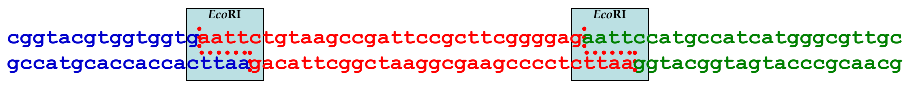
Lecture 4: DNA Restriction Mapping

Study Chapter 4.1-4.3

Recall Restriction Enzymes

(from High-Throughput Biology Lecture)

- Restriction enzymes break DNA whenever they encounter specific base sequences
- They occur reasonably frequently within long sequences (a 6-base sequence target appears, on average, 1:4096 bases)
- Can be used as molecular scissors


EcoRI
EcoRI
cggtacgtggtggtgaattctgtaagccgattccgcttcggggagaattccatgccatcatgggcgttgc
gccatgcaccaccacttaagacattcggctaaggcgaagcccctcttaaggtacggtagtacccgcaacg

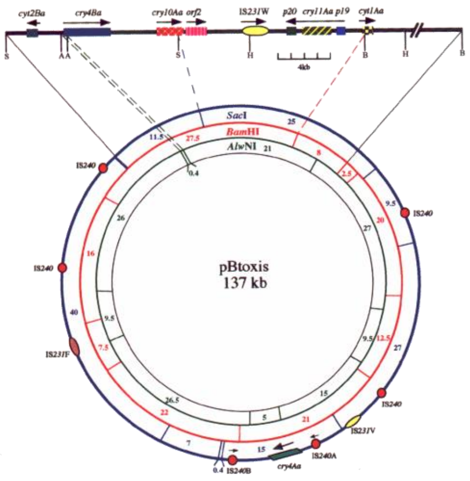
Restriction Enzyme Uses

- Recombinant DNA technology
  - make novel DNA constructs,
  - add fluorophores
  - add other probes
- Digesting DNA into pieces that can be efficiently and reliably replicated through PCR (Polymerase Chain Reaction)
- Cutting DNA for genotyping via Microarrays
- Sequence Cloning
  - Inserting sequences into a host cell, via vectors
- cDNA/genomic library construction
  - Coding DNA, is a byproduct of transcription
  - Targeted sequencing (ex. RRBS)
- DNA restriction mapping
  - A rough map of a DNA fragment

3

DNA Restriction Maps

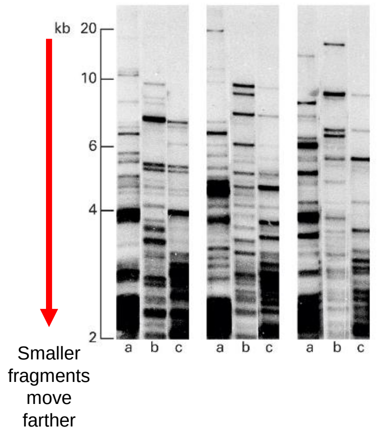
- A map of the restriction sites in a DNA sequence
- If the DNA sequence is known, then constructing a restriction map is trivial
- Restriction maps are a cheap alternative to sequencing for unknown sequences



4

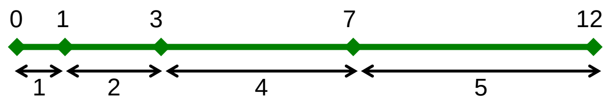
Consider the DNA Mapping Problem

- Begin with an isolated strand of DNA
- *Digest* it with restriction enzymes
  - Breaks strand into variable length fragments
- Use *gel electrophoresis* to sort fragments according to size
  - Can accurately sort DNA fragments that differ in length by a single nucleotide, and estimate their relative abundance
- Use fragment "lengths" to reassemble a map of the original strand

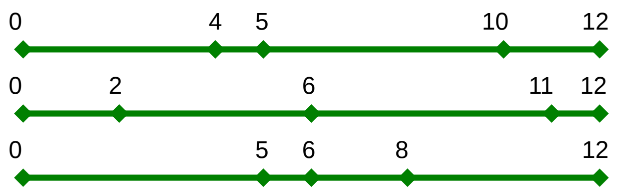


Single Enzyme Digestion

- What can be learned from a single complete digest?

