Lecture 2: Stable Machings

Sam McCauley

February 5, 2024

Welcome Back!

- Reminder: assignment due Wednesday
- New TA hours on website
- Today: names!
- Any questions before we start?

Correctness Continued

Example 0: Finding Maximum

```
1  indexOfLargest = 0
2  for j = 1 to i:
3      if A[j] > A[indexOfLargest]
4          indexOfLargest = j
```

- What does this code do?
- *Intuitively*, in 1-2 sentences, why?
- What Invariant does it satisfy?
  - One answer: after k iterations, `indexOfLargest` contains the index of the largest element in $A[0] \ldots A[k]$.

Example 0: Finding Maximum

```
1  indexOfLargest = 0
2  for j = 1 to i:
3      if A[j] > A[indexOfLargest]
4          indexOfLargest = j
```

Proof.

I.H.: After k iterations, `indexOfLargest` contains the index of the largest element in $A[0] \ldots A[k]$.

Base case: after 0 iterations, `indexOfLargest` is 0; $A[0]$ is the largest element in $A[0] \ldots A[0]$.

Inductive Step: (contd. next slide)

Example 1: Finding Maximum

```
1  findMax(A, i):
2  indexOfLargest = 0
3  for j = 1 to i:
4      if A[j] > A[indexOfLargest]
5          indexOfLargest = j
```

Proof.

I.H.: After k iterations, `indexOfLargest` contains the index of the largest element in $A[0] \dots A[k]$.

Induc. Step: Assume I.H. is true for some k.

After $k+1$st iteration, if $A[k+1] > A[\texttt{indexOfLargest}]$, then $\texttt{indexOfLargest} = k+1$, and the I.H is true for $k+1$ since $A[k+1]$ is the largest element in $A[0] \dots A[k+1]$.

Otherwise, `indexOfLargest` remains the same, and the I.H. is true for $k+1$ since $A[\texttt{indexOfLargest}]$ is the largest element in $A[0] \dots A[k+1]$. $\square$

Example 1: Selection Sort

```
1  selectionSort(A):
2      for i = |A|-1 to 0:
3          indexOfLargest = 0
4          for j = 1 to i:
5              if A[j] > A[indexOfLargest]
6                  indexOfLargest = j
7          swap(A, i, indexOfLargest)
8
9  swap(A, i, j): // swaps A[i] and A[j]
10     temp = A[i]
11     A[i] = A[j]
12     A[j] = temp
```

- What does the inner loop of selection sort do?
- *Intuitively*, in 1-2 sentences, why is this algorithm correct?
- How can we turn this into an inductive proof?

Proofs in CS 256

- Proofs are a language for you to communicate with me
- Level of detail?
  - Pretend you are explaining to a skeptical classmate.
  - Practice your explanation on a skeptical rubber duck
  - When in doubt: write anything you assume.

Example 2: Insertion Sort

```
1  insertionSort(A):
2      for i = 0 to |A| - 1:
3          j = i
4          while j > 0 and A[j-1] > A[j]:
5              swap(A[j-1], A[j]) # swaps A[j-1] and A[j]
6              j = j - 1
```

- What invariant can we guarantee after the outer loop executes *i* times?
- *Intuitively*, in 1-2 sentences, why is this algorithm correct?
- How can we turn this into an inductive proof?
    - Good at-home exercise. For the sake of time (and reference), I have a proof on the next slide.

Insertion Sort Inductive Proof of Correctness

Theorem

After k iterations of the outer loop, the items in $A[0]$ through $A[k-1]$ are in sorted order.

Proof: By induction. **Base case:** for $k = 1$, $A[0]$ is always in sorted order.

Inductive step: Assume true for some $k \geq 1$. During the $k+1$st iteration of the outer loop, the inner loop maintains that for any j: all items from $A[j]$ to $A[k]$ are in sorted order.

After the inner loop completes, all items from $A[0]$ to $A[j-1]$ are in sorted order (by the I.H. since they were unchanged), and are less than $A[j]$ (otherwise the loop would not stop). Thus, when the $k+1$st iteration of the outer loop completes, all items from $A[0]$ through $A[k]$ are in sorted order.

