Dynamic Programming

Ashley Montanaro

Centre for Quantum Information and Foundations,
Department of Applied Mathematics and Theoretical Physics,
University of Cambridge

28 January 2013

Introduction

Dynamic programming is a way of finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

Introduction

Dynamic programming is a way of finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea:

- Start out with a problem you want to solve.
- Find a naïve exponential-time recursive algorithm.
- Speed up the algorithm by storing solutions to subproblems.
- Speed it up further by solving subproblems in a more efficient order.

Example: Fibonacci numbers

The Fibonacci numbers are defined as follows:

- $F_0 = 0$;
- $F_1 = 1$;
- $F_n = F_{n-1} + F_{n-2} \quad (n \geqslant 2)$.

They occur (for example) in biology. The first few are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Calculating the Fibonacci numbers

Imagine we want to calculate the n'th Fibonacci number F_n. The following algorithm is immediate from the definition:

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

Calculating the Fibonacci numbers

Imagine we want to calculate the n'th Fibonacci number F_n.
The following algorithm is immediate from the definition:

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

However, $F(n)$ has running time exponential in n!

Exercise: prove this.

Calculating the Fibonacci numbers

This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.

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

Calculating the Fibonacci numbers

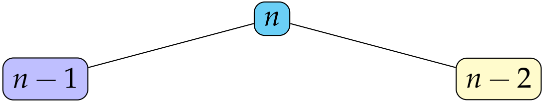
This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.

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

Calculating the Fibonacci numbers

This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.

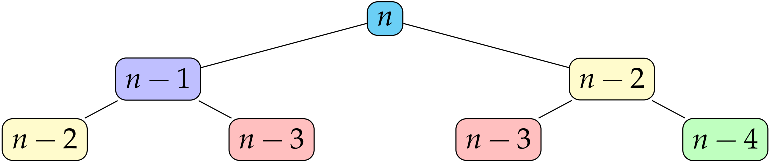
```
int F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```



Calculating the Fibonacci numbers

This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.

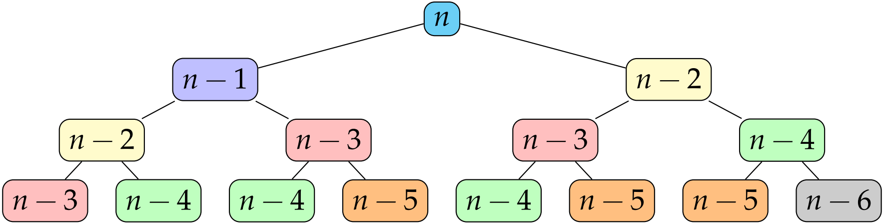
```
int F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```



Calculating the Fibonacci numbers

This naïve algorithm is inefficient: it repeatedly recomputes the answers to subproblems.

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



Improving the algorithm

We can make the algorithm more efficient by storing the results of these recursive calls.

```
int memo_F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (undefined(F[n]))
        F[n] = memo_F(n-1) + memo_F(n-2);
    return F[n];
}
```

This process is known as memoization.

The performance of this algorithm

What is the algorithm's running time now?

```
int memo_F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (undefined(F[n]))
        F[n] = memo_F(n-1) + memo_F(n-2);
    return F[n];
}
```

The performance of this algorithm

What is the algorithm's running time now?

```
int memo_F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (undefined(F[n]))
        F[n] = memo_F(n-1) + memo_F(n-2);
    return F[n];
}
```

- Each entry in the memory is only computed once, so there are only $O(n)$ integer additions.
- Each integer addition can be performed in time $O(n)$, so the total running time is $O(n^2)$.

Improving the algorithm further

Something a bit unnatural about this algorithm: the numbers are requested from the top down, but filled in from the bottom up.

```
int memo_F(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    if (undefined(F[n]))
        F[n] = memo_F(n-1) + memo_F(n-2);
    return F[n];
}
```

- That is, the F array is computed in the order $F[0], F[1], \ldots, F[n]$.
- This leads to an unnecessarily large number of recursive calls being made.

Improving the algorithm further

We can get rid of the recursion by simply computing the Fibonacci numbers in ascending order.

```
int asc_F(int n) {
    F[0] = 0;
    F[1] = 1;
    for (i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Improving the algorithm further

We can get rid of the recursion by simply computing the Fibonacci numbers in ascending order.

