CSC263 Week 12

Larry Zhang
Announcements

➔ No tutorial this week

➔ PS5-8 being marked

➔ Course evaluation:
 ◆ available on Portal
 ◆ http://uoft.me/course-evals
Lower Bounds
So far, we have mostly talked about upper-bounds on algorithm complexity, i.e., $O(n \log n)$ means the algorithm takes at most $cn \log n$ time for some $c$.

However, sometime it is also useful to talk about lower-bounds on algorithm complexity, i.e., how much time the algorithm at least needs to take.
Scenario #1

You, implement a sorting algorithm with worst-case runtime $O(n \log \log n)$ by next week.

Okay Boss, I will try to do that ~

You try it for a week, cannot do it, then you are fired...
Scenario #2

You, implement a sorting algorithm with worst-case runtime $O(n \log \log n)$ by next week.

No, Boss. $O(n \log \log n)$ is below the lower bound on sorting algorithm complexity, I can’t do it, nobody can do it!
Why learn about lower bounds

Know your limit

- we always try to make algorithms faster, but if there is a limit that you cannot exceed, you want to know

Approach the limit

- Once you have an understanding about of limit of the algorithm’s performance, you get insights about how to approach that limit.
Lower bounds on sorting algorithms
**Upper bounds:** We know a few sorting algorithms with worst-case $O(n \log n)$ runtime.

Is $O(n \log n)$ the best we can do?

Actually, yes, because the lower bound on sorting algorithms is $\Omega(n \log n)$, i.e., a sorting algorithm needs at least $cn \log n$ time to finish in worst-case.
actually, more precisely ... 

The lower bound $n \log n$ applies to only all comparison based sorting algorithms, with no assumptions on the values of the elements.

It is possible to do faster than $n \log n$ if we make assumptions on the values.
Example: sorting with assumptions

Sort an array of $n$ elements which are either 1 or 2.

ściś 2 1 1 2 2 2 1

➔ Go through the array, count the number of 1’s, namely, $k$
➔ then output an array with $k$ 1’s followed by $n-k$ 2’s
➔ This takes $O(n)$. 
Now prove it
the worst-case runtime of comparison based sorting algorithms is in $\Omega(n \log n)$
Sort \( \{x, y, z\} \) via comparisons

Assume \( x, y, z \) are distinct values, i.e., \( x \neq y \neq z \)

A tree that is used to decide what the sorted order of \( x, y, z \) should be ...
The decision tree for sorting \( \{x, y, z\} \)

A tree that contains a complete set of decision sequences
Each **leaf node** corresponds to a possible **sorted order** of \{x, y, z\}, a decision tree need to contain **all possible orders**.

How many possible orders for n elements? \(n!\)

So number of leaves \(L \geq n!\)
Now think about the **height** of the tree

A **binary** tree with height $h$ has at most $2^h$ leaves

So number of leaves $L \leq 2^h$

So number of leaves $L \geq n!$
So,

\[2^h \geq n!\]

\[h \geq \log(n!) \in \Omega(n \log n)\]

So number of leaves \(L \leq 2^h\)

Not trivial, will show it later

So number of leaves \(L \geq n!\)

\[h \in \Omega(n \log n)\]
What does $h$ represent, really?

The worst-case # of comparisons to sort!

\[ h \in \Omega(n \log n) \]
What did we just show?

The worst-case number of comparisons needed to sort $n$ elements is in $\Omega(n \log n)$.

Lower bound proven!
Appendix: the missing piece

Show that $\log (n!)$ is in $\Omega (n \log n)$

$\log (n!)$

$= \log 1 + \log 2 + ... + \log n/2 + ... + \log n$

$\geq \log n/2 + ... + \log n$  \hspace{1cm} (n/2 + 1 of them)

$\geq \log n/2 + \log n/2 + ... + \log n/2$  \hspace{1cm} (n/2 + 1 of them)

$\geq n/2 \cdot \log n/2$

$\in \Omega (n \log n)$
other lower bounds
The problem

Given \( n \) elements, determine the maximum element.

