduced from) from above:

$$H(P)(n) \lessapprox h(n) + g(n) \cdot H(Q)(f(n)).$$

Shorter notation, allowing more parameters:

$$H(P) \lessapprox h + g \cdot H(Q) \circ f.$$

But we're interested in showing that problems are hard, not easy.

Flipping the Inequality

In the inequality we saw before, we bounded the hardness of the "superproblem" (the one that we reduced from) from above:

$$H(P)(n) \lessapprox h(n) + g(n) \cdot H(Q)(f(n)).$$

Shorter notation, allowing more parameters:

$$H(P) \lessapprox h + g \cdot H(Q) \circ f.$$

But we're interested in showing that problems are hard, not easy. Let's flip the inequality!

$$H(Q)(f(n)) \gtrapprox (H(P)(n) - h(n))/g(n).$$

Hardness lower bounds

$$H(Q)(f(n)) \gtrapprox (H(P)(n) - h(n))/g(n).$$

So, if we already knew that the original problem P is hard, then the existence of a reasonably fast reduction (h, g, f small) would show that Q is also hard.

Hardness lower bounds

$$H(Q)(f(n)) \gtrapprox (H(P)(n) - h(n))/g(n).$$

So, if we already knew that the original problem P is hard, then the existence of a reasonably fast reduction (h, g, f small) would show that Q is also hard.

Concrete example:

$$H(\textsc{Maxflow})(n, n^2) \gtrapprox \frac{1}{n}(H(\textsc{Widget Scheduling})(n) - n^3).$$

This should be surprising.

Isn't this curious? By using Q as a subroutine to solve another problem P, we showed that some of the hardness of P can rub off on Q.

Hardness is work you can't avoid. If we started with a hard problem and only did a little work, the remaining problem that we reduced to must still be hard.

Pulling hardness up by the bootstraps (2)

But there's a chicken-and-egg problem here. How do we know that P is hard to begin with?

Pulling hardness up by the bootstraps (2)

But there's a chicken-and-egg problem here. How do we know that P is hard to begin with?

Maybe we tried looking really hard for a fast solution, but couldn't find one.

Pulling hardness up by the bootstraps (2)

But there's a chicken-and-egg problem here. How do we know that P is hard to begin with?

Maybe we tried looking really hard for a fast solution, but couldn't find one.

But for a single problem, this is hardly persuasive. We spent decades looking for a reasonably fast (quasipolynomial) algorithm for Graph Isomorphism, and only found a candidate in 2017.

Pulling hardness up by the bootstraps (2)

What if we could do this with lots of problems, none of which we can solve efficiently?

Pulling hardness up by the bootstraps (2)

What if we could do this with lots of problems, none of which we can solve efficiently?

Imagine if we had something like a universal subroutine, which we can call to make significant progress towards any problem. (So much progress that only little work is left to do.)

Too ambitious?

Pulling hardness up by the bootstraps (3)

Okay, fine. What about a universal subroutine with which we can make significant progress towards any problem of some class? Say, any program a computer can solve in polynomial time?

Pulling hardness up by the bootstraps (3)

Okay, fine. What about a universal subroutine with which we can make significant progress towards any problem of some class? Say, any program a computer can solve in polynomial time?

At that point, we're putting the science in Computer Science: one data point (we can't solve this problem fast) is anecdote, but many have some sort of persuasive power. "We've never once seen a cannonball fall up, over many experiments."

Pulling hardness up by the bootstraps (4)

Here's an interesting class of problems: NP, the class of problems whose solutions a computer can verify to be correct in polynomial time. (Intuitively, imagine something like Sudoku puzzles. We might not how to solve them, but the rules are simple enough to check.)

Pulling hardness up by the bootstraps (4)

Here's an interesting class of problems: NP, the class of problems whose solutions a computer can verify to be correct in polynomial time. (Intuitively, imagine something like Sudoku puzzles. We might not how to solve them, but the rules are simple enough to check.)

(Equivalently, the class of problems whose solutions a nondeterministic computer can solve in polynomial time. We will see later what this means.)

Pulling hardness up by the bootstraps (4)

Here's an interesting class of problems: NP, the class of problems whose solutions a computer can verify to be correct in polynomial time. (Intuitively, imagine something like Sudoku puzzles. We might not how to solve them, but the rules are simple enough to check.)

(Equivalently, the class of problems whose solutions a nondeterministic computer can solve in polynomial time. We will see later what this means.)

Can we find a universal subroutine for these? Still too ambitious?

Pulling hardness up by the bootstraps (5)

Let's pretend it isn't. How would we go about this?

Pulling hardness up by the bootstraps (5)

Let's pretend it isn't. How would we go about this?

Intuitively, for a subroutine to make real progress towards solving a problem, it must do some work on an aspect of the problem. For a subroutine to make progress towards multiple problems, it must do some work that is common to all these problems.

What's common to all the problems in NP...?

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

- have solutions verifiable on a computer

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

- have solutions verifiable on a computer
- in polynomial time.

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

- have solutions verifiable on a computer
- in polynomial time.

We would do well to come up with a formal definition of these. For starters, what does it mean to do something on a computer?

Pulling hardness up by the bootstraps (6)

Well, actually, we said two things when defining the class: they are problems that

- have solutions verifiable on a computer
- in polynomial time.

We would do well to come up with a formal definition of these. For starters, what does it mean to do something on a computer?

Hence, next time: an excursion into Computability Theory.