In [14]:

```python
"""
Simple multithreaded algorithm to show how the 4 phases of a genetic algorithm works
(Evaluation, Selection, Crossover and Mutation)
https://en.wikipedia.org/wiki/Genetic_algorithm
Author: D4rkia
"""

from __future__ import annotations

# Python program to show time by process_time()
from time import process_time

import os, psutil
import matplotlib.pyplot as plt

import random

# Maximum size of the population.  bigger could be faster but is more memory expensive
N_POPULATION = 50000
# Number of elements selected in every generation for evolution the selection takes
# place from the best to the worst of that generation must be smaller than N_POPULATION
N_SELECTED = 50
# Probability that an element of a generation can mutate changing one of its genes this
# guarantees that all genes will be used during evolution
MUTATION_PROBABILITY = 0.4
# just a seed to improve randomness required by the algorithm
random.seed(random.randint(0, 1000))
process = psutil.Process(os.getpid())

# Start the stopwatch / counter
t1_start = process_time()

def basic(target: str, genes: list[str], debug: bool = True) -> tuple[int, int, str]:
    """
    Verify that the target contains no genes besides the ones inside genes variable.
    >>> from string import ascii_lowercase
    >>> basic("doctest", ascii_lowercase, debug=False)[2]
    'doctest'
    >>> genes = list(ascii_lowercase)
    >>> genes.remove("e")
```

```
    >>> basic("test", genes)
    Traceback (most recent call last):
    ...
    ValueError: ['e'] is not in genes list, evolution cannot converge
    >>> genes.remove("s")
    >>> basic("test", genes)
    Traceback (most recent call last):
    ...
    ValueError: ['e', 's'] is not in genes list, evolution cannot converge
    >>> genes.remove("t")
    >>> basic("test", genes)
    Traceback (most recent call last):
    ...
    ValueError: ['e', 's', 't'] is not in genes list, evolution cannot converge
    """

    # Verify if N_POPULATION is bigger than N_SELECTED
    if N_POPULATION < N_SELECTED:
        raise ValueError(f"{N_POPULATION} must be bigger than {N_SELECTED}")
    # Verify that the target contains no genes besides the ones inside genes variable.
    not_in_genes_list = sorted({c for c in target if c not in genes})
    if not_in_genes_list:
        raise ValueError(
            f"{not_in_genes_list} is not in genes list, evolution cannot converge"
        )

    # Generate random starting population
    population = []
    for _ in range(N_POPULATION):
        population.append("".join([random.choice(genes) for i in range(len(target))]))

    # Just some logs to know what the algorithms is doing
    generation, total_population = 0, 0

    # This loop will end when we will find a perfect match for our target
    while True:
        generation += 1
        total_population += len(population)

        # Random population created now it's time to evaluate
        def evaluate(item: str, main_target: str = target) -> tuple[str, float]:
            """
```

```
        Evaluate how similar the item is with the target by just
        counting each char in the right position
        >>> evaluate("Helxo Worlx", Hello World)
        ["Helxo Worlx", 9]
        """
        score = len(
            [g for position, g in enumerate(item) if g == main_target[position]]
        )
        return (item, float(score))

    # Adding a bit of concurrency can make everything faster,
    #
    # import concurrent.futures
    # population_score: list[tuple[str, float]] = []
    # with concurrent.futures.ThreadPoolExecutor(
    #                                   max_workers=NUM_WORKERS) as executor:
    #     futures = {executor.submit(evaluate, item) for item in population}
    #     concurrent.futures.wait(futures)
    #     population_score = [item.result() for item in futures]
    #
    # but with a simple algorithm like this will probably be slower
    # we just need to call evaluate for every item inside population
    population_score = [evaluate(item) for item in population]

    # Check if there is a matching evolution
    population_score = sorted(population_score, key=lambda x: x[1], reverse=True)
    if population_score[0][0] == target:
        return (generation, total_population, population_score[0][0])

    # Print the Best result every 10 generation
    # just to know that the algorithm is working
    if debug and generation % 10 == 0:
        print(
            f"\nGeneration: {generation}"
            f"\nTotal Population:{total_population}"
            f"\nBest score: {population_score[0][1]}"
            f"\nBest string: {population_score[0][0]}"
        )

    # Flush the old population keeping some of the best evolutions
    # Keeping this avoid regression of evolution
    population_best = population[: int(N_POPULATION / 3)]
```

```
        population.clear()
        population.extend(population_best)
        # Normalize population score from 0 to 1
        population_score = [
            (item, score / len(target)) for item, score in population_score
        ]

        # Select, Crossover and Mutate a new population
        def select(parent_1: tuple[str, float]) -> list[str]:
            """Select the second parent and generate new population"""
            pop = []
            # Generate more child proportionally to the fitness score
            child_n = int(parent_1[1] * 100) + 1
            child_n = 10 if child_n >= 10 else child_n
            for _ in range(child_n):
                parent_2 = population_score[random.randint(0, N_SELECTED)][0]
                child_1, child_2 = crossover(parent_1[0], parent_2)
                # Append new string to the population list
                pop.append(mutate(child_1))
                pop.append(mutate(child_2))
            return pop

        def crossover(parent_1: str, parent_2: str) -> tuple[str, str]:
            """Slice and combine two string in a random point"""
            random_slice = random.randint(0, len(parent_1) - 1)
            child_1 = parent_1[:random_slice] + parent_2[random_slice:]
            child_2 = parent_2[:random_slice] + parent_1[random_slice:]
            return (child_1, child_2)

        def mutate(child: str) -> str:
            """Mutate a random gene of a child with another one from the list"""
            child_list = list(child)
            if random.uniform(0, 1) < MUTATION_PROBABILITY:
                child_list[random.randint(0, len(child)) - 1] = random.choice(genes)
            return "".join(child_list)

        # This is Selection
        for i in range(N_SELECTED):
            population.extend(select(population_score[int(i)]))
            # Check if the population has already reached the maximum value and if so,
            # break the cycle.  if this check is disabled the algorithm will take
            # forever to compute large strings but will also calculate small string in
```

```python
            # a lot fewer generations
            if len(population) > N_POPULATION:
                break


if __name__ == "__main__":
    target_str = (
        "This is a genetic algorithm to evaluate, combine, evolve, and mutate a string!"
    )
    genes_list = list(
        " ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklm"
        "nopqrstuvwxyz.,;!?+-*#@^'èéòà€ù=)(&%$£/\\"
    )
    print(
        "\nGeneration: %s\nTotal Population: %s\nTarget: %s"
        % basic(target_str, genes_list)
    )
    # Stop the stopwatch / counter
    t1_stop = process_time()
    print("Elapsed time:", t1_stop, t1_start)
    print("Elapsed time during the whole program in seconds:", t1_stop-t1_start)
    print("Memory Usage:", (process.memory_info().rss))  # in Mbits
```

```
Generation: 10
Total Population:208722
Best score: 32.0
Best string: j!fsWi!wa -eke'éc rògE&-thM PK #v pP=€l,wFQmbin%rU€voésZ,zan* oét!teuaèGtnKnE!

Generation: 20
Total Population:385382
Best score: 44.0
Best string: ChisWiswa gene'éc rZgoG-thM PK #v lRaiJ,wFombin%M €voész,zan* out!teuawGtninE!

Generation: 30
Total Population:562042
Best score: 53.0
Best string: ChisWis a gene'éc rZgoG-thm Po #v luaiJ,wFombin%M evoész,zand mut!teua Gtning!

Generation: 40
Total Population:738702
Best score: 60.0
Best string: ChisWis a gene'wc +ZgoGithm Po #valuaiJ,Qcombin%M evolsz,zand mutate a stning!
```

```
Generation: 50
Total Population:915362
Best score: 65.0
Best string: ThisWis a gene'ic aZgorithm Po evaluaiJ,wcombin%M evolsz,zand mutate a stning!

Generation: 60
Total Population:1092022
Best score: 68.0
Best string: ThisWis a genetic aZgorithm Po evaluatJ, combin%M evoljz,zand mutate a stning!

Generation: 70
Total Population:1268682
Best score: 74.0
Best string: This is a genetic aZgorithm Po evaluate, combine, evolve,zand mutate a stning!

Generation: 80
Total Population:1445342
Best score: 76.0
Best string: This is a genetic aZgorithm to evaluate, combine, evolve, and mutate a stning!

Generation: 90
Total Population:1622002
Best score: 77.0
Best string: This is a genetic aWgorithm to evaluate, combine, evolve, and mutate a string!

Generation: 100
Total Population:1798662
Best score: 77.0
Best string: This is a genetic aZgorithm to evaluate, combine, evolve, and mutate a string!

Generation: 110
Total Population:1975322
Best score: 77.0
Best string: This is a genetic aZgorithm to evaluate, combine, evolve, and mutate a string!

Generation: 112
Total Population: 2010654
Target: This is a genetic algorithm to evaluate, combine, evolve, and mutate a string!
Elapsed time: 97.1875 77.0625
Elapsed time during the whole program in seconds: 20.125
Memory Usage: 89403392
```

