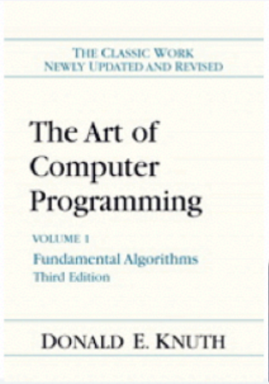
Introduction
CISC4080: Computer Algorithms
CIS, Fordham Univ.

Instructor: X. Zhang

What is an algorithm?

“ An algorithm is a finite, definite, effective procedure, with some input and some output. ”

— Donald Knuth



- All algorithms solve some well-defined problem
- **Problem**: specified by the input, and what’s the desired output
  - e.g., what’s the input, desired output of sorting problem?
  - An algorithm is correct if it always generates desired output
- Algorithm is a term coined to honor **Muḥammad ibn Mūsā al-Khwārizmī (**Arabized as al-Khwarizmi and formerly Latinized as *Algorithmi),* 9th century Persian polymath who laid out basic methods for
  - adding, multiplying and dividing **decimal numbers**
  - extracting square roots
  - calculating digits of pi

Algorithms that you've seen

- Linear **Search**: search for an item with a matching key in an array (unsorted)
- Binary **Search**

- Bubble Sort, Insertion Sort, Selection Sort
- Search in a binary search tree: algorithm + data structure

- Graph algorithms: traversal, shortest path, …

Case study: Linear Search

- What’s the problem? i.e., what’s the input and the desired output?
    - **Input:**
    - **Output**:

Case study: Linear Search

- What's the problem? i.e., what's the input and the desired output?
  - **Input:** a list of elements, L[0…n-1], and a value t
    - L[0…n-1] refers to a ADT list made up of n elements: L[0], L[1], … L[n-1]
    - assumption: t is of same type as L[i]
  - **Output**:
    - if t appears in L, returns the index of its first appearance
    - if t does not appear in L, returns -1

C++ code

```
/* linear search
  @param: L an array of integer
  @param: n size of array (or vector L)
  @param t: the target value to search for in L
  @pre-condition: ??
  @post-condition: if t appears in L, returns the index of its first appearance
          • if t does not appear in L, returns -1
*/
int LinearSearch (int L[ ], int n, int t)
{
   int loc=-1;
    // iterates through list, compare each element with t,
   for (int i=0;i<=n-1; i++)
        if (L[i]==t)
            loc=i;

    return loc;
}
```

Question 1: Is this correct? i.e., does it return the desired output?

Question 2: How many comparison operations is carried out?

Pseudocode

Pseudocode: language-neutral, data type neural

```
/* linear search
  @param: L a list of elements of size n
  @param t: the target value to search for in L
  @pre-condition: ??
  @post-condition: ???
*/
LinearSearch (L[0…n-1], t) //omit the return type
{
    // iterates through list, compare each element with t
    for i=0 to n-1  //default increment is 1
          if L[i]==t    //omit () around condition
               return i    //when matches, return right away!

   return -1
}
```

Usually not a good idea to use a break or return inside a loop

Why study algorithms?

- Computers system: circuit layout, scheduling algorithms for OS or data center, compiler (code optimization), ...
- Internet: web search, packet routing… => graph algorithms
- Security:
  - encryption algorithms (such as RSA public/private key algorithms, among many algorithms)
  - cryptographic hash function: generate checksum to verify the authenticity of data
- Multimedia, Computer Graphics
  - Compression algorithms used in MP3, JPEG technology
  - One component in MP3 is Huffman method (a greedy algorithm)

Why study algorithms?

- Biology: Bioinformatics combines mathematics, statistics and computer science to study biological molecules, such as DNA, RNA, and protein structures…
  - Edit distance (6.3) for suggesting correction in spell checker ==> sequence alignment (DNA, RNA, protein)
- Social networks analysis
  - Community detection
  - Link prediction: predict whether there will be links between two nodes ==> Graph problem
- Algorithms in ML and AI
  - Traditional AI: A* search, Maze Solving
  - K-mean clustering algorithms, linear regression
  - Reinforcement Learning: Bellman algorithm, Q-learning algorithm…

Introduction to algorithm analysis

- Consider calculation of Fibonacci sequence, in particular, the n-th number in sequence:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Leonardo Fibonacci (c. 1170 – c. 1250)

Fibonacci helped the spread of the decimal system in Europe, primarily through the publication in the early 13th century of his Book of Calculation, the Liber Abaci. (Source: Wikipedia)

Fibonacci Sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Formally,

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

- Fibonacci Seq. Calculation Problem:
    - Input: an natural number n
    - Output: value of n-th term of Fibonacci sequence
- Can you write a recursive algorithm to solve this problem?