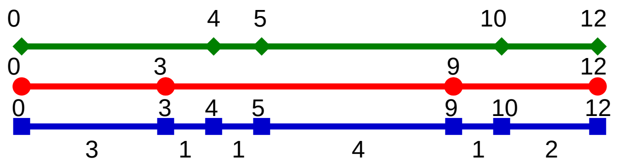


- Not much. There are many possible answers



Double Enzyme Digestion

- An alternative approach is to digest with two different enzymes in three stages
  - First, with restriction enzyme A
  - Second, with restriction enzyme B
  - Third, with both enzymes, A & B



- The inputs are three sets of restriction fragment lengths [1,2,4,5], [3,3,6], [1,1,1,2,3,4]

7

Double Digest Problem

- Given two sets of intervals on a common line segment between two disjoint interior point sets, and a third set of intervals between all points, reconstruct the positions of the points.

Input:

dA – fragment lengths from the digest with enzyme ***A***.

dB – fragment lengths from the digest with enzyme ***B***.

dX – fragment lengths from the digest with *both* ***A*** and ***B***.

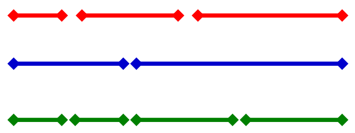
Output: ***A*** – location of the cuts for the enzyme ***A***.

B – location of the cuts for the enzyme ***B***.

8

Class Exercise

- Suppose you are asked to assemble a map from three digests
  - A = [1,2,3]
  - B = [2,4]
  - AB = [1,1,2,2]
- How do you solve for the map?
- How do you state your strategy as a general purpose algorithm?



9

Set Permutations

- Given a set [A,B,C,D] find all permutations

[A,B,C,D]	[B,A,C,D]	[C,A,B,D]	[D,A,B,C]
[A,B,D,C]	[B,A,D,C]	[C,A,D,B]	[D,A,C,B]
[A,C,B,D]	[B,C,A,D]	[C,B,A,D]	[D,B,A,C]
[A,C,D,B]	[B,C,D,A]	[C,B,D,A]	[D,B,C,A]
[A,D,B,C]	[B,D,A,C]	[C,D,A,B]	[D,C,A,B]
[A,D,C,B]	[B,D,C,A]	[C,D,B,A]	[D,C,B,A]

- How many?
  - 1st choice = n
  - 2nd choice = n-1
  - 3rd choice = n-2

N! permutations of N elements

10! = 3628800
24! = 620448401733239439360000

10

A Brute Force Solution

- Test all permutations of A and B checking they are compatible with some permuation of AB

```
def doubleDigest(seta, setb, setab, circular = False):
    a = Permute(seta)
    while (a.permutationsRemain()):
        ab = Permute(setab)
        while (ab.permutationsRemain()):
            if compatible(a.order, ab.order):
                b = Permute(setb)
                while (b.permutationsRemain()):
                    if (circular):
                        for i in xrange(len(setab)):
                            abShift = shift(ab.order, i)
                            if compatible(b.order, abShift):
                                return (a.order, b.order, ab.order, i)
                    else:
                        if compatible(b.order, ab.order):
                            return (a.order, b.order, ab.order, 0)
    return (aState, bState, abState, -1)
```

len(a)!

len(b)!

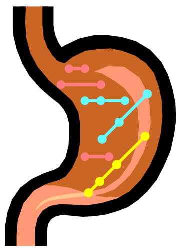
len(ab)!

How to Improve Performance?

