PDSP2025, Lecture 22, 4 November 2025

Searching and Sorting

Setup

- Use `time` library to time executions

In [1]:
```
import time
```

Naive search by scanning the list

In [2]:
```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Binary search

In [3]:
```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Checking correctness on input `[0,2,...,50]`

In [4]:
```
l = list(range(0,51,2))

for i in range(51):
  print((i,naivesearch(i,l)),end=",")
print()

for i in range(51):
  print((i,binarysearch(i,l)),end=",")
print()
```

```
(0, True),(1, False),(2, True),(3, False),(4, True),(5, False),(6, True),
(7, False),(8, True),(9, False),(10, True),(11, False),(12, True),(13, Fal
se),(14, True),(15, False),(16, True),(17, False),(18, True),(19, False),
(20, True),(21, False),(22, True),(23, False),(24, True),(25, False),(26,
True),(27, False),(28, True),(29, False),(30, True),(31, False),(32, Tru
e),(33, False),(34, True),(35, False),(36, True),(37, False),(38, True),(3
9, False),(40, True),(41, False),(42, True),(43, False),(44, True),(45, Fa
lse),(46, True),(47, False),(48, True),(49, False),(50, True),
(0, True),(1, False),(2, True),(3, False),(4, True),(5, False),(6, True),
(7, False),(8, True),(9, False),(10, True),(11, False),(12, True),(13, Fal
se),(14, True),(15, False),(16, True),(17, False),(18, True),(19, False),
(20, True),(21, False),(22, True),(23, False),(24, True),(25, False),(26,
True),(27, False),(28, True),(29, False),(30, True),(31, False),(32, Tru
e),(33, False),(34, True),(35, False),(36, True),(37, False),(38, True),(3
9, False),(40, True),(41, False),(42, True),(43, False),(44, True),(45, Fa
lse),(46, True),(47, False),(48, True),(49, False),(50, True),
```

Performance comparison across 10^4 worst case searches in a list of size 10^5

- Looking for odd numbers in a list of even numbers

In [5]:
```
l = list(range(0,200000,2))

starttime = time.perf_counter()
for i in range(3001,23000,2):
  v = naivesearch(i,l)
elapsed = time.perf_counter() - starttime
print()
print("Naive search", elapsed)

starttime = time.perf_counter()
for i in range(3001,23000,2):
  v = binarysearch(i,l)
elapsed = time.perf_counter() - starttime
print()
print("Binary search", elapsed)
```

```
Naive search 17.956641599004797

Binary search 1.9248965570004657
```

Selection sort

In [6]:
```
def SelectionSort(L):
   n = len(L)
   if n < 1:
      return(L)
   for i in range(n):
      # Assume L[:i] is sorted
      mpos = i
      # mpos is position of minimum in L[i:]
      for j in range(i+1,n):
        if L[j] < L[mpos]:
           mpos = j
      # L[mpos] is the smallest value in L[i:]
      (L[i],L[mpos]) = (L[mpos],L[i])
```

```
      # Now L[:i+1] is sorted
   return(L)
```

Selection sort performance is more or less the same for all inputs

In [7]:
```
import random
random.seed(2025)
inputlists = {}
inputlists["random"] = [random.randrange(100000) for i in range(5000)]
inputlists["ascending"] = [i for i in range(5000)]
inputlists["descending"] = [i for i in range (4999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    SelectionSort(tmplist)
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 0.41388963800272904
ascending 0.3815801900054794
descending 0.390925444997265
```

Insertion sort, iterative

In [8]:
```
def InsertionSort(L):
   n = len(L)
   if n < 1:
      return(L)
   for i in range(n):
      # Assume L[:i] is sorted
      # Move L[i] to correct position in L[:i]
      j = i
      while(j > 0 and L[j] < L[j-1]):
        (L[j],L[j-1]) = (L[j-1],L[j])
        j = j-1
      # Now L[:i+1] is sorted
   return(L)
```

Insertion sort preformance

- On already sorted input, performance is very good
- On reverse sorted input, performance is worse than selection sort

In [9]:
```
import random
random.seed(2025)
inputlists = {}
inputlists["random"] = [random.randrange(100000) for i in range(5000)]
inputlists["ascending"] = [i for i in range(5000)]
inputlists["descending"] = [i for i in range (4999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    InsertionSort(tmplist)
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 0.5835713570049847
ascending 0.0002267620002385229
descending 1.1273996609961614
```