A recursive algorithm

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

```
Algorithm: Fib (n)
{
    //base cases: when do we know the answers right away?
    // n is 0 or 1


    //general case: when we need to use previous terms to calculate current one?




}
```

Discussion: Three questions:

- Is it correct?
- How long does it take to calculate 5th term, 50th term?
- Can we do better? (faster?)

How long does it take…

- you to finish a homework assignment?
  - Right before I started to work on the homework, I checked the clock, and it’s Wednesday, Sept 9, 10:15:00 am
  - I then worked on the homework (without taking any break).
  - Right after I finished it, I checked the clock again and it’s Wednesday, Sept 9, 1:20:00pm
  - How long did it take me to finish the homework?
- Let’s see how to apply same idea to measure algorithm running time!

System Time

- All computer systems maintain a system time/clock.
- It keeps track number of seconds and nanoseconds (10^{-9} second) passed since some starting time, called the *epoch*.
    - Unix and POSIX-compliant systems use Jan 1st, 1970 00:00:00 as the epoch
- System clock tells you how many seconds and nanoseconds has passed since the epoch

clock_gettime()

A library function, **clock_gettime** retrieves the time of a system clock

Usage example

```
#include <time.h>
/* where the following type is defined
  timespec type:
  struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};  */

struct timespec t; // declare a variable t of type timespec

clock_gettime(CLOCK_REALTIME, &t);
// Retrieve the time of specified clock, CLOCK_REALTIME
```

Measuring running time

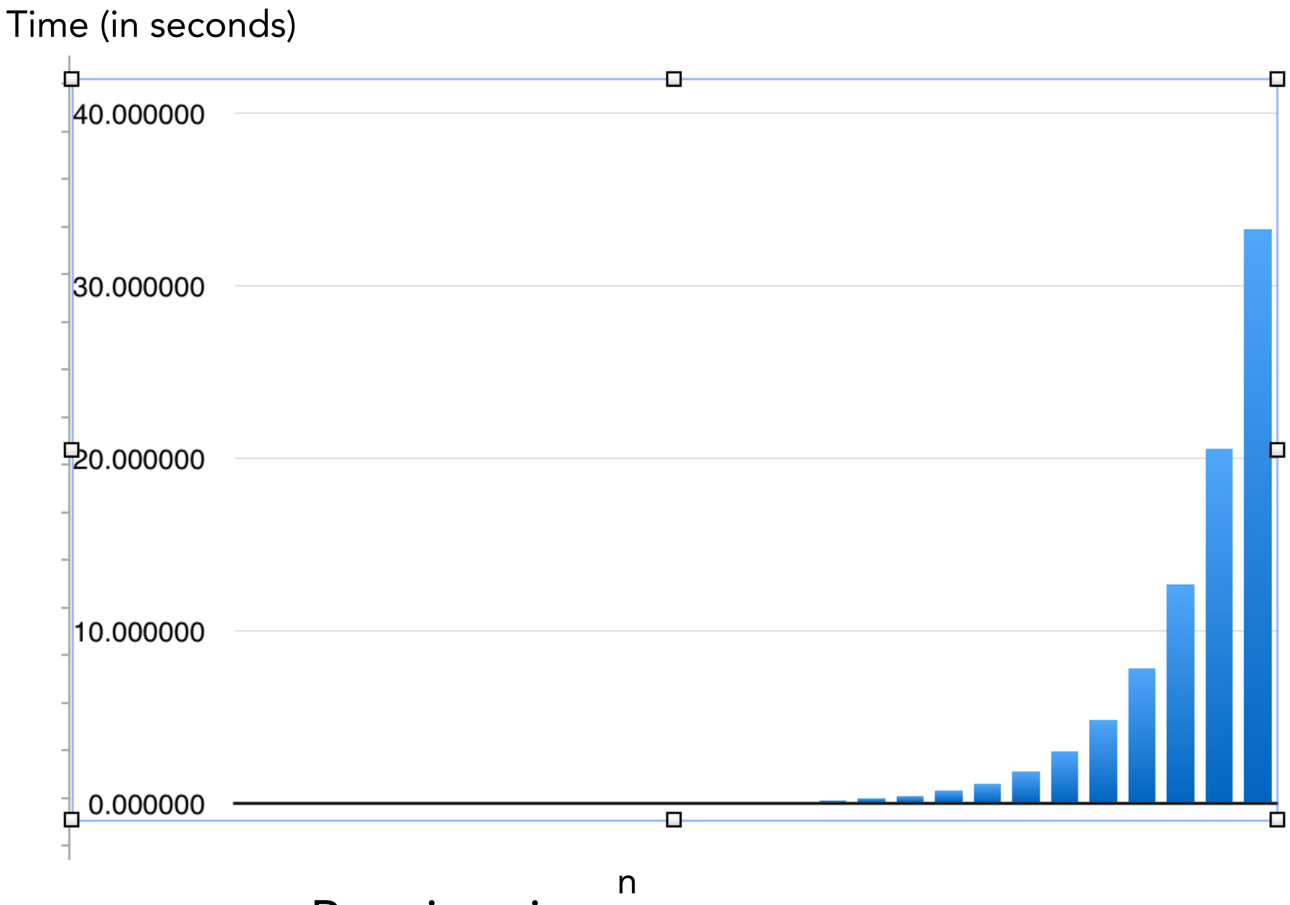
```
#include <time.h>
 …
struct timespec t1, t2; //t1, t2 are two variables of type timeval
 …
clock_gettime(CLOCK_REALTIME, &t1);
```
result = Fib(n); // or any other algorithm we want to measure running time of …
```
clock_gettime (CLOCK_REALTIME, &t2);

// calculate how many seconds have passed between t1 and t2
double timeInSeconds = (t2.tv_sec-t1.tv_sec) +
      (t2.tv_nsec-t2.tv_nsec)/1e9;
```