- What strategy can we use to solve the double restriction map problem faster?
- Is there a branch-and-bound strategy?
  - Does the given code *really* test every permutation?
  - How does compatible( ) help?
  - Does the order of the loops help?
- Could you do all permutations of A and B, then compute the intervals and compare to AB?
- The double digest problem is truly a hard problem (NP-complete). No one knows an algorithm whose execution time does not grow slower than some exponent in the size of the inputs.

Partial Digestion Problem

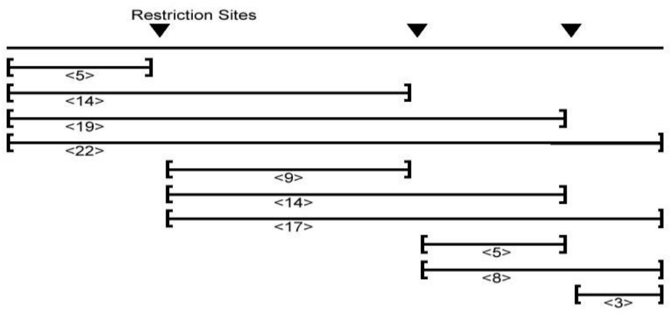
- Another way to construct a restriction map
- Expose DNA to the restriction enzyme for a limited amount of time to prevent it from cutting at all restriction sites (partial digestion)
- Generates the set of all possible restriction fragments between every pair of (not necessarily consecutive) points
- The set of fragment sizes is used to determine the positions of the restriction sites
- We assume that the multiplicity of a repeated fragment can be determined, i.e., multiple restriction fragments of the same length can be determined (e.g., by observing twice as much fluorescence for a double fragment than for a single fragment)



Partial Digestion Illustration

- A complete set of pairwise distances between points. In the following example a set of 10 fragments is generated.

L = {3, 5, 5, 8, 9, 14, 14, 17, 19, 22}



Pairwise Distance Matrix

- Often useful to consider partial digests in a distance matrix form
- Each entry is the distance between a pair of point positions labeled on the rows and columns
- The distance matrix for *n* points has *n(n-1)/2* entries, therefore we expect that many digest values as inputs
- Largest value in L establishes the segment length
- Actual non-zero point values are a subset of L

	0	5	14	19	22
0	-	5	14	19	22
5		-	9	14	17
14			-	5	8
19				-	3
22					-

15

Partial Digest Problem

- Given all pairwise distances between points on a line, reconstruct the positions of those points.

Input: A multiset of pairwise distances L, containing $\frac{n(n-1)}{2}$ elements

Output: A set X, of *n* integers, such that the set of pairwise distances ΔX = L

16

Homometric Solutions

	0	1	3	4	5	7	12	13	15
0		1	3	4	5	7	12	13	15
1			2	3	4	6	11	12	14
3				1	2	4	9	10	12
4					1	3	8	9	11
5						2	7	8	10
7							5	6	8
12								1	3
13									2
15									

	0	1	3	8	9	11	12	13	15
0		1	3	8	9	11	12	13	15
1			2	7	8	10	11	12	14
3				5	6	8	9	10	12
8					1	3	4	5	7
9						2	3	4	6
11							1	2	4
12								1	3
13									2
15									

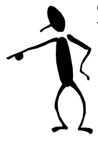
- The solution of a PDP is not always unique
- Two distinct point sets, A and B, can lead to indistinguishable distance multisets, ΔA = ΔB

17

Brute Force PDP Algorithm

- Basic idea: Construct all combinations of *n* - 2 integers between 0 and max(L), and check to see if the pairwise distances match.

