Lecture 24: Conclusion

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

August 3, 2021

Overview

- Course Overview and Recap
    - What have we learned?
    - High-Level Principles
    - Interconnectedness
- Where do we go from here?
    - Next steps
    - “Real” Life

What was this course about?

In your previous algorithms/optimization/data structures course, you learned some of the following:

- combinatorial techniques (divide-and-conquer, greedy algorithms, dynamic programming, local search, etc.)
- data structures (heaps, balanced trees, etc.)
- optimization (linear programming, SDP, integer programming)

What was this course about?

In your previous algorithms/optimization/data structures course, you learned some of the following:

- combinatorial techniques (divide-and-conquer, greedy algorithms, dynamic programming, local search, etc.)
- data structures (heaps, balanced trees, etc.)

The techniques above emphasized **two computational models** (sequential & deterministic computation, query model).

What was this course about?

In this course we used the algorithmic lens to:

- explore several models of computation:
    1. deterministic sequential
    2. randomized sequential
    3. randomized parallel
    4. sublinear-time
    5. memory constrained (streaming)
    6. distributed
    7. online (competitive analysis)
    8. algebraic
    9. interactive

What was this course about?

In this course we used the algorithmic lens to:

- explore several models of computation:
  1. deterministic sequential
  2. randomized sequential
  3. randomized parallel
  4. sublinear-time
  5. memory constrained (streaming)
  6. distributed
  7. online (competitive analysis)
  8. algebraic
  9. interactive
- expand your algorithmic toolkit
  1. amortized analysis
  2. use of randomness
  3. concentration inequalities
  4. dealing with NP-complete problems (approximation algorithms)
  5. exploring the limits of approximation algorithms

allow us to convert "expectation" into "typical"

if my algorithm does well on average with concentration we show that it behaves like average most of the time

High-level principles in algorithms

- When encountering a problem, follow template:
    1. what *model of computation* is most suitable?
       That is: what are the most important *resources* and *constraints* that we need to respect?

High-level principles in algorithms

- When encountering a problem, follow template:
  1. what *model of computation* is most suitable?
     That is: what are the most important *resources* and *constraints* that we need to respect?
  2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
     That is: are there any impossibility results which are known in the computational model?

High-level principles in algorithms

- When encountering a problem, follow template:
  1. what *model of computation* is most suitable?
     That is: what are the most important *resources* and *constraints* that we need to respect?
  2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
     That is: are there any impossibility results which are known in the computational model?
  3. what is the simplest/most natural algorithm to solve it, and what is its complexity?

③' are there any simpler versions of the problem (special cases) that we can solve?
If so, can we generalize the simple algorithm?

High-level principles in algorithms

- When encountering a problem, follow template:
    1. what *model of computation* is most suitable?
       That is: what are the most important *resources* and *constraints* that we need to respect?
    2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
       That is: are there any impossibility results which are known in the computational model?
    3. what is the simplest/most natural algorithm to solve it, and what is its complexity?
    4. Can we do better?

High-level principles in algorithms

- When encountering a problem, follow template:
    1. what *model of computation* is most suitable?
       That is: what are the most important *resources* and *constraints* that we need to respect?
    2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
       That is: are there any impossibility results which are known in the computational model?
    3. what is the simplest/most natural algorithm to solve it, and what is its complexity?
    4. Can we do better?
    5. Is the problem hard? If so, can we hope to *relax the guarantees*?

High-level principles in algorithms

- When encountering a problem, follow template:
    1. what *model of computation* is most suitable?
       That is: what are the most important *resources* and *constraints* that we need to respect?
    2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
       That is: are there any impossibility results which are known in the computational model?
    3. what is the simplest/most natural algorithm to solve it, and what is its complexity?
    4. Can we do better?
    5. Is the problem hard? If so, can we hope to *relax the guarantees*?
    6. Can we do better?