* Live demo

* Question: How the running time of Fib() grows when with n?

Example (Fib1: recursive)

```
Recursive Fib calculator
n   F(n)      T(n)ofFib1 (s)
1   1   2.72e-07
2   1   1.67e-07
3   2   2.49e-07
4   3   3e-07
5   5   3.95e-07
6   8   3.98e-07
7   13  5.46e-07
8   21  7.25e-07
…
13  233 4.256e-06
14  377 6.848e-06
15  610 1.0964e-05
16  987 1.7656e-05
17  1597   2.8207e-05
18  2584   4.5365e-05
19  4181   7.3375e-05
…
33  3524578    0.0195626
34  5702887    0.0318968
35  9227465    0.0516566
36  14930352   0.0839258
…
40  102334155  0.605248
41  165580141  0.932493
42  267914296  1.51373
43  433494437  2.40282
44  701408733  3.89735
45  1134903170 6.31143
46  1836311903 10.2351
47  -1323752223   16.5711
48  512559680  26.8665
49  -811192543 43.6285
```



Running time seems to grows exponentially as n increases

How long does it take to Calculate F(100)?

Why?

- Draw recursive function call tree for fib1(5)
  - Observation: wasteful repeated calculation

```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n - 1) + fib1(n - 2)
```

Can we do better?