Compare this Python code to the pseudocode on page 88 in the book



```
def bruteForcePDP(L, n):
    L.sort()
    M = max(L)
    X = intsBetween(0,M,n-2)
    while (X.combinationsRemain()):
        dX = allPairsDist(X.intSet())
        dX.sort()
        if (dX == L):
            print "X =", X.intSet()
```

18

Set Combinations

- Combinations of A things taken B at a time
- Order is unimportant
  [A,B,C] ≡ [A,C,B] ≡ [B,A,C] ≡ [B,C,A] ≡ [C,A,B] ≡ [C,B,A]
- All combinations of *n* items in *k* positions
  [1,1,0,0], [1,0,1,0],[1,0,0,1],[0,1,1,0],[0,1,0,1],[0,0,1,1]
- Smaller than a factorial

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Interesting relation $\sum_{k=0}^{n} \binom{n}{k} = 2^n$

19

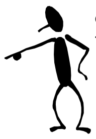
BruteForcePDP Performance

- BruteForcePDP takes $O(\boldsymbol{max(L)}^{\,n-2})$ time since it must examine all possible sets of positions.
- The problem scales with the size of the largest pairwise distance
- Suppose we multiply each element in *L* by a constant factor?
- Should we consider *every* possible combination of *n* - 2 points? (Consider our observations concerning distance matrices)

20

Another Brute Force PDP Approach

- Recall that the actual point values are a subset of L's values. Thus, rather than consider all combinations of possible points, we need only consider $n - 2$ combinations of values from L.



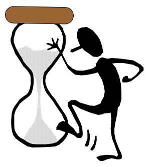
Compare this Python code to the pseudocode on page 88 in the book

```
def anotherBruteForcePDP(L, n):
    L.sort()
    M = max(L)
    X = intsFromL(L,n-2)
    while (X.combinationsRemain()):
        dX = allPairsDist(X.intSet())
        dX.sort()
        if (dX == L):
            print "X = ", X.intSet()
```

21

Efficiency of AnotherBruteForcePDP

- It's more efficient, but still slow
- If $\boldsymbol{L} = \{2, 998, 1000\}$ ($\boldsymbol{n} = 3$, $\boldsymbol{M} = 1000$), BruteForcePDP will be extremely slow, but AnotherBruteForcePDP will be quite fast
- Fewer sets are examined, but runtime is still exponential: $O(\boldsymbol{n}^{2n-4})$
- Is there a better way?



22

A Practical PDP Algorithm

1. Begin with ***X*** = {0}
2. Remove the largest element in ***L*** and place it in ***X***
3. See if the element *fits* on the right or left side of the restriction map
4. When it fits, find the other lengths it creates and remove those from ***L***
5. Go back to step 3 until ***L*** is empty

Defining delta(y, X)

- Before describing PartialDigest, we first define a helper function:

  delta(***y***, ***X***)

  as the multiset of all distances between point ***y*** and the points in the set ***X***

  $$\text{delta}(y, X) = \{|y - x_1|, |y - x_2|, \ldots, |y - x_n|\}$$

  ex. [3,6,11] = delta(8,[5,14,19])

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0 }

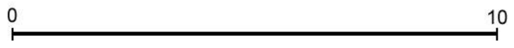
An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0 }

Remove 10 from ***L*** and insert it into ***X***. We know this must be the total length of the DNA sequence because it is the largest fragment.

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 10 }

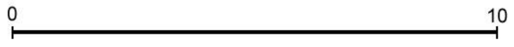


27

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 10 }

Remove 8 from ***L*** and make ***y*** = 2 or 8. But since the two cases are symmetric, we can assume ***y*** = 2.

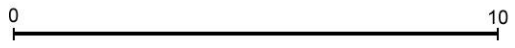


28

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 10 }

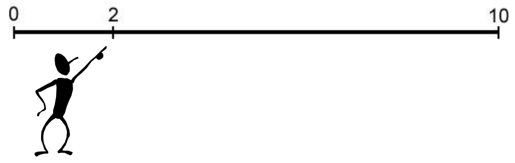
Find the distances from $y = 2$ to other elements in X.
D($\boldsymbol{y}$, $\boldsymbol{X}$) = {8, 2}, so we remove {8, 2} from $\boldsymbol{L}$ and add 2 to $\boldsymbol{X}$.