Insertion Sort 3-sentence Explanation of Correctness

The algorithm maintains the invariant that after k iterations of the outer loop, items in $A[0]$ through $A[k]$ are in sorted order.

This is maintained because on the $k+1$st iteration, the inner loop repeatedly swaps the element e that began in $A[k+1]$ with the previous element if e is smaller than the previous element.

The inner loop therefore maintains that $A[0]$ through $A[k]$ are in the same order, and it places the e in the correct position; therefore, $A[0]$ through $A[k+1]$ are in sorted order.

Power of Invariants

- Can help figure out why algorithms work
- Or don't work! Great for bug finding
- *No universal rule* for finding invariants. Some tips:
  - Try small examples, see what happens
  - What are we trying to solve? What kind of partial work is helpful?
  - What internal state would make the algorithm *wrong*? Can this happen?

Correctness in CS 256

- I will frequently ask you to *explain* correctness
- I will only occasionally ask you to *prove* correctness

Questions about Correctness?

Running Time

Two Broad Questions about Algorithms

- Correctness: does this algorithm work?
- Running time: how fast is this algorithm?

What do we want out of a running time guarantee?

- Is a guarantee (is *always* as fast as we say)

- Platform-independent

IBM
Blue Gene/P

What do we want out of a running time guarantee?

- Is a guarantee (is *always* as fast as we say)
- Platform-independent
    - Analyze as data becomes *large*

Big-O notation

- Ignore constants (they are platform-dependent)
- Analyze performance as input size n becomes large

Definition: $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that:

$$\forall n > n_0, \; f(n) \leq c \cdot g(n)$$

Big-O notation

- Ignore consta

I will not ask you to *formally* prove functions are big-O of others in this class. But I may ask you *if* one is big-O of another (without proof).

- Analyze performance as input size n becomes large

Definition: $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that:

$$\underbrace{\forall n > n_0,}_{\text{large } n} \quad \underbrace{f(n) \leq c \cdot g(n)}_{\text{ignore constants}}$$

Running time example 1: Selection Sort

```
1  selectionSort(A):
2      for i = |A|-1 to 0:
3          indexOfLargest = 0
4          for j = 1 to i:
5              if A[j] > A[indexOfLargest]
6                  indexOfLargest = j
7          swap(A[i], A[indexOfLargest])
```

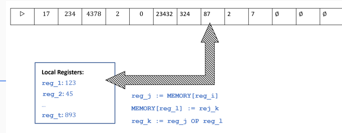
Running time example 1: Selection Sort

```
1  selectionSort(A):
2      for i = |A|-1 to 0: # c₁ time (per loop)
3          indexOfLargest = 0 # c₂ time
4          for j = 1 to i: # c₃ time (per loop)
5              if A[j] > A[indexOfLargest] # c₄ time
6                  indexOfLargest = j # c₅ time
7          swap(A[i], A[indexOfLargest]) # c₆ time
```

Total running time? [On Board #1]

$O(n^2)$ time

Simplifying running time



- We always use big-*O* for running time in this class
- So no need to track constants!
- Assume all basic operations take time 1 (or any *c* basic operations)
    - *Aside:* Formally, we work in the “Word RAM Model.”
    - Access array items, manipulate numbers, execute instructions in time 1
    - We won’t use this model formally in this class

Selection Sort Simplified

```
1  selectionSort(A):
2      for i = |A|-1 to 0:
3          indexOfLargest = 0
4          for j = 1 to i:
5              if A[j] > A[indexOfLargest]
6                  indexOfLargest = j
7          swap(A[i], A[indexOfLargest])
```

Running time for $n = |A|$:

$$\sum_{i=n-1}^{0}\left(1+\sum_{j=0}^{i}1\right) = \sum_{i=n-1}^{0} i+1 = \sum_{j=1}^{n} j = n(n+1)/2 = O(n^2).$$

Running Time

- Running time of for loops is straightforward
- For while loops, we need to account for time more carefully.

