Taking Stock

- Finished basic chapters (Chapters 1-4)

Taking Stock

- Finished basic chapters (Chapters 1-4)
- What's Left?
  - Graphs (Chapters 5 & 6).
  - Algorithmic Strategies (Chapters 7 (Comb Search) & 8 (Dynamic Programming))
  - Chapter 9 (NP Completeness)

Algorithmic Strategies

1. Divide and Conquer.
2. Greedy.
3. Exhaustive Search.
4. Dynamic Programming.
5. Backtracking.

Dynamic Programming

- Dynamic programming is a very powerful, general tool for solving optimization problems on left-right-ordered items such as character strings.

Dynamic Programming

- Dynamic programming is a very powerful, general tool for solving optimization problems on left-right-ordered items such as character strings.
- Once understood it is relatively easy to apply, it looks like magic until you have seen enough examples.

Greedy Algorithms

- *Greedy* algorithms focus on making the best local choice at each decision point.

Greedy Algorithms

- *Greedy* algorithms focus on making the best local choice at each decision point.
- In the absence of a correctness proof such greedy algorithms are very likely to fail.

Greedy Strategy for TSP

A popular solution starts at some point p_0 and then walks to its nearest neighbor p_1 first, then repeats from p_1, etc. until done.

Greedy Strategy for TSP

A popular solution starts at some point p_0 and then walks to its nearest neighbor p_1 first, then repeats from p_1, etc. until done.

Pick and visit an initial point p_0
$p = p_0$
$i = 0$
While there are still unvisited points
    $i = i + 1$
    Let p_i be the closest unvisited point to p_{i-1}
    Visit p_i
Return to p_0 from p_i

Dynamic Programming

Dynamic programming gives us a way to design custom algorithms which

1. Systematically search all possibilities (thus guaranteeing correctness) while

Dynamic Programming

Dynamic programming gives us a way to design custom algorithms which

1. Systematically search all possibilities (thus guaranteeing correctness) while
2. storing partial results to avoid recomputing (thus providing efficiency).

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1}, a_1 = 2$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1}, a_1 = 2 \longrightarrow a_n = 2^n$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1}, a_1 = 2 \longrightarrow a_n = 2^n$
- Factorials: $a_n = na_{n-1}, a_1 = 1$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1}, a_1 = 2 \longrightarrow a_n = 2^n$
- Factorials: $a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n!$

Recurrence Relations

A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: $a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1}, a_1 = 2 \longrightarrow a_n = 2^n$
- Factorials: $a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n!$

Computer programs can easily evaluate the value of a given recurrence even without the existence of a nice closed form.

Computing Fibonacci Numbers

Definition:

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$

Computing Fibonacci Numbers

Definition:

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$$

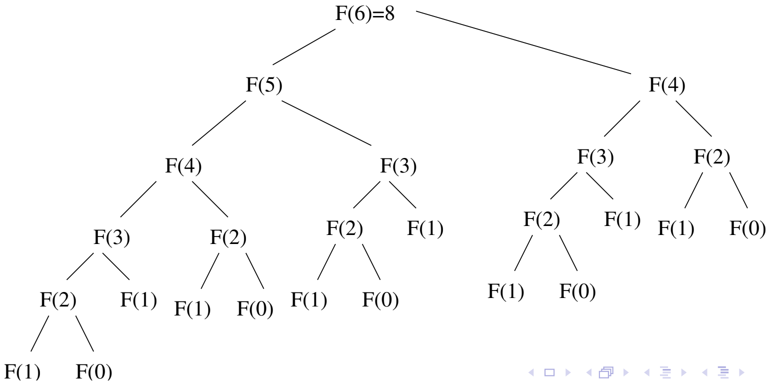
Recursive Algorithm:

```
int F(int i) {
  if (i == 0) return 0;
  if (i == 1) return 1;
  return F(i-1) + F(i-2);
}
```

Run Fibonacci code

Perils of Recursive Algorithms

Implementing as a recursive procedure is easy, but slow because we repeat calculations.



How Slow?

- $$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$$

How Slow?

- $$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$$
- $F_n \approx 1.6^n$.

How Slow?

- $$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$$
- $F_n \approx 1.6^n$.
- Since our recursion tree has 0 and 1 as leaves, computing F_n requires $>> 1.6^n$ calls!

How Slow?

- $$F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$$
- $F_n \approx 1.6^n$.
- Since our recursion tree has 0 and 1 as leaves, computing F_n requires $>> 1.6^n$ calls!
- Each call is $O(1)$, so exp time algorithm!

What about Dynamic Programming?

We can calculate F_n in linear time by storing values in array $F[0..n]$:

$F[0] = 0$
$F[1] = 1$
For $i = 2$ to n
    $F[i] = F[i-1] + F[i-2]$

What about Dynamic Programming?

We can calculate F_n in linear time by storing values in array $F[0..n]$:

$F[0] = 0$
$F[1] = 1$
For $i = 2$ to n
    $F[i] = F[i-1] + F[i-2]$

Moral: we traded space for time.

But "I LIKE Recursive Algorithms"

But “I LIKE Recursive Algorithms”

```
int F(int i) {
  if F[i] has “already been computed”, return F[i];
  if (i == 0) F[i] = 0;
  if (i == 1) F[i] = 1;
  F[i] = F(i-1) + F(i-2); // recursive calls
  return F[i];
}
```

Avoiding Recomputation by Storing Partial Results

- The trick to dynamic programming is to see that the naive recursive algorithm computes the same subproblems over and over again.

Avoiding Recomputation by Storing Partial Results

- The trick to dynamic programming is to see that the naive recursive algorithm computes the same subproblems over and over again.
- If so, storing the answers in a table instead of recomputing can lead to an efficient algorithm.

Avoiding Recomputation by Storing Partial Results

- The trick to dynamic programming is to see that the naive recursive algorithm computes the same subproblems over and over again.
- If so, storing the answers in a table instead of recomputing can lead to an efficient algorithm.
- Thus we must first hunt for a correct recursive algorithm – later we can worry about speeding it up by using a results matrix.

Binomial Coefficients

An important class of counting numbers are the *binomial coefficients*, where $\binom{n}{k}$ counts the number of ways to choose k things from n possibilities.

- *Committees* – How many ways are there to form a k-member committee from n people? By definition, $\binom{n}{k}$.

Binomial Coefficients

An important class of counting numbers are the *binomial coefficients*, where $\binom{n}{k}$ counts the number of ways to choose k things from n possibilities.

- *Committees* – How many ways are there to form a k-member committee from n people? By definition, $\binom{n}{k}$.
- *Paths Across a Grid* – How many ways are there to travel from the upper-left corner of an $n \times m$ grid to the lower-right corner by walking only down and to the right?

Number of Paths in a Grid

Every path must consist of $n + m$ steps, n downward and m to the right, so there are $\binom{n+m}{n}$ such sets/paths.

Computing Binomial Coefficients

Since

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

you can, in principle, compute them straight from factorials.

Computing Binomial Coefficients

Since

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

you can, in principle, compute them straight from factorials.

However, intermediate calculations can *easily* cause arithmetic overflow even when the final coefficient fits comfortably within an int.

Pascal's Recurrence

A more stable way to compute binomial coefficients is using the recurrence relation:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

It works because the *n*th element either appears or does not appear in one of the $\binom{n}{k}$ subsets of k elements.

Basis Cases

- How many ways are there to choose 0 things from a set?

Basis Cases

- How many ways are there to choose 0 things from a set?
- The right term of the sum drives us up to $\binom{k}{k}$. How many ways are there to choose k things from a k-element set?

Binomial Coefficient Worksheet

Implementation: Demo on Board

```
int binomial_coefficient(int n,int k)
{
      int bc[n+1][k+1]; // table of binomial coefficients

      for (int i=0; i≤n; i++) bc[i][0] = 1;
      for (int j=0; j≤k; j++) bc[j][j] = 1;

      for (i=1; i<=n; i++)
            for (j=1; j<i and j ≤ k; j++)
                  bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

      return( bc[n][k] );
}
```

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm.

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence so you always have what you need.

What's Next?

Next, we will study the application of dynamic programming to optimization problems.

Edit Distance

Misspellings make *approximate pattern matching* an important problem

Edit Distance

Misspellings make *approximate pattern matching* an important problem

If we are to deal with inexact string matching, we must first define a cost function telling us how far apart two strings are, i.e., a distance measure between pairs of strings.