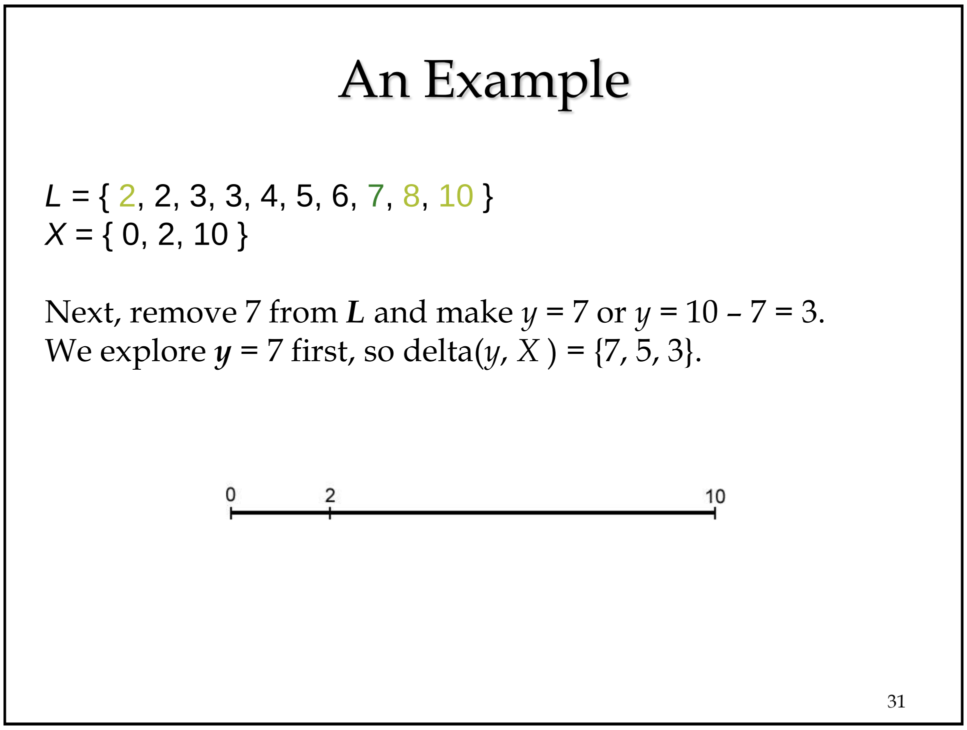
29

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 10 }



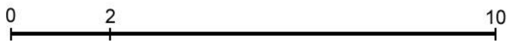
30



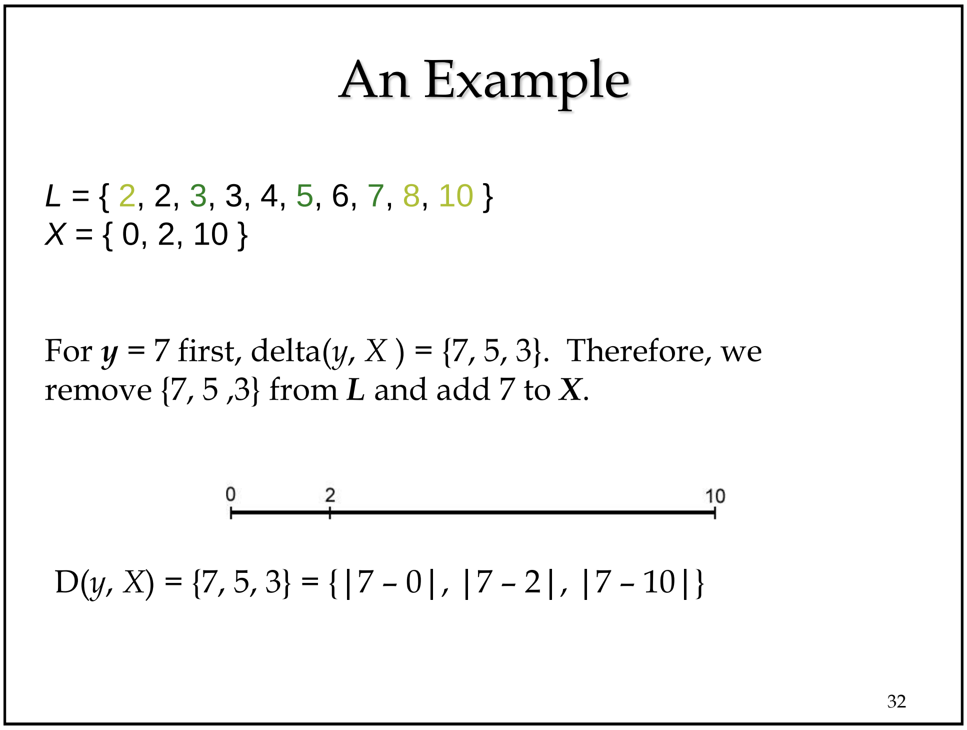
An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 10 }