Insertion sort, recursive

Setup

- Set recursion limit to maxint, $2^{31} - 1$
  - This is the highest value Python allows

In [10]:
```
import sys
sys.setrecursionlimit(2**31-1)
```

In [11]:
```
def Insert(L,v):
   n = len(L)
   if n == 0:
     return([v])
   if v >= L[-1]:
     return(L+[v])
   else:
     return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
   n = len(L)
   if n < 1:
      return(L)
   L = Insert(ISort(L[:-1]),L[-1])
   return(L)
```

Recursive insertion sort is slower than iterative

- Input of 2000 (40%) takes more time than 5000 for iterative
  - Overhead of recursive calls
- Performance pattern between unsorted, sorted and random is similar

In [12]:
```
import random
random.seed(2025)

inputlists = {}
inputlists["random"] = [random.randrange(100000) for i in range(2000)]
inputlists["ascending"] = [i for i in range(2000)]
inputlists["descending"] = [i for i in range (1999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    ISort(tmplist)
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 2.9386179680004716
ascending 0.006556373999046627
descending 4.456001231003029
```

Merge sort

In [13]:
```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (n-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

In [14]:
```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

A simple input to check correctness

In [15]:
```
mergesort([i for i in range(0,100,2)]+[j for j in range (1,100,2)])
```

Out[15]: [0,
1,
2,
3,
4,
5,
6,
7,
8,
9,
10,
11,
12,
13,
14,
15,
16,
17,
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,

```
 60,
 61,
 62,
 63,
 64,
 65,
 66,
 67,
 68,
 69,
 70,
 71,
 72,
 73,
 74,
 75,
 76,
 77,
 78,
 79,
 80,
 81,
 82,
 83,
 84,
 85,
 86,
 87,
 88,
 89,
 90,
 91,
 92,
 93,
 94,
 95,
 96,
 97,
 98,
 99]
```

Perfomance on large inputs, 10^6, random and sorted

In [16]:
```
import random
random.seed(2025)
inputlists = {}
inputlists["random"] = [random.randrange(100000000) for i in range(100000
inputlists["ascending"] = [i for i in range(1000000)]
inputlists["descending"] = [i for i in range (999999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    mergesort(tmplist)
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 2.7101028859979124
ascending 1.4198947430049884
descending 1.4313436170050409
```

Quicksort

In [17]:
```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else:  # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return
```

Small input to check correctness

In [18]:
```
qlist = [1,3,5,0,2,4,17,2,-5,6,4,3]
quicksort(qlist,4,8)
print(qlist)
```

```
[1, 3, 5, 0, 2, 2, 4, 17, -5, 6, 4, 3]
```

Quicksort performance

- Random input of size 10^6
- Sorted inputs of size 2×10^4

In [19]:
```
import random
random.seed(2025)
inputlists = {}
inputlists["random"] = [random.randrange(100000000) for i in range(100000
inputlists["ascending"] = [i for i in range(15000)]
inputlists["descending"] = [i for i in range (14999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    quicksort(tmplist,0,len(tmplist))
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 1.9635030670033302
ascending 4.193553910001356
descending 5.847825593002199
```

Randomized quicksort

- Choose the pivot position uniformly at random between `l` and `r-1`
- Swap pivot to `L[l]` and proceed as usual

In [20]:
```
import random
def quicksortrand(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)

  # Choose a random postion between l and r-l as pivot, swap with L[l]
  randpivot = random.randrange(r-l)
  (L[l],L[l+randpivot]) = (L[l+randpivot],L[l])

  # Rest is same as before
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else:  # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksortrand(L,l,lower)
  quicksortrand(L,lower+1,upper)
  return(L)
```

In [21]:
```
import random
random.seed(2025)
inputlists = {}
inputlists["random"] = [random.randrange(100000000) for i in range(100000
inputlists["ascending"] = [i for i in range(15000)]
inputlists["descending"] = [i for i in range (14999,-1,-1)]
inputlists["ascendingbig"] = [i for i in range(1000000)]
inputlists["descendingbig"] = [i for i in range (999999,-1,-1)]
for k in inputlists.keys():
    tmplist = inputlists[k][:]
    starttime = time.perf_counter()
    quicksortrand(tmplist,0,len(tmplist))
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
random 2.1547414349988685
ascending 0.01822181899478892
descending 0.018493495001166593
ascendingbig 1.622358353000891
descendingbig 1.747888535996026
```