



IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms Using Python
Professor. Madhavan Mukund
Selection Sort

So, binary search is a success story for us because it works in log in time. But to apply binary search, we first have to have the list sorted. So, now we will look at this problem of sorting a list.

(Refer Slide Time: 0:19)

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

Hand-drawn diagram: A stack of papers with an arrow pointing to the bottom paper, labeled 'Min'.

Madhavan Mukund Selection Sort

Sorting a list not only allows us to do binary search, it allows us to do a number of things better. For instance, sometimes we want to find the median value, we want to find that value such that half the values in the list are bigger and half are smaller. So, if I have managed to sort the list, the median will automatically be the middle value. So, by just probing the middle value in the list, I know that this is the median.

Another possible problem is to find out whether this list has got unique values or whether there are duplicates, like are there two SIM cards, which have been registered with the same Aadhaar number. Now if I can sort the thing by Aadhaar number, then if I find duplicates, they must be next to each other because all the values are sorted, two identical values will appear adjacent in the sorted list.

So, if I go through the sorted list, and if I find, if I do not find any value, which is the same as the previous of the next value, I know that all the values are unique. If I, if there are duplicates, I will find them. And more general than this duplicate problem is if there are multiple values and many

things like for example, these are marks in an exam or something, if I sought my marks, and I want to know how many students got 44, and only students got 72, and how many silver 99.

In this sorted list, all the blocks will come together all the 40 fours will come as a block all the 99 will come as a block and so on so I can build a frequency table. So, sorting is a general first step, which makes many subsequent steps easier to solve. So, it is a very important and very fundamental question in computing, how to sort a list efficiently and effectively.

So, that is the question we are going to address in the remaining lectures in this week. How do we sort a list and what is the complexity of sorting each of these different ways of sorting. So, we are going to start with two very intuitive ways of sorting, which you would typically do when you are asked to do something by hand.

So, let us look at a very practical problem, which you may be asked to do by hand, supposing are the teaching assistant for a course, and the instructor has graded the exams. So the instructor is graded the exams. But of course, the exams are graded in the order in which they got, came to the instructor could be in roll number order; it could be the order in which the person physically handed in the exam paper as they left the hall.

So, there is no particular logic to or correlation between the order in which the exams are and the marks that the students have. So, now what the instructor would like to do is assign grades. So, you need to sort the papers. So, you need to get the papers in ascending or descending order. So, let us say you want the highest from the top, so let us say descending order.

So, now, as the TA it is your job to do this. So, you are given this huge stack of exam papers in areas to sort it in descending order with the highest marks on the top. So, how would you go about it? So, here is one natural strategy, which we do use quite often, when we are looking at quantities like this and trying to establish this, we will scan the entire list.

And then as we are going along, we all know how to keep track of the maximum and the minimum in a list, you keep the first value as your hypothetical minimum or maximum every time you see a smaller or larger number, you will update the minimum or the maximum as the case may be.

So, by doing one scan of the list, you can find the maximum value. Let us assume for simplicity that all these values are distinct, there is only one mark, one paper with each mark, it does not really matter, because if you find it again, you will find it again. So, it is not a problem. But let us assume that there is only one, one paper for each value of mark.

So, you find the maximum and then what you do, you take it and you move it aside, or you find the minimum and you move it aside depending on which so your problem here is you are trying to find the final list in order. So, it is better to put the first the minimum at the bottom. So, you have this whole pile of papers. So, you find the minimum one, and then you move it from here to the bottom.

(Refer Slide Time: 4:17)

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

Madhavan Mukund Selection Sort

Now you do the same thing again. So, you take that pile. And then you again, find the minimum and put it on top of the first paper, you want to find the minimum put on top the second paper. So, in this way in the second list is growing from minimum to maximum, because you are finding the minimum at each time and then appending it to this list. So, this is a simple strategy where eventually the second pile is going to be in order from bottom to top from minimum to maximum, which is what your instructor asked you to do.

(Refer Slide Time: 4:43)

Sorting a list



74 32 89 55 21 64

Bottom Top



Sorting a list



74 32 89 55 21 64

21



सिद्धिर्भवति कर्मजा

Sorting a list



74 32 89 55 21 64

21 32

Navigation icons

Madhavan Mukund

Selection Sort

Sorting a list



74 32 89 55 21 64

21 32 55 64 74 89

Bottom

Navigation icons

Madhavan Mukund

Selection Sort


So, let us say that this is the hypothetically, this is our top and this is our bottom. So, it is easier to do left to right. So, this is my initial set of marks, six marks sorted like this. So, the first, the lowest mark is sorry. I think they should take it. Yes, bottom to top, the lowest mark is 74 and the highest mark is 6, I say the top most paper is 64 mark.