Example 2: Insertion Sort

```
1  insertionSort(A):
2      for i = 0 to |A| - 1:
3          j = i
4          while j > 0 and A[j-1] > A[j]:
5              swap(A[j-1], A[j]) # swaps A[j-1] and A[j]
6              j = j - 1
```

- How many steps is the inner loop at most?
  - $O(n)$. (More precisely: $O(i)$)
- What is the final running time?
  - $O(n^2)$

Ignoring Constants Motivation

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Gale-Shapeley Stable Matching

Perfect Stable Matching: Problem Setup

- Medical students need to be matched to residencies
- n students, n hospital openings
- Each student ranks what hospital they want to go to
    - Orders all n hospitals
- Each hospital ranks all students

Perfect Stable Matching Example

	1st	2nd	3rd
OH	Chris	Aamir	Beth
NH	Aamir	Chris	Beth
MA	Aamir	Chris	Beth

	1st	2nd	3rd
Aamir	NH	MA	OH
Beth	MA	OH	NH
Chris	MA	NH	OH

- Definition of Perfect matching: every student is matched to every hospital
  - What is an easy algorithm to create a Perfect matching? [On Board #2]
- Question: what qualities might we want to see out of a *good* matching?

Unstable Pairs

- A matching is *unstable* if there exists a (student, hospital) pair that would rather have each other than their current match
- Such a pair wants to ignore our system, and match each other (maybe leaving others unmatched!)
- Let's say Chris is matched to MA, Beth is matched to New Hampshire, and Aamir is matched to Ohio. [On Board #3]
  - Who wants to leave the algorithm? What is the unstable pair?
- Answer: Aamir and Massachusetts. Aamir would rather have Massachusetts than Ohio; Massachusetts would rather have Aamir than Chris.

Stable Matching

- In stable matching: If a student s is matched to a hospital h, then for any hospital h' that s prefers to h, h' is already matched to someone they prefer to s
- And the reverse: if a hospital h is matched to a student s, any student s' that h prefers is matched to a hospital that s' prefers to h
- Intuitively: if a student calls up a hospital trying to improve their match, the hospital will always respond that they already are matched to a student they prefer

Stable Matching Example

[On Board #4]

I claim that I made a stable matching on the board. How can we check this?

Does a stable matching always exist?

- Not obvious!
- How can we prove this?
- Our plan: create an algorithm to find a stable matching. Prove that it works.

Stable Matching: First Attempt

Question: how would you match students to hospitals in practice? This looks a little like undergraduate admissions...

- Proceed in rounds
- Each unmatched hospital makes offer to its top unmatched candidate
- Each student accepts the top offer and rejects others

Let's try it with our example [On Board #5] and see if it works

It does not work. (Chris, MA) is an unstable pair! (Chris prefers MA to OH; MA prefers Chris to Beth.)

What went wrong?

- In a single round, MA offered to Aamir and OH offered to Chris
- Chris took OH since it was the best offer in that round
- But Aamir rejected MA! MA was free, but Chris never saw the offer.
- The issue with this algorithm is that if a hospital is rejected by a student, it might not be considered “on the market” by other students
- Solution: do not permanently commit to our matchings, allow students to trade up

Gale-Shapely Stable Matching Algorithm

Proceed in rounds. We keep track of “matched” pairs, but do not permanently match them until the end

- Each unmatched hospital offers to its top choice *that it has not offered to yet*
- Each student matches to top offer
- If a student gets a better offer, it rejects current offer and matches to best new offer

Gale-Shapely Pseudocode

```
1  To begin, all students and hospitals are unmatched
2  while ∃ unmatched hospital who has not proposed to every
       student:
3      Let h be such a hospital
4      Let s be highest-ranked student on h's list that h has not
           proposed to
5      if s is unmatched:
6          s and h become matched
7      else:
8          let h′ be the hospital s is currently matched to
9          if s prefers h to h′:
10             s and h become matched
11             h′ becomes unmatched
12 return All matches
```

Let's do this out [On Board #6]

Gale-Shapeley Running Time

- How long does it take to find an unmatched hospital?
- Can you come up with an $O(1)$ time solution?
  - Store unmatched hospitals in a linked list. All of the following take $O(1)$ time: (1) find a new unmatched hospital (2) remove it from the list (3) add a new unmatched hospital to the list
- How long does it take to find the next student the hospital has not proposed to? Can you come up with an $O(1)$ time solution?
- Store students in a linked list in order of preference. We always take the first item from the linked list.

