451: Dynamic Programming I

G. Miller, K. Sutner
Carnegie Mellon University
2020/09/08/03

The usual definition of the Fibonacci numbers uses recursion:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2) \qquad n \geq 2
\end{aligned}
$$

In a decent environment, this definition translates directly into code:

```
fib[0] = 0;
fib[1] = 1;
fib[x_] :=  fib[x-1] + fib[x-2];
```

This is rather painless from the programmer's perspective, and correctness is obvious (assuming the system is properly implemented).

As Kleene has shown, recursion is the only power tool one needs to obtain all of computability (abstract computability, much more than what we are looking for in algorithms).

To make sure the mechanism of recursion works and everything terminates we need a well-order $\prec$ on the problem domain D.

Total Recall:

A well-order $\langle D, \prec \rangle$ is a total order with the special property that for all $\emptyset \neq X \subseteq D$, there is a $\prec$-minimal element in X.

Alternatively, there is no infinite descending chain

$$x_0 \succ x_1 \succ x_2 \succ \ldots \succ x_n \succ x_{n+1} \succ \ldots$$

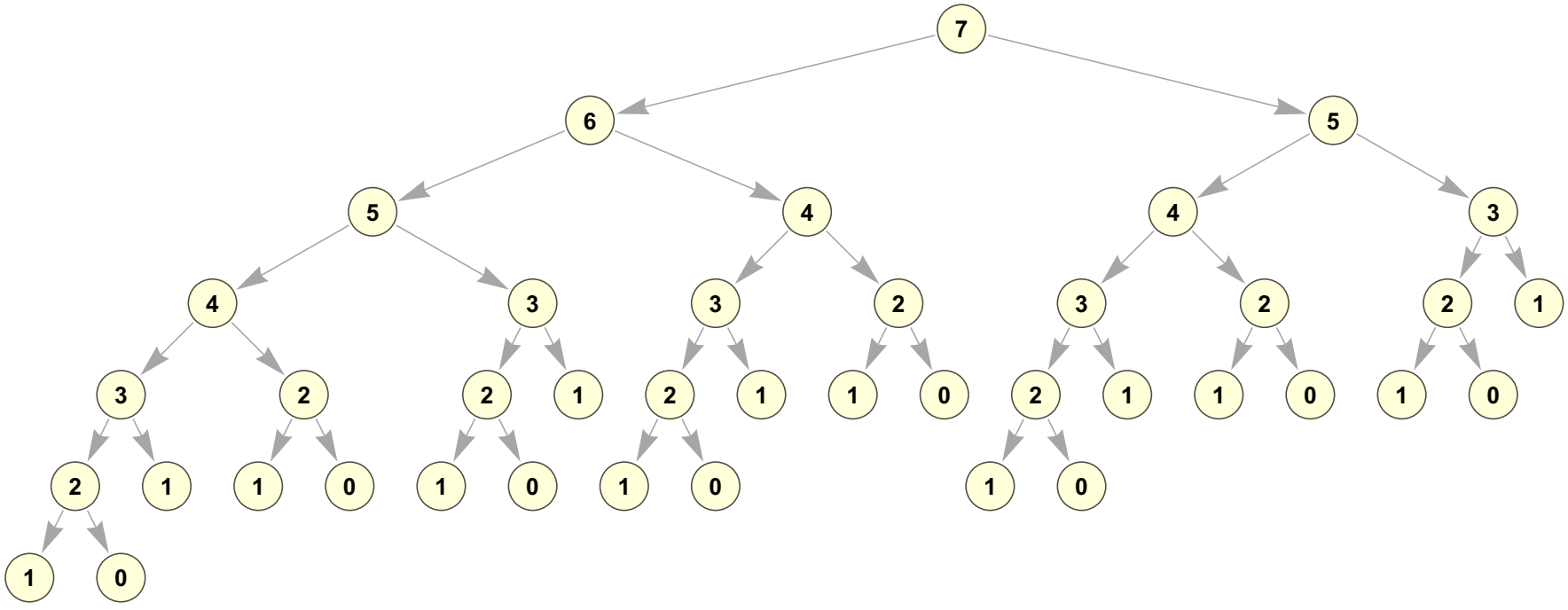
defun $f(x : D)$
    if x is minimal
    then
        return answer
    else
        compute $y_i = f(x_i)$ for $i = 1, \ldots, k$
        return answer constructed from the y_i and x

Here it is understood that $x_i \prec x$ for all x.

