Review lecture
COMP4500/7500
Advanced Algorithms & Data Structures

October 26, 2020

Admin

- Tutorials this week will be question and answer sessions, where you can come and get help in your preparation for the final exam. (There are no set questions scheduled.)

Motivation for the subject

Expand your abilities to

- analyse,
- critique,
- design, and
- implement

advanced data structures and algorithms.

Motivation

Towards this:

- Understanding how efficiency is measured and compared

Motivation

Towards this:

- Understanding how efficiency is measured and compared
- Representing problems as graphs

Motivation

Towards this:

- Understanding how efficiency is measured and compared
- Representing problems as graphs
- Dynamic programming

Motivation

Towards this:

- Understanding how efficiency is measured and compared
- Representing problems as graphs
- Dynamic programming
- Greedy strategies

Motivation

Towards this:

- Understanding how efficiency is measured and compared
- Representing problems as graphs
- Dynamic programming
- Greedy strategies
- Probabilistic analysis and randomised algorithms

Motivation

Towards this:

- Understanding how efficiency is measured and compared
- Representing problems as graphs
- Dynamic programming
- Greedy strategies
- Probabilistic analysis and randomised algorithms

Also:

- understanding the limitation of our knowledge (e.g. P vs NP problem)
- being able to recognise problems for which there is no known tractable solution

Course recap

Lectures (approx):

1 Mathematical background and asymptotic notation

2 Divide and conquer algorithms and solving recurrences

3-5 Graphs

6-8 Dynamic programming and Greedy algorithms

9 Amortised analysis

10 Computational complexity

11 Probabilistic Analysis and Randomized Algorithms (e.g. quicksort)

Related “chunks”

Chunks:

- Measuring efficiency:
  - asymptotic notation,
  - recurrences and solving them (e.g. Master Method)
  - amortised analysis
  - average case analysis and expected time complexity of randomised algorithms
- Graphs
- Dynamic/greedy programming
- Computational complexity

The exam itself

- 2 hours + 30min prepare and upload exam solution document
- Answer all questions.
- 100 marks
- Worth 50% (comp4500,COMP7500) or 60% If midterm result is lower than final exam result
- Open book exam

Exam late submission

- From 1-15 minutes = 20%
- From 15-30 minutes = 50%
- More than 30 minutes = 100%

Exam late submission

- From 1-15 minutes = 20%
- From 15-30 minutes = 50%
- More than 30 minutes = 100%

Penalties are calculated on the total mark. For example:

- Late submission by 10 minutes.
- Total mark available: 50.
- Marks received: 40.
- Penalty applied: $50 \times 20\% = 10$.
- Final mark with penalty applied: 30/50.

Exam information

- Exam Information available on Blackboard examination
- The examination duration will be 2 hours and 30 minutes.
- Download exam paper from test section COMP4500/COMP7500 Semester Two Final Examination 2020 – QUESTION PAPER
- Once completed upload results file to assignment section COMP4500/COMP7500 Semester Two Final Examination 2020 – SUBMISSION
- You must commence your exam at the time listed in your personalised timetable.
- If you experience any technical difficulties during the examination, contact the Library AskUs service for advice as soon as practicable

Academic integrity

- You cannot cut-and-paste material other than your own work as answers.
- You are not permitted to consult any other person – whether directly, online, or through any other means – about any aspect of this examination during the period that it is available.
- If it is found that you have given or sought outside assistance with this examination, then that will be deemed to be cheating.

Exam technique

- Start with easiest questions
- Don’t wast your time googling questions as you won’t find answers online
- Read questions carefully and be comprehensive with your answers
- Even if you don’t know the answer: TRY
- Nothing you write is worthless
- Exam is mainly designed to test your ability of solving problems... DO NOT SHARE ANSWERS

Recap: Execution Time Analysis

Definition (Worst case)

$T(n) =$ Maximum execution time over all inputs of size n

Definition (Average case)

$T(n) =$ Average execution time over all inputs of size n, weighted by the probability of the input

Definition (Best case)

$T(n) =$ Mimimum execution time over all inputs of size n

Recap: Asymptotic notation

- Ignores implementation dependent constants:
  - machine speed
  - compiler
- Difference in order outweigh constant factors:
  - Merge sort $\Theta(n \lg n)$
  - Insertion sort $\Theta(n^2)$

