Lecture 2: Big O and Stable Matchings

Sam McCauley

February 10, 2025

Welcome Back!

- Reminder: problem set due Wednesday; daily homeworks starting today
- TA hours on website
- TA hours and office hours in common room for now
- Names today! :)
- Handout on tips for big-O and log rules posted after class
- Any questions before we start?

Correctness Continued

Example 0: Finding Maximum

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

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

Proof.

I.H.: After k iterations (for some $j \in \{1, \ldots, i-1\}$), `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) $\square$

Example 1: Finding Maximum

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

Proof.

I.H.: After k iterations (for some $j \in \{1, ..., i-1\}$), $\texttt{indexOfLargest}$ contains the index of the largest element in $A[0] \ldots A[k]$.

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 $A[0] \ldots A[k+1]$.

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

Example 1: Selection Sort

```
selectionSort(A):
    for i = |A|-1 to 0:
        indexOfLargest = 0
        for j = 1 to i:
            if A[j] > A[indexOfLargest]
                indexOfLargest = j
        swap(A, i, indexOfLargest)

swap(A, i, j): // swaps A[i] and A[j]
    temp = A[i]
    A[i] = A[j]
    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?

Example 1: Selection Sort

```
selectionSort(A):
    for i = |A|-1 to 0:
        indexOfLargest = 0
        for j = 1 to i:
            if A[j] > A[indexOfLargest]
                indexOfLargest = j
        swap(A, i, indexOfLargest)

swap(A, i, j): // swaps A[i] and A[j]
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

- Invariant: after k iterations of the outer loop, the last k positions in A contain the k largest elements in sorted order.
- How could we turn this into an inductive proof? What is the inductive hypothesis?

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

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

- What invariant can we guarantee after the outer loop executes i times?
  - Does the selection sort invariant work?
  - No! The largest element isn't in the correct place after one loop; nor is the smallest.
  - Idea: Items in $A[0]$ through $A[i]$ are in increasing order
- *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 in the slides.

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 increasing 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 increasing order.

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 increasing order.

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

After the inner loop completes, all items from $A[0]$ to $A[j-1]$ are in increasing 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 increasing 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:

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

Big-O Discussion

- In the past, you've used big-O to talk about running time
- But really it's just a way to compare if a function is at least as big as another
  - Can be *bigger!*
- You can say "this algorithm takes $O(n^2)$ time"
- More formally, what you mean is: "the function of the total number of operations taken by this algorithm in the worst case is bounded by $O(n^2)$"
- You can say big O for things other than running time! You ignore constants and assume n is large.
- **In pairs:** Let's say a graph has n vertices. In big-O notation, how many edges does it have?
  (Hint: there can be at most one edge for each pair of vertices in a graph.)

Big-O Notation: Some Useful Assumptions

In this class you can assume:

- If the function is a polynomial, can just take the element with the largest exponent
  - Example: $.3n^5 + 1000n^2 + 2n = O(n^5)$
- Logs are smaller than any polynomial
  - Example: $\log n = O(n^{.01})$
- Exponents are *larger* than any polynomial
  - Example: $n^{100} = O(2^n)$
- $O(1)$ is any constant independent of n
  - Example: $2000 = O(1)$ or $.01 = O(1)$.

Running time example 1: Selection Sort

```
selectionSort(A):
    for i = |A|-1 to 0:
        indexOfLargest = 0
        for j = 1 to i:
            if A[j] > A[indexOfLargest]
                indexOfLargest = j
        swap(A[i], A[indexOfLargest])
```

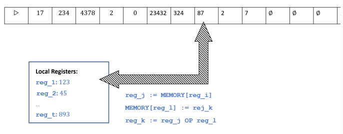
Running time example 1: Selection Sort

```
selectionSort(A):
    for i = |A|-1 to 0: # c1 time (per loop)
        indexOfLargest = 0 # c2 time
        for j = 1 to i: # c3 time (per loop)
            if A[j] > A[indexOfLargest] # c4 time
                indexOfLargest = j # c5 time
        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

```
selectionSort(A):
    for i = |A|-1 to 0:
        indexOfLargest = 0
        for j = 1 to i:
            if A[j] > A[indexOfLargest]
                indexOfLargest = j
        swap(A[i], A[indexOfLargest])
```

Running time for $n = |A|$:

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

This is how we will analyze running time in this class.

Running Time

- Running time of for loops is usually straightforward since we know how many times they run.

- For while loops, we need to account for time more carefully.

Example 2: Insertion Sort

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

- How many steps is the inner loop at most?
    - $O(i)$. ($O(n)$ is also an OK answer here)
- 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-Shapley 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