ALGORITHM DESIGN

PART 1

ADS1, S2023

A VIEW BACK IN TIME

GUESS MY WORD

not independent

→ We need efficient ways of solving a given problem → Algorithms

&

→ We need efficient ways of representing data needed to solve the problem → Data Structures

general problem?

lists

stacks

⋮

BUT WHY?

To ensure the quality of software

ADS = tools to analyze

Big-Oh

efficiency — do the thing fast

correctness — do the right thing

?

invariants

THE NEXT THREE WEEKS

- Algorithm correctness
  - Invariants
- Algorithm design
  - Brute force algorithms
  - Randomized algorithms
  - Greedy algorithms
  - Dynamic programming
  - Divide-and-conquer algorithms
  - Backtracking algorithms

Today

next week

in 2 weeks

INVARIANTS

```
power(m,n):
  r = 1
  for i from 1 to n:
      r = r * m
  return r
```

computes m^n

but how do we know?

LOOP INVARIANTS

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

Statements that

→ are true when we first reach the loop

→ if they are true before running through the loop, they are also true after

In this case:

I_1: m, n, r and i are integers

I_2: $r = m^i$

LOOP INVARIANTS

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

I_1: m, n, r and i are integers

I_2: $r = m^i$

Prove I_1 is true before we encounter the loop the first time:

m, n ✓

$i = 0$ ✓

$r = 1$ ✓

LOOP INVARIANTS

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

I_1: m, n, r and i are integers

I_2: $r = m^i$

prove that IF I_1 true before a run of the loop, it must also be true after

Before: assume m, n, r, i integers

After: $m \to m$ still integer

$n \to n$ — " —

$r \to r*m$ — " —

$i \to i+1$ — " —

INDUCTION PROOF

If we can show

↳ 1. Base case: True before

↳ 2. Induction step: If true at some point, then true after next loop run

then it's always true

LOOP INVARIANTS

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

prove I_2 true before

$r = 1 \quad i = 0$

$r = m^i$

$1 = m^0 = 1$ ✓

I_1: m, n, r and i are integers

I_2: $r = m^i$

LOOP INVARIANTS

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

I_1: m, n, r and i are integers

I_2: $r = m^i$

prove that IF I_2 true before loop, also true after

Before: $r_B = m^{i_B}$ → before

After: $r_A = m^{i_A}$?

loop: $r_A = m \cdot r_B$

$i_A = i_B + 1$

so $m \cdot r_B = m^{i_B+1} = m \cdot m^{i_B}$

$m \cdot r_B = m \cdot r_B$ ✓

FINALLY …

```
power(int m, int n):
    i = 0
    r = 1
    while i < n:
        r = r * m
        i = i + 1
    return r
```

I_1: m, n, r and i are integers
I_2: $r = m^i$

I_1 and I_2 also true at the end

$i = n$

$r = m^i = m^n$

So m^n is returned

WORKFLOW

Set up invariants fulfilling:
1) I true before
2) If true before one run, also true after

I will be true when we exit the loop

Use I & loop condition to make conclusion

ANOTHER EXAMPLE

```
square(int n):
    i = 0
    r = 0
    while i < n:
        r = r + 2*i + 1
        i = i + 1
    return r
```

I_1: n, r and i are integers

I_2: $r = i^2$

Prove I_1

Base case: n ✓ i=0 ✓ r=0 ✓

Induction: $n \to n$ ✓

$r \to r+2i+1$ ✓

$i \to i+1$ ✓

ANOTHER EXAMPLE

```
square(int n):
    i = 0
    r = 0
    while i < n:
        r = r + 2*i + 1
        i = i + 1
    return r
```

I_1: n, r and i are integers

I_2: $r = i^2$

Prove I_2

Base case: $0 = 0^2$ ✓

Induction: $r \to r + 2i + 1$

$i \to i + 1$

If $r = i^2$ is $(r + 2i + 1) = (i+1)^2$?

