

# COMP 3522

Object Oriented Programming in C++  
Week 8 Day 2

# Agenda

1. Static
2. Genetic Algorithm
  - Assignment 2
3. Midterm  
walkthrough

# COMP

# 3522

STATIC

# Static declarator

- In C++ member variables can be static:
  - Only **one copy** per class exists
  - Permit a single resource to be **shared** between instances
  - **Independent static storage** for life of program
  - This can be useful for the Singleton design pattern (we'll look at it later, but the name is a hint)
  - **static keyword used with declaration** not definition

```
class X { static int n; }; // incomplete declaration  
int X::n = 1; // definition
```

# Accessing statics

- Two forms can be used:

1. Qualified name

**Class::member**

2. Member access expression

**Class->member or Class.member**

- Exist even if no objects have been defined

# Initializing static

- **Can't** be initialized inside the class definition

```
class X {  
    static int m = 5; //ERROR  
    static int n; //OK  
  
};
```

```
int X::n = 5; //OK
```

# Static constants: const

- **Can** be initialized with an initializer in which every expression is a constant expression right inside the class definition

```
class X {  
    const static int m = 9; //initialized inside  
    const static int k;  
    static const double * pointer;  
};  
  
const int X::k = 3; //initialized outside  
const double * X::pointer = new double[3];
```

# Static constants: constexpr

- **Must** be initialized with an initializer in which every expression is a constant expression right inside the class definition

```
class X {  
    constexpr static int arr[] = { 1, 2, 3 }; // OK  
    constexpr static std::complex<double> n  
                                   = {1,2}; // OK  
    constexpr static int k; // Error  
};
```



# Static member functions

- Can only access static data and invoke static member functions
- There is no `*this` pointer
- Cannot be virtual or const

# Applications?

- Classes that don't need to be instantiated to be used
- Static calculator
  - A class that contains a series of common math operations
  - Add, subtract, multiply, divide
  - Instead of instantiating a Calculator class, just call it statically

//available only within scope of c variable

```
Calculator c;
```

```
c.add(num1, num2);
```

//available globally

```
Calculator::add(num1, num2)
```

[calculator.cpp & staticCalculator.cpp](#)

# GENETIC ALGORITHMS

# Genetic algorithms

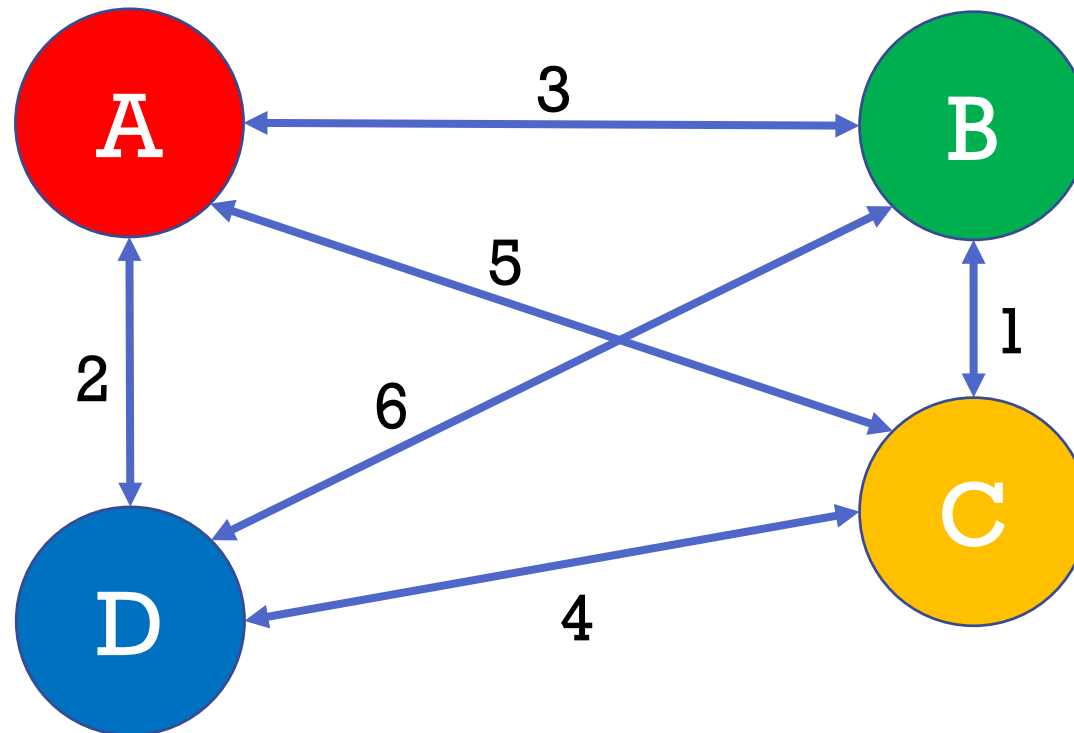
- Computing has been helpful in many problem domains
- Many problem domains have, in turn, lent problem solving strategies to computing
- Genetic algorithms are inspired by the process of natural selection described in contemporary biology
- Useful for large problems with solutions that are difficult to find
- Useful for optimization, search problems.

