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, Fals
e),(12, True),(13, False),(14, True),(15, False),(16, True),(17, False),(18, True),(19, False),(20, True),(21, False),(22, Tr
ue),(23, False),(24, True),(25, False),(26, True),(27, False),(28, True),(29, False),(30, True),(31, False),(32, True),(33, F
alse),(34, True),(35, False),(36, True),(37, False),(38, True),(39, False),(40, True),(41, False),(42, True),(43, False),(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, Fals
e),(12, True),(13, False),(14, True),(15, False),(16, True),(17, False),(18, True),(19, False),(20, True),(21, False),(22, Tr
ue),(23, False),(24, True),(25, False),(26, True),(27, False),(28, True),(29, False),(30, True),(31, False),(32, True),(33, F
alse),(34, True),(35, False),(36, True),(37, False),(38, True),(39, False),(40, True),(41, False),(42, True),(43, False),(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 [6]:
```
l = list(range(0,100000,2))

starttime = time.perf_counter()
for i in range(3001,13000,2):
  v = naivesearch(i,l)
elapsed = time.perf_counter() - starttime
print()
print("Naive search", elapsed)

starttime = time.perf_counter()
for i in range(3001,13000,2):
  v = binarysearch(i,l)
elapsed = time.perf_counter() - starttime
print()
print("Binary search", elapsed)
```

```
Naive search 10.29776527301874

Binary search 1.4463745940010995
```

Selection sort

In [8]:
```
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 [9]:
```
import random
random.seed(2021)
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 1.7182374670228455
ascending 1.3549897029879503
descending 1.49373773302068
```

Insertion sort, iterative

In [10]:
```
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 [11]:
```
import random
random.seed(2021)
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 2.4838963110232726
ascending 0.0008862329996190965
descending 4.982716853002785
```

Insertion sort, recursive

In [12]:
```
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 [13]:
```
import random
random.seed(2021)
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)
/tmp/ipykernel_369207/2627148913.py in <module>
      8     tmplist = inputlists[k][:]
      9     starttime = time.perf_counter()
---> 10     ISort(tmplist)
     11     elapsed = time.perf_counter() - starttime
     12     print(k,elapsed)

/tmp/ipykernel_369207/2398291761.py in ISort(L)
     12     if n < 1:
     13         return(L)
---> 14     L = Insert(ISort(L[:-1]),L[-1])
     15     return(L)

... last 1 frames repeated, from the frame below ...

/tmp/ipykernel_369207/2398291761.py in ISort(L)
     12     if n < 1:
```

Setup

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

In [14]:
```
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 [15]:
```
import random
random.seed(2021)

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 14.011942709999857
ascending 0.04380735001177527
descending 23.527039036998758
```

Merge sort

In [16]:
```
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 [17]:
```
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 [18]:
```
mergesort([i for i in range(0,1000,2)]+[j for j in range (1,1000,2)])
```

```
 174,
 175,
 176,
 177,
 178,
 179,
 180,
 181,
 182,
 183,
 184,
 185,
 186,
 187,
 188,
 189,
 190,
 191,
 192,
```

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

In [19]:
```
import random
random.seed(2021)
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 10.105712931021117
ascending 5.8624570870015305
descending 5.302236225979868
```