Analysis of algorithms

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 15, 17 Oct 2023

Measuring performance

- Example of validating SIM cards against Aadhaar data
  - Naive approach takes thousands of years
  - Smarter solution takes a few minutes
- Two main resources of interest
  - Running time — how long the algorithm takes
  - Space — memory requirement
- Time depends on processing power
  - Impossible to change for given hardware
  - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
  - Easier to configure, augment
- Typically, we focus on time rather than space

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - How do we account for this?

Input size

- Running time depends on input size
  - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$
- Different inputs of size n may take different amounts of time
  - How do we account for this?

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$
- Clever algorithm: $t(n) \approx n \log_2 n$
  - $\log_2 n$ — number of times you need to divide n by 2 to reach 1
  - $\log_2(n) = k \Rightarrow n = 2^k$

Orders of magnitude

- When comparing $t(n)$, focus on orders of magnitude
  - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
  - For small values of n, $f(n) < g(n)$
  - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
  - What happens in the limit, as n becomes large
- Typical growth functions
  - Is $t(n)$ proportional to $\log n, \ldots, n^2, n^3, \ldots, 2^n$?
    - Note: $\log n$ means $\log_2 n$ by default
  - Logarithmic, polynomial, exponential, ...

Orders of magnitude

Input size	Values of $t(n)$						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7	10^{12}			
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

Measuring running time

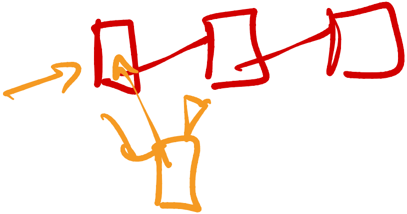
- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations

Measuring running time

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations
- Typical basic operations
  - Compare two values
  - Assign a value to a variable

Measuring running time

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations
- Typical basic operations
  - Compare two values
  - Assign a value to a variable
- Exchange a pair of values?



```
(x,y) = (y,x)        t = x
                     x = y
                     y = t
```

y = x

l1 = l2   l1 = l2[:]

  - If we ignore constants, focus on orders of magnitude, both are within a factor of 3
  - Need not be very precise about defining basic operations

What is the input size

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - Number of objects we want to rearrange
  - Number of vertices and number edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
  - Magnitude of n is not the correct measure
  - Arithmetic operations are performed digit by digit
    - Addition with carry, subtraction with borrow, multiplication, long division ...
  - Number of digits is a natural measure of input size
    - Same as $\log_b n$, when we write n in base b

Which inputs should we consider?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the "average" behaviour
  - Difficult to compute
  - Average over what? Are all inputs equally likely?
  - Need a probability distribution over inputs
- Instead, worst case input
  - Input that forces algorithm to take longest possible time
    - Search for a value that is not present in an unsorted list
    - Must scan all elements
  - Pessimistic — worst case may be rare
  - Upper bound for worst case guarantees good performance

Summary

- Two important parameters when measuring algorithm performance
  - Running time, memory requirement (space)
  - We mainly focus on time
- Running time $t(n)$ is a function of input size n
  - Interested in orders of magnitude
  - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes
- We focus on worst case inputs
  - Pessimistic, but easier to calculate than average case
  - Upper bound on worst case gives us an overall guarantee on performance

Searching in a list

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 15, 17 Oct 2023

Search problem

- Is value `v` present in list `l`?

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list

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

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list

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

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`

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

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`
- Worst case complexity is $O(n)$

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

Searching a sorted list

- What if `l` is sorted in ascending order?

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
  - If midpoint is `v`, the value is found
  - If `v` less than midpoint, search the first half
  - If `v` greater than midpoint, search the second half
  - Stop when the interval to search becomes empty

```
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:]))
```

upto m-1

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
  - If midpoint is `v`, the value is found
  - If `v` less than midpoint, search the first half
  - If `v` greater than midpoint, search the second half
  - Stop when the interval to search becomes empty
- Binary search

```
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:]))
```

Binary search

- How long does this take?

```
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:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 // 2 = 0$, so next call reaches empty interval