# Genetic algorithm

```
{  
    initialize population  
    evaluate population  
    while (termination criteria not reached)  
    {  
        select solutions for next population  
        perform crossover and mutation  
        evaluate population  
    }  
}
```

# Travelling salesperson problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



# Travelling salesperson problem

- Input:
  - List of cities to visit
- Requirements:
  - Visit all the cities
    - Return to original city
  - Minimize travelling distance

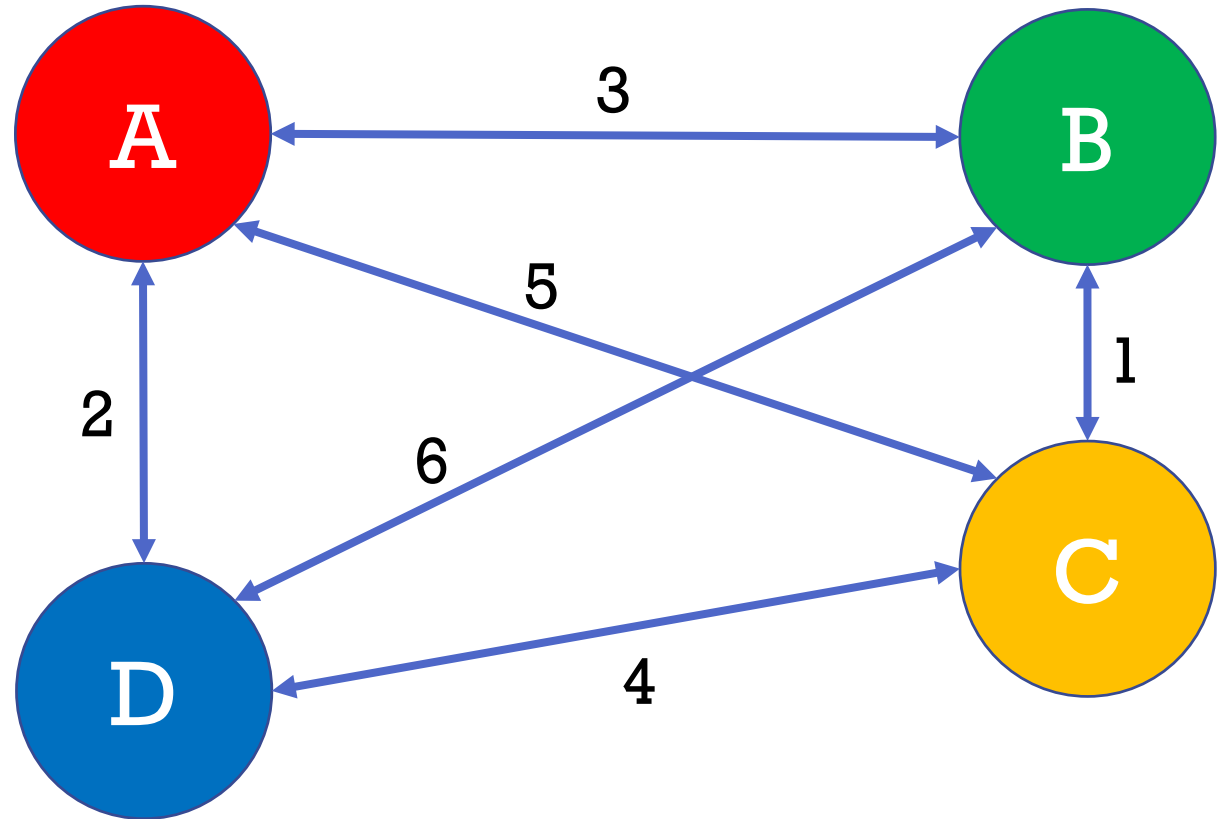
This sounds easy (we can do this by hand with a few cities)

$O(n!)$  – this becomes impractical with 20 cities.