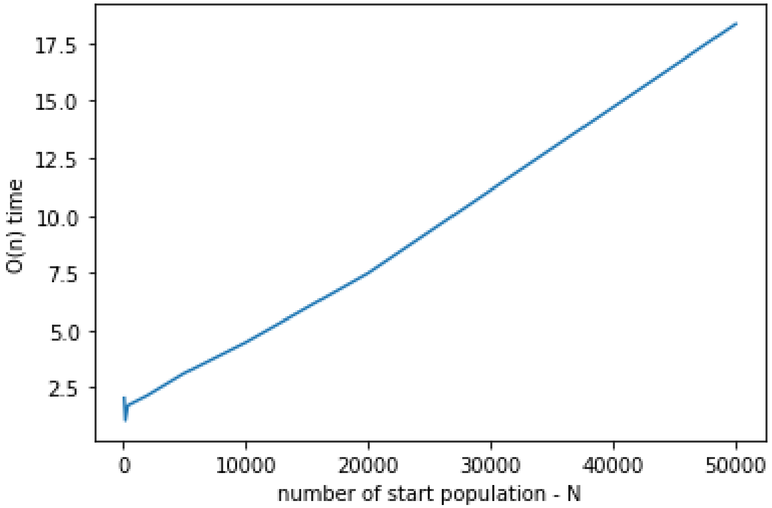
In [20]:

```
# time axe in y
```

```python
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()  # Create a figure containing a single axes.
ax.plot([100, 200, 400, 2000, 5000, 10000, 20000, 50000], [2.046875, 1.0625, 1.71875, 2.15625, 3.125, 4.46875, 7.4843
plt.ylabel('O(n) time')
plt.xlabel('number of start population - N')
```