Edit Distance

Misspellings make *approximate pattern matching* an important problem

If we are to deal with inexact string matching, we must first define a cost function telling us how far apart two strings are, i.e., a distance measure between pairs of strings.

A reasonable distance measure minimizes the cost of the *changes* which have to be made to convert source string to target.

String Edit Operations

There are three natural types of changes:

- Substitution/Match: Change a single character from source s to a different character in text t, such as changing "shot" to "spot".

String Edit Operations

There are three natural types of changes:

- Substitution/Match: Change a single character from source s to a different character in text t, such as changing "shot" to "spot".
- Insertion: Insert a single character into source s to help it match target t, such as changing "ago" to "agog".

String Edit Operations

There are three natural types of changes:

- Substitution/Match: Change a single character from source s to a different character in text t, such as changing "shot" to "spot".
- Insertion: Insert a single character into source s to help it match target t, such as changing "ago" to "agog".
- Deletion: Delete a single character from source s to help it match target t, such as changing "hour" to "our".

Visualization Examples

Why is this an Optimization Problem?

Convert IAGO to AGOG

Why is this an Optimization Problem?

Convert IAGO to AGOG

Solution 1					Solution 2			
I	A	G	O	-	I	A	G	O
D	M	M	M	I	S	S	S	S
-	A	G	O	G	A	G	O	G

Why is this an Optimization Problem?

Convert IAGO to AGOG

Solution 1					Solution 2			
I	A	G	O	-	I	A	G	O
D	M	M	M	I	S	S	S	S
-	A	G	O	G	A	G	O	G

$\text{Cost}(\text{Solution 1}) = 2$

$\text{Cost}(\text{Solution 2}) = 4$

Recursive Algorithm

We can compute the edit distance with a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.

Recursive Algorithm

We can compute the edit distance with a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.
If we knew the cost of editing the three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly.

Recursive Algorithm

We can compute the edit distance with a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.

If we knew the cost of editing the three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly.

We can learn this cost, through the magic of recursion.

Recursive Edit Distance Code

```
const int MATCH = 0 // enumerated type symbol for match/substitute
const int INSERT = 1 // enumerated type symbol for insert
const int DELETE = 2 // enumerated type symbol for delete

int string_compare(char *s, char *t, int i, int j)
{
      int option[3]; // cost of the three options
      int lowest_cost;

      if (i == 0) return j; //  DPM: explain
      if (j == 0) return i;

      option[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
      option[INSERT] = string_compare(s,t,i,j-1) + 1;
      option[DELETE] = string_compare(s,t,i-1,j) + 1;

      lowest_cost = option[MATCH];
      if (option[INSERT] < lowest_cost) lowest_cost = option[INSERT];
      if (option[DELETE] < lowest_cost) lowest_cost = option[DELETE];
      return( lowest_cost ); }
```

Speeding it Up

- This program is absolutely correct but takes exponential time because it recomputes values again and again and again!

Speeding it Up

- This program is absolutely correct but takes exponential time because it recomputes values again and again and again!
- But there can only be $|s| \cdot |t|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the parameters of recursive calls.

Speeding it Up

- This program is absolutely correct but takes exponential time because it recomputes values again and again and again!
- But there can only be $|s| \cdot |t|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the parameters of recursive calls.
- By storing the values for each of these (i, j) pairs in a table, we can avoid recomputing them and just look them up as needed.

The Dynamic Programming Table

The table is a two-dimensional matrix *m* where each cell contains the cost of the optimal solution of this subproblem, and a parent pointer explaining how we got to this location:

```
class cell {
        public:
                int cost; // cost of reaching this cell
                int parent; // parent cell
};

cell m[MAXLEN+1][MAXLEN+1];
```

Differences between Recursive & Dynamic Programming Versions

- First, it gets its intermediate values using table lookup instead of recursive calls.

Differences between Recursive & Dynamic Programming Versions

- First, it gets its intermediate values using table lookup instead of recursive calls.
- Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit-sequence later.

Evaluation Order

- To determine the value of cell (i, j) we need three values to be sitting and waiting for us, namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$.

Evaluation Order

- To determine the value of cell (i, j) we need three values to be sitting and waiting for us, namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$.
- Any evaluation order with this property will do, including the row-major order used in this program.