  Merge sort is ultimately better for large enough n no matter what the constant factors

Recap: Asymptotic notation

For functions f and g

$f \in O(g)$ f is asymptotically bounded above by g to within a constant factor

- $n \in O(n^2)$
- $64,000n \in O(n)$

Recap: Asymptotic notation

For functions f and g

$f \in O(g)$ f is asymptotically bounded above by g to within a constant factor

- $n \in O(n^2)$
- $64,000n \in O(n)$

$f \in \Omega(g)$ f is asymptotically bounded below by g to within a constant factor

- $g \in O(f)$
- $n^2 \in \Omega(n)$

Recap: Asymptotic notation

For functions f and g

$f \in O(g)$ f is asymptotically bounded above by g to within a constant factor

- $n \in O(n^2)$
- $64,000n \in O(n)$

$f \in \Omega(g)$ f is asymptotically bounded below by g to within a constant factor

- $g \in O(f)$
- $n^2 \in \Omega(n)$

$f \in \Theta(g)$ f is asymptotically bounded above and below by g to within a constant factor

- $f \in O(g) \wedge f \in \Omega(g)$
- $42n \in \Theta(n)$
- $n \notin \Theta(n^2)$

Recap: Solving recurrences.

- Running time can often be described by a recurrence.
- Iteration and substitution methods apply widely but can be unwieldy.
- The Master Method is less general, but covers many common cases. It is also quite direct.

Recap: Master method. Slides week 2

- $T(n) = aT(n/b) + f(n)$
- n/b can be $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$
- In each case we compare $n^{\log_b a}$ with $f(n)$

  Case 1 $n^{\log_b a}$ "larger than" $f(n)$
  solution $\Theta(n^{\log_b a})$

  Case 2 $n^{\log_b a}$ "similar to" $f(n)$
  so solution $n^{\log_b a} \lg n$ (i.e., multiply by $\lg n$ factor)

  Case 3 $f(n)$ "larger than" $n^{\log_b a}$
  so solution $\Theta(f(n))$

Recap: Master method. Slides week 2

Case 1 $n^{\log_b a}$ must be "polynomial larger than" $f(n)$
i.e., $n^{\log_b a} \in \Omega(n^{\epsilon} f(n))$, $\epsilon > 0$

Case 3 Two requirements:

- $f(n)$ must be "polynomial larger than" $n^{\log_b a}$
- $f(n)$ must be "regular":
  i.e., $af(n/b) < cf(n)$, $c < 1$
  (regularity is usual)

Recap: Amortised analysis.

- Amortised analysis typically applies to data structures, and their set of operations, rather than a specific algorithm.

Recap: Amortised analysis.

- Amortised analysis typically applies to data structures, and their set of operations, rather than a specific algorithm.
  - we are interested in the worst-case cost of a **sequence of operations** as opposed to one operation in isolation

Recap: Amortised analysis.

- Amortised analysis typically applies to data structures, and their set of operations, rather than a specific algorithm.
  - we are interested in the worst-case cost of a **sequence of operations** as opposed to one operation in isolation
- Finds the worst-case cost of a sequence of operations on the data structure, starting from an initial value.

Recap: Amortised analysis.

- Amortised analysis typically applies to data structures, and their set of operations, rather than a specific algorithm.
  - we are interested in the worst-case cost of a **sequence of operations** as opposed to one operation in isolation
- Finds the worst-case cost of a sequence of operations on the data structure, starting from an initial value.
- The amortised cost is then the average cost

Recap: Amortised analysis.

- Amortised analysis typically applies to data structures, and their set of operations, rather than a specific algorithm.
  - we are interested in the worst-case cost of a **sequence of operations** as opposed to one operation in isolation
- Finds the worst-case cost of a sequence of operations on the data structure, starting from an initial value.
- The amortised cost is then the average cost

Three techniques:

1. Aggregate analysis
2. Accounting method
3. Potential method

Example: stack operations

Consider standard stack operations PUSH and POP

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$
- Hence, any sequence of n stack operations must be $O(n^2)$

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$
- Hence, any sequence of n stack operations must be $O(n^2)$
- But can we prove a better bound?