Next, remove 7 from ***L*** and make *y* = 7 or *y* = 10 – 7 = 3.
We explore ***y*** = 7 first, so delta(*y*, X) = {7, 5, 3}.



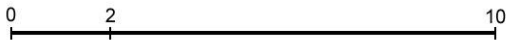
31



An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 10 }

For ***y*** = 7 first, delta(*y*, X) = {7, 5, 3}. Therefore, we remove {7, 5 ,3} from ***L*** and add 7 to ***X***.

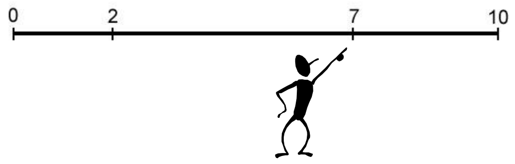


D(*y*, *X*) = {7, 5, 3} = {|7 – 0|, |7 – 2|, |7 – 10|}

32

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 7, 10 }

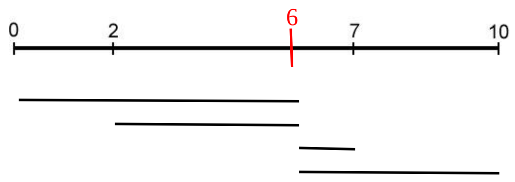


33

An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 7, 10 }

Next, take 6 from ***L*** and make ***y*** = 6. Unfortunately, delta(*y*, *X*) = {6, 4, 1 ,4}, which is not a subset of ***L***. Therefore, we won't explore this branch.

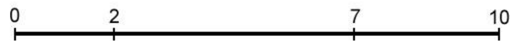


34

An Example

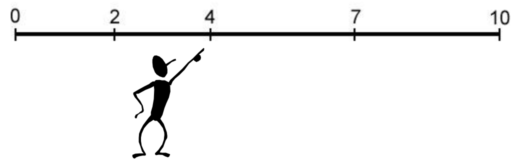
L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 7, 10 }

This time make *y* = 4. delta(*y*, *X*) = {4, 2, 3 ,6}, which is a subset of ***L*,** so we explore this branch. We remove {4, 2, 3 ,6} from ***L*** and add 4 to ***X***.



An Example

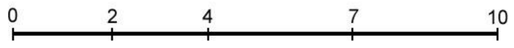
L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 4, 7, 10 }



An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 4, 7, 10 }

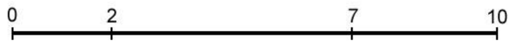
L is now empty, so we have a solution, which is *X*.



An Example

L = { 2, 2, 3, 3, 4, 5, 6, 7, 8, 10 }
X = { 0, 2, 7, 10 }

To find other solutions, we backtrack (remove old insertions and try different ones).