Dynamic Programming Edit Distance

```
int string_compare(char *s, char *t)
{
    int option[3];// cost of the three options

    initialize zeroth column;// DPM: brief explanation
    initialize zeroth row;

    for (i=1; i<strlen(s); i++)
        for (int j=1; j<strlen(t); j++) {
            option[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
            option[INSERT] = m[i][j-1].cost + 1;
            option[DELETE] = m[i-1][j].cost + 1;

            m[i][j].cost = option[MATCH]; // Rest is about finding min
            m[i][j].parent = MATCH;
            if (option[INSERT] < m[i][j].cost) {
                m[i][j].cost = option[INSERT];
                m[i][j].parent = INSERT;
            }
            .... Repeat for DELETE
```

Example

Check out
http://www.planatscher.net/bioinfodemos/

	T	a	g	o	g
S	0	1	2	3	4
i	1	1	2	3	4
a	2	1	2	3	4
g	3	2	1	2	3
o	4	3	2	1	2

Reconstructing the Path

- The dynamic programming implementation above returns the cost of the optimal solution, but not the solution itself.

Reconstructing the Path

- The dynamic programming implementation above returns the cost of the optimal solution, but not the solution itself.
- Solution is described by a path through the table, starting from the initial configuration (the pair of empty strings $(0, 0)$) down to the final goal state (the pair of full strings $(|s|, |t|)$).

Reconstructing the Path (2)

- Reconstructing these decisions is done by walking backward from the goal state, following `parent` pointers.

Reconstructing the Path (2)

- Reconstructing these decisions is done by walking backward from the goal state, following `parent` pointers.
- The `parent` field for `m[i,j]` tells us whether the edit at (i,j) was `MATCH`, `INSERT`, or `DELETE`.

Reconstructing the Path (2)

- Reconstructing these decisions is done by walking backward from the goal state, following `parent` pointers.
- The `parent` field for `m[i,j]` tells us whether the edit at (i, j) was `MATCH`, `INSERT`, or `DELETE`.
- if `MATCH`, look at $m[i, j]$ and $m[i-1, j-1]$ to determine if it's a match or a substitute.

Difference between Optimization v/s Non-Optimization Problems

- The need to make decisions in addition to simple computations.
- Decision-making translates into an optimization construct (e.g., `min` or `max`) in the recurrence relation.
- The need to keep track of decisions and reconstruct solution from them.

Longest Common Subsequence

The *longest common subsequence* (not substring) between "democrat" and "republican" is eca.

Longest Common Subsequence

The *longest common subsequence* (not substring) between “democrat” and “republican” is eca.
A common subsequence is defined by all the identical-character matches in an edit trace.

Longest Common Subsequence

The *longest common subsequence* (not substring) between "democrat" and "republican" is eca.
A common subsequence is defined by all the identical-character matches in an edit trace.
To maximize the number of such matches, we must prevent substitution of non-identical characters.

```
int match(char c, char d)
{
        if (c == d) return(0);
        else return(MAXLEN); // used to be 1
}
```

Maximum Monotone Subsequence (MMS)

A numerical sequence is *monotonically increasing* if the ith element is larger than the $(i-1)$st element.

Maximum Monotone Subsequence (MMS)

A numerical sequence is *monotonically increasing* if the ith element is larger than the $(i-1)$st element. The MMS problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.

Maximum Monotone Subsequence (MMS)

A numerical sequence is *monotonically increasing* if the ith element is larger than the $(i-1)$st element. The MMS problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.
One solution to MMS("243517698") is "23568."

Any ideas?

Reduction to LCS

In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.

MMS("243517698") $=$
LCS("243517698" , "123456789")

Dividing the Work

The job of scanning in a shelf of books is to be split between k workers. To avoid rearranging the books or separating them into piles, we divide the shelf into k regions and assign each region to a worker.

Dividing the Work

The job of scanning in a shelf of books is to be split between k workers. To avoid rearranging the books or separating them into piles, we divide the shelf into k regions and assign each region to a worker.

What is the fairest way to divide the shelf up?