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$
- Hence, any sequence of n stack operations must be $O(n^2)$
- But can we prove a better bound?

Intuition:

- MULTIPOP will only iterate while the stack is not empty.

Example: stack operations

Consider standard stack operations PUSH and POP, with the addition of

```
MULTIPOP(S, k)
  while S is not empty and k > 0
      POP(S)
      k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$
- Hence, any sequence of n stack operations must be $O(n^2)$
- But can we prove a better bound?

Intuition:

- MULTIPOP will only iterate while the stack is not empty.
- Each element is pushed exactly once and popped exactly once, hence after m pushes there must be at most m pops

Stack operations: aggregate method

Analyse:

1 **for** $i = 1..n$
2 PUSH(..)
or
3 POP(..)
or
4 MULTIPOP(..)

Stack operations: aggregate method

Analyse:

```
1   for i = 1..n
2         PUSH(..)
          or
3         POP(..)
          or
4         MULTIPOP(..)
```

Arguing this is $O(n)$ is somewhat clumsy using the *aggregate* method, which we saw for dynamic tables.
Consider more sophisticated techniques:

- accounting method
- potential method

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$
2. Assign an *amortised cost*, $\hat{c}_i$, to each method.
   1. PUSH: 2
   2. POP: 0
   3. MULTIPOP(S, K): 0

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$
2. Assign an *amortised cost*, $\hat{c}_i$, to each method.
   1. PUSH: 2
   2. POP: 0
   3. MULTIPOP(S, K): 0

For *any* sequence of stack operations, the amortised cost must be an upper bound on the actual cost

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$
2. Assign an *amortised cost*, $\hat{c}_i$, to each method.
   1. PUSH: 2
   2. POP: 0
   3. MULTIPOP(S, K): 0

For *any* sequence of stack operations, the amortised cost must be an upper bound on the actual cost

Then, one can use the amortised cost in place of the (more complicated) actual cost.

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$
2. Assign an *amortised cost*, $\hat{c}_i$, to each method.
   1. PUSH: 2
   2. POP: 0
   3. MULTIPOP(S, K): 0

For *any* sequence of stack operations, the amortised cost must be an upper bound on the actual cost

Then, one can use the amortised cost in place of the (more complicated) actual cost.
In the above case every operation has constant amortised cost, hence a sequence of n operations is $O(n)$.

Stack operations: accounting method

We must show that the total amortised cost minus the total actual cost is never negative:

$$\left(\sum_{i=1}^{n} \hat{c}_i\right) \geq \left(\sum_{i=1}^{n} c_i\right)$$

(for all sequences of all possible lengths n)

Stack operations: accounting method

We must show that the total amortised cost minus the total actual cost is never negative:

$$\left(\sum_{i=1}^{n} \hat{c}_i\right) \geq \left(\sum_{i=1}^{n} c_i\right)$$

(for all sequences of all possible lengths n)

	actual cost	amortised cost
op	c_i	$\hat{c}_i$
PUSH	1	2
POP	1	0
MULTIPOP(S, k)	k'	0
	$= \text{MIN}(\#S, k)$	

Stack operations: accounting method

We must show that the total amortised cost minus the total actual cost is never negative:

$$\left(\sum_{i=1}^{n} \hat{c}_i\right) \geq \left(\sum_{i=1}^{n} c_i\right)$$

(for all sequences of all possible lengths n)

	actual cost	amortised cost
op	c_i	$\hat{c}_i$
PUSH	1	2
POP	1	0
MULTIPOP(S, k)	k'	0
	$= \text{MIN}(\#S, k)$	

Intuition: the extra credit in PUSH pays for the later (MULTI)POP.

Accounting method

Let MP(k) abbreviate MULTIPOP(S, k)

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH

Accounting method

Let MP(k) abbreviate MULTIPOP(S, k)

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1

Accounting method

Let MP(k) abbreviate MULTIPOP(S, k)

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7

Accounting method

Let MP(k) abbreviate MULTIPOP(S, k)

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7
amortised cost	2	2	0	2	0	2

Accounting method

Let MP(k) abbreviate MULTIPOP(S, k)

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7
amortised cost	2	2	0	2	0	2
running total	2	4	4	6	6	8

The running total of the amortised cost is always bigger than the running total of the actual cost.

Potential method