For example, in the natural numbers we can use the standard order $x_i < x$.

Length-lex order on words also works, but lexicographic order fails.

Incidentally, a while ago there were people who thought that recursion has no place in a real programming language.



The call tree of `fib[7]`, an unmitigated disaster! There is a ton of recomputation going on: e.g., the call to argument 4 appears 3 times. We get exponentially man calls.

There are two standard ways to deal with this issue:

Top-Down Aka memoization: keep the recursive program, but store all computed results in a hash table, do not recompute.

Bottom-Up Lovingly construct a lookup table by hand, replace recursive calls by filling in the table in a corresponding order.

Memoization has the great advantage that it requires no further thought, everything happens automatically.

```
fib[0] = 0;
fib[1] = 1;
fib[x_] :=  ( fib[x] = fib[x-1] + fib[x-2] );
```

Very elegant and easy to use.

If efficiency is paramount, one may be better off by constructing a table of known values by hand.

Just to be clear: the Fibonacci example is a bit lame, obviously one could compute the values

$$F(0), F(1), F(2), \ldots, F(n)$$

bottom-up by using an array of size $n + 1$.

Which method is easily improved, since $F(i)$ depends only on $F(i-1)$ and $F(i-2)$ we can easily reduce the space requirement to $O(1)$ in this case, using plain iteration.

Definition

Dynamic programming refers to a type of algorithm that solves large problem instances by systematically breaking them up into smaller sub-instances, solving these separately and then combining the partial solutions.

The key difference to divide-and-conquer is that the sub-instances are not required to be independent: two sub-instances may well contain the same sub-sub-instances.

Bookkeeping becomes an important task here: we have to keep track of all the sub-instances and their associated solutions. We tackle this by constructing a table (usually one- or two-dimensional) for all the sub-instances and their solutions.

Here are the key ingredients in any dynamic programming algorithm.

- Find a suitable notion of (smaller) sub-instance for any given instance.
- Find a recursive way to express the solution of an instance in terms of the solutions of the sub-instance(s).
- For bottom-up: organize this recursive computation into a neat table, possibly with some pre-computation.

It is very helpful when the number of sub-instances is $O(1)$. Alas, in general one may have to deal with an unbounded number of sub-instances.

Again: Constructing an explicit table may be more efficient, but it tends to be more painful for the programmer.

In the RealWorld$^{\mathrm{TM}}$, building the table comes down to two key problems:

Boundary We have to figure out the correct boundary values and initialize the table correspondingly.

Dependency We have to fill the table in the correct order: never read a position without having written to it first.

Definition

Consider a sequence $A = a_1, a_2, \ldots, a_n$. A subsequence of A if is any sequence $S = s_1, s_2, \ldots, s_k$ such that there is an index sequence $1 \leq p_1 < p_2 < \ldots < p_k \leq n$ with

$$s_i = a_{p_i}.$$

Given two sequences A and B, S is a common subsequence if S is a subsequence of both A and B.
S is a longest common subsequence (LCS) if its length is maximal among all common subsequences.

So this is an optimization problem: we have to maximize the length of a common subsequence.