$i^2 + 2i + 1 = (i+1)^2$ ✓

ANOTHER EXAMPLE

```
square(int n):
    i = 0
    r = 0
    while i < n:
        r = r + 2*i + 1
        i = i + 1
    return r
```

I_1: n, r and i are integers

I_2: $r = i^2$

$i = n$

$r = i^2 = n^2$

So the algorithm returns … n^2

ALGORITHM DESIGN

- Brute force algorithms
- Randomized algorithms
- Greedy algorithms
- Dynamic programming
- Divide-and-conquer algorithms
- Backtracking algorithms

BRUTE FORCE ALGORITHMS

Generate all candidates for solutions and check individually

→ Always finds a solution

→ Easy to implement

→ Astronomical time and space complexities

BRUTE FORCE SORTING

4 | 8 | 3 | 1

4	8	3	1
4	8	1	3
4	3	8	1
4	3	1	8
4	1	8	3
4	1	3	8
8	4	3	1
8	4	1	3
8	3	4	1
8	3	1	4
8	1	4	3
8	1	3	4

3	4	8	1
3	4	1	8
3	8	4	1
3	8	1	4
3	1	4	8
3	1	8	4
1	4	8	3
1	4	3	8
1	8	4	3
1	8	3	4
1	3	4	8
1	3	8	4

$O(n!)$