Gale-Shapely Running Time

```
1  To begin, all students and hospitals are unmatched
2  while ∃ unmatched hospital who has not proposed to every
       student:
3      Let h be such a hospital
4      Let s be highest-ranked student on h's list that h has not
           proposed to
5      if s is unmatched:
6          s and h become matched
7      else:
8          let h′ be the hospital s is currently matched to
9          if s prefers h to h′:
10             s and h become matched
11             h′ becomes unmatched
12 return All matches
```

- We showed: inner loop takes $O(1)$ time
- How many times can the outer loop run?
- Answer: n^2: each time a hospital proposes to a *new* student

Gale-Shapely Running Time

```
1  To begin, all students and hospitals are unmatched
2  while ∃ unmatched hospital who has not proposed to every
       student:
3      Let h be such a hospital
4      Let s be highest-ranked student on h's list that h has not
           proposed to
5      if s is unmatched:
6          s and h become matched
7      else:
8          let h′ be the hospital s is currently matched to
9          if s prefers h to h′:
10             s and h become matched
11             h′ becomes unmatched
12 return All matches
```

- Outer loop runs n^2 times, each time it runs takes $O(1)$
- Therefore: $O(n^2)$ running time
- Challenge question: why is this the best possible?

Gale-Shapeley Correctness

Need to show:

- The returned matching is perfect
- The returned matching is stable

First, let's show the matching is perfect.

- Invariant: Once a student is proposed to, they will always stay in a match; furthermore they only swap to a hospital they prefer
- Why must each student eventually get a match?
  - If a student s is unmatched, some hospital h is unmatched.
  - h must not have proposed to s
  - So h is an unmatched hospital that has not proposed to each student

Question: How would you turn the above into a proof?

Gale-Shapeley Correctness

Proof that the returned matching is stable by contradiction:

Assume for returned matching M, $\exists (h, s) \notin M$ such that $(h, s') \in M$ and $(h', s) \in M$, where h prefers s over s', and s prefers h over h'.

h must have offered to s before s'. Since $(h, s) \notin M$, either s broke the match to h, or s rejected the offer from h.

- A student only breaks a match if it receives a better offer. So if s broke the match to h, s must prefer h' to h; contradiction.
- If s rejected the offer from h, it must have been matched to a hospital h'' which it prefers to h. But again, then s must prefer h' to h'', and therefore prefer h' to h, a contradiction.

Gale-Shapeley Wrapup

- Nobel prize in economics for this algorithm
- Traditionally called “stable marriage” problem
- Interesting further questions: are there multiple stable matching? Who benefits from this algorithm (do hospitals get their preference? Students?) What happens if there are ties, or partial preferences, or an uneven number of people?
    - Discussed in Algorithmic Game Theory

More Asymptotic Notation

Big Ω

- One can think of big-O as way to say $\leq$ (ignoring constants and for large n)
- Some true but not useful statements:
  - True: Insertion sort takes $O(n^{10})$ time
  - True: Finding the minimum element in an array using a linear scan takes $O(2^n)$ time
- Big-Ω notation: like big-O, but for $\geq$
- True: Insertion sort takes $\Omega(n^2)$ time
- True: Insertion sort takes $\Omega(n)$ time
- Wrong: Insertion sort takes $\Omega(n^3)$ time

[On Board #7] Definition of big Ω

Big Θ

- We have $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- A version of "$=$" ignoring constants and for large n
- Insertion sort takes $\Theta(n^2)$ time
- True: Insertion sort takes $O(n^3)$ time
- Wrong: Insertion sort takes $\Theta(n \log n)$ time

Questions about Running time and Correctness?