Welcome to Algorithms!

L1: Introduction

Introduction

- Class is about *designing* and *analyzing algorithms*. What does that mean?
  - *Algorithm*: A well-defined procedure that takes an input and computes some output.
  - *Design of algorithms*: The process of coming up with algorithms. This is both an art and a science. There are no recipes, but there exist some general techniques and we'll see some of them.
  - *Analysis of algorithms*: Abstract/mathematical comparison of algorithms without actually implementing them. Think of analysis as a measure of the quality of your algorithm and you'll use it to justify design decisions when you write programs.

This week's goals

The goal for this first week is to review fundamental algorithms for searching and sorting and their analysis.

- Searching: linear search, binary search
- Quadratic sorting: bubble sort, selection sort, insertion sort
- Analysis: best and worst-case running times using $O()$ as introduced in Data Structure.

1 Searching

The search problem: Given an input array A of arbitrary size which we denote by n, $A[0...n-1]$ and a target element t, find if t occurs in A (yes/no answer).

- Variations: find index where it occurs; find all occurences; count number of occurences.

There are two basic ways to search: the so-called *linear search* (why is it called linear?), and *binary search*. Both of these should be familiar — you may want to find your notes and/or other resources to review these.

2 Sorting

The sorting problem: Given an input array A of of arbitrary size which we denote by n, $A[0...n-1]$, and a target element t, find if t occurs in A (yes/no n elements, permute the elements such that the sequence is now sorted, i.e. $A[0] \leq A[1] \leq A[2] \leq ... \leq A[n-1]$.

- Note that the way sorting is defined above, the input array is permuted so that it's sorted. This is called *in-place sorting*; we'll come back to this in a few lectures. For now, just make a mental note that the other possibiity is for the sorting algorithm to create a new array which is the sorted version of the input array. Why does this matter? Creating a new array uses space, and when the array is xxxl, space matters. In-placeness is a desirable feature of a sorting algorihm, and one we need to be aware of.
- We'll warm up with the following algorithms: bubble sort, selection sort, insertion sort
- You may have seen these, or some of these, in Data Structures. If not, no worries, you get to see them now!
- Go through each algorithm and try to answer the questions. They are meant to help you understand the algorithms, as well as get used to think with an "algorithms hat" on.
- Note: All in-lecture questions are at collaboration-level 0, that is, no restrictions. Feel free to work with peers and search the www if that helps you learn better.

2.1 Bubblesort

```
Bubble-Sort(A)
1  For $k = 1$ to $n-1$
2      // do a bubble pass
3      For $i = 0$ to $n-2$
4          if $A[i] > A[i+1]$: swap
```

Study questions:

1. Run the algorithm on $A = [3, 1, 5, 7, 4, 6, 2]$, and show the array after every iteration.
2. What can you say about the last element in A after one bubble pass? Make a statement and try to argue why it's true. (Hint: there is one element that is always guaranteed to be in the right place after one pass...which one? why?)
3. What happens after two bubble passes? Use the example above to guide you. (Hint: there are two elements that are guaranteed to be in the right places...which ones?)
4. Using this insight, argue that after $n-1$ bubble passes the input is sorted. This is saying that the algorithm is correct.

5. Because after $n-1$ passes the input is guaranteed to be sorted (see above), this tells us that $n-1$ bubble passes are *sufficient* to sort. Now we ask the following question: Are $n-1$ passes *necessary*? In other words, is there an array of n elements that is **not** sorted after $n-2$ bubble passes and therefore needs the last $(n-1)$-th pass to be sorted? If the answer is no, then we could write a better algorithm that still sorts correctly but does fewer passes.

   Come up with an array A that needs precisely $n-1$ bubble passes to be sorted.

2.2 Selection sort

The idea of selection sort is to select the smallest element, swap it with the first element, and repeat on the remaining elements. We start by writing it using high-level statements, such as the one in line 2 below . This is something we often do when we try to come up with algorithms: we sketch the algorithm in terms of high-level blocks until we get the overall idea right, and we fill in the detail of each block later. It helps to keep the focus on the big picture before taking care of the details.

```
SELECTION-SORT($A$)
1  For $i = 0$ to (?)
2       $k$ = the index of the smallest element among $A[i], A[i+1], ..., A[n-1]$
3       swap $A[i]$ with $A[k]$
```

Study questions:

1. Show how this works on the array $A = (3, 1, 5, 7, 4, 6, 2)$.

2. Fill in the omitted detail in the “for” loop (ie what should the ? be in line 1 of the algorithm? ie, how many iterations are necessary?).