WHEN SHOULD I USE BRUTE FORCE ALGORITHMS?

- Probably never
- If # of potential candidates can somehow be narrowed down (backtracking)
- If you need a solution that can't be achieved otherwise

ALGORITHM DESIGN

- Brute force algorithms
- Randomized algorithms
- Greedy algorithms
- Dynamic programming
- Divide-and-conquer algorithms
- Backtracking algorithms

RANDOMIZED ALGORITHMS

Incorporate some degree of randomness

→ Eliminate "bad input"

QUICKSORT

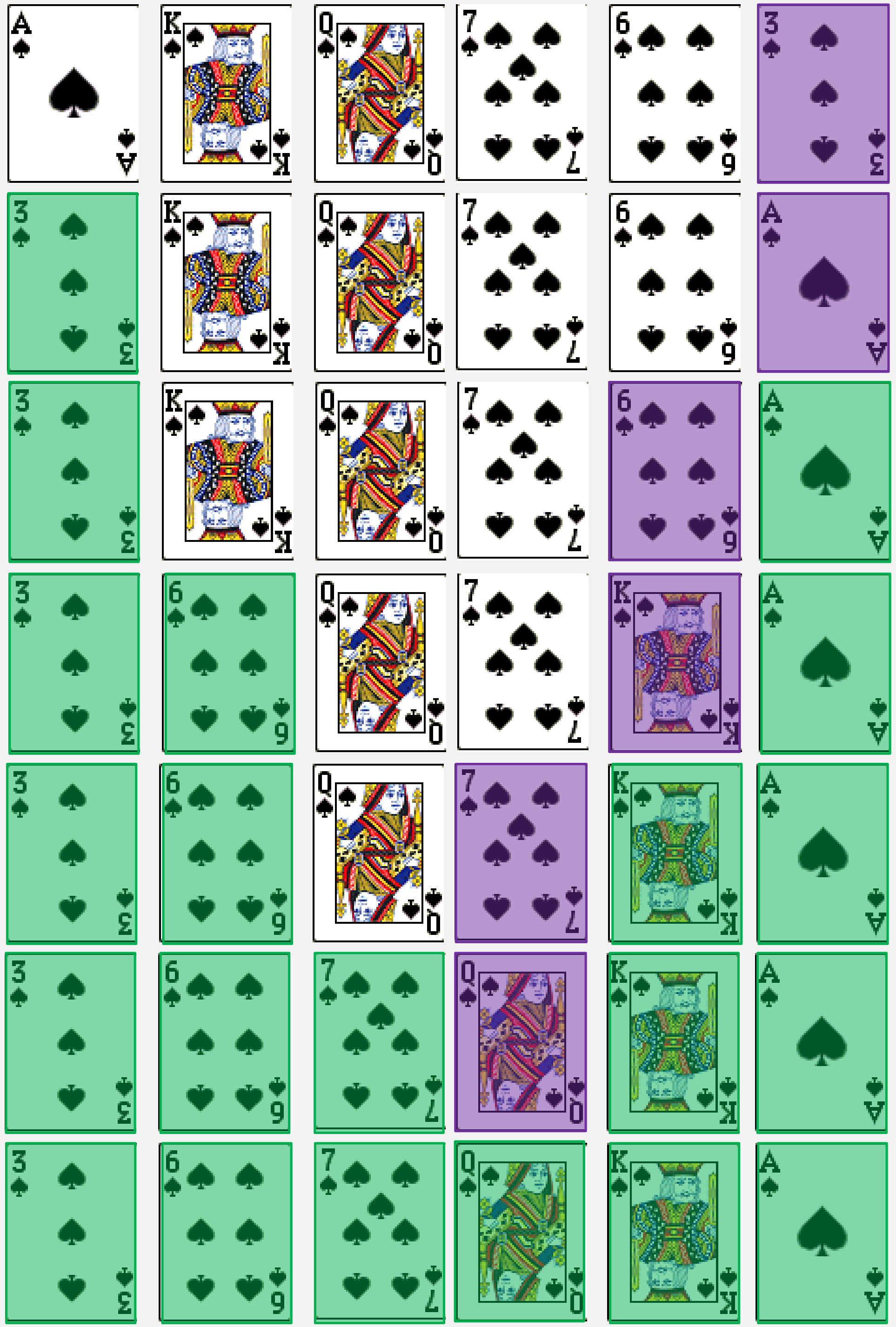
```
QuickSort(list, p, r):
    if p < r:
        q = Partition(list, p, r)
        QuickSort(list, p, q - 1)
        QuickSort(list, q + 1, r)


call QuickSort(A, 1, length(list))
```

```
Partition(list, p, r):
    x = list[r]
    i = p – 1
    for j = p to r – 1:
        if list[j] <= x:
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[r]
    return i + 1
```