```
int asc_F(int n) {
    F[0] = 0;
    F[1] = 1;
    for (i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

- This algorithm clearly uses $O(n)$ additions and stores $O(n)$ integers.

Improving the algorithm further

We can get rid of the recursion by simply computing the Fibonacci numbers in ascending order.

```
int asc_F(int n) {
    F[0] = 0;
    F[1] = 1;
    for (i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

- This algorithm clearly uses $O(n)$ additions and stores $O(n)$ integers.
- This may be the natural algorithm one would come up with when first looking at the problem, but the point is that here we found it almost completely mechanically.

$F_n$al notes on Fibonacci numbers

Although this problem was very simple, it illustrates the basic concepts behind dynamic programming:

1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
2. Write down a naïve recursive algorithm based on this presentation.
3. Memoize the recursive algorithm.
4. Finally, restructure the algorithm to compute solutions in an efficient order.

$F_n$al notes on Fibonacci numbers

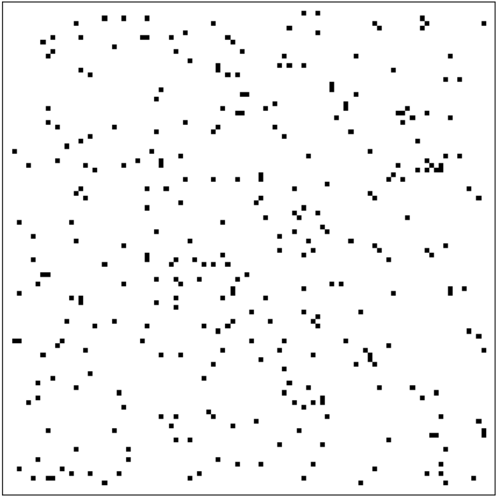
Although this problem was very simple, it illustrates the basic concepts behind dynamic programming:

1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
2. Write down a naïve recursive algorithm based on this presentation.
3. Memoize the recursive algorithm.
4. Finally, restructure the algorithm to compute solutions in an efficient order.

Exercise: give an improved algorithm which computes F_n in time $o(n^2)$.

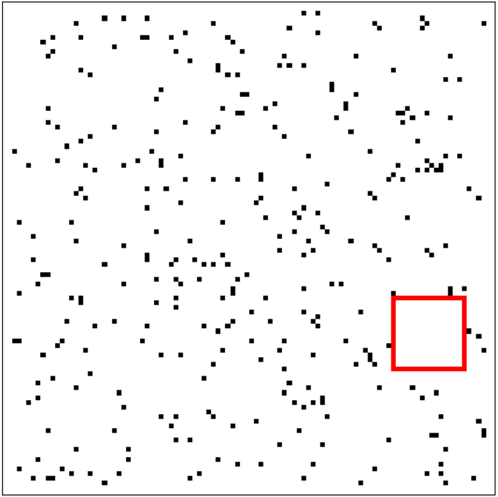
Example: largest empty square

Consider the following problem: given an $n \times n$ monochrome image, find the largest empty square, i.e. square avoiding any black points.



Example: largest empty square

Consider the following problem: given an $n \times n$ monochrome image, find the largest empty square, i.e. square avoiding any black points.



Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

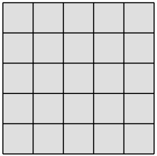
- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:

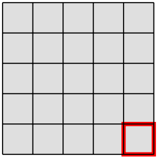


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:

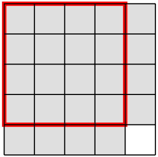


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:

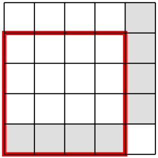


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:

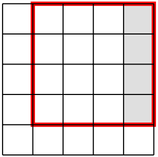


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:

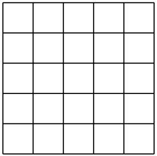


Dynamic programming to the rescue

A recursive formulation of this problem is as follows.

- An $m \times m$ square S is empty if and only if:
  - The bottom right pixel in S is empty;
  - The three $(m-1) \times (m-1)$ squares in the top left, top right and bottom left of S are all empty.

Proof by picture:



Dynamic programming to the rescue

So let `les`(x, y) be the size of the largest empty square whose bottom right-hand corner is at position (x, y).

Dynamic programming to the rescue

So let $\texttt{les}(x, y)$ be the size of the largest empty square whose bottom right-hand corner is at position (x, y).

Then:

- If the pixel (x, y) isn't empty, $\texttt{les}(x, y) = 0$.

Dynamic programming to the rescue

So let `les`(x, y) be the size of the largest empty square whose bottom right-hand corner is at position (x, y).

Then:

- If the pixel (x, y) isn't empty, `les`$(x, y) = 0$.
- If (x, y) is empty and in the first row or column, `les`$(x, y) = 1$.

Dynamic programming to the rescue

So let $\texttt{les}(x, y)$ be the size of the largest empty square whose bottom right-hand corner is at position (x, y).

Then:

- If the pixel (x, y) isn't empty, $\texttt{les}(x, y) = 0$.
- If (x, y) is empty and in the first row or column, $\texttt{les}(x, y) = 1$.
- If (x, y) is empty and not in the first row or column, then

  $$\texttt{les}(x, y) = \min(\texttt{les}(x-1, y-1), \texttt{les}(x, y-1), \texttt{les}(x-1, y)) + 1.$$

Dynamic programming to the rescue

So let $\texttt{les}(x, y)$ be the size of the largest empty square whose bottom right-hand corner is at position (x, y).

Then:

- If the pixel (x, y) isn't empty, $\texttt{les}(x, y) = 0$.
- If (x, y) is empty and in the first row or column, $\texttt{les}(x, y) = 1$.
- If (x, y) is empty and not in the first row or column, then

  $$\texttt{les}(x, y) = \min(\texttt{les}(x-1, y-1), \texttt{les}(x, y-1), \texttt{les}(x-1, y)) + 1.$$

This immediately suggests a recursive algorithm!

A recursive algorithm

The following algorithm computes the size of the largest empty square whose bottom right-hand corner is (x, y).