What if we have 200 cities, or 200,000 cities, or ALL the cities and towns and villages in the world?

# Let's travel around 4 cities

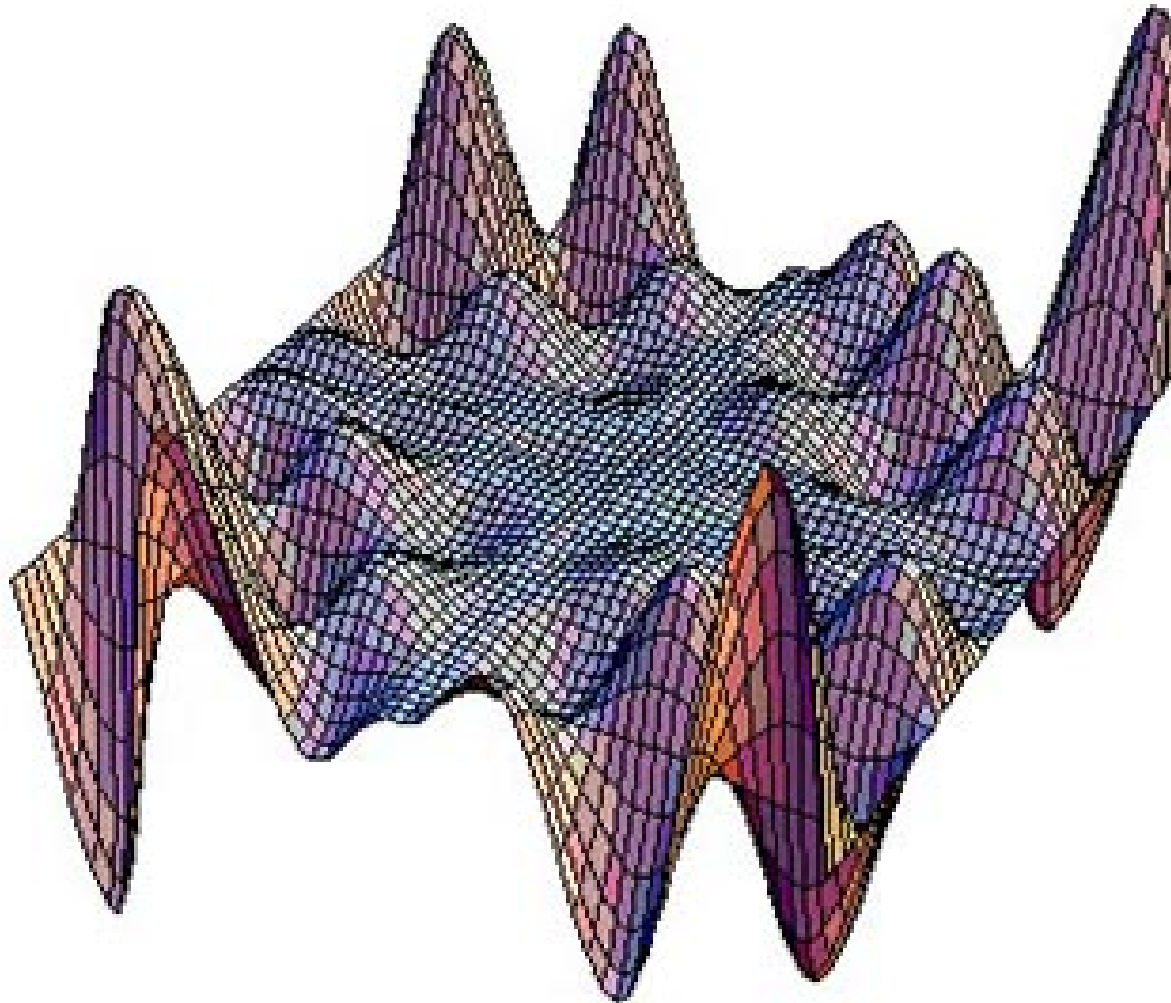
- **ABCD**A = 3+1+4+2 = 10
- ACBDA = 5+1+6+2 = 14
- ADBCA = 2+6+1+5 = 14
- **ADC**BA = 2+4+1+3 = 10
- ...
- DBCAD = 6+1+5+2=14
- DBACD = 6+3+5+4=18



- 24 possibilities = 4! (4 factorial)



# Imagine the 'solutionscape'



Some tours  
are very  
short

Some tours  
are very  
long

There are  
too many to  
find a global  
solution.

