CS1110

Lecture 21: **More sequence algorithms**

Announcements

Two morals from A4:
- Sometimes even seemingly random human behavior can be modeled and predicted precisely (fraction-converted fixed point).
- A good enough idea (small *t*) promoted by even a small but vocal group (large *d*) can *change the whole world*.

Typo in A6 _drawHBar spec: see Piazza @309.
No office hours *next* Wed-Fri (can't start grading until Thu Apr 18)

Slides by D. Gries, L. Lee, S. Marschner, W. White

Invariants: Keep in mind

- At heart, an invariant is just a way to *document what you want your variables to mean*.
  This is why you want your code to keep the invariant true; you want to keep things consistent in your program, and in your head.

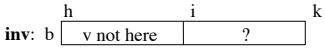
- In our notation, both
  `b[i+1..i]` and
  `b[i..i-1]`
  denote an empty sequence.

Linear search in unsorted lists

Goal: *Given* unsorted list b, search range h..k-1 for k >=h, and target value v, *return* index n of v's first occurrence in b[h..k-1] (-1 if not found)

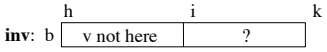
Restated as postcondition: if n=-1, then v is not in b[h..k-1]. Otherwise, v = b[n] and v is not in b[h..n-1].

Idea: keep an index i, marking position of next thing unchecked; everything to its left has been verified to not be v.



If b[i] = v, we can stop and return i as n; if i==k, v isn't in b.

Linear Search




```
def linear_search(b,h,k,v):
    """(see previous)"""
    i = h
    # inv says: b[h..i-1] not v; start: b[h..h-1] not v;
    # end: b[i] is v or i is k.
    while i < k and b[i] != v:
        i = i + 1
    n = i if i < j else -1
    return n
```

Analyzing the Loop

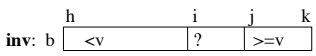
1. Does the initialization make **inv** true?
2. Is **post** true when **inv** is true and **condition** is false?
3. Does the repetend make progress?
4. Does the repetend keep **inv** true?

Binary search in *sorted* lists

Goal: *Given* **sorted** list b, search range h..k for k >=h, and target value v, *return* index n of v's first occurrence in b[h..k-1] (-1 if not found)

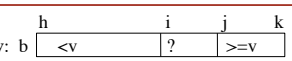
Restated as postcondition: if n=-1, then v is not in b[h..k-1]. Otherwise, v = b[n] and v is not in b[h..n-1].

Idea: keep indices i and j, marking position of next thing not known to be < v, and the first thing known to be >=v. Check halfway btwn 'em.



If b[i] = v, return i as n; if i> k or i ==j and b[i] not v or i=k+1, v isn't in b.

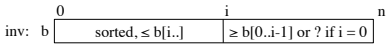
(most of) Binary search implementation




```
def bin_search(b,h,k,v):  # omitting the last return for space
    """(see previous)"""
    Q1: (A) i = h; j=k    (B) i=h-1; j = k+1   (C) i = h-1; j = k   (D) i = h; j = k+1
    # inv: b[h..i-1] < v, b[j..k] >=v, i <= j; start: b[h..h-1] < v, b[k+1..k] >= v
    while Q2:    (A) i==j      (B) i < j     (C) i <=j
        if b[i] == v:
            return i
        mid = (i+j)/2
        if b[mid] < v:
            Q3:    (A) i = mid      (B) i = mid+1
        else:
            j = mid    # may skip vast section of b
```

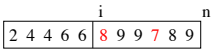
Sorting: Selection Sort

Selection Sort:

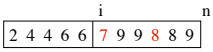


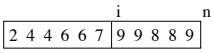
INITIALIZE AND COMPLETE

```
while ...:
    j = b[0:n].index(min(b[i:n]))
```



Note the swap of the reds

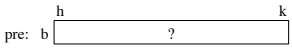




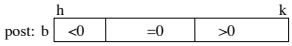
(blank for room to write)

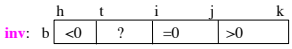
Famous "Sort-Like" Example

- Dutch national flag: tri-color
  - Sequence of h..k of red (<0), white (=0), blue (>0) "pixels"
  - Arrange to put <0 first, then =0 , then >0, return "split pts"



(values in h..k are unknown)





b[h..t-1] <0, b[t..i-1] unknown, b[i..j] =0, b[j+1..k] >0

Dutch National Flag Algorithm

```
def dnf(b, h, k):
    """(DNF explanation omitted for space.)
    Returns: split-points as a tuple (i,j)"""
    # init?
    # inv: b[h..t-1] < 0, b[t..i-1] ?, b[i..j] = 0, b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            #  what?

        elif b[i-1] == 0:
            # what?
        else:
            # what?

    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
```