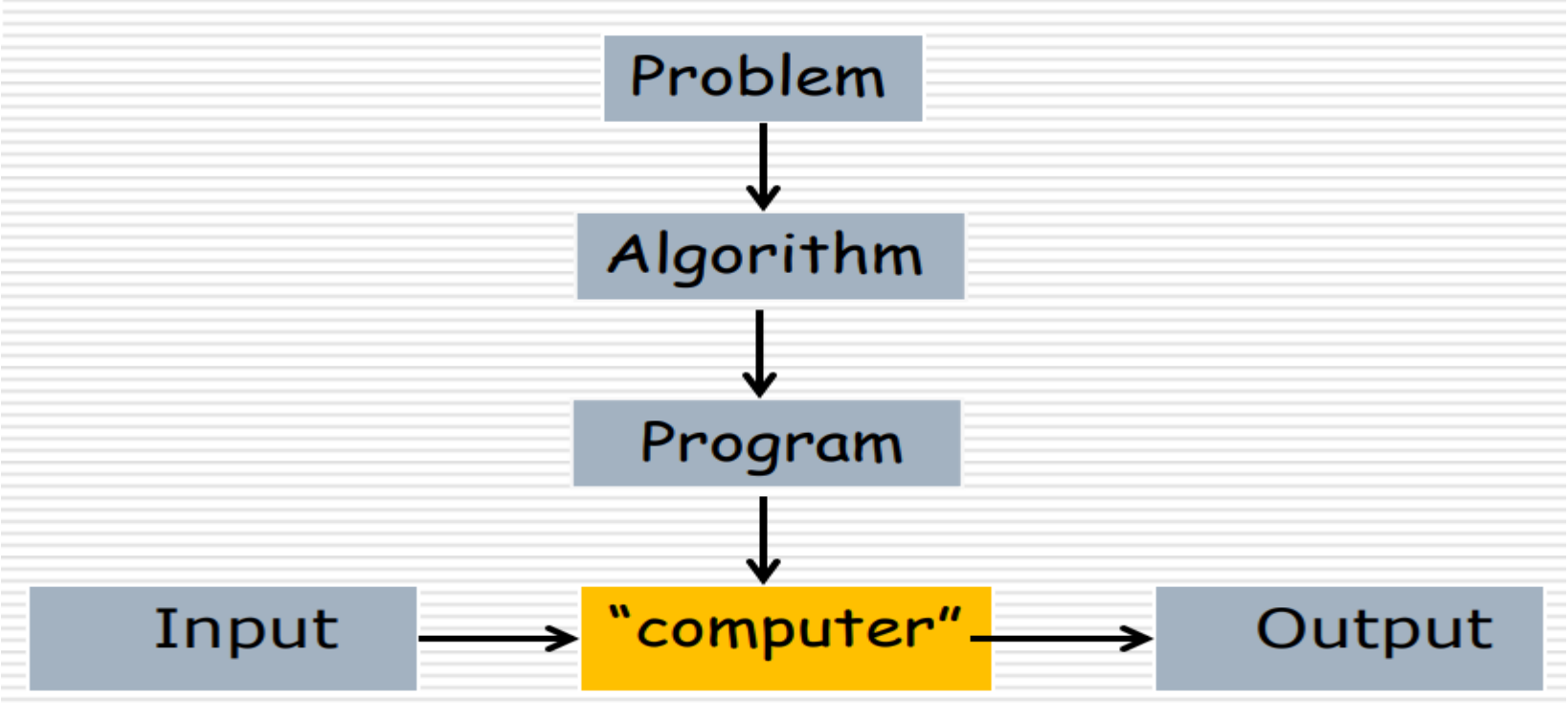
Unit 1: Introduction to Algorithms

- Fundamentals of Algorithmic Problem Solving
- Space and Time Complexity
- Order of Growth
- Asymptotic Notations

CY43/CI43 – DAA

Algorithm

- An algorithm is a sequence of unambiguous instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Examples of Algorithms

- Computing Greatest Common Divisor of Two non-negative, not-both zero Integers
- gcd(m, n): the largest integer that divides both m and n
- Euclid's Algorithm:
  - gcd(m, n) = gcd(n, m mod n

Greatest Common Divisor (Euclid's Algorithm), gcd(m, n)

- Step 1: If n = 0, return value of m as the answer and stop; otherwise, proceed to Step 2.
- Step 2: Divide m by n and assign the value of the remainder to r.
- Step 3: Assign the value of n to m and the value of r to n. Go to Step 1

Pseudocode for (Euclid's Algorithm), gcd(m, n)

- ALGORITHM Euclid(m, n)
- // Computes gcd(m, n) by Euclid's algorithm
- // Input: Two nonnegative, not-both-zero integers m and n
- //Output: Greatest common divisor of m and n

```
while n ≠ 0 do
        r = m mod n
        m = n
        n = r
        return m
```

Question: GCD(50, 30) how many division Operations are required to compute GCD using Euclid algorithm ?