Out[20]: Text(0.5, 0, 'number of start population - N')

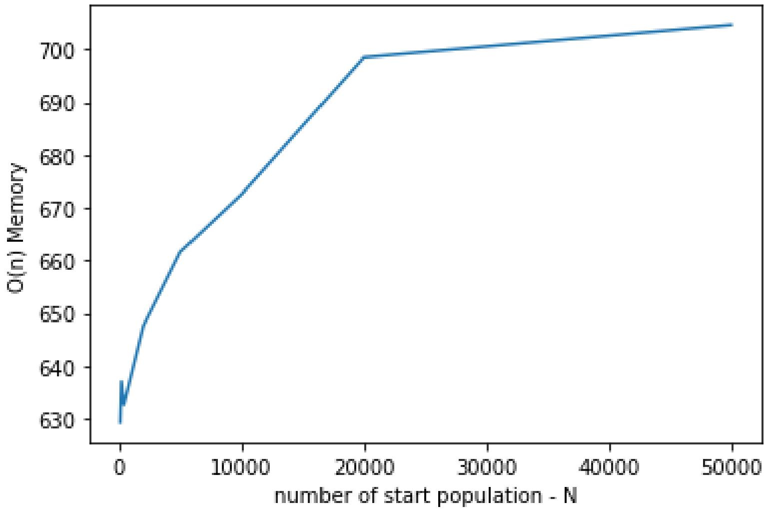


In [21]:

```python
# memory (Mbits) axe in y
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()  # Create a figure containing a single axes.
ax.plot([100, 200, 400, 2000, 5000, 10000, 20000, 50000], [629.342208, 637.108224, 632.684544, 647.626752, 661.61868
plt.ylabel('O(n) Memory')
plt.xlabel('number of start population - N')
```

Out[21]: Text(0.5, 0, 'number of start population - N')



O(n) Memory
700
690
680
670
660
650
640
630
0
10000
20000
30000
40000
50000
number of start population - N

In [ ]: