Algorithms and Complexity

Dynamic programming

Julián Mestre

School of Information Technologies
The University of Sydney

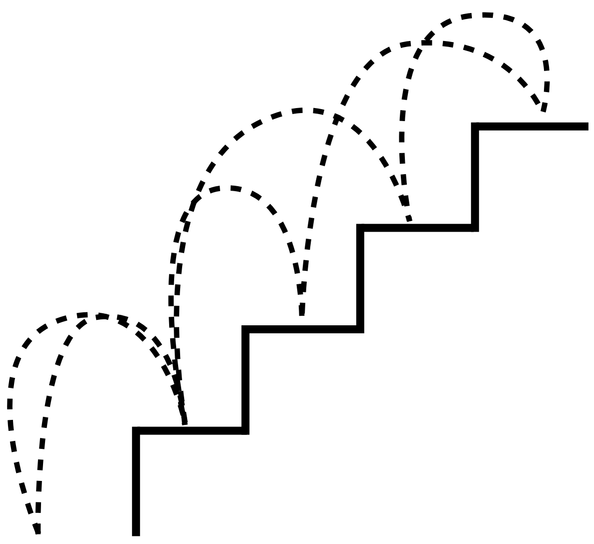
THE UNIVERSITY OF SYDNEY

A counting problem

How many way are there to go up a staircase with n steps when you can go up one or two steps at a time?

Let F(n) be this number. Then

- F(0) = 1
- F(1) = 1
- F(2) = 2
- F(3) = 3
- F(4) = 5

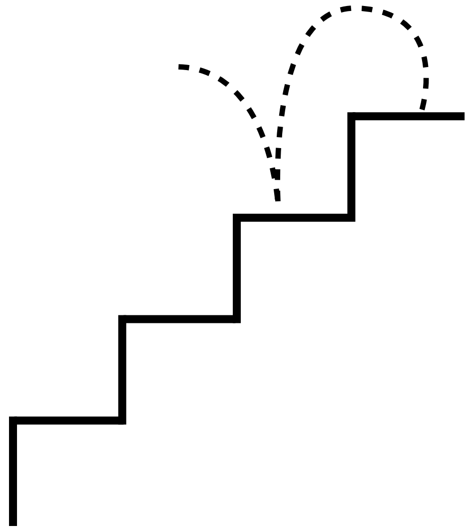


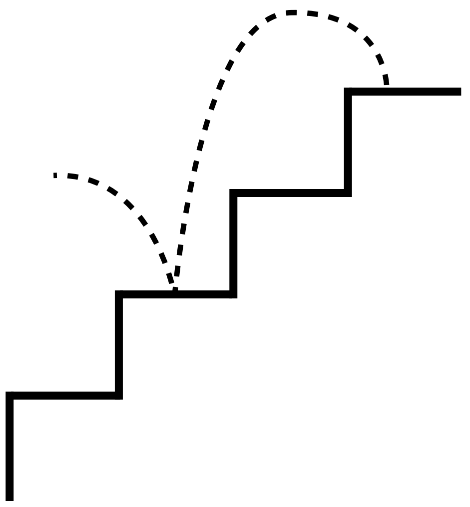
What's the pattern?

A counting problem

To see the pattern, condition on the last move

- There are F(n-1) ways of ending with a 1-step move
- There are F(n-2) ways of ending with a 2-step move





```
def fib(n):
  if n <= 1:
    return 1
  else:
    return fib(n-1) +
           fib(n-2)
```

$F(n) = F(n-1) + F(n-2)$ for $n > 1$

Time complexity

Let $T(n)$ be the running time of our algorithm then

$$T(n) = T(n-1) + T(n-2) + O(1) \quad \text{for } n > 1$$

which is at least $\Omega(2^{n/2})$

What can be done to speed up the algorithm?

Reuse computation!