# Some terms we are using

**City:** A location that has a name and x/y coordinates.

**Cities\_to\_visit:** an invariant (unchanging) list of City structs that we want to visit. The “master list.”

**Tour:** a list of pointers to the cities we want to visit. We can shuffle the pointers easily to compare different orderings of cities without modifying the “master list”.

**Population:** a collection of candidate Tours. We keep the population “sorted,” i.e., the “fittest” tours are at the front of the list.

# Some terms we are using

**Fitness:** Each candidate Tour in the population has a fitness, i.e., how “good” it is. For us, a fit Tour has a short travel distance. A Tour with a shorter distance has better fitness.

**Elite:** Each generation, we can designate one or more Tours that are so amazing they don’t cross, they get carried over to the next ‘generation’. These Tours are “elite.”

**Crosses and Crossover:** Each generation we create new Tours by crossing “parents.” The crossover algorithm is basic.

# Some terms we are using

**Parents:** Each iteration, we select some parents from the Population of Tours and use the parents' contents to generate a new Tour for the next iteration.

**Mutation:** Each iteration, we randomly “mess up” a few of the Tours in our population. This mimics the random mutations that take place as cells divide, etc.

**Mutation rate:** If we ‘roll’ less than the rate, we swap a few cities in the Tour being mutated.

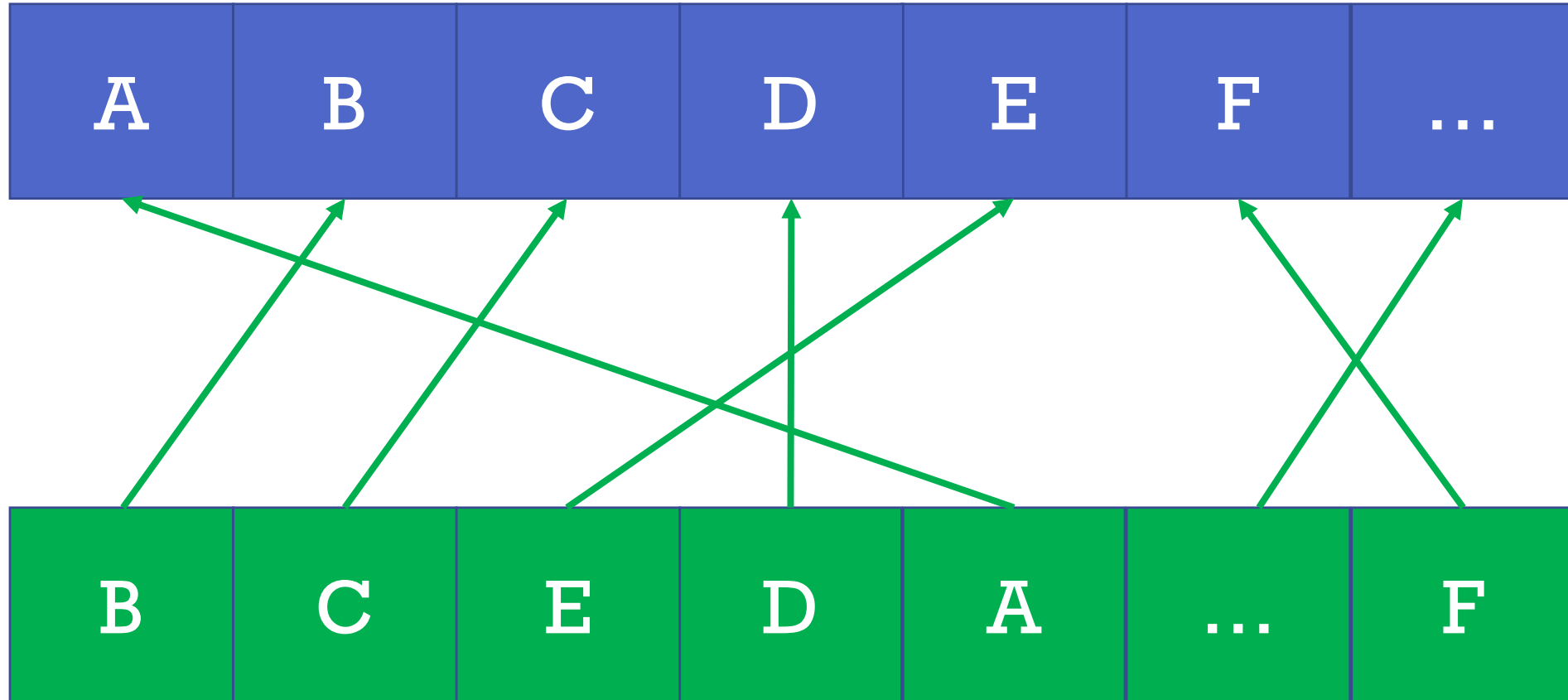
# Our genetic algorithm

```
{  
    create cities  
    evaluate cities' fitness  
    while (fitness < improvement or iterations < 1000)  
    {  
        move elite to front  
        perform crossover and mutation of tours  
        evaluate cities fitness  
    }  
}
```