High-level principles in algorithms

- When encountering a problem, follow template:
  1. what *model of computation* is most suitable?
     That is: what are the most important *resources* and *constraints* that we need to respect?
  2. What *guarantees* are *possible* in the model of computation? Can we hope to get an optimal answer to the problem?
     That is: are there any impossibility results which are known in the computational model?
  3. what is the simplest/most natural algorithm to solve it, and what is its complexity?
  4. Can we do better?
  5. Is the problem hard? If so, can we hope to *relax the guarantees*?
  6. Can we do better?
  7. Can we show that our algorithm is the best?
     That is, we could try to reduce it to another problem which is known to be optimum (perhaps under certain complexity assumptions)

- Approximate diameter problem (sublinear time) no complexity assumptions
- Hardness of approximation (P vs NP, UGC)

Example: Amortized Analysis

- In data structures, oftentimes one cares about worst-case per query
    - internet's client-server model

Example: Amortized Analysis

- In data structures, oftentimes one cares about worst-case per query
  - internet's client-server model
- sometimes, we don't care about worst-case per query, but worst-case overall
  - use of data structures in sequential algorithms
  - minimum spanning tree

(UNION-FIND)

don't care about worst-case per query

Example: Amortized Analysis

discussion about model and guarantees

- In data structures, oftentimes one cares about worst-case per query
  - internet's client-server model
- sometimes, we don't care about worst-case per query, but worst-case overall
  - use of data structures in sequential algorithms
  - minimum spanning tree
- Learned how to use amortized analysis to provide better overall guarantees

vanilla amortization	count all costs
charging scheme	assign charges to operations
potential function	assign charges and potential to data structure

expansion of our algorithmic toolkit.

Similar input setting - different models

Warning: computational model not only dictated by how input is given

- When input comes to you "online" (as a stream of events)

  YOLO: you only look once

  we may have different computational models/goals

Similar input setting - different models

- When input comes to you "online" (as a stream of events)

  YOLO: you only look once

  we may have different computational models/goals
- We learned:
  - data streaming: *memory* is our main constraint. Content with *approximation* of best answer
  - Examples: median, heavy hitters, distinct elements

best answer may be impossible to obtain (worst-case)

Similar input setting - different models

- When input comes to you "online" (as a stream of events)

  YOLO: you only look once

  we may have different computational models/goals
- We learned:
  - data streaming: *memory* is our main constraint. Content with *approximation* of best answer

    model #1
  - Examples: median, heavy hitters, distinct elements
  - online algorithms: want *fast updates*, need to *decide on the spot*. Want to do as best as we can compared to *best in hindsight* (algorithms that can see entire input beforehand)

    model #2

    goal different from streaming
  - *competitive analysis*
  - Examples: multiplicative weights update, paging, k-server

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)
- Quite useful in many settings:
    - when we don't know what to do
    - when problem has some adversarial input format

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)
- Quite useful in many settings:
  - when we don't know what to do
  - when problem has some adversarial input format
  - when we have to make decisions before seeing the whole input (streaming)

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)
- Quite useful in many settings:
  - when we don't know what to do
  - when problem has some adversarial input format
  - when we have to make decisions before seeing the whole input (streaming)
  - when problem can be encoded in algebraic format (polynomial identity testing)
  - when we need to construct objects which are abundant, but "hard" to construct (hash functions, graph sparsification, dimension reduction, expanders)

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)
- Quite useful in many settings:
    - when we don't know what to do
    - when problem has some adversarial input format
    - when we have to make decisions before seeing the whole input (streaming)
    - when problem can be encoded in algebraic format (polynomial identity testing)
    - when we need to construct objects which are abundant, but "hard" to construct (hash functions, graph sparsification, dimension reduction, expanders)
    - when we need to estimate number of objects which are hard to count (random walks)

Randomness gives us power

- We learned how to use randomness in algorithms
- To use randomness, tradeoff between an algorithm which works *all the time* (deterministic) to one that works *most of the time*
- Usually faster (or the only option)
- Quite useful in many settings:
  - when we don't know what to do
  - when problem has some adversarial input format
  - when we have to make decisions before seeing the whole input (streaming)
  - when problem can be encoded in algebraic format (polynomial identity testing)
  - when we need to construct objects which are abundant, but "hard" to construct (hash functions, graph sparsification, dimension reduction, expanders)
  - when we need to estimate number of objects which are hard to count (random walks)
  - approximation algorithms (randomized rounding)

Dealing with NP-hard problems

- when faced with an NP-hard (optimization) problems, still want to solve them (as best as we can)
- to that task, important to relax the guarantees (or make more assumptions)
- instead of trying to get best solution, get a solution which is *guaranteed* to be close to best

Dealing with NP-hard problems

- when faced with an NP-hard (optimization) problems, still want to solve them (as best as we can)
- to that task, important to relax the guarantees
- instead of trying to get best solution, get a solution which is *guaranteed* to be close to best
- formulate problems as *integer programs* or *quadratic programs* and relax them to problems we can solve: *linear programs* or *semidefinite programs*

Dealing with NP-hard problems

- when faced with an NP-hard (optimization) problems, still want to solve them (as best as we can)
- to that task, important to relax the guarantees
- instead of trying to get best solution, get a solution which is *guaranteed* to be close to best
- formulate problems as *integer programs* or *quadratic programs* and relax them to problems we can solve: *linear programs* or *semidefinite programs*
- when we have enough structure, deterministic methods are good

solutions are nice (half-integral, low denominator)

Dealing with NP-hard problems

- when faced with an NP-hard (optimization) problems, still want to solve them (as best as we can)
- to that task, important to relax the guarantees
- instead of trying to get best solution, get a solution which is *guaranteed* to be close to best
- formulate problems as *integer programs* or *quadratic programs* and relax them to problems we can solve: *linear programs* or *semidefinite programs*
- when we have enough structure, deterministic methods are good
- when integer programs have nice vertex solutions, easy to obtain deterministic rounding

their LP relaxations

Dealing with NP-hard problems

- when faced with an NP-hard (optimization) problems, still want to solve them (as best as we can)
- to that task, important to relax the guarantees
- instead of trying to get best solution, get a solution which is *guaranteed* to be close to best
- formulate problems as *integer programs* or *quadratic programs* and relax them to problems we can solve: *linear programs* or *semidefinite programs*
- when we have enough structure, deterministic methods are good
- when integer programs have nice vertex solutions, easy to obtain deterministic rounding
- when the above does not happen, randomness to the rescue

Interaction gives us power and limitations

- We learned about complexity of proofs, and how we can use interaction to give different characterization of complexity classes

Interaction gives us power and limitations

- We learned about complexity of proofs, and how we can use interaction to give different characterization of complexity classes
- PCP theorem: NP can be characterized as problems that have proofs which can be verified by making *3* random queries to the proof!

Interaction gives us power and limitations

- We learned about complexity of proofs, and how we can use interaction to give different characterization of complexity classes
- PCP theorem: *NP* can be characterized as problems that have proofs which can be verified by making *3* random queries to the proof!
- Also saw how we can use interactive proofs (PCPs) to construct reductions which preserve a gap between YES and NO instances

used for hardness of approximation

Interaction gives us power and limitations

- We learned about complexity of proofs, and how we can use interaction to give different characterization of complexity classes
- PCP theorem: *NP* can be characterized as problems that have proofs which can be verified by making *3* random queries to the proof!
- Also saw how we can use interactive proofs (PCPs) to construct reductions which preserve a gap between YES and NO instances
- Interaction not only good for hardness of approximation - saw how to use interaction to give *zero knowledge proofs*

  How to convince someone that you know something without revealing any knowledge on how you do it.

Algebraic/Arithmetic Algorithms

- Also learned about algebraic/arithmetic models of computation
    - when your problem is algebraic in nature, this is the most natural model

Algebraic/Arithmetic Algorithms

- Also learned about algebraic/arithmetic models of computation
  - when your problem is algebraic in nature, this is the most natural model
  - widely used for some of the most used algorithms in real life: matrix multiplication, discrete Fourier transform

Algebraic/Arithmetic Algorithms

- Also learned about algebraic/arithmetic models of computation
  - when your problem is algebraic in nature, this is the most natural model
  - widely used for some of the most used algorithms in real life: matrix multiplication, discrete Fourier transform
- Used widely for design of parallel algorithms, since “linear algebra can be done in parallel”

Algebraic/Arithmetic Algorithms

- Also learned about algebraic/arithmetic models of computation
    - when your problem is algebraic in nature, this is the most natural model
    - widely used for some of the most used algorithms in real life: matrix multiplication, discrete Fourier transform
- Used widely for design of parallel algorithms, since “linear algebra can be done in parallel”
- More generally: “any polynomial you can compute is a determinant”