Instead of focusing on the cost of operations, focus on the data structure

1. As before, determine the *actual cost* of each operation
2. Define a *potential function*, Φ, on the data structure.
3. The *amortised cost* of an operation is the actual cost plus the *change in potential*

Potential method

Instead of focusing on the cost of operations, focus on the data structure

1. As before, determine the *actual cost* of each operation
2. Define a *potential function*, Φ, on the data structure.
3. The *amortised cost* of an operation is the actual cost plus the *change in potential*

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$$

Potential method

Instead of focusing on the cost of operations, focus on the data structure

1. As before, determine the *actual cost* of each operation
2. Define a *potential function*, Φ, on the data structure.
3. The *amortised cost* of an operation is the actual cost plus the *change in potential*

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$$

The obligation is to show that $\Phi(D_i) \geq \Phi(D_0)$ after every operation.

Potential method

Instead of focusing on the cost of operations, focus on the data structure

1. As before, determine the *actual cost* of each operation
2. Define a *potential function*, Φ, on the data structure.
3. The *amortised cost* of an operation is the actual cost plus the *change in potential*

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$$

The obligation is to show that $\Phi(D_i) \geq \Phi(D_0)$ after every operation.
This is normally trivial as $\Phi(D_0) = 0$ (the "potential" of the initial value of the data structure is 0)

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$
2. $\Phi(S) = \#S$ (size of the stack)

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$
2. $\Phi(S) = \#S$ (size of the stack)
3. Change in potential $(\Phi(D_i) - \Phi(D_{i-1}))$:
   1. PUSH: 1 (the size has increased by one)
   2. POP: -1 (the size has decreased by one)
   3. MULTIPOP: $-k'$ (the size has decreased by k')

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$
2. $\Phi(S) = \#S$ (size of the stack)
3. Change in potential $(\Phi(D_i) - \Phi(D_{i-1}))$:
   1. PUSH: 1 (the size has increased by one)
   2. POP: -1 (the size has decreased by one)
   3. MULTIPOP: $-k'$ (the size has decreased by k')
4. Amortised cost (actual cost + change in potential):
   1. PUSH: 2 (= 1 + 1)
   2. POP: 0 (= 1 + -1)
   3. MULTIPOP: 0 (= $k' + -k'$)

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$
2. $\Phi(S) = \#S$ (size of the stack)
3. Change in potential ($\Phi(D_i) - \Phi(D_{i-1})$):
   1. PUSH: 1 (the size has increased by one)
   2. POP: -1 (the size has decreased by one)
   3. MULTIPOP: $-k'$ (the size has decreased by k')
4. Amortised cost (actual cost + change in potential):
   1. PUSH: 2 (= 1 + 1)
   2. POP: 0 (= 1 + -1)
   3. MULTIPOP: 0 (= $k' + -k'$)

Obligation: the potential function is never negative (trivial)

Therefore, all operations have constant amortised time.

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7
$\Phi(S)$	1	2	1	2	0	1

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7
$\Phi(S)$	1	2	1	2	0	1
Change	+1	+1	-1	+1	-2	+1

Potential method

n	1	2	3	4	5	6
op	PUSH	PUSH	POP	PUSH	MP(2)	PUSH
actual cost	1	1	1	1	2	1
running total	1	2	3	4	6	7
$\Phi(S)$	1	2	1	2	0	1
Change	+1	+1	-1	+1	-2	+1
amortised cost	2	2	0	2	0	2

For this simple example, potential and accounting method give almost exactly the same intuition

Recap: Graphs.

- Graphs are a widely-used concept in theory and practice:
  - Used to represent concepts such as course dependencies
  - Used to represent places on a map

Recap: Graphs.

- Graphs are a widely-used concept in theory and practice:
  - Used to represent concepts such as course dependencies
  - Used to represent places on a map
- Often need to *traverse* a graph in some manner, to determine particular properties:
  - Shortest paths (connect source to destination)
  - Minimum spanning trees (connect all dots)

Recap: Graphs.

- Representation as: Adjacency list; Adjacency matrix
- Can be (un)directed, (a)cyclic, weighted

Recap: Graphs.

- Representation as: Adjacency list; Adjacency matrix
- Can be (un)directed, (a)cyclic, weighted

BFS(G, v)