How many comparisons are needed \textit{at least}?
A similar problem
How many matches need to be played to determine a champion out of 16 teams?

Each match **eliminates** at most 1 team.

Need to eliminate **15** teams in order to determine a champion.

So, need at least **15** matches.
The problem

Given \( n \) elements, determine the maximum element.

How many comparisons are needed \textbf{at least}?

Need at least \( n-1 \) comparisons
Insight: approach the limit

How to design a maximum-finding algorithm that reaches the lower bound $n-1$?

➔ Make every comparison count, i.e., every comparison should guarantee to eliminate a possible candidate for maximum/champion.

➔ No match between losers, because neither of them is a candidate for champion.

➔ No match between a candidate and a loser, because if the candidate wins, the match makes no contribution (not eliminating a candidate)
These algorithms reach the lower bound

- Linear scanning
- Tournament
Challenge question

Given $n$ elements, what is the lower bound on the number of comparisons needed to determine both the maximum element and the minimum element?

Hint: it is smaller than $2(n-1)$
The “playoffs” argument kind-of serves as a proof of lower bound for the maximum-finding problem.

But this argument may not work for other problems.

We need a more general methodology for formal proofs of lower bounds.
proving lower bounds using Adversarial Arguments
How does your opponent smartly cheat in this game?

➔ While you ask questions, the opponent alters their ships’ positions so that they can “miss” whenever possible, i.e., construct the worst possible input (layout) based on your questions.

➔ They won’t get caught as long as their answers are consistent with one possible input.
If we can prove that, no matter what sequence of questions you ask, the opponent can always craft an input such that it takes at least 42 guesses to sink a ship.

Then we can say the **lower bound** on the complexity of the “sink-a-ship” problem is 42 guesses, no matter what “guessing algorithm” you use.
more formally ...

To prove a lower bound $L(n)$ on the complexity of problem $P$,

we show that for every algorithm $A$ and arbitrary input size $n$, there exists some input of size $n$ (picked by an imaginary adversary) for which $A$ takes at least $L(n)$ steps.
Example: search unsorted array

Problem:
Given an unsorted array of n elements, return the index at which the value is 42. (assume that 42 must be in the array)
Possible algorithms

➔ Check through indices 1, 2, 3, ..., n
➔ Check from n, n-1, n-2, ..., to 1
➔ Check all odd indices 1, 3, 5, ..., then check all even indices 2, 4, 6, ...
➔ Check in the order 3, 1, 4, 1, 5, 9, 2, 6, ...

Prove: the lower bound on this problem is $n$, no matter what algorithm we use.
Proof: (using adversarial argument)

→ Let \( A \) be an \textit{arbitrary} algorithm in which
the first \( n \) indices checked are \( i_1, i_2, \ldots, i_n \).

→ Construct (adversarially) an input array \( L \)
such that \( L[i_1], L[i_2], \ldots, L[i_{n-1}] \) are \textit{not} \( 42 \),
and \( L[i_n] \) is \( 42 \).

→ Because \( A \) is arbitrary, therefore the lower bound on the complexity of solving this
problem is \( n \), no matter what algorithm is used.
proving lower bounds using Reduction
The idea

→ Proving one problem’s lower bound using another problem’s known lower bound.

→ If we know problem B can be solved by solving an instance of problem A, i.e., A is “harder” than B

→ and we know that B has lower bound $L(n)$

→ then A must also be lower-bounded by $L(n)$
Example:

Prove: ExtractMax on a binary heap is lower bounded by $\Omega(\log n)$.

Suppose ExtractMax can be done faster than $\log n$, then HeapSort can be done faster than $n \log n$, because HeapSort is basically ExtractMax $n$ times.

But HeapSort, as a comparison based sorting algorithm, has been proven to be lower bounded by $\Omega(n \log n)$. Contradiction, so ExtractMax must be lower bounded by $\Omega(\log n)$.
WE'RE DONE!
Final thoughts
what did we learn in CSC263
Data structures are the underlying skeleton of a good computer system.