Middle-school procedure, gcd(m, n)

- Step 1: Find prime factors of m.
- Step 2: Find prime factors of n.
- Step 3: Identify all common prime factors of m and n
- Step 4: Compute product of all common factors and return product as the answer.

Consecutive Integer Checking, gcd(m, n)

- Step 1: Assign the value of min(m, n) to t.
- Step 2: Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4.
- Step 3: Divide n by qt If the remainder is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
- Step 4: Decrease the value of t by 1. Go to Step 2.

Algorithm Analysis

- Same problem, but different algorithms, based on different ideas and having dramatically different speeds.

C++ Program - Analysis of the methods to find the GCD of two numbers

```
#include<iostream.h>
#include<conio.h>
#include<time.h>
long int euclid(long int m,long int n)
{
clock_t start,end;
start=clock();
long int r;
while(n!=0)
{
r=m%n;
m=n;
n=r;
}
end=clock();
cout<<endl<<"Time
taken:"<<(end-start)/CLK_TCK<<"
sec";
return m;
}
```

Contd.....

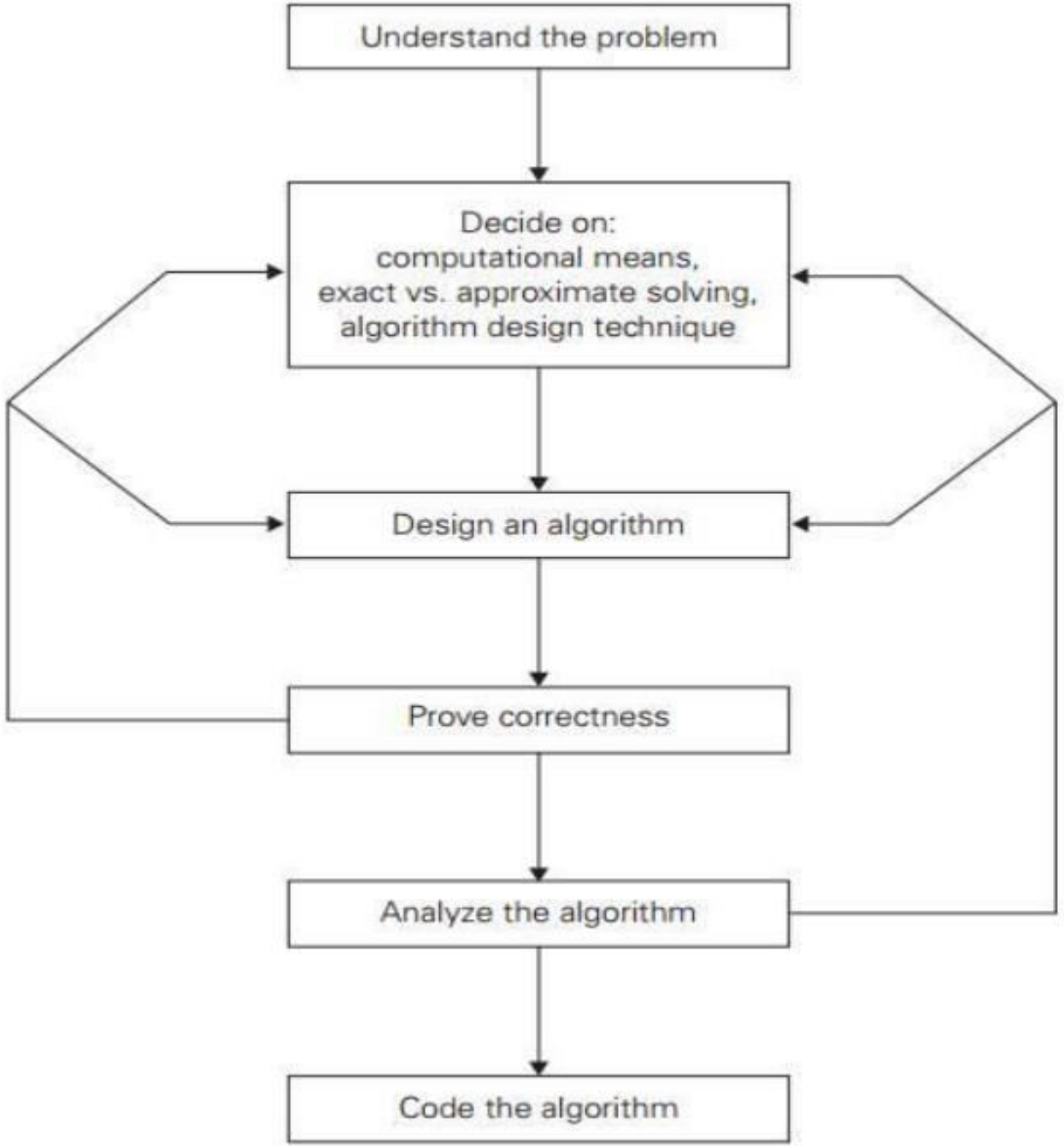
```
long int con(long int m,long int n)
{
clock_t start,end;
start=clock();
long int t,r,g;
if(m>n)
{ t=n; }
else
{ t=m; }
a:do
{
r=m%t;
if(r!=0)
t--;
} while(r!=0);
if(r==0)
{
r=n%t;
if(r==0)
g=t;
else
{
t--;
goto a;
}
}
end=clock();
cout<<"Time taken :"<<(endstart)/CLK_TCK<<"
sec";
return g;
} /*End of the function con*
```