```
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:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 // 2 = 0$, so next call reaches empty interval
- $O(\log n)$ steps

```
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:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$

as $O(1)$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n // 2) + 1$, $n > 0$
- Solve by "unwinding"

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n // 2) + 1$, $n > 0$
- Solve by "unwinding"
- $T(n) \quad = T(n // 2) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n // 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n // 2) + 1 \\ &= (T(n // 4) + 1) + 1 \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n /\!/ 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n /\!/ 2) + 1$, $n > 0$
- Solve by "unwinding"
- $T(n) \quad = T(n /\!/ 2) + 1$
  $\quad\quad = (T(n /\!/ 4) + 1) + 1 = T(n /\!/ 2^2) + \underbrace{1 + 1}_{2}$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \mathbin{//} 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \mathbin{//} 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \mathbin{//} 2) + 1 \\ &= (T(n \mathbin{//} 4) + 1) + 1 = T(n \mathbin{//} 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n \mathbin{//} 2^k) + \underbrace{1 + \cdots + 1}_{k} \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n \,//\, 2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n // 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n // 2) + 1 \\ &= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n // 2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \\ &= (T(0) + 1) + \log n = 2 + \log n \end{aligned}$$

list takes time $O(m)$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

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

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

len(l) takes const. time

Checking l[m] takes const time

Binary search on a linked list

Checking $l[m]$ takes time $O(m)$

$T(0) = 1$

$T(n) = T(n/2) + n/2$

$T(n/4) + n/4$

$T(n/8) + n/8$

$T(n) = $

$\underbrace{n/2 + n/4 + n/8 + \cdots}$

$\Downarrow$

n

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time
- In a sorted list, we can determine that v is absent by examining just $\log n$ values!

Naïve Sorting Algorithms

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 15, 17 Oct 2023

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time

Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time
- Eventually, the new pile is sorted in descending order

Sorting a list

74	32	89	55	21	64

Sorting a list

74 32 89 55 ~~21~~ 64

21

Sorting a list

74 ~~32~~ 89 55 ~~21~~ 64

21 32

Sorting a list

74 ~~32~~ 89 ~~55~~ ~~21~~ 64

21 32 55

Sorting a list

74	~~32~~	89	~~55~~	~~21~~	~~64~~

21	32	55	64

Sorting a list

~~74~~	~~32~~	89	~~55~~	~~21~~	~~64~~

21	32	55	64	74

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ ~~64~~

21 32 55 64 74 89

Selection sort

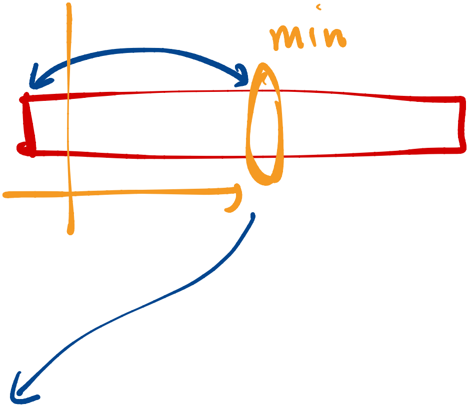
- Select the next element in sorted order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...

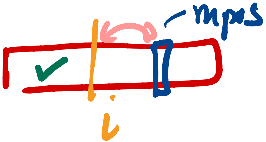


Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order

Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order




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

Analysis of selection sort

- Correctness follows from the invariant

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$

$\frac{n \cdot (n+1)}{2}$

$n + (n-1) + \cdots + 1$

$1 + 2 + \cdots + n$

$n+1 \quad n+ \quad n+1$

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

Analysis of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$
- $T(n)$ is $O(n^2)$

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

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two

Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two
- Do this for the remaining papers
  - Insert each one into correct position in the second pile

Sorting a list

74 32 89 55 21 64

Sorting a list

~~74~~ 32 89 55 21 64

74

Sorting a list

~~74~~ ~~32~~ 89 55 21 64

32 74

Sorting a list

~~74~~	~~32~~	~~89~~	55	21	64

32	74	89

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ 21 64

32 55 74 89

Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ 64

21 32 55 74 89

Sorting a list

~~74~~	~~32~~	~~89~~	~~55~~	~~21~~	~~64~~

21	32	55	64	74	89

Insertion sort

- Start building a new sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`

```
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
        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)
```

i

1 2 4 6 ~~3~~0

1 2 4 3 6

1 2 3 4 6

Analysis of iterative insertion sort

- Correctness follows from the invariant

```
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
        j = i
        while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency

```
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
        j = i
        while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times

```
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
    j = i
    while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates *n* times
  - Inner loop: *i* steps to insert `L[i]` in `L[:i]`

```
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
    j = i
    while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$

```
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
    j = i
    while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$

```
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
    j = i
    while(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)
```

Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$
- $T(n)$ is $O(n^2)$

```
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
        j = i
        while(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)
```

Summary

- Selection sort and insertion sort are intuitive sorting algorithms

Summary

- Selection sort and insertion sort are intuitive sorting algorithms
- Selection sort
  - Repeatedly find the minimum (or maximum) and append to sorted list
  - Worst case complexity is $O(n^2)$
  - Every input takes this much time
  - No advantage even if list is arranged carefully before sorting

Summary

- Selection sort and insertion sort are intuitive sorting algorithms
- Selection sort
  - Repeatedly find the minimum (or maximum) and append to sorted list
  - Worst case complexity is $O(n^2)$
  - Every input takes this much time
  - No advantage even if list is arranged carefully before sorting
- Insertion sort
  - Create a new sorted list and repeatedly insert elements into the sorted list
  - Worst case complexity is $O(n^2)$
  - Unlike selection sort, not all cases take time n^2
  - If list is already sorted, `Insert` stops in 1 step
  - Overall time can be close to $O(n)$