So, first I scan through this whole thing and I find the minimum. And the minimum in this case is 21. So, I take this 21 out, and I move it to a new list. Now I have five papers remaining. So, again, I look through the whole thing, and I find the second minimum, the minimum amount, what remains and that is this 32. So, 32 is my next candidate.

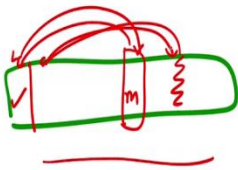
So, I will take this 32 and move it aside, then I will find this 55 and move it aside and so on. So, 55 then 64 then 74 and then 89. So, this is the algorithm in work. So now at this, again, this is my top, and this is my bottom.

(Refer Slide Time: 5:50)


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

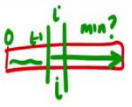


```

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)
          
```

Selection sort





So, this is called selection sort, because we select the next item in sorted order and move it into the correct place, move it into the correct business cases, just append it, we do not have to do any work on the second pile, the second pile is growing naturally in sorted order. We are just adding it to that.

Now, for many reasons, we would like to avoid using a second list. So, we would like to ideally use the space that we have in the list. So, supposing we have a large list, we do not want to duplicate it. So, we want to make this same algorithm work within the same list that we have. So, a strategy for that is that you take the list and now you go through it.

And so you take this list, and you go through it. And perhaps you find the minimum here. So, what you now do is you move this to the beginning, and you move this here, so we just exchange the values here and now the minimum is at the first position. Now I go through this whole list, and maybe I will find the second minimum at this position.

So, now I move the second minimum here and move this year. So, this makes it possible for me to do this strategy without creating a new list. Because remember, in the previous example as I move things, I cross them out, so they were no longer there. So, here the crossing out in this thing corresponds to saying that they are moved into position and now their position has been taken by someplace and not value, which I have not seen before.

So, eventually this list will be rearranged. If I keep moving the minimum to the beginning of each segment, I will rearrange it in ascending order. So, here is a Python implementation of this selection sort. So, what you do is you first compute the length of the list. So, if the list has is empty, if the length is 0, and is less than one, then I just return the list as it is, I do not have to do any work.

Now otherwise, I am going to build up this segment, so at any given point, I am going to be looking at some index i so I am looking at $L[i]$ and I am going to scan from here onwards and look for the minimum and then move it to $L[i]$. So, I am not going to touch the part on the left. So, basically 0 to $i - 1$, this part is already sorted.

So, we are going to assume that the slice or the prefix of the list up to $i - 1$ index is sorted. So, when I am at 0 this means that nothing is sorted because there is no $i - 1$. So, now, in order to find the minimum, I use our usual algorithm, I assume that the current position is my minimum and I keep looking at all the values from the next position onwards, and if I find a value which is smaller than the minimum that I assumed, then I replace my minimum position from i to j .

So, this just computes j as the minimum, I mean, it scans through all j and keeps updating $mpos$, to be the minimum position, the value with position with a minimum value from i onwards. Once you have done this scan, now what you want to do is swap that thing, so that is what just, swap that to the current beginning.


So, we take a L of i , which was where we started the scan, and exchange it with the minimum position. Now if L of i was already the minimum position, it just gets exchanged with itself, which is no use. But if we did find a smaller one further to the right, then we would bring it here. So, what have we achieved? We have achieved that now I have moved this boundary from i , i minus 1 to i .

So, earlier before I did this pass, 0 to i minus 1 was sorted. Now I found the correct thing to put in position i so that 0 to i is sorted and I keep doing this from for i going from 0 to n minus 1. So, eventually, every time I move, one more extension of the sorted segment happens and the whole thing becomes sorted.

So, it is important to kind of think of these algorithms in this way because unless you do this, it is not going to be possible to really convince yourself you have handled it properly. So, these are what are called invariance.

(Refer Slide Time: 9:58)

Selection sort

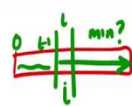



- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
 - Swap the minimum element into the first position
 - Swap the second minimum element into the second position
 - ...
- Eventually the list is rearranged in place in ascending order

```

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)

```





Madhavan Mukund
Selection Sort

So, this is what is called an invariant. So, at the beginning of the loop, the invariant that I have is up to i sorted, that is 0 to i minus 1 in terms of indices and what the loop does is it extends the

invariant to one more step. So, earlier I had L colon I was sorted now, L column i plus 1 is sorted and then you have to just verify that what happens here is a valid calculation to ensure that we make this progress from L 0 to i to L 0 to i plus 1.