Note: we are dealing with scattered subsequences here, the elements need not be contiguous.

$$A = 3, 5, 4, 3, 1, 5, 4, 2, 4, 5, 3, 1, 3, 5, 2, 1, 5, 2, 1, 3$$

$$B = 4, 3, 5, 1, 5, 3, 1, 3, 3, 2, 2, 2, 5, 4, 4, 4, 4, 5, 4, 4$$

Claim: An LCS for A and B is

$$S = 3, 5, 1, 5, 3, 1, 3, 5, 5$$

This is not exactly obvious.

It's not hard to check that S really is a subsequence: use a greedy approach to find an occurrence of S in A and B:

$$A = 3, 5, 4, 3, 1, 5, 4, 2, 4, 5, 3, 1, 3, 5, 2, 1, 5, 2, 1, 3$$

$$B = 4, 3, 5, 1, 5, 3, 1, 3, 3, 2, 2, 2, 5, 4, 4, 4, 4, 5, 4, 4$$

But it is far from clear that we have the longest possible subsequence.

Uniqueness is another interesting question.

So suppose we have sequences $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_n$.

What are the sub-instances that we need to consider?

It is tempting to just shorten the sequence by chopping off the last item.

To keep notation simple we will write

$$A\,a$$

for a sequence with last element a and initial segment A.

Let $a \neq b$. Then the recursion is fairly clear:

$$\mathsf{LCS}(Ac, Bc) = \mathsf{LCS}(A, B)\, c$$

$$\mathsf{LCS}(Aa, Bb) = \mathsf{LCS}(A, Bb) \text{ or } \mathsf{LCS}(Aa, B)$$

Read this with a grain of salt – the "or" in the second case means we don't know which sub-instance is going to produce the solution.

Exercise

Explain how to make sense out of these equations.

Here is a standard trick: first ignore the optimal solution, and only compute the optimal value. Then, in a second round, construct the optimal solution from the optimal value.

Let's write $\mathsf{lcs}(A, B)$ for the length of any LCS for A and B.

Then assuming $a \neq b$:

$$\mathsf{lcs}(Ac, Bc) = \mathsf{lcs}(A, B) + 1$$

$$\mathsf{lcs}(Aa, Bb) = \max(\, \mathsf{lcs}(A, Bb), \mathsf{lcs}(Aa, B)\,)$$

There is nothing fishy about these equations, they are literally correct.

If we use memoizing, this is essentially the solution.

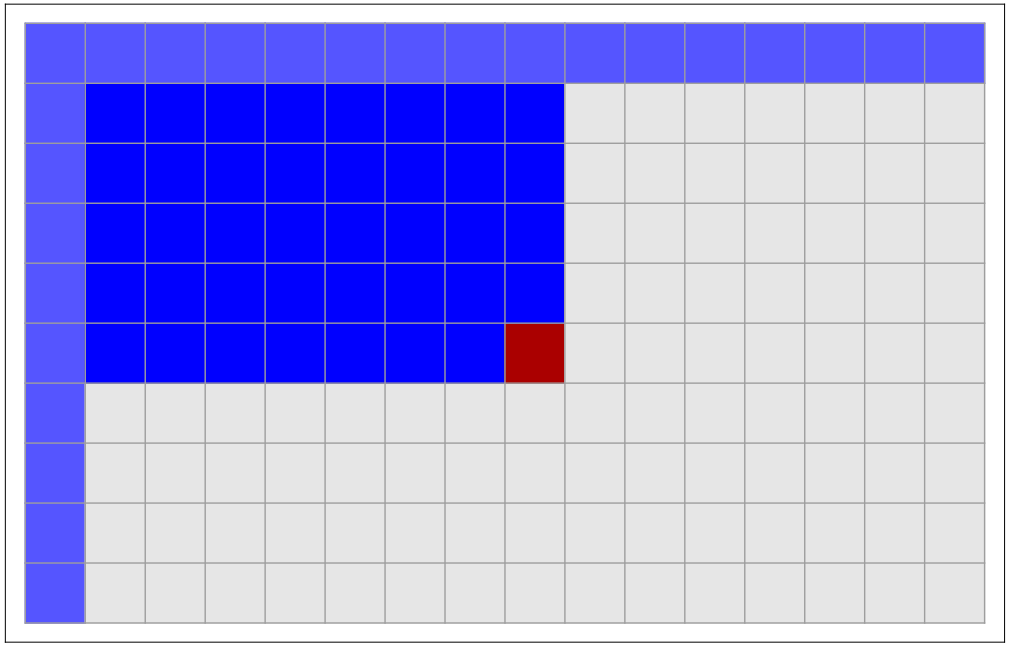
Write A_i for the prefix of A of length i: $A_i = a_1, a_2, \ldots, a_{i-1}, a_i$ and likewise for B_j. We compute an $(n+1) \times (m+1)$ table:

$$L(i,j) = \mathsf{lcs}(A_i, B_j)$$

Initialization is easy: $L(0,j) = L(i,0) = 0$. Update:

$$L(i,j) = \begin{cases} L(i-1,j-1)+1 & \text{if } a_i = b_j, \\ \max(L(i-1,j), L(i,j-1)) & \text{otherwise.} \end{cases}$$

for all $0 < i, j$.



Possible approaches:

- column-by-column, top-down
- row-by-row, left-to-right
- sweeping counterdiagonal

$A = d, d, d, c, c, b, c, a$ and $B = d, c, c, d, b, c, a, c, c, d$ produces

1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2
1	1	1	2	2	2	2	2	2	3
1	2	2	2	2	3	3	3	3	3
1	2	3	3	3	3	3	4	4	4
1	2	3	3	4	4	4	4	4	4
1	2	3	3	4	5	5	5	5	5
1	2	3	3	4	5	6	6	6	6

We have left off the 0 row and column.

So lcs$(A, B) = 6$ and an LCS is d, c, c, b, c, a.

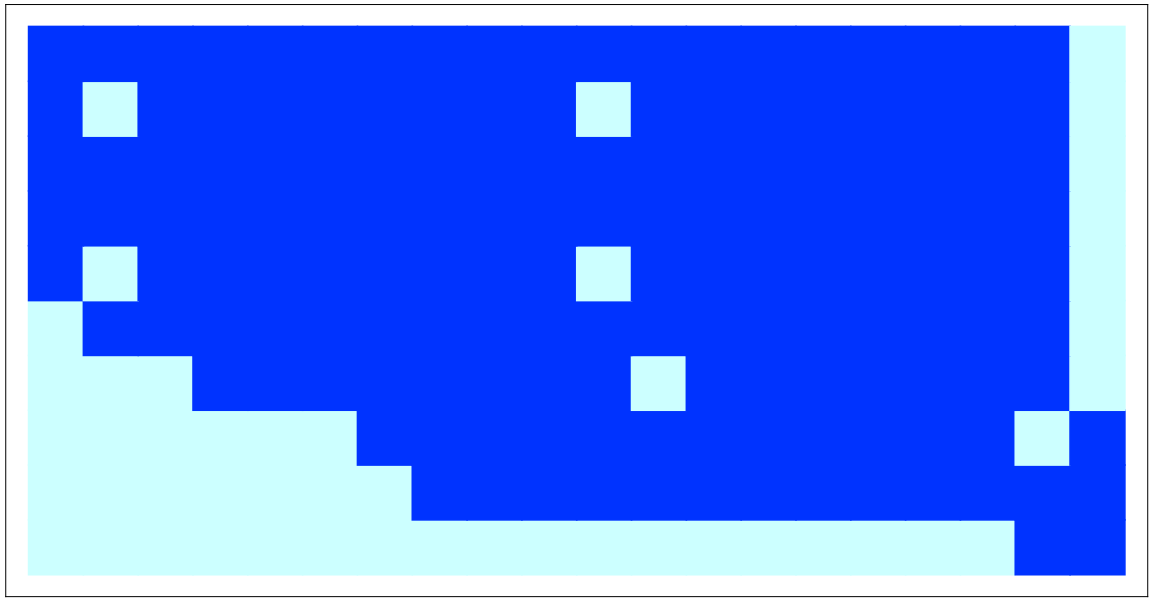
Read off d, c, c, b, c, a by following diagonal moves.