Distributed Computation

- Algorithms which run on a network, or multiprocessors within a computer which share memory (or not)

they must work together to achieve a certain goal

Distributed Computation

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
    - Resource Management
    - Data Management and Transmission
    - Synchronization
    - Consensus
    - many more

Distributed Computation

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
  - Resource Management
  - Data Management and Transmission
  - Synchronization
  - Consensus
  - many more
- Challenges in this setting:
  - Concurrent Activity
  - Uncertainty of order of events
  - Failure and recovery of processors or channels

Distributed Computation

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
  - Resource Management
  - Data Management and Transmission
  - Synchronization
  - Consensus
  - many more
- Challenges in this setting:
  - Concurrent Activity
  - Uncertainty of order of events
  - Failure and recovery of processors or channels
- Many models
  - *Memory & Communication*: shared memory, message-passing
  - *Timing*: synchronous (rounds), asynchronous, partially synchronous (bounds on message delay, processor speeds, clock rates)
  - *Failures*: processor (stop, Byzantine), communication (message loss/altered), system state corruption

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not
- we saw how randomness is applied in many different settings, and common techniques such as hashing, fingerprinting, polynomial identity testing

derandomization , pseudo-randomness

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not
- we saw how randomness is applied in many different settings, and common techniques such as hashing, fingerprinting, polynomial identity testing
- interactive proofs, from proof complexity, used to prove that certain algorithms cannot be approximated up to a certain point

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not
- we saw how randomness is applied in many different settings, and common techniques such as hashing, fingerprinting, polynomial identity testing
- interactive proofs, from proof complexity, used to prove that certain algorithms cannot be approximated up to a certain point
- hashing and fingerprinting highly used in interactive proofs

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not
- we saw how randomness is applied in many different settings, and common techniques such as hashing, fingerprinting, polynomial identity testing
- interactive proofs, from proof complexity, used to prove that certain algorithms cannot be approximated up to a certain point
- hashing and fingerprinting highly used in interactive proofs
- Algorithms used to prove lower bounds (recent trend - highly recommended!)

Interconnectedness in Algorithms and Complexity

- We saw many different settings for algorithms, important problems and techniques
- Is this just a bunch of tricks though?
- Certainly not
- we saw how randomness is applied in many different settings, and common techniques such as hashing, fingerprinting, polynomial identity testing
- interactive proofs, from proof complexity, used to prove that certain algorithms cannot be approximated up to a certain point
- hashing and fingerprinting highly used in interactive proofs
- Algorithms used to prove lower bounds (recent trend - highly recommended!)
- Algorithms in forms of reductions, used to prove that even easy problems cannot be improved! (fine-grained complexity)

- Course Overview and Recap
  - What have we learned?
  - High-Level Principles
  - Interconnectedness

- Where do we go from here?
  - Next steps
  - “Real” Life

How can I learn more?

Consider taking more advanced courses next term!
See graduate course openings at:

- Current graduate course offerings for next term!

  `https://cs.uwaterloo.ca/current-graduate-students/courses/current-course-offerings/fall-2021-course-offerings`

- Or, try out some of the research opportunities at UW!

Research

Consider doing a URA, URF or USRA with a U Waterloo faculty!
See research openings at:

- Undergraduate Research Assistantship (URA):

  `https://cs.uwaterloo.ca/computer-science/current-undergraduate-students/research-opportunities/undergraduate-research-assistantship-ura-program`

- Undergraduate Research Fellowship (URF):

  `https://grec.cs.uwaterloo.ca/`

- Undergraduate Research Internship (URI):

  `https://cs.uwaterloo.ca/current-undergraduate-students/research-opportunities/undergraduate-research-internship-uri-program`

- For Canadians, please check out NSERC's USRA:

  `https://cs.uwaterloo.ca/usra`

But is this theory stuff useful?

- Certainly so - and lately the gap between theory and practice has been quite short
- intense use of theoretical cryptography and distributed computing in cryptocurrencies
- cryptography highly used in e-commerce
- several algorithms used in computational biology
- Markov chains used in page rank, simulations of physical systems
- many more applications

Questions

Questions?