```
 1  v.distance = 0
 2  v.colour = grey
 3  Q.initialise()
 4  Q.enqueue(v)
 5  while not Q.isEmpty()
 6        current = Q.next()
 7        for u in adjacent[current]
 8              if u.colour == white
 9                    u.distance = current.distance + 1
10                    u.colour = grey
11                    Q.enqueue(u)
12        current.colour := black
```

Recap: Graphs.

- Representation as: Adjacency list; Adjacency matrix
- Can be (un)directed, (a)cyclic, weighted

BFS(G, v)

```
 1  v.distance = 0
 2  v.colour = grey      v.parent = NULL
 3  Q.initialise()
 4  Q.enqueue(v)
 5  while not Q.isEmpty()
 6        current = Q.next()
 7        for u in adjacent[current]
 8              if u.colour == white
 9                    u.distance = current.distance + 1
10                    u.colour = grey      u.parent = current
11                    Q.enqueue(u)
12        current.colour := black
```

Recap: Dynamic programming.

- Polynomial-time algorithms for complex problems
- Store answers to subproblems rather than recalculate

Recap: Dynamic programming.

- Polynomial-time algorithms for complex problems
- Store answers to subproblems rather than recalculate
- Strategy:
  1. Devise an intuitive/concise definition of the problem as a recursive function (a *recurrence*)
  2. Systematically use this as the basis for filling in an array containing solutions to subproblems

Example: Longest common subsequence

Problem: find the longest (non-contiguous) sequence of characters shared between two strings

S_1 : A B C B C
S_2 : C A B B D

Example: Longest common subsequence

Problem: find the longest (non-contiguous) sequence of characters shared between two strings

S_1 :	A	B		B	
S_2 :		A	B	B	

Example: Longest common subsequence

Definition (Length of longest common subsequence (LCS))

$$
\begin{aligned}
LCS(\langle\rangle, S_2) &= LCS(S_1, \langle\rangle) = \langle\rangle \\
LCS(S_1.X, S_2.X) &= LCS(S_1, S_2).X \\
LCS(S_1.X, S_2.Y) &= \text{MAX}(LCS(S_1, S_2.Y), LCS(S_1.X, S_2)) \\
&\quad \text{provided } X \neq Y
\end{aligned}
$$

Example: Longest common subsequence

Definition (Length of longest common subsequence (LCS))

$$
\begin{aligned}
LCS(\langle\rangle, S_2) &= LCS(S_1, \langle\rangle) = \langle\rangle \\
LCS(S_1.X, S_2.X) &= LCS(S_1, S_2).X \\
LCS(S_1.X, S_2.Y) &= \text{MAX}(LCS(S_1, S_2.Y), LCS(S_1.X, S_2)) \\
&\quad \text{provided } X \neq Y
\end{aligned}
$$

Rewrite where S_1 and S_2 are (global, constant) arrays of length N and M. Index from 0. Solution is given by $LCS(0, 0)$.

Definition (Length of longest common subsequence (LCS))

$$
\begin{aligned}
LCS(N, j) &= LCS(i, M) = 0 \\
LCS(i, j) &= LCS(i+1, j+1) + 1 \\
&\quad \text{provided } S_1[i] = S_2[j], \text{ and } i < N \text{ and } j < M \\
LCS(i, j) &= \text{MAX}(LCS(i+1, j), LCS(i, j+1)) \\
&\quad \text{provided } S_1[i] \neq S_2[j], \text{ and } i < N \text{ and } j < M
\end{aligned}
$$

Example: Dynamic implementation of LCS

LCS-LENGTH-DYN(S_1, S_2, N, M)

Example: Dynamic implementation of LCS

LCS-LENGTH-DYN(S_1, S_2, N, M)

```
1  T = new int[N + 1, M + 1]
2  for j = 0 to M        T[N, j] = 0      // Base cases
3  for i = 0 to N        T[i, M] = 0
```

Example: Dynamic implementation of LCS

LCS-LENGTH-DYN(S_1, S_2, N, M)