Iterative algorithm

```
def fib(n):

  M = array of length n + 1
  M[0] = 1
  M[1] = 1
  for i in range(2, n+1):
    M[i] = M[i-1] + M[i-2]
  return M[n]
```

Weighted Interval Scheduling

Motivation:

- Users submitting requests to use some common resource (e.g., a classroom)
- Each request has a time window where the resource is needed
- Users cannot share the resource
- Users have priority

Input:

- Set of weighted intervals $\{I_1, I_2, \ldots, I_n\}$ where $I_i = (\ s(i), f(i)\ )$ and $w_i > 0$

Task:

- Find a maximum weight subset of intervals that do not intersect

The structure of optimum

Assume intervals are sorted so that $f_1 \leq f_2 \leq \cdots \leq f_n$

Let $OPT(i)$ be an optimal solution for intervals $\{1, \ldots, i\}$

Obs 1: If interval $n \notin OPT(n)$ then $OPT(n) = OPT(n-1)$

Obs 2: If interval $n \in OPT(n)$ then $OPT(n) = \{n\} \cup OPT(p(n))$,
where $p(i)$ is the largest index such that $f_{p(i)} \leq s_i$

Let $M(i)$ be the value of the optimal solution then

$$M(i) = \max \{ M(i-1), w(i) + M(p(i)) \} \text{ for } i > 0$$

Recursive algorithm

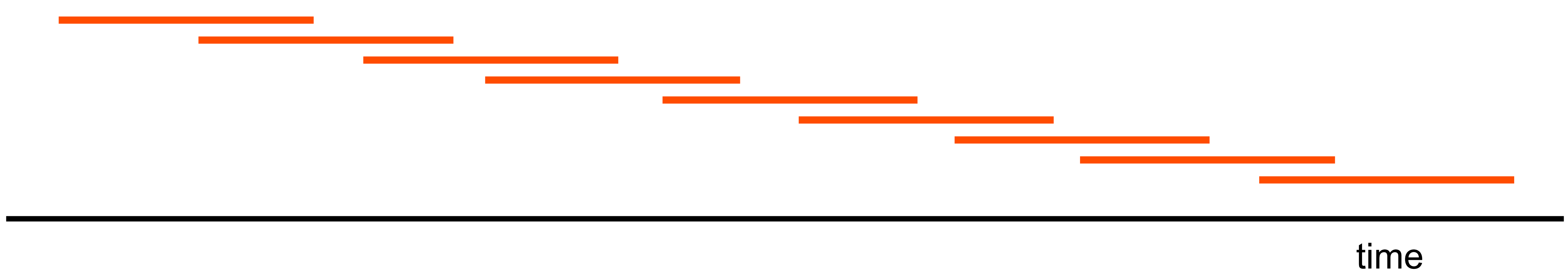
```
def MWIS(intervals,w):

  def helper(i):
    if i == 0:
      return 0
    return max(helper(i-1),
               w[i] + helper(p[i]))


  sort intervals in increasing f-value
  compute p-values for each interval

  return helper(n)
```

Time complexity

Let $T(n)$ be the running time of the algorithm. In the instance below $p(i) = i - 2$ for each $i > 2$ so

$$T(n) \geq T(n-1) + T(n-2) + O(1)$$

which be already saw is exponential!



Iterative algorithm

```
def MWIS(intervals,w):

  n = len(intervals)
  sort intervals in increasing f-value
  compute p-values for each interval

  M = array of length n + 1
  M[0] = 0
  for i in range(1, n+1):
    M[i] = max(M[i-1], w[i] + M[p[i]])
  return M[n]
```

Time complexity

Sorting the intervals takes $O(n \log n)$ time

Computing the p-values can be done in $O(n \log n)$ time

There are $n-2$ iterations each taking $O(1)$,
so computing M takes $O(n)$ time

Overall the algorithm runs in $O(n \log n)$ time

What about finding the solution?

```
def MWIS(intervals,w):

  def helper(i):
    if i == 0:
      return []
    if M[i] == M[i-1]:
      return helper(i-1)
    else:
      return helper(p[i]) + [i]

  sort intervals in increasing f-value
  compute p-values for each interval
  compute M-values

  return helper(n)
```

Longest increasing subsequence

Input:

- Unsorted array A with n numbers

Task:

- Find longest sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$

Knapsack

Motivation:

- You need to design an investment portfolio for yourself
- There are n investment options, each having associated a profit and cost.
- Investment decision are binary decision (all or nothing)
- Given your budget, find the best investment option

Input:

- Set of pairs $(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and knapsack capacity W

Task:

- Find $S \subseteq \{1, \ldots, n\}$ maximizing $v(S)$ subject to $w(S) \leq W$

Recap: Dynamic programming (DP)

DP algorithms have three distinctive ingredients

- A big subproblem is broken up into smaller subproblems
- The solution of a subproblem can be expressed recursively
- There is an ordering of the subproblems from "small" to "large" such that to solve a subproblem we only need the solution to "smaller" subproblems

Correctness depends on the correctness of the recurrence

Time complexity is usually dominated by

# of DP states * time it takes to fill one state