Contd…. Analysis of 2 algorithms

```
void main()
{
long int x,y;
clrscr();
cout<<"\t\t  ANALYSIS OF THE TWO ALGORITHMS"<<endl<<endl;
cout<<"GCD - EUCLID'S ALG : "<<endl;
cout<<"enter two numbers:";
cin>>x>>y;
cout<<endl<<endl<<"GCD : "<<euclid(x,y);
cout<<endl<<endl<<"----------
----------";
cout<<endl<<endl<<"GCD - CONSECUTIVE INTEGER CHECKING ALG :
"<<endl<<endl;
cout<<endl<<endl<<"GCD : "<<con(x,y);
getch();
}
```

Fundamentals of Algorithmic Problem Solving

- Sequence of steps in the process of design and analysis of algorithms



Understanding the problem

- Ask questions, do a few small examples by hand, think about special cases, etc.
- An input is an instance of the problem the algorithm solves
- Specify exactly the set of instances the algorithm needs to handle
- Decide on
- Exact vs. approximate solution
- Approximate algorithm: Cannot solve exactly, e.g., extracting square roots, solving nonlinear equations, etc.
- Appropriate Data Structure

Analyze algorithm

- Time efficiency: How fast it runs
- Space efficiency: How much extra memory it uses
- Simplicity: Easier to understand, usually contains fewer bugs, sometimes simpler is more efficient, but not always.

Coding algorithm

- Write in a programming language for a real machine
- Standard tricks:
  - Compute loop invariant (which does not change value in the loop) outside loop
  - Replace expensive operation by cheap ones.

Analysis of Algorithms

- Reasons to Analyze Algorithms
- Predict Performance
- Compare Algorithms
- Provide Guarantees
- Understand theoretical basis.
- Primary Practical Reason: Avoid Performance Bugs

Performance measure of the algorithm

- Two kinds of efficiency:
  - Space Efficiency or Space Complexity
  - Time Efficiency or Time Complexity

Two kinds of Algorithm Efficiency

- Analyzing the efficiency of an algorithm (or the complexity of an algorithm) means establishing the amount of computing resources needed to execute the algorithm.

There are two types of resources:

- Memory space. It means the amount of space used to store all data processed by the algorithm.
- Running time. It means the time needed to execute all the operations specified in the algorithm.

- Space efficiency: Deals with the space required by the algorithm
- Time efficiency: It indicates how fast an algorithm runs.

What is Space complexity?

- For any algorithm, memory is required for the following purposes
- Memory required to store program instructions
- Memory required to store constant values
- Memory required to store variable values

- Space complexity of an algorithm can be defined as follows
  - Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

What is Space complexity?

- Generally, when a program is under execution it uses the computer

memory for THREE reasons.

They are:

- Instruction Space: It is the amount of memory used to store compiled version of instructions.
- Data Space: It is the amount of memory used to store all the variables and constants.
- Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.