	d	c	c	d	b	c	a	c	c	d
d	1	1	1	1	1	1	1	1	1	1
d	1	1	1	2	2	2	2	2	2	2
d	1	1	1	2	2	2	2	2	2	3
c	1	2	2	2	2	3	3	3	3	3
c	1	2	3	3	3	3	3	4	4	4
b	1	2	3	3	4	4	4	4	4	4
c	1	2	3	3	4	5	5	5	5	5
a	1	2	3	3	4	5	6	6	6	6

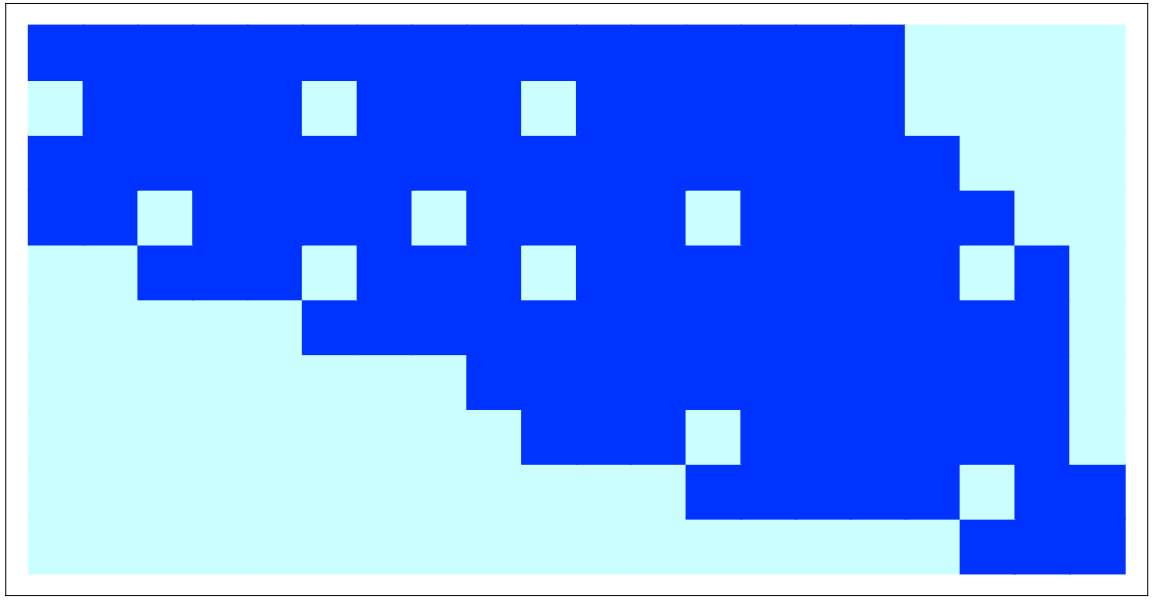
Exercise

Explain how to turn this into an algorithm. How would you generate all LCS?

Here is a typical call structure with memoizing:



Around 75% of the matrix is used.



About 60% of the matrix is used.

Exercise

What is the least amount of memory needed with memoizing? How important is this going to be in practical applications?

Exercise

Can memoizing ever require mn entries? How important is this going to be in practical applications?

Exercise

Implement both methods and try to determine which one is better.