Implementation

```
def partialDigest(L):
    width = max(L)
    L.remove(width)
    X = [0, width]
    Place(L, X)

def Place(L, X):
    if (len(L) == 0):
        print(sorted(X))
        return
    y = max(L)
    dyX = delta(y, X)
    if (dyX.subset(L)):
        X.append(y)
        for x in dyX.items:
            L.remove(x)
        Place(L, X)
        X.remove(y);
        for x in dyX.items:
            L.append(x)
    w = max(X) - y
    dwX = delta(w, X)
    if (dwX.subset(L)):
        X.append(w)
        for x in dwX.items:
            L.remove(x)
        Place(L, X)
        X.remove(w)
        for x in dwX.items:
            L.append(x)
    return
```

Checks distances from the "0" end

Checks distances from the "width" end

This PDP algorithm outputs all solutions. In fact, it might even repeat solutions

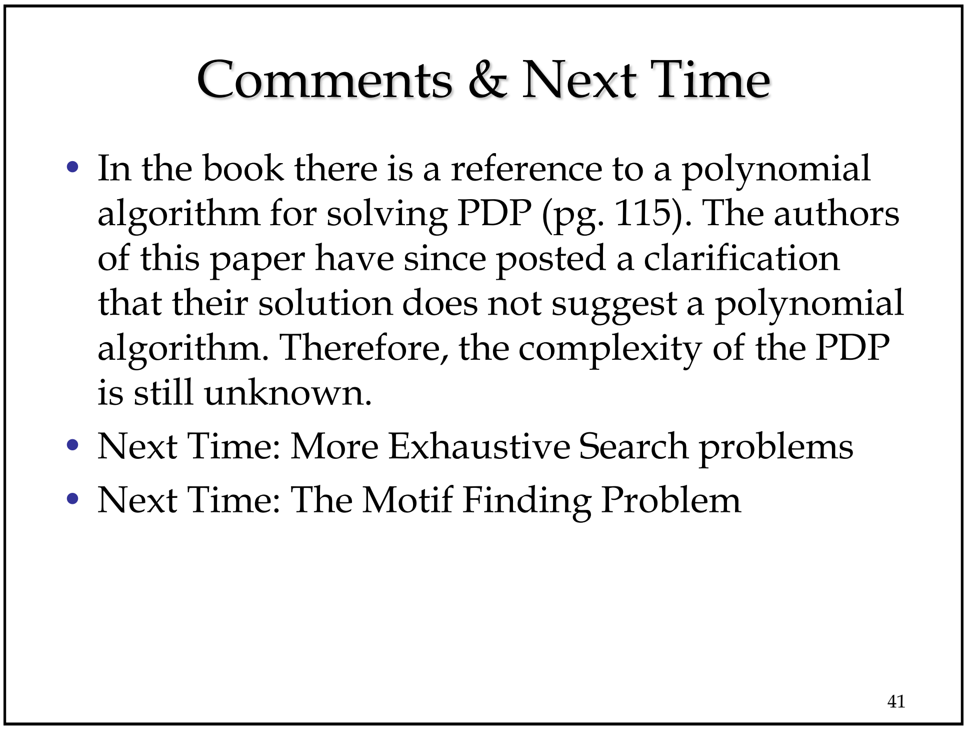
Analysis

- Let $T(n)$ be the maximum time that partialDigest takes to solve an n-point instance of PDP
- If, at every step, there is only one viable solution, then partialDigest reduces the size of the problem by one on each recursive call

$$T(n) = T(n-1) + O(n) \rightarrow O(n^2)$$

- However, if there are two alternatives then

$$T(n) = 2T(n-1) + O(n) \rightarrow O(2^n)$$



Comments & Next Time

- In the book there is a reference to a polynomial algorithm for solving PDP (pg. 115). The authors of this paper have since posted a clarification that their solution does not suggest a polynomial algorithm. Therefore, the complexity of the PDP is still unknown.
- Next Time: More Exhaustive Search problems
- Next Time: The Motif Finding Problem

41