```
 1  T = new int[N + 1, M + 1]
 2  for j = 0 to M        T[N, j] = 0      // Base cases
 3  for i = 0 to N        T[i, M] = 0

 4  for i = N − 1 to 0
 5        for j = M − 1 to 0
 6              if S1[i] == S2[j]
 7                    T[i, j] = T[i + 1, j + 1] + 1
 8              else
 9                    T[i, j] = MAX(T[i + 1, j], T[i, j + 1])
10  return T[0,0]
```

Recap: Computational complexity

We want to know whether a problem can or can't be solved in polynomial time.

But in fact there is no general way to determine that.

Recap: Computational complexity

- The class P is the set of concrete decision problems that are polynomial-time solvable.

Recap: Computational complexity

- The class P is the set of concrete decision problems that are polynomial-time solvable.
- The class NP is the set of concrete decision problems for which a solution (a certificate) can be checked (verified) in polynomial time.

Recap: Computational complexity

- The class P is the set of concrete decision problems that are polynomial-time solvable.
- The class NP is the set of concrete decision problems for which a solution (a certificate) can be checked (verified) in polynomial time.
- A concrete decision problem B is NP-hard when every problem $A \in NP$ is polynomial-time reducible to B.

Recap: Computational complexity

- The class P is the set of concrete decision problems that are polynomial-time solvable.
- The class NP is the set of concrete decision problems for which a solution (a certificate) can be checked (verified) in polynomial time.
- A concrete decision problem B is NP-hard when every problem $A \in NP$ is polynomial-time reducible to B.
  - If any one of these problems can be solved by a polynomial-time algorithm, then all problems in NP can be solved by a polynomial time algorithm.

Recap: Computational complexity

- The class P is the set of concrete decision problems that are polynomial-time solvable.
- The class NP is the set of concrete decision problems for which a solution (a certificate) can be checked (verified) in polynomial time.
- A concrete decision problem B is NP-hard when every problem $A \in NP$ is polynomial-time reducible to B.
  - If any one of these problems can be solved by a polynomial-time algorithm, then all problems in NP can be solved by a polynomial time algorithm.
- The class NPC (NP-complete) consists of all the problems in NP that are also NP-hard.
  - the “hardest” problems in NP

Recap: Computational complexity

- The existence of *NPC* makes it “likely” (follow common sense/experience) that
  - NPC problems do *not* have a polynomial-time solution
  - $P \neq NP$

Recap: Computational complexity

If you can prove that the problem you are trying to solve is NP-complete, you know that it is unlikely that it is polynomial-time solvable.

Recap: Computational complexity

If you can prove that the problem you are trying to solve is NP-complete, you know that it is unlikely that it is polynomial-time solvable.

To show that your problem is *NP*-complete you must show that it is :

- in *NP*
- is *NP*-hard – can use a reduction based proof

NP Hard - Reduction based proof

Assume we know that problem Y is NP Hard.

How can we prove that another problem Z is NP Hard?

NP Hard - Reduction based proof

Assume we know that problem Y is NP Hard.

How can we prove that another problem Z is NP Hard?

If you can show that Y is polynomial-time reducible Z, then Z is also NP-hard.

NP Hard - Reduction based proof

Assume we know that problem Y is NP Hard.

How can we prove that another problem Z is NP Hard?

If you can show that Y is polynomial-time reducible Z, then Z is also NP-hard.

Why does that work?

NP Hard - Reduction based proof

Assume we know that problem Y is NP Hard.

How can we prove that another problem Z is NP Hard?

If you can show that Y is polynomial-time reducible Z, then Z is also NP-hard.

Why does that work?

If

- every problem $A \in NP$ is *polynomial-time reducible* to Y,
- and Y is *polynomial-time reducable* to Z,

then

- every problem $A \in NP$ is *polynomial-time reducible* to Z.

What to study

- Go over the lecture material
- Redo the revision exercises
- Read the relevant chapters in the textbook
- Look back over assignment material
- Practice past exam papers
- Practice online exam papers

Survey

Please complete the SECaT survey for this course and for tutorials

Good luck

It's been a pleasure teaching you, and I wish you all the best in your final exam

Questions?