# Question 1 of 3

Recall that pseudocode is an informal, English-like language that describes an algorithm. In Lecture 0, David showed the following pseudocode for finding Mike Smith in a phone book.

|  |
| --- |
| 1 Pick up phone book  2 Open to middle of phone book  3 Look at page  4 If Smith is on page  5 Call Mike  6 Else if Smith is earlier in book  7 Open to middle of left half of book  8 Go back to line 3  9 Else if Smith is later in book  10 Open to middle of right half of book  11 Go back to line 3  12 Else  13 Quit |

1. Write pseudocode for an algorithm that would identify the tallest person in a room.
2. What is the running time of your algorithm in Big O notation?
   * *Hint: if your algorithm has one or more loops, how many times do the loops execute if there are N people in the room?*

## Answers

2. Line up (or assemble) the persons in any random order
3. Measure the height of the first person in the line and remember this height and the person in two variables (say **tallestHeight** and **tallestPerson**)
4. Do the following until there is no more person to measure
   1. Measure the next person’s height. Call this **currentHeight** and **currentPerson**.
   2. If **currentHeight** is larger than the **tallestHeight**, then assign **currentHeight** to **tallestHeight** and **currentPerson** to **tallestPerson**
   3. If there are more people left in the room,

go to (a) and repeat the process,

* 1. Else

go to Step 4.

1. The variables **tallestHeight** and **tallestPerson** now refer to the tallest person in the room. Return them and quit.
2. Since we are trying to identify the tallest person in a group of people, we need to look at all the members. Therefore, the worst-case time complexity is *O*(*n*).

# Question 2 of 3

Recall from lecture how we implemented a phone book in C using the below struct.

|  |
| --- |
| typedef struct  {  string name;  string number;  }  person; |

1. Why was it arguably better design to use one array of structs than to use two arrays, one to store names and one to store phone numbers?
2. Imagine you were to use the above struct to implement an app for your contacts, like the one on your mobile phone. What are two additional fields might you want to add to the struct, and what should their types be?

## Answers

1. Using one array of **structs** allows us to move both name and number together as a single element. For example, let us say that we want to move the 3rd element in the array corresponding to Joan (number = 617-123-4567) to position 7. Since both the name (Joan) and her number (617-123-4567) are part of a single struct in the array, we only need to move this single element to position 7. If we maintain name and number as two separate arrays of structs, then every time we move a name in the first array, we also need to move the corresponding number in the second array. This requires more work.

Secondly, using two separate arrays of **structs** requires more space than using a single array of structs.

1. We can add (1) street and (2) city of the person as additional fields to the above **struct**. Both of these fields are of type string. (Other typical fields that are useful for a phone record is (3) house number and (4) state. House number can be of type int and state can be of type string.)

# Question 3 of 3

Imagine that you have an unsorted collection of items (maybe they're notes for class, or a collection of old receipts) that you expect you'll need to search. When might it make more sense to sort the collection of items first before searching, and when might it make more sense to leave the collection unsorted?

*Hint: Consider algorithmic efficiency. What's the cost (i.e., running time) of linear search? Of binary search? Of sorting?*

## Answers

If the collection has any searchable key (such as dates in old receipts or dates the notes were taken), then we can first sort the collection before searching. This will be more efficient, especially if the collection is large and there will be repeated searches. Since the cost of sorting (say *O*(*n* log *n*) or *O*(*n*2) depending on the algorithm used) is incurred only once, but a search using binary search only costs *O*(log *n*), this method is more efficient (than searching in an unsorted collection).

If the collection does not have any searchable key (such as dates in old receipts or dates the notes were taken), we can leave them unsorted. In this case, there is no overhead for the cost of sorting, and the cost of the search is that of a linear search or *O*(*n*). In the worst-case scenario, we will have to search until the last element to find the item of interest.