# Our algorithm

1. Create our master list of cities named A, B, C, ..., R, S, T. Our master list is 20 Cities long.
2. Create a Population of Tours. The Population contains 30 candidate Tours. Each Tour contains pointers that point to the cities in the master list. Each Tour is shuffled randomly.

Master city list – doesn't change



Tour – pointers to cities

Have 30 tours each with random pointers to cities

# Our algorithm

3. Find the shortest travelling distance in the randomly shuffled Tours. That's our starting point.

Population – list of tours

Tour 1 = 100 cost
Tour 2 = 200 cost
...
Tour 29 = 70 cost
Tour 30 = 150 cost

**ELITE!**



# Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)

1. Find the best tour in the Population, call it an Elite, and move it to the front of the list so we can keep an eye on it.

Population – list of tours

Tour 29 = 70 cost
Tour 2 = 200 cost
...
Tour 28 = 100 cost
Tour 30 = 150 cost

Move ELITE to front

- Identified the best tour of the existing tours
- Next create a new list of tours based on the existing tours

# Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)
  2. Create a temporary list of Tours called Crosses.
  3. For each remaining Tour in our population, create a new Tour that is generated by crossing parents

Original Population

Elite Tour = 70 cost
Tour 2 = 200 cost
...
Tour 28 = 100 cost
Tour 30 = 150 cost



Crosses – new population

Elite Tour = 70 cost
?
?
?
?

## Original Population

### Step 4.3 Crossing parents

Elite Tour = 70 cost
Tour 2 = 200 cost
...
Tour 28 = 100 cost
Tour 30 = 150 cost

Set 1

Tour 26 = 500 cost
Tour 11 = 700 cost
Tour 27 = 800 cost
Tour 14 = 900 cost
Tour 30 = 150 cost

Set 2

Tour 13 = 700 cost
Tour 2 = 200 cost
Tour 12 = 300 cost
Tour 10 = 600 cost
Tour 9 = 350 cost

- Pick two sets of 5 random tours from the original population

## Step 4.3 Crossing parents

- Find the **fittest tour** in each set.
- These two parents will be crossed to generate a new child

Set 1

Tour 26 = 500 cost
Tour 11 = 700 cost
Tour 27 = 800 cost
Tour 14 = 900 cost
Tour 30 = 150 cost

Parent 1

Set 2

Tour 13 = 700 cost
Tour 2 = 200 cost
Tour 12 = 300 cost
Tour 10 = 600 cost
Tour 9 = 350 cost

Parent 2

## Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of Parent 1, copy everything up to and including index 1 from parent 1 to child

Parent 1

Tour 30 = 150 cost  
A B C D E

Parent 2

Tour 2 = 200 cost  
D C A B E

Child tour  
?????

## Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of Parent 1, copy everything up to and including index 1 from parent 1 to child

Parent 1

Tour 30 = 150 cost

A B C D E



Parent 2

Tour 2 = 200 cost

DCABE

Child tour


A????

## Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of Parent 1, copy everything up to and including index 1 from parent 1 to child

Parent 1

Tour 30 = 150 cost  
A B C D E



Parent 2

Tour 2 = 200 cost  
D C A B E

Child Tour  
A **B** ???

## Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in Parent 2 and child, copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost  
A B C D E

Parent 2

Tour 2 = 200 cost  
D C A B E



Child Tour  
A B D ??



## Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities, copy over non-duplicate cities to child

Parent 1

Tour 30 = 150 cost  
A B C D E

Parent 2

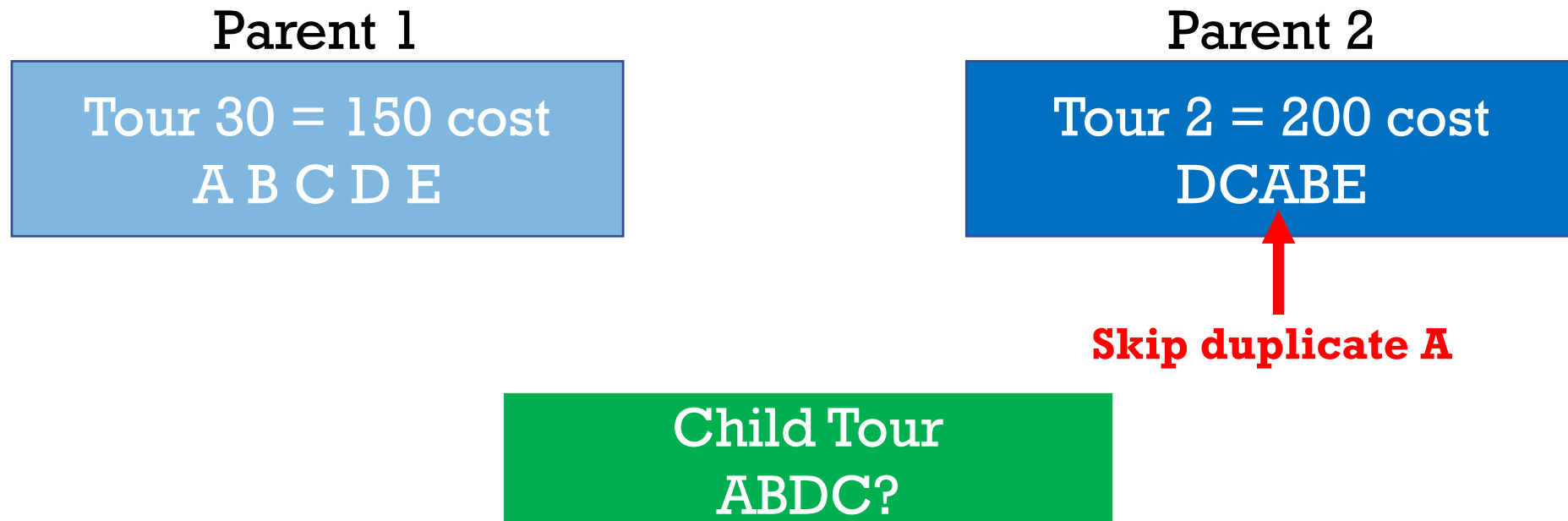
Tour 2 = 200 cost  
D C A B E



Child Tour  
A B D C ?

## Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities, copy over non-duplicate cities to child



## Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities, copy over non-duplicate cities to child

Parent 1

Tour 30 = 150 cost  
A B C D E

Parent 2

Tour 2 = 200 cost  
D C A B E

**Skip duplicate B**

Child Tour  
A B D C ?

## Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities, copy over non-duplicate cities to child

Parent 1

Tour 30 = 150 cost  
A B C D E

Parent 2

Tour 2 = 200 cost  
D C A B E



Child Tour  
A B D C E

# Our algorithm

- This is our new merged tour. Repeat previous steps for the rest of the new population of tours

Merged Tour 1  
ABDCE

4. (Loop) While we haven't reached our goal (or still have iterations)

4. Replace all the Tours in our Population (except the Elite Tour) with the new crosses

Elite Tour = 70 cost
Merged Tour 1
Merged Tour 2
...
Merged Tour 28
Merged Tour 29

# Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)
5. Mutate the population (except the Elite) by smudging around some of the Cities in each Tour.

Elite Tour = 70 cost

Merged Tour 1 (mutate few city pointers)

Merged Tour 2 (mutate few city pointers)

...

Merged Tour 28 (mutate few city pointers)

Merged Tour 29 (mutate few city pointers)



Merged Tour 2

ABCDEF...

Merged Tour 2 (after 15% mutation)

BACDFE...

# Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)
  6. Evaluate the fitness (distance) and report it.

Elite Tour = 70 cost
Merged Tour 1
Merged Tour 2
Merged Tour ...
Merged Tour 28
Merged Tour 29



Elite Tour = 70 cost
Merged Tour 1 = 300 cost
Merged Tour 2 = 50 cost
Merged Tour ...
Merged Tour 28 = 170 cost
Merged Tour 29 = 210 cost

**ELITE!**

5. Bam. You just implemented a genetic algorithm in C++!