Complexity

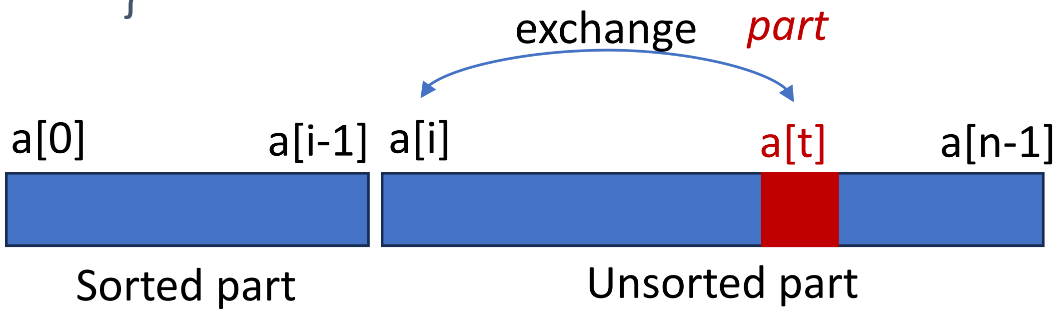
Ch 1

Two algorithms. Which one has better performance?

- **Selection sort**

```
for (i = 0; i < n-1; i++)
  {  /*Find the minimum element in a[i:n-1]*/
     int t = i;
     for (j = i+1; j < n; j++)
        if (a[j] < a[t])
          t = j;
     /* Swap the minimum element with the a[i]*/
     if(t != i)
        swap(&a[t], &a[i]);
  }
```

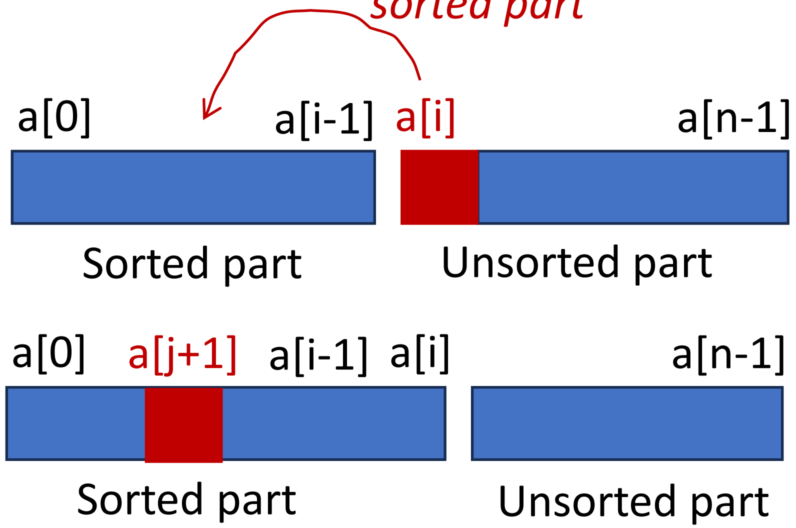
Select the minimum element in unsorted part



- **Insertion sort**

```
for (i = 1; i < n; i++)
{/* insert a[i] into a[0:i-1] */
   int t = a[i];
   int j;
   for (j = i - 1; j >= 0 && t < a[j]; j--)
      a[j + 1] = a[j];
   a[j + 1] = t;
}
```

Insert a[i] into the sorted part



Performance analysis

Obtaining estimates of time and space that are machine *independent*.

- Space complexity:
  The amount of memory that it needs to run to completion.

- Time complexity:
  The amount of computer time that it needs to run to completion.

Space complexity

$$S(P) = c + S_p(I)$$

Total space requirement

= Fixed space requirement + Variable space requirements

Do **not** depend on the number and size of the program *P*'s input and output.

Depends on the number and size of input and output associated with the instance *I*.

Example 1

```
float abc(float a, float b, float c)
{
    return a+b+c*c+(a-b+c)/c+5.00;
}
```

This function has only fixed space requirements.

The variable space requirement $S_{abc}(I) = 0$.

Example 2

In C, the address of the first element is passed.

```
float sum(float list[], int n)
{
    float tempsum=0;
    int i;
    for (i = 0; i < n; i++)
        tempsum += list[i];
    return tempsum;
}
```

C passes all parameters by value. In this case, C does not copy the array list. The variable space requirement $S_{sum}(I) = 0$.

Example 3

```
float rsum(float list[], int n)
{
    if (n) return rsum(list,n-1)+list[n-1];
    return 0;
}
```

One recursive call requires K bytes for 2 parameters and the return address.

If the initial length of list[] is N,

the variable space requirement $S_{rsum}(N) = N*K$ bytes.