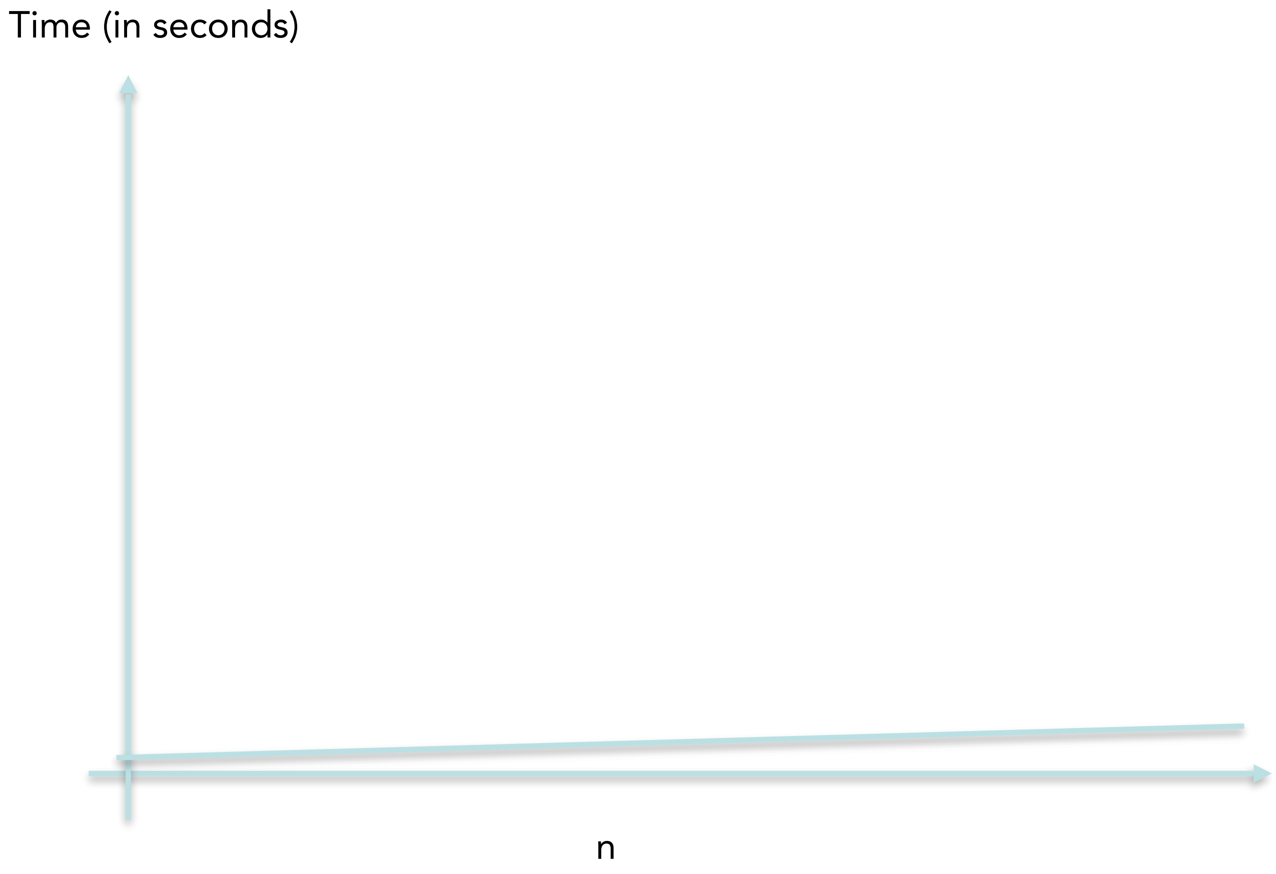
```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n - 1) + fib1(n - 2)
```

- Idea: Store solutions to subproblems in array (key of Dynamic Programming)

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

Example (Fib2: iterative)

```
Iterative Fib calculator
n   F(n)      T(n)ofFib2 (s)
1   1   2.8e-07
2   1   6.5e-08
3   2   1.76e-07
4   3   2.06e-07
5   5   1.92e-07
6   8   2.18e-07
7   13  1.88e-07
8   21  2.14e-07
9   34  1.67e-07
10  55  2.11e-07
11  89  2.92e-07
12  144 1.61e-07
13  233 1.71e-07
14  377 2.41e-07
15  610 2.71e-07
…
46  18363119033.78e-07
47  -1323752223   3.69e-07
48  512559680 3.9e-07
49  -8111925433.96e-07
50  -2986328634.11e-07
51  -1109825406   4.54e-07
52  -1408458269   4.43e-07
53  17766836214.37e-07
54  368225352 4.43e-07
55  21449089734.31e-07
56  -1781832971   4.38e-07
57  363076002 5.32e-07
58  -1418756969   4.62e-07
59  -1055680967   4.63e-07
60  18205293604.77e-07
```



Increase very slowly as n increases

Analytic approach

- Is it possible to find out how running time grows when input size grows, **analytically?**
    - Does running time stay constant, increase linearly, logarithmically, quadratically, … exponentially?
- Yes: analyze pseudocode/code, calculate total number of steps in terms of input size, and study its order of growth
    - results are general: not specific to language, run time system, caching effect, other processes sharing computer
    - shed light on effects of larger problem size, faster CPU, …

Course Overview (1)

