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, Fal
se),(12, True),(13, False),(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, True),(3
3, False),(34, True),(35, False),(36, True),(37, False),(38, True),(39, False),(40, True),(41, False),(42, True),(43, Fals
e),(44, True),(45, False),(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, Fal
se),(12, True),(13, False),(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, True),(3
3, False),(34, True),(35, False),(36, True),(37, False),(38, True),(39, False),(40, True),(41, False),(42, True),(43, Fals
e),(44, True),(45, False),(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 21.177326752993395

Binary search 2.5567972129938425
```

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(2023)
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 0.6429454580065794
ascending 0.6363217170001008
descending 0.6335939559940016
```

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(2023)
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 1.1455828940088395
ascending 0.00039821400423534214
descending 2.23140712500026
```

Insertion sort, recursive

In [10]:
```
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)
```

In [11]:
```
import random
random.seed(2023)
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()
    ISort(tmplist)
    elapsed = time.perf_counter() - starttime
    print(k,elapsed)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In [11], line 10
      8 tmplist = inputlists[k][:]
      9 starttime = time.perf_counter()
---> 10 ISort(tmplist)
     11 elapsed = time.perf_counter() - starttime
     12 print(k,elapsed)

Cell In [10], line 14, in ISort(L)
     12 if n < 1:
     13    return(L)
---> 14 L = Insert(ISort(L[:-1]),L[-1])
     15 return(L)

Cell In [10], line 14, in ISort(L)
     12 if n < 1:
     13    return(L)
---> 14 L = Insert(ISort(L[:-1]),L[-1])
     15 return(L)

    [... skipping similar frames: ISort at line 14 (2969 times)]

Cell In [10], line 14, in ISort(L)
     12 if n < 1:
     13    return(L)
---> 14 L = Insert(ISort(L[:-1]),L[-1])
     15 return(L)

Cell In [10], line 11, in ISort(L)
     10 def ISort(L):
---> 11    n = len(L)
     12    if n < 1:
     13       return(L)

RecursionError: maximum recursion depth exceeded while calling a Python object
```

Setup

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

In [12]:
```
import sys
sys.setrecursionlimit(2**31-1)
```

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 [13]:
```
import random
random.seed(2023)

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 5.81613647499762
ascending 0.017258646999835037
descending 10.139818099996774
```

Merge sort

In [14]:
```
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 [15]:
```
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 [16]:
```
mergesort([i for i in range(0,1000,2)]+[j for j in range (1,1000,2)])
```

```
 318,
 319,
 320,
 321,
 322,
 323,
 324,
 325,
 326,
 327,
 328,
 329,
 330,
 331,
 332,
 333,
 334,
 335,
 336,
 337
```

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

In [17]:
```
import random
random.seed(2023)
inputlists = {}
inputlists["random"] = [random.randrange(100000000) for i in range(1000000)]
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 4.532351800997276
ascending 2.5078760080068605
descending 2.5784778259985615
```

Iterative implementation of binary search

- Each recursive call halves the interval to search
- Maintain the endpoints explicitly and update with each iteration

In [18]:
```
def binarysearchiterative(v,L):
    if len(L) == 0:
        return(True)
    left = 0
    right = len(L)
    while (right - left > 0):
        mid = (left + right) // 2
        if v == L[mid]:
            return(True)
        if v < L[mid]:
            right = mid
        else:
            left = mid+1
    return(False)
```

Checking correctness

In [19]:
```
evenlist = [n for n in range(50) if n%2 == 0]
print(evenlist)
for i in range(0,60,3):
    print(i,binarysearchiterative(i,evenlist))
```

```
3 False
6 True
9 False
12 True
15 False
18 True
21 False
24 True
27 False
30 True
33 False
36 True
39 False
42 True
45 False
48 True
51 False
54 False
57 False
```

In [21]:
```
l = list(range(0,200000,2))
starttime = time.perf_counter()
for i in range(3001,23000,2):
  v = binarysearchiterative(i,l)
elapsed = time.perf_counter() - starttime
print()
print("Binary search", elapsed)
```

```
Binary search 0.047113468986935914
```