Exercise: factorial function *n!*

A function *f* to compute the factorial of a number *n*

$$f(n) = n * f(n-1) \quad \text{for } n > 1$$
$$f(0) = 1 \quad \text{for } n <= 1$$

Given that $n=N$, the size of the input parameter$=K$, and the size of return address$=M$.

- Q1: What is the variable space requirement for **iterative** factorial function?
- Q2: What is the variable space requirement for **recursive** factorial function?

Please reply your answers of Q1 and Q2 via the following link:

Group members: 1~3 people

Exercise: factorial function *n!*

- **Iterative**

```
double iterFact(int n)
{   int i;
    double answer;
    if ((n == 0) || (n == 1)) return 1.0;
    answer = 1.0;
    for (i = n; i > 1; i--)
       answer *= i;
    return answer;
}
```

- **Recursive**

```
double recurFact(int n)
{
   if ((n==0) || (n==1))  return 1.0;
   return n*recurFact(n-1);
}
```

Time complexity

Total time requirement $T(P)$

= Compile time + Execution time

- Use system clock to time the program

  or

- Count the number of operations

Do **not** depend on the instance characteristics.

Depends on the characteristics of instance I.

Count program step (1)

- <u>Program step</u>: A segment with execution time **independent** from the instance characteristics

```
float sum(float list[], int n)
{
    float tempsum=0;
    int i;
    for (i = 0; i < n; i++)
        tempsum += list[i];

    return tempsum;
}
```

step ++; for assignment

For each iteration

step ++; the for loop (i=0~n-1)

step ++; for assignment

step ++; last execution of for (i=n)

step ++; to return value

Total number of steps: 2*n+3

Count program step (2)

- <u>Program step</u>: A segment with execution time **independent** from the instance characteristics

```
float rsum(float list[], int n)
{
    if (n)
        return rsum(list,n-1)+list[n-1];

    return 0;
}
```

For n=0

step ++; for if condition

step ++; for return

Number of steps: 2

For n>0

step ++; for if condition

step ++; for return

Number of steps: 2n

Total number of steps: 2*n+2

For given parameters, computing time may be different.

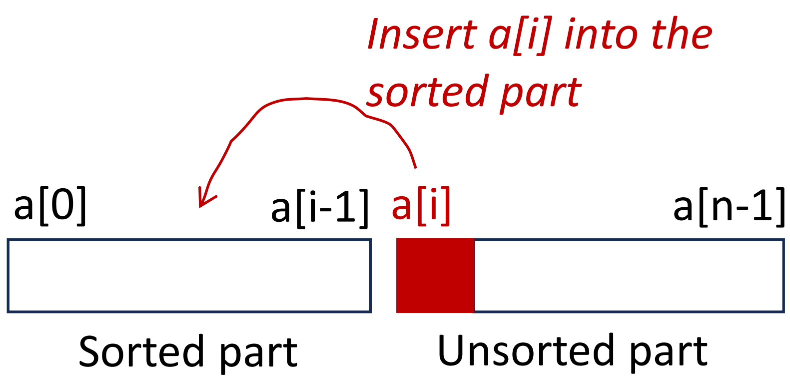
- **Best-case count**
  Minimum number of steps that can be executed.

- **Worst-case count**
  Maximum number of steps that can be executed.

- **Average count**
  Average number of steps executed.

Worst-case comparison count

Insertion sort



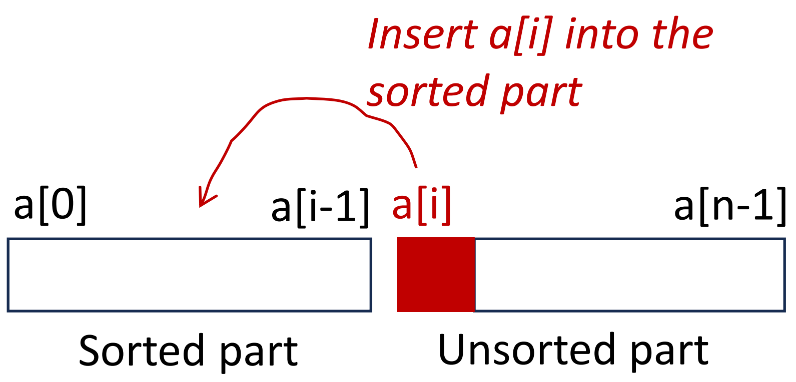

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
```

- a[0:i-1] = [1,2,3,4] and t = 0 → 4 compares
- a[0:i-1] = [1,2,3,…,i] and t = 0 → i compares

For a list in decreasing order:

total compares = $1 + 2 + 3 + \ldots + (n\text{-}1) = (n\text{-}1)n/2 = ½(n^2)\text{-}½(n)$

Best-case comparison count



Insertion sort

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
```

- a[0:i-1] = [1,2,3,4] and t = 5 → 1 compare
- a[0:i-1] = [1,2,3,...,i] and t = i+1 → 1 compare

For a list in increasing order:

total compares = 1 + 1 + 1 + ... + 1 = *n*-1

Asymptotic complexity

- Sometimes determining exact step counts is difficult and not useful.