Exercise

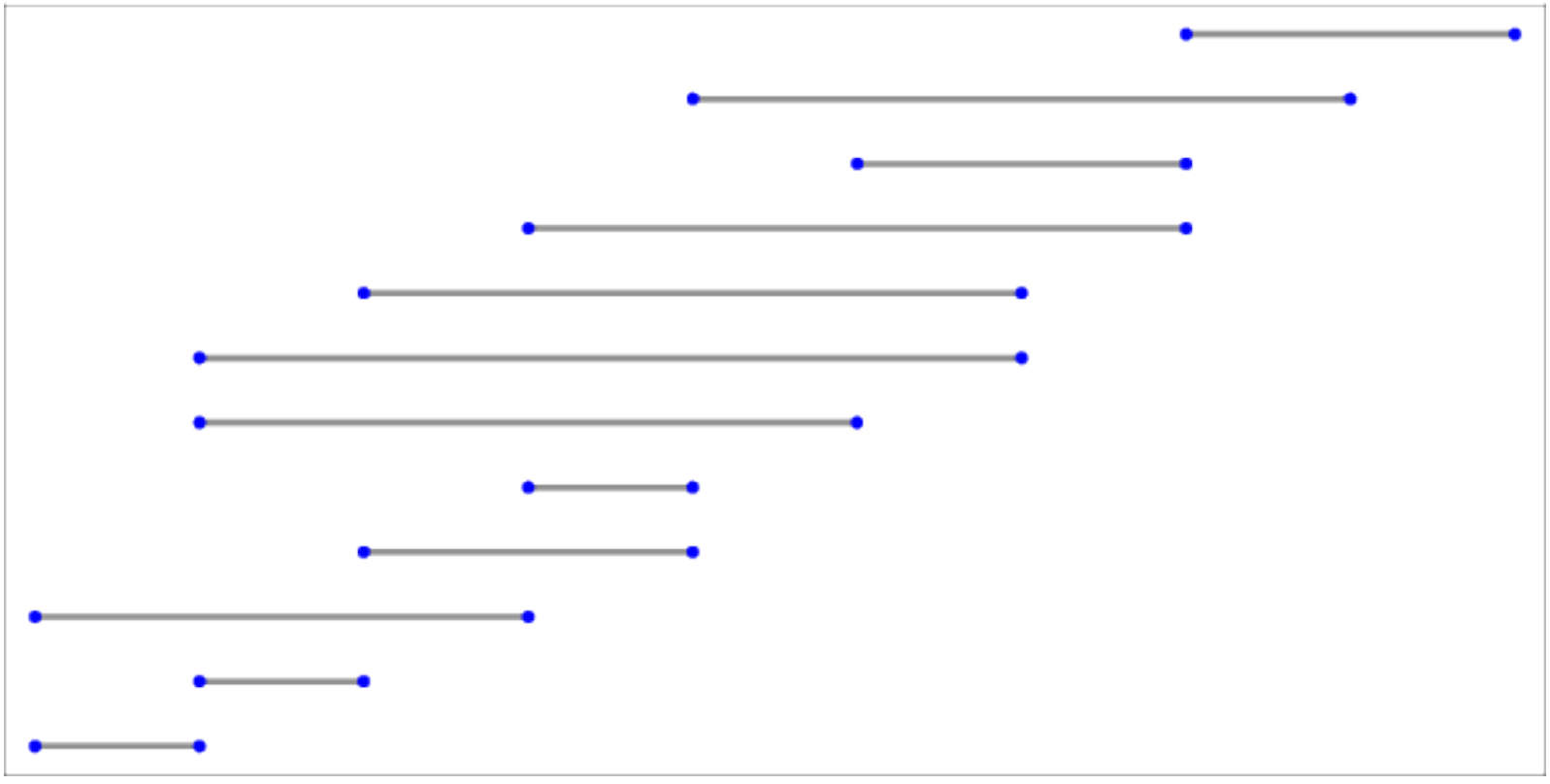
How would one go about constructing an actual LCS in the memoizing case?

Suppose you have a resource (such as a conference room) and requests for its use. The requests have the form (t_1, t_2) where $t_1 < t_2 \in \mathbb{N}$ (the begin and end time of a meeting).