If you will get to design such a system yourself and make fundamental decisions, what you learned from CSC263 should give you some clues on what to do.
→ Understand the nature of the system / problem, and model them into structured data
→ Investigate the probability distribution of the input
→ Investigate the real cost of operations
→ Make reasonable assumptions and estimates where necessary
→ Decide what you care about in terms of performance, and analyse it
  ◆ “No user shall experience a delay more than 500 milliseconds” -- worst-case analysis
  ◆ “It’s ok some rare operations take a long time” -- average-case analysis
  ◆ “what matter is how fast we can finish the whole sequence of operations” -- amortized analysis
In CSC263, we learned to be a computer scientist, not just a programmer.
what we did NOT learn

but are now ready to learn
Awesomer kinds of heaps

➔ Sometimes we want to be able to merge two heaps into one heap, with binary heap we can do it in $O(n)$ time worst-case.

➔ Using binomial heap, we can do merge in $O(\log n)$ time worst-case

➔ Using Fibonacci heap, we can do merge (as well as Max/Insert/IncreaseKey) in $O(1)$ time amortized.
Awesomer kinds of search trees

➔ We learned BST and AVL tree, and there are others called red-black tree, 2-3 tree, splay tree, AA tree, scapegoat tree, etc.

➔ There is B-tree, optimized for accessing big blocks of data (like in a hard drive)

➔ There is B+ tree, which is even better than B-tree (widely used in database systems).

➔ You’ll learn about these in CSC443.
Awesomer kinds of hashing

➔ **Universal hashing** which provably guarantees simple uniform hashing

➔ **Perfect hashing** guarantees **worst-case** $O(1)$ time for searching, instead of **average-case** $O(1)$ time
Shortest paths in a graph

➔ We learned how to get shortest paths using BFS on a graph

➔ We did NOT learn how to get shortest (weighted) paths on a weighted graph.
  ◆ Dijkstra, Bellman-Ford, ...

➔ You’ll learn about them in CSC358 / 373
Greedy algorithms

➔ We learned that Kruskal’s and Prim’s MST algorithms are greedy

➔ What property is satisfied by the problems that can be perfectly solved by greedy algorithms?

➔ Will learn in CSC373
Dynamic programming

➔ Pick an interesting algorithm design problem, very likely it involves dynamic programming

➔ Will learn in CSC373
P vs NP, approximation algorithms

➔ We learned a bit about lower bounds.

➔ There are some problems, we can prove they cannot be perfectly solved in polynomial time.

➔ For these problems, we have to design some approximation algorithms.

➔ Will learn in CSC373 / 463
As our circle of knowledge expands, so does the circumference of darkness surrounding it.
Final Exam Prep
Topics covered: all of them

➔ Heaps
➔ BST, AVL tree, augmentation
➔ Hashing
➔ Randomized algorithms, Quicksort
➔ Graphs, BFS, DFS, MST
➔ Disjoint sets
➔ Lower bounds
➔ Analysis: worst-case, average-case, amortized.
Types of questions

➔ Short-answer questions testing basic understanding.
➔ Trace operations we learned on a data structure
➔ Implement an ADT using a data structure
➔ Analysis runtimes
  ◆ best / worst-case
  ◆ average-case
  ◆ amortized cost
➔ Given a real-world problem, design data structures / algorithms to solve it.
Study for the exam

➔ Review lecture notes/slides
➔ Review tutorial problems
➔ Review all problem sets / assignments
➔ Practice with past exams (available at old exam repository at UofT library)
➔ Come to **office hours** whenever confused.
Larry’s pre-exam office hours

➔ All Thursdays 2-4pm
➔ All Fridays 2-4pm
➔ Monday, April 13, 4-6pm
➔ Tuesday, April 14, 4-6pm
➔ Wednesday, April 15, 4-6pm
➔ Monday, April 20, 4-6pm
➔ Tuesday, April 21, 4-6pm
Exam Time & Location

Wednesday, April 22nd, PM 2:00 - 5:00

Locations:

➔ A - HO: NR 25
➔ HU - NGO: ST VLAD
➔ NGU - WI: UC 266
➔ WL- Z: UC 273

double-sided, handwritten aid-sheet

Go to the right location.
All the best!