Selection Sort

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 2

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:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$

```
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)
```

Summary

- Selection sort is an intuitive algorithm to sort a list

Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list

Summary

- Selection sort is an intuitive algorithm to sort a list
- 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