3. Now think about how you would implement line 2 in the algorithm. Namely, write an algorithm FIND_SMALLEST(A, i, n) which takes as input an array A of n elements, and an index i such that $0 \leq i < n$, and finds the index of the smallest element among $A[i], A[i+1], ..., A[n-1]$

2.3 Insertion sort

Insertion sort works similarly with sorting a deck of cards: Initially all cards are on the table and your hand is empty. You take one card at a time from the table and insert it in the hand in the right place, until the pile of cards on the table is finished. Along the way you make sure to keep the cards in your hand sorted.

The algorithm is easy to understand, but a little more complicated to write if we insist to do it by permuting A, that is, without allocating a new array to perform the insertion. It starts with one card in hand, and $n-1$ on the table. It has two nested loops: the first loop runs precisely $n-1$ times, and every time it runs it looks at the next element $A[k]$. The inner loop performs the insert of $A[k]$ into $A[0...k-1]$. This is done by finding the correct place where $A[k]$ fits in $A[0], ..., A[k-1]$, placing it there and copying all elements to its right (ie larger than $A[k]$) one space over to the right in order to make space for inserting $A[k]$. In fact the shift to the right can be done while searching for the right place for $A[k]$. The algorithm is described below.

```
INSERTION-SORT(A)
1  For $k = 1$ to $n - 1$
2       //invariant: $A[0] \leq A[1] \leq A[2] \leq ...A[k-1]$
3       $key = A[k]$
4       $i = k - 1$
5       while $i \geq 0$ and $A[i] > key$
6            $A[i+1] = A[i]$
7            $i = i - 1$
8       $A[i+1] = key$
```

Study questions:

1. Show how this works on $A = (3, 1, 5, 7, 4, 6, 2)$. The goal of this exercise is to go through the algorithm and understand how it work.
2. Note the *invariant* in line 2. An invariant is something that we claim to be true. In this case, the invariant is inside the loop so we claim it is true at at every iteration of the loop; sometimes it is refered to as a *loop invariant*. What does this invariant state for the first iteration of the loop($k = 1$)? Spell it out. Is it true?
3. What part of the array A corresponds to the "hand"?
4. Describe in a short sentence what the inner loop does.
5. For a given value of k, how many times does the inner while loop run, in the best case?
6. Same question as above, but the worst-case.
7. Assume $n = 7$ and give a best-case and worst-case input for insertion-sort.
8. Note: So the invariant is trivially true for $k = 1$. To prove that the invariant is true for every k one would use the technique called *induction*. If you have seen induction, try to sketch the proof. If not, no worries, move on.

Once you understand it you'll probably agree that this little piece of code is quite elegant! It sorts A while using only three additional variables (i, k, key), so it is very space efficient. Does it matter that an algorithm is space-efficient? Sometimes. If A has trillions of elements, being stingy with space is a good idea.

3 Analysis review

We'll spend all next week to formally define analysis, but for now we want to review the concepts from Data Structures.

- The goal of analysis is to model theoretically the performance of an algorithm (without implementing it)
- By default performance means running time; but could also mean memory, bandwidth, disk accesses, etc.
- The running time of an algorithm is roughly the number of instructions it executes $\times$ the time taken by an instruction (assuming all instructions are equal, more on that later). The time taken to execute an instruction is the same on a given platform, and factoring it in will not help distinguish between algorithms. So we ignore it and consider that running time = nb. of instructions
- Inputs of different sizes will have different running times, thus the running time is a function of the input size n, and is usually denoted by $T(n)$.
- Consider various inputs of a given size. For e.g. linear search can run precisely one comparison and find the target as the first element in the array. Or, it can run through the entire array without finding the target. Thus $T(n)$ depends on the particular input of size n.
- Consider all possible inputs of a size n. The *best case runnig time* is the shortest possible running time $T(n)$ on an input of size n. For e.g. linear search can run precisely one comparison and find the target as the first element in the array
- Similarly, the *worst case runnig time* is the largest possible running time $T(n)$ on an input of size n. For e.g. the worst case of linear search is running through the whole array and not finding the target.
- In Data Structures you expressed the running time using the big-oh notation. Basically this means ignoring constants and looking for the dominant term. For e.g. $O(n), O(n^2)$, and so on. We'll introduce this formally next week.
- Example: The best case of linear search is $O(1)$ (i.e. a constant), and corresponds to when the target is the first element in the array.
- Example: The worst case of linear search is $O(n)$ and corresponds to an array of size n and a target that does not occur in the array.

Study questions:

1. What is the best case and worst case running time of Bubblesort, as written above? What sort of input arrays trigger these cases?
2. What is the best case and worst case running time of Selection sort? What sort of input arrays trigger these cases?
3. What is the best case and worst case running time of Insertion sort? What sort of input arrays trigger these cases?