(Refer Slide Time: 10:38)

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)
```

Madhavan Mukund

Selection Sort

Analysis of selection sort

■ Correctness follows from the invariant

■ Efficiency

- Outer loop iterates n times
- Inner loop: $n - i$ steps to find minimum in $L[i:]$
- $T(n) = n + (n-1) + \dots + 1$

$\text{total} = \frac{n(n+1)}{2}$

$n + (n-1) + \dots + 2 + 1$

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

$n \cdot (n+1) = 2 \cdot \text{total}$

```
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)
```

Madhavan Mukund

Selection Sort

So, the correction, correctness of the algorithm follows from the invariant but correctness is an important step that we should we should never ignore, because it is totally pointless to have an efficient algorithm which is wrong. So, ultimately, when you design an algorithm, the first thing you have to make sure is doing the job that is supposed to do, then you can worry about its complexity.

So, establishing correctness is cannot be dismissed, you must show that you are always going to work. So, here I claim that these comments you have written, you have to will, so I have kind of informally convinced you, but you have to kind of formulate convince yourself that this invariant holds the beginning, each iteration will actually update the invariant as I claim and therefore, at the end of the loop, the invariant will establish the property I want, which is the entire slice from 0 to n minus one is actually sorted. So, that is the first thing.

Next, we have to worry about the efficiency. So, how long does it take? Well, here, we can use this, because there is no recursion and all that we can just look at the loop structure. So, we can see that there is an outer loop which works, which takes n steps and inside this outer loop we have this inner loop.

And how much time does this inner loop take when it goes from i to n minus 1. So, it is going to take n minus i steps to find this minimum. So, therefore, for each of the outer iterations, I am going to take n minus i steps, and i is going to keep changing. So, I am going to take initially, I am going to take go through on any elements to find the minimum, absolute minimum and bring it to the first position, then I am going to go through n minus 1 elements, find the minimum, bring it to the second position, and so on.

So, the total time algorithm is going to take is n plus n minus 1 plus up to 1. So, you should probably know what this adds up to. But in case you do not, here is a simple way to do it. So, have n plus n minus 1 plus 2 plus 1. So, what does this add up to? So, this is a trick which is attributed to Gauss as a school child. So, he said, let me write it in reverse the same sum, let me write it again.

So, I have just written it out from right to left. But now if I add it, column by column, then this column adds up n plus 1, this column adds up to n plus 1. So, every column adds up to n plus 1. And how many columns are there? Clearly, there are n because n to 1. So, there are n columns, each of it adds up to n plus 1.

But the total sum is n into n plus 1, but that is 2 times this because I have added the list once forwards and ones backwards. So, this is 2 times the total amount. So therefore, my total is going to be n into n plus 1 by 2. So, if you do not remember this formula, this is one easy way to reconstruct it.

(Refer Slide Time: 13:47)

Analysis of selection sort

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

Handwritten notes: $n^2 + n$ with a green checkmark and a red 'X' over the n term.

Madhavan Mukund Selection Sort

So, T of n , if n if it is a summation from 1 to n is just n into n plus 1 by 2, it is the number of elements times the number of elements plus 1 by 2. And as we have seen, this is big O of n squared. So, all we need is the highest term. So, this is n squared by 2 plus n by 2. So basically, in our asymptotic, order of magnitude way of doing things, we throw away this, we throw away this and we say this is order of n squared. So, selection sort is actually an order n squared algorithm.

(Refer Slide Time: 14:21)

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

Madhavan Mukund Selection Sort

So, it is an intuitive algorithm to sort a list because we do it all the time actually, when we do it with small values, when we are asked to find them. Whenever somebody gives you a kind of set of things and asks you to arrange it, this is very often the kind of thing that you might want to do. So, we just repeatedly find the minimum or the maximum and append it to the sorted list.

Now, one of the features of this algorithm is that not only is the worst case big $O(n^2)$, but actually every case is big $O(n^2)$ and that is because if you look at this loop, so this loop will scan the entries from $i + 1$ to n regardless of what the status of those entries is, it is not really influenced by whether those entries are already in sorted order or not, it is just going to scan every one of them and try to update the minimum.

So, it has no performance improvement on whether this list was already sorted, not sorted in a particular order, it is as good or as bad whether you had a completely random order or whether you had some nice order to start with. So, this is actually some case where the worst case is really every case. So, selection sort is really big $O(n^2)$ in every case, so there is no advantage, even if the list is arranged carefully before sorting.