- Today: overview, introduction
- Sorting, Recursion, Data Structure review
  - vector, queue, stack, heap (priority queue), hash table
- Algorithm analysis (running time and space), Big-O notations
  - How does running time grows when problem size grows?
    - constant, linear, quadratic, cubic, … exponential?
- Divide-and-conquer paradigm, master theorem
  - n logn sorting algorithms
  - solving problem divide-and-conquer way

Course Overview (2)

- Backtracking and Recursion
  - How to enumerate all subsets?
  - How to solve maze?
- Graph Algorithms (or, for relational data)
  - Review BFS, DFS traversal
  - Dijkstra algorithm
  - Minimum Spanning Tree algorithms
  - Travel Salesman Problem, and more
- Dynamic Programming
  - optimized recursive algorithm where we use table to remember subproblems solutions…

Summary

- Basic concepts: algorithm, problem, input, output, input instance, correctness
- Case study: selection sort
    - Pseudocode convention
- Tracing an algorithm
- Correctness and efficiency
    - Importance of efficient algorithm: Fibonacci example
- Roadmap of the course

Case study: Selection Sort

- Idea: select smallest element and swap it to first entry, and then select second smallest element and swap it with second entry, and continue until the list is sorted

Index	0	1	2	3	4	5
L[0…5]	4	3	1	7	6	2

Case study: Selection Sort

Index	0	1	2	3	4	5	
L[0…5]	4	3	1	7	6	2	Find smallest element, L[2]

Case study: Selection Sort

Index	0	1	2	3	4	5	
L[0…5]	4	3	1	7	6	2	
Passes	1	3	4	7	6	2	Swap L[2] with L[0]

Case study: Selection Sort

Index	0	1	2	3	4	5	
L[0…5]	**4**	**3**	**1**	**7**	**6**	**2**	
	1	**3**	**4**	**7**	**6**	**2**	L[0] now stores smallest element No need to work on it any more

Case study: Selection Sort

Index	0	1	2	3	4	5	
L[0…5]	4	3	1	7	6	2	
	1	3	4	7	6	2	Find second smallest element, L[5], or smallest in L[1…5]

Case study: Selection Sort

Index	0	1	2	3	4	5
L[0…5]	4	3	1	7	6	2
	1	3	4	7	6	2
	1	2	4	7	6	3

Swap L[5] with L[1],
L[0], L[1] are done!

Case study: Selection Sort

Index	0	1	2	3	4	5
L[0…5]	4	3	1	7	6	2
	1	3	4	7	6	2
	1	2	4	7	6	3
	1	2	3	7	6	4
	1	2	3	4	6	7
	1	2	3	4	6	7

Continue until we have one
Element left!
List is sorted…

Selection Sort: n=6

Index	0	1	2	3	4	5
L[0…5]	4	3	1	7	6	2
	1	3	4	7	6	2
	1	2	4	7	6	3
	1	2	3	7	6	4
	1	2	3	4	6	7
	1	2	3	4	6	7

Question and observations:

1. **A pass:** is the operations of finding smallest element in remaining list, and swap it to front. How many passes have we carried out here?

2. How many swap operations are carried out by selection sort on this list?

SelectionSort (L[0…n-1])

```
{
    for s=0 to n-1 //pass #
        // find index of smallest
        // element in L[s…n-1]



        //swap smallest element
        // with L[s] if it's not at front
        if (minIndex!=s)
            swap (L[s], L[minIndex]);

}
```

Index	0	1	2	3	4	5
L[ ]	4	3	1	7	6	2
	1	3	4	7	6	2
	1	2	4	7	6	3
	1	2	3	7	6	4
	1	2	3	4	6	7
	1	2	3	4	6	7

Selection Sort (L[0…n-1])

```
{
  for s=0 to n-2 {
      // find index of smallest element in L[s…n-1]
     minIndex = s;
     for k=s+1 to n-1
         if L[k]<L[minIndex]: minIndex=k

     //swap it with first element in the sublist
     if (minIndex!=s)
          swap (L[s], L[minIndex]);
   }
}
```

Tracing: with input a[0…5]={4,3,1,7,6}, n=6

```
SelectionSort (L[0…n-1])
{
    for s=0 to n-2 {
     // find index of smallest element in L[s…n-
      minIndex = s;
      for k=s+1 to n-1
           if L[k]<L[minIndex]
                minIndex=k

     //swap it with first element in the sublist
     if (minIndex!=s)
          swap (L[s], L[minIndex]);
  }
}
```