→ pick last element

QUICKSORT ON BAD INPUT



$O(n^2)$

RANDOMIZED QUICKSORT

```
RandQuickSort(list, p, r):
    if p < r:
        q = RandPartition(list, p, r)
        RandQuickSort(list, p, q - 1)
        RandQuickSort(list, q + 1, r)
```

```
RandPartition(list, p, r):
    i = random number btwn p and r
    swap A[r] and A[i]
    return Partition(list, p, r)
```

call RandQuickSort(A, 1, length(list))

still $O(n^2)$ worst case, but no particular input structure gives you worst case

TWO GENERAL STRATEGIES

Problem: Find a 0 in a bitstring containing half 0's, half 1's.

Linear search: worst case $\frac{n}{2} = O(n)$ [if input bad]

$\Theta(1)$

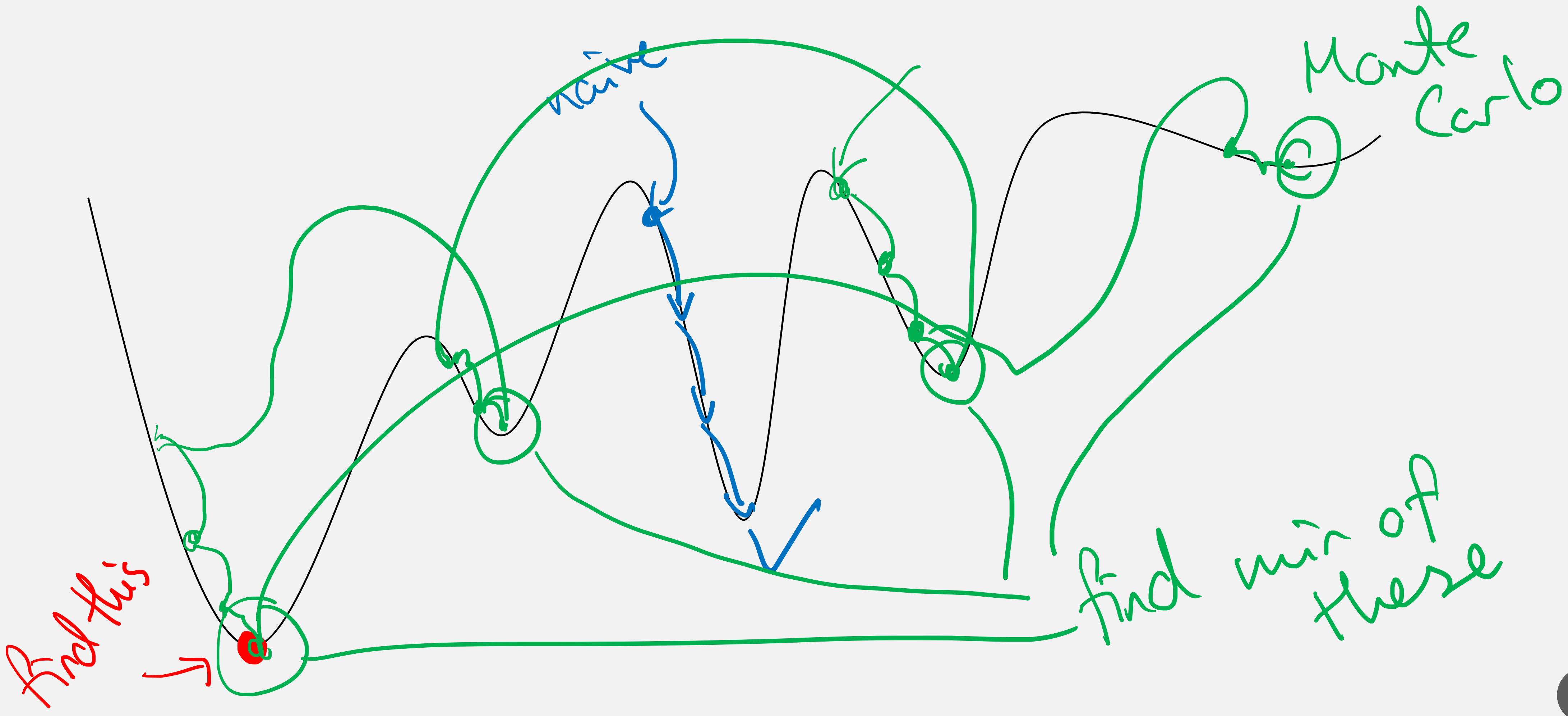
while 0 not found:
    randomly select new element

Las Vegas algorithm
- always finds correct solution
- may not terminate

repeat k times:
    randomly select new element
    if 0 found:
        break

Monte Carlo algorithm
- always terminates
- may not find correct solution

RANDOMIZED OPTIMIZATION



WHEN SHOULD I USE RANDOMIZED ALGORITHMS?

When a particular input leads to bad behaviour

ALGORITHM DESIGN

- Brute force algorithms
- Randomized algorithms
- Greedy algorithms
- Dynamic programming
- Divide-and-conquer algorithms
- Backtracking algorithms

GREEDY ALGORITHMS

- At each step, you choose what seems most promising right now

- Accept that you won't necessarily find the best solution

THE COIN CHANGE PROBLEM

Given an amount and a set of coins, return the fewest number of coins that sum to the amount.

½ 1 2

5 10 20

$$\begin{array}{r} 37 \\ -20 \\ \hline 17 \end{array} \quad \begin{array}{r} 17 \\ -10 \\ \hline 7 \end{array} \quad \begin{array}{r} 7 \\ -5 \\ \hline 2 \end{array} \quad \begin{array}{r} 2 \\ -2 \\ \hline 0 \end{array}$$

$20+10+5+2 \Rightarrow 4$ coins

THE COIN CHANGE PROBLEM

The Danish monetary system is optimized for a greedy strategy.