- Using big-O to represent space and time complexity

$\frac{1}{2}(n^2)-\frac{1}{2}(n)$ vs. n^2+3n-4: which one is better?

- Both are $O(n^2)$
- There performance are similar.

Time complexity of insertion sort

- Worst case

When the list is in decreasing order

$\frac{1}{2}(n^2)-\frac{1}{2}(n) \rightarrow O(n^2)$

- Best case

When the list is in increasing order

$n-1 \rightarrow O(n)$

How do various functions grow with *n*?

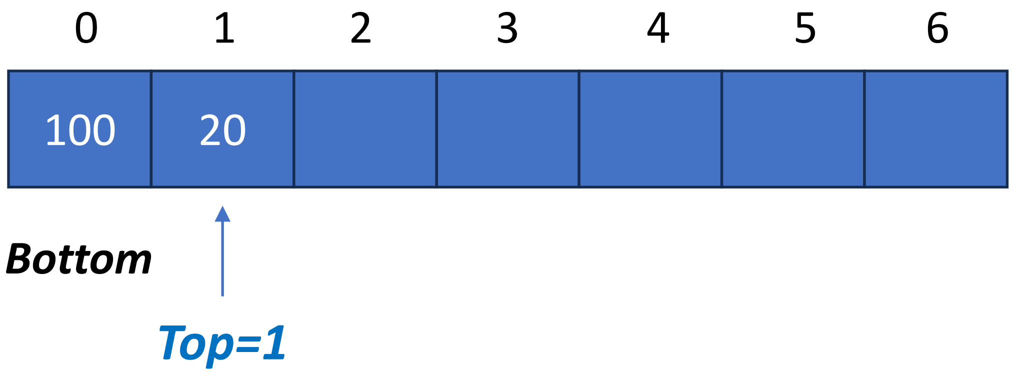
	linear		quadratic	cubic	exponential
log n	**n**	**n log n**	**n^2**	**n^3**	**2^n**
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	6
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

On a computer performing 1 billion (10^9) steps/second

2^{40} steps require 18.3 mins, 2^{50} steps require 13 days, and 2^{60} steps require 310 years.

Sometimes, improving algorithm may be more useful than improving hardware.

Stacks: Using variable *top* to implement operations



- IsEmpty(): check whether *top* >= 0
- IsFull(): check whether *top* == MAX_STACK_SIZE-1
- Top(): if not empty, return stack[*top*]

Time complexity

O(1)

O(1)

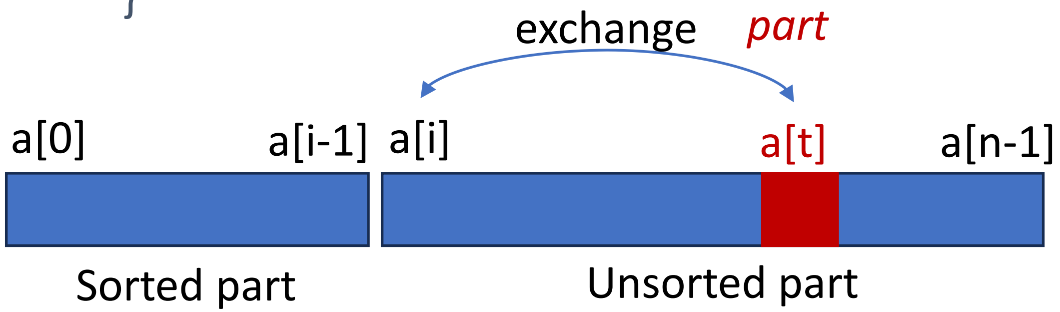
O(1)

Exercise

- **Selection sort**

```
for (i = 0; i < n-1; i++)
  {  /*Find the minimum element in a[i:n-1]*/
     int t = i;
     for (j = i+1; j < n; j++)
        if (a[j] < a[t])
           t = j;
      /* Swap the minimum element with the a[i]*/
     if(t != i)
        swap(&a[t], &a[i]);
  }
```

Select the minimum element in unsorted part



Please reply your answers of Q3 and Q4 via the following link:

Group members: 1~3 people

- Q4: Given a list in **increasing** order, what is the time complexity of selection sort?
- Q5: Given a list in **decreasing** order, what is the time complexity of selection sort?