S	minIndex	0	1	2	3	4	5
Before		**4**	**3**	**1**	**7**	**6**	**2**
0	2	**4**	**3**	**1**	**7**	**6**	**2**
1	5	1	**3**	**4**	**7**	**6**	**2**
2	5	1	2	**4**	**7**	**6**	**3**
3	5	1	2	3	**7**	**6**	**4**
4	4	1	2	3	4	**6**	**7**
5	5	1	2	3	4	6	**7**

sorted part (colored in gray, L[0…s-1]) grows

unsorted part (colored in black, L[s…n-1]) shrinks

Selection Sort (L[0…n-1]) on Input a[0…5]={3,4,1,2,5,6}, n=6

```
{
  for s=0 to n-2 {
      // find index of smallest element in L[s…n-1]
      minIndex = s;
      for k=s+1 to n-1
          if L[k]<L[minIndex]    //Comparison operations
              minIndex=k

      //swap it with first element in the sublist
      if (minIndex!=s)
          swap (L[s], L[minIndex]);
  }
}
```

Outer loop control s	Inner loop header	Comp op#	minIndex after inner lop	L[] after swap
0	for k=1 to 5	5	2	{1,4,3,2,5,6}
1	for k=2 to 5	4		
2	For k=3 to 5	3		
3	For k=4 to 5	2		
4	For k=5 to 5	1		

Is it correct?

- Testing with a few test cases
- How to reason about correctness of algorithm (i.e., for all inputs, it generates correct outputs)?
  - One extreme: Your program should be like a proof.
  - At least document your algorithm/code.

```
SelectionSort (L[0…n-1])
{
    for s=0 to n-2 {
        // find index of smallest element in L[s…n-1] <= We think this is self-evident!
        minIndex = s;
        for k=s+1 to n-1
            if L[k]<L[minIndex]
                minIndex=k

        //swap it with first element in the sublist
        if (minIndex!=s)
          swap (L[s], L[minIndex]);
       //at this point: L[s] stores (0+1)-th smallest element in the list
     }
  }
```

Is it efficient?

- How much resources does it take … to sort a list of n elements?
  - Especially when n is very very large.
  - time (CPU resource) and space (main memory) ==> how many operations/statements are executed, how much variables are used?

```
SelectionSort (L[0…n-1])
{
    for s=0 to n-2 {
        minIndex = s;
        for k=s+1 to n-1
            if L[k]<L[minIndex]
                minIndex=k

        //swap it with first element in the sublist
        if (minIndex!=s)
           swap (L[s], L[minIndex]);

    }
}
```

Selection Sort recursively

- To sort list L[0…n-1] into ascending order
  - Start by selecting smallest element in list L[0…n-1], and swap it with L[0]
  - Now **we only need to sort L[1…n-1]**
    - Do it recursively, i.e., by calling the function itself, but with smaller part of the list, L[1…n-1]
- **In general, when solving a problem recursively, add additional parameters to function**
  - allow passing starting and ending indices => to specify a smaller instance of the problem

SelectionSort (L[], left, right)

```
// arrange L[left… right] into ascending order
// precondition: left<=right
{


    // find index of smallest element in L[left…right]
     minIndex = left;
     for k=left+1 to right
          if L[k]<L[minIndex]
               minIndex=k

    //swap it with first element in the sublist
    if (minIndex!=left)
         swap (L[left], L[minIndex]);

    SelectionSort (L, left+1, right)     //recursive call to sort the rest of the list
  }
```

Discussion:

What's the base case?

What's the recursive case?

What is the three-questions rule?

SelectionSort (L[], left, right)

```
// arrange L[left… right] into ascending order
// left<=right
{
     if (left==right)
            return;

     // find index of smallest element in L[left…right]
       minIndex = left;
       for k=left+1 to right
              if L[k]<L[minIndex]
                   minIndex=k

      //swap it with first element in the sublist
      if (minIndex!=left)
           swap (L[left], L[minIndex]);

      SelectionSort (L, left+1, right)     //recursive call to sort the rest of the list
   }
```

Discussion:

What's the base case?

What's the recursive case?

What is the three-questions rule?

Could you trace through its execution? e.g., input L[0…5]={3,4,1,2,5,6}, left=0, right=5?