

Linear Search

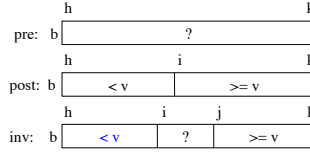
```
def linear_search(b,c,h):
    """Returns: first occurrence of c in b[h..]"""
    # Store in i the index of the first c in b[h..]
    i = h
    # invariant: c is not in b[0..i-1]
    while i < len(b) and b[i] != c:
        i = i + 1
    # post: c is not in b[h..i-1]
    # i >= len(b) or b[i] == c
    return i if i < len(b) else -1
```

Analyzing the Loop

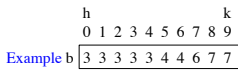
1. Does the initialization make **inv** true?
2. Is **post** true when **inv** is true and **condition** is false?
3. Does the repetend make progress?
4. Does the repetend keep the invariant **inv** true?

Binary Search

- Look for value v in **sorted** segment b[h..k]

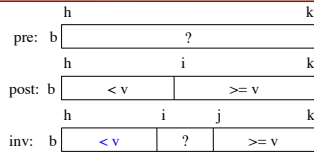


New statement of the invariant guarantees that we get **leftmost** position of v if found



- if v is 3, set i to 0
- if v is 4, set i to 5
- if v is 5, set i to 7
- if v is 8, set i to 10

Binary Search

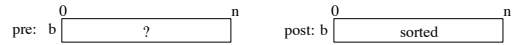


New statement of the invariant guarantees that we get **leftmost** position of v if found

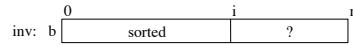
```
i = h; j = k+1;
while i != j:
```

- Looking at b[i] gives **linear search from left**.
- Looking at b[j-1] gives **linear search from right**.
- Looking at middle: b[(i+j)/2] gives **binary search**.

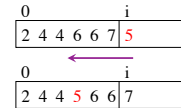
Sorting: Arranging in Ascending Order



Insertion Sort:



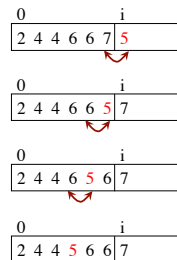
```
i = 0
while i < n:
    # Push b[i] down into its
    # sorted position in b[0..i]
    i = i+1
```



Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1
def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

swap shown in the lecture about lists



Insertion Sort: Performance

```
def push_down(b, i):
    """Push value at position i into
    sorted position in b[0..i-1]"""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

Insertion sort is an **n²** algorithm

Total Swaps: 0 + 1 + 2 + 3 + ... (n-1) = (n-1)*n/2

Algorithm "Complexity"

- **Given:** a list of length n and a problem to solve
- **Complexity:** rough number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

Complexity	n=10	n=100	n=1000
n	0.01 s	0.1 s	1 s
n log n	0.016 s	0.32 s	4.79 s
n ²	0.1 s	10 s	16.7 m
n ³	1 s	16.7 m	11.6 d
2 ⁿ	1 s	4x10 ¹⁹ y	3x10 ²⁹⁰ y

Major Topic in 2110: Beyond scope of this course

Sorting: Changing the Invariant

pre: b [0 ? n] post: b [0 sorted n]

Selection Sort:

inv: b [0 sorted, ≤ b[i..] i ≥ b[0..i-1] n] First segment always contains smaller values

```

i = 0
while i < n:
    j = index of min of b[i..n-1]
    swap(b,i,j)
    i = i+1
    
```

Selection sort also is an n² algorithm

Partition Algorithm

- Given a list segment b[h..k] with some value x in b[h]:

pre: b [h x ? k]

- Swap elements of b[h..k] and store in j to truthify post:

post: b [h <= x i i+1 k]

change: b [h 3 5 4 1 6 2 3 8 1 k]

into b [h 1 2 1 3 5 4 6 3 8 k]

or b [h 1 2 3 1 3 4 5 6 8 k]

- x is called the **pivot value**
 - x is not a program variable
 - denotes value initially in b[h]

Sorting with Partitions

- Given a list segment b[h..k] with some value x in b[h]:

pre: b [h x ? k]

- Swap elements of b[h..k] and store in j to truthify post:

post: b [h <= y y >= y x i i+1 k]

Partition Recursively

Recursive partitions = sorting

- Called **QuickSort** (why???)
- Popular, fast sorting technique

QuickSort

```

def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    j = partition(b, h, k)
    # b[h..j-1] <= b[j] <= b[j+1..k]
    # Sort b[h..j-1] and b[j+1..k]
    quick_sort(b, h, j-1)
    quick_sort(b, j+1, k)
    
```

- **Worst Case:** array already sorted
 - Or almost sorted
 - n² in that case
- **Average Case:** array is scrambled
 - n log n in that case
 - Best sorting time!

pre: b [h x ? k]

post: b [h <= x i i+1 k]

Final Word About Algorithms

- **Algorithm:**
 - Step-by-step way to do something
 - Not tied to specific language
- **Implementation:**
 - An algorithm in a specific language
 - Many times, not the "hard part"
- Higher Level Computer Science courses:
 - We teach advanced algorithms (pictures)
 - Implementation you learn on your own