Space Complexity= Instruction space + Data space + Stack space

Stack Allocation of Space

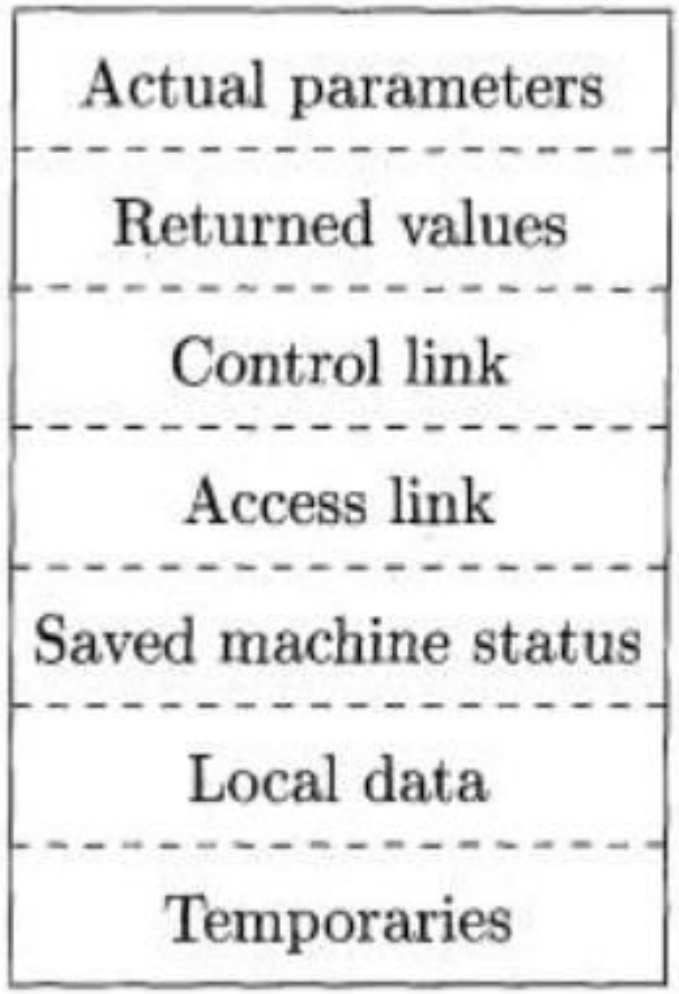
Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return).

An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

A *control link,* pointing to the activation record of the caller.

The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency.



Calculating Space Complexity

- To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following.
- 1 byte to store Character value,
- 2 bytes to store Integer value,
- 4 bytes to store Floating Point value,
- 6 or 8 bytes to store double value

Calculating Space Complexity

Calculating the Data Space required for the following given code

```
int square(int a)
{
return a*a;
}
```

Data Space Required:

- For int a → **2 Bytes**
- For returning a*a → **2 Bytes**

Total **4 Bytes**

Data Space Required:

- This code requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.
- That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.
- If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Performance measure of the algorithm

- What is Time complexity?
- Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.
- Time complexity of an algorithm can be defined as follows.
- The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution

What is Time complexity?

- Generally, running time of an algorithm depends upon the following.
- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32 bit machine or 64 bit machine
- Read and Write speed of the machine.
- The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
- Input data

Calculating Time Complexity

	Cost or Number Operations in the Statement
int sum = 0, i;	1 (initializing zero to sum)
for(i = 0; i < n; i++)	1+1+1 (i=0, i<n, i++)
sum = sum + A[i];	1+ 1 (Addition and Assigning result to sum)
return sum;	1 (returning sum)

Calculating Time Complexity

Example, Calculating the Time Complexity required for the following given code

```
int sum(int a, int b) {
return a+b;
}
```

	Time Required	
To **calculate** a+b	1 Unit of time	
For **returning** a+b	1 Unit of time	
Total	**2 Units of time**	

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Calculating Time Complexity

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**
int sum = 0, i;	1	**1**
for(i = 0; i < n; i++)	1+1+1	**1+(n+1)+n** (i=0 gets executed one time, i<n gets executed (n+1) times, i++ gets executed n times
sum = sum + A[i];	1+ 1	
return sum;	1	

Calculating Time Complexity

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1+1+1	1+(n+1)+n	2n+2
sum = sum + A[i];	1+ 1	n + n	2n
return sum;	1	1	1
		Running Time T(n)	4n+4

Calculating Time Complexity

- Cost is the amount of computer time required for a single operation in each line.
- Repetition is the amount of computer time required by each operation for all its repetitions.
- Total is the amount of computer time required by each operation to execute.
- So above code requires '4n+4' Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.
- Totally it takes '4n+4' units of time to complete its execution and it is Linear Time Complexity.
- If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity

Stable Marriage Problem

There is a set Y = $\{m_1,\ldots,m_n\}$ of n men and a set X = $\{w_1,\ldots,w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A *marriage matching* M is a set of n pairs (m_i, w_j).

A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable

{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Stable Marriage Algorithm (Gale-Shapley)

Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:

Proposal The selected free man *m* proposes to *w*, the next woman on his preference list

Response If *w* is free, she accepts the proposal to be matched with *m*. If she is not free, she compares *m* with her current mate. If she prefers *m* to him, she accepts *m*'s proposal, making her former mate free; otherwise, she simply rejects *m*'s proposal, leaving *m* free

Step 2 Return the set of *n* matched pairs

Example

Free men:
Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea
Lea accepted

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea
Lea rejected

Example (cont.)

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Sue
Sue accepted

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Sue
Sue rejected

Example (cont.)

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Lea
Lea replaced Bob
with Tom

Free men:
Bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann
Ann accepted

Analysis of the Gale-Shapley Algorithm

- The algorithm terminates after no more than n^2 iterations with a stable marriage output
- The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men
- A man (woman) optimal matching is unique for a given set of participant preferences
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training