```
int les(x,y) {
    if (!empty(x,y)) return 0;
    if ((x == 1) || (y == 1)) return 1;
    return min(les(x-1,y-1),
               les(x,y-1),
               les(x-1,y)) + 1;
}
```

Once this has been done, taking the maximum of `les`(x, y) over all x, y gives the size of the largest empty square in the whole image.

A memoized recursive algorithm

Next step: memoize this algorithm. . .

```
int memo_les(x,y) {
    if (!empty(x,y)) return 0;
    if ((x == 1) || (y == 1)) return 1;
    if (undefined(les[x,y]))
        les[x,y] = min(memo_les(x-1,y-1),
                       memo_les(x,y-1),
                       memo_les(x-1,y)) + 1;
    return les[x,y];
}
```

This algorithm now only makes $O(n^2)$ integer additions!

A bottom-up version of the algorithm

Finally, observe that the `les` array gets filled in from the top left. Rewriting this as an iterative algorithm, we get

```
int asc_les(n) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      if (!empty(x,y))
        les[x,y] = 0;
      else if ((x == 1) || (y == 1))
        les[x,y] = 1;
      else
        les[x,y] = min(les[x-1,y-1],
                       les[x,y-1],
                       les[x-1,y]) + 1;
    }
  }
}
```

Conclusions

- Dynamic programming is a simple, yet powerful, technique for developing efficient algorithms.

Conclusions

- Dynamic programming is a simple, yet powerful, technique for developing efficient algorithms.
- The process of developing such an algorithm can sometimes be almost completely mechanical:
  1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
  2. Write down a naïve recursive algorithm.
  3. Memoize the recursive algorithm.
  4. Finally, restructure the algorithm to compute solutions in an efficient order.

Conclusions

- Dynamic programming is a simple, yet powerful, technique for developing efficient algorithms.
- The process of developing such an algorithm can sometimes be almost completely mechanical:
  1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
  2. Write down a naïve recursive algorithm.
  3. Memoize the recursive algorithm.
  4. Finally, restructure the algorithm to compute solutions in an efficient order.

Some further reading:

- Some excellent lecture notes by Jeff Erickson: `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf`
- *Algorithms* ch. 6 (Dasgupta, Papadimitriou and Vazirani).