Example: $k = 3$

If each book is the same length, partition equally:

100 100 100 | 100 100 100 | 100 100 100

Example: $k = 3$

If each book is the same length, partition equally:

$$100\ 100\ 100 \mid 100\ 100\ 100 \mid 100\ 100\ 100$$

But what if the books are not the same length?

$$100\ 200\ 300 \mid 400\ 500\ 600 \mid 700\ 800\ 900$$

Example: $k = 3$

If each book is the same length, partition equally:

$$100\ 100\ 100 \mid 100\ 100\ 100 \mid 100\ 100\ 100$$

But what if the books are not the same length?

$$100\ 200\ 300 \mid 400\ 500\ 600 \mid 700\ 800\ 900$$

Which part would you volunteer to do?!!

Example: $k = 3$

If each book is the same length, partition equally:

$$100\ 100\ 100 \mid 100\ 100\ 100 \mid 100\ 100\ 100$$

But what if the books are not the same length?

$$100\ 200\ 300 \mid 400\ 500\ 600 \mid 700\ 800\ 900$$

Which part would you volunteer to do?!!
How can we find the fairest possible partition, i.e.

$$100\ 200\ 300\ 400\ 500 \mid 600\ 700 \mid 800\ 900$$

The Linear Partition Problem

Input: A given arrangement S of nonnegative numbers $(s_1, \ldots, s_n)$ and an integer k.

Problem: Partition S into k ranges, so as to minimize the maximum sum over all the ranges.

The Linear Partition Problem

Input: A given arrangement S of nonnegative numbers $(s_1, \ldots, s_n)$ and an integer k.

Problem: Partition S into k ranges, so as to minimize the maximum sum over all the ranges.

Does taking the average value of a part $\sum_{i=1}^{n} s_i / k$ from the left always work?

Recursive Idea

Notice that the kth partition starts right after we place the $(k-1)$st divider. But, where can we place this last divider?

Recursive Idea

Notice that the kth partition starts right after we place the $(k-1)$st divider. But, where can we place this last divider?
A: Between the ith and $(i+1)$st books for some i.

What is the cost of this?

Recursive Idea(2)

The total cost will be the larger of two quantities,

1. (Easy) the cost of the last partition $\sum_{j=i+1}^{n} s_j$
2. (Not so easy) the cost of the largest partition formed to the left of i. How to proceed????

Recursive Idea(2)

The total cost will be the larger of two quantities,

1. (Easy) the cost of the last partition $\sum_{j=i+1}^{n} s_j$
2. (Not so easy) the cost of the largest partition formed to the left of i. How to proceed????

To figure this, we need to partition the elements $(s_1, \ldots, s_i)$ as equitably as possible among $k - 1$ ranges.

Recursive Idea(2)

The total cost will be the larger of two quantities,

1. (Easy) the cost of the last partition $\sum_{j=i+1}^{n} s_j$
2. (Not so easy) the cost of the largest partition formed to the left of i. How to proceed????

To figure this, we need to partition the elements $(s_1, \dots, s_i)$ as equitably as possible among $k-1$ ranges.

But isn't this just a smaller instance of the same problem!

Dynamic Programming Recurrence

Define $M[n, k]$ to be the minimum possible cost over all partitions of $(s_1, \ldots, s_n)$ into k ranges.

$$M[n, k] = \min_{i=1}^{n} \{ \max(M[i, k-1], \sum_{j=i+1}^{n} s_j) \}$$

with the basis cases of

$$M[1, k] = s_1 \text{ and} M[j, 1] = \sum_{i=1}^{j} s_i$$

Run Time

- Number of cells $\times$ run time per cell.
- A total of $k \cdot n$ cells in the table.
- Each cell depends on n others (see recurrence), and can be computed in linear time, for a total of $O(kn^2)$.

When can you use DP?

- DP computes recurrences efficiently by storing partial results; thus, it can only be efficient when there aren't too many partial results to compute!

When can you use DP?

- DP computes recurrences efficiently by storing partial results; thus, it can only be efficient when there aren't too many partial results to compute!
- DP works best on objects which are linearly ordered and cannot be rearranged; e.g., chars in a string, row of books.

When can you use DP?

- DP computes recurrences efficiently by storing partial results; thus, it can only be efficient when there aren't too many partial results to compute!
- DP works best on objects which are linearly ordered and cannot be rearranged; e.g., chars in a string, row of books.
- Whenever your objects are ordered in a left-to-right way, you should smell DP!