1 15 25

Greedy

$30 = 25 + 1 + 1 + 1 + 1 + 1$

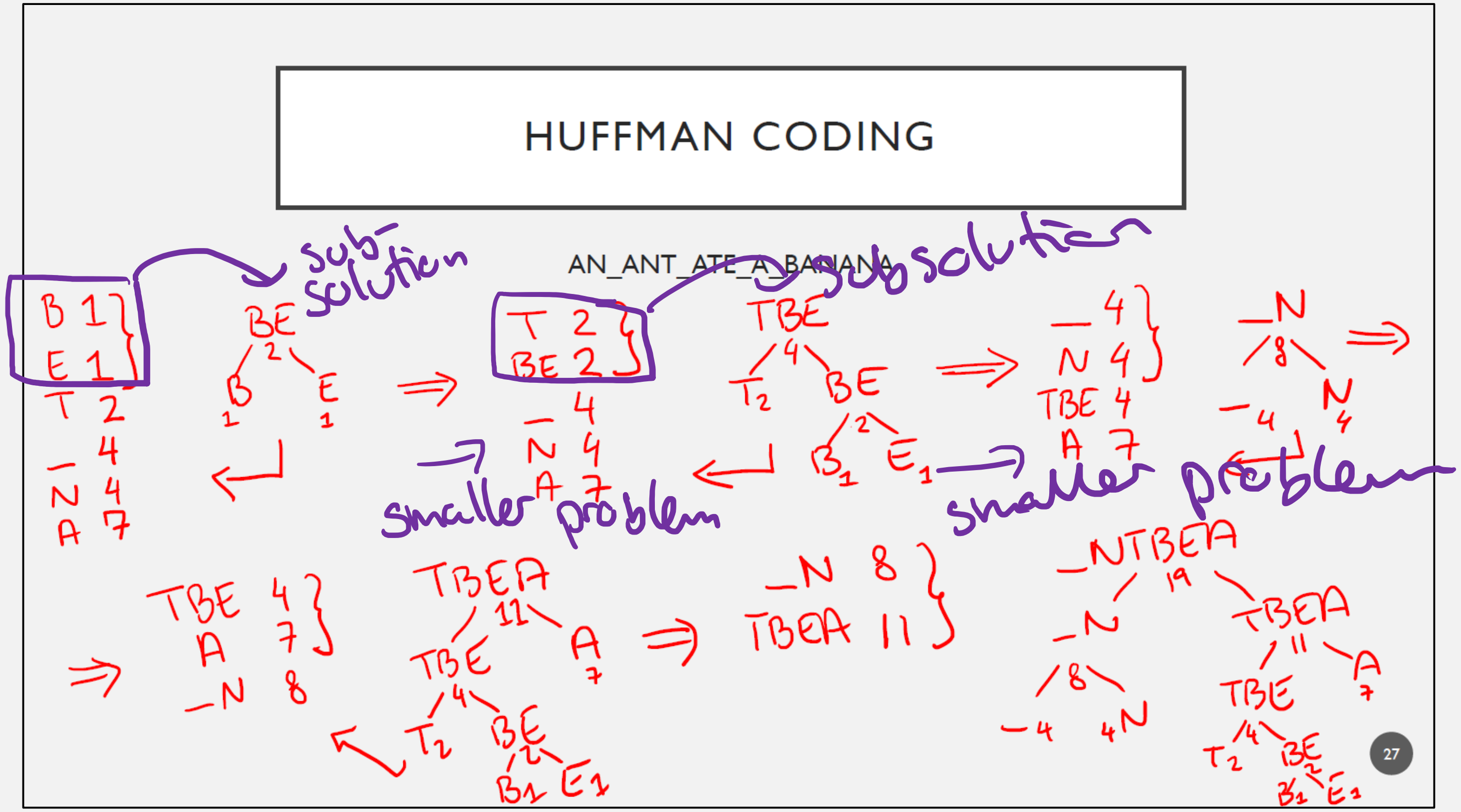
6 coins

Optimal

$30 = 15 + 15$

2 coins

HUFFMAN CODING



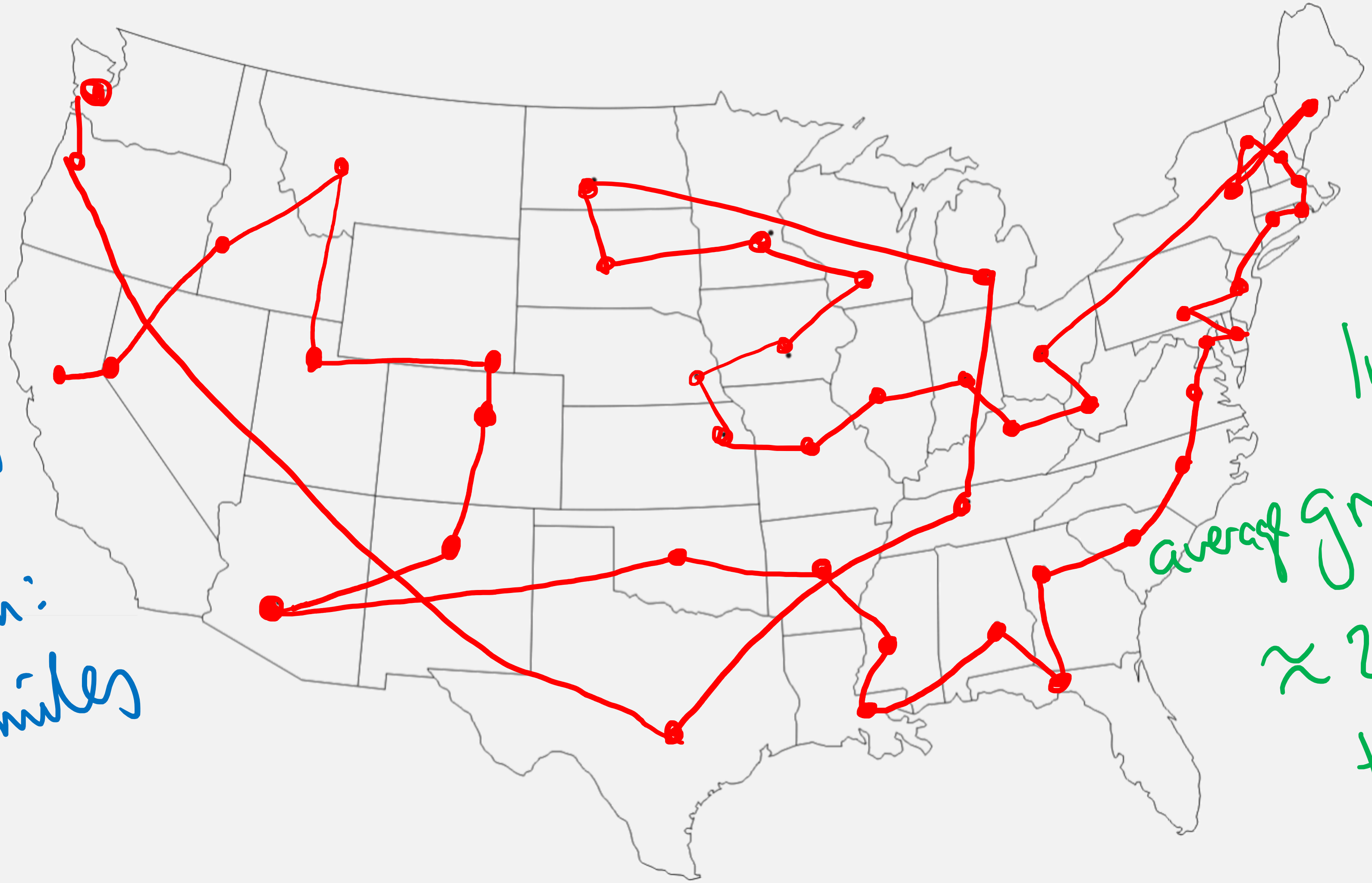
At each step, take the two least frequent symbols, disregard the rest → greedy

A GREEDY APPROACH TO THE TRAVELING SALESMAN PROBLEM

Best greedy
~15000 miles

Worst greedy
~19000 miles

Best known:
~12000 miles



In general
average greedy
≈ 25% longer
than optimal

WHEN SHOULD I USE GREEDY ALGORITHMS?

→ If an optimal solution can be found from a solution to a single smaller subproblem

OR

→ If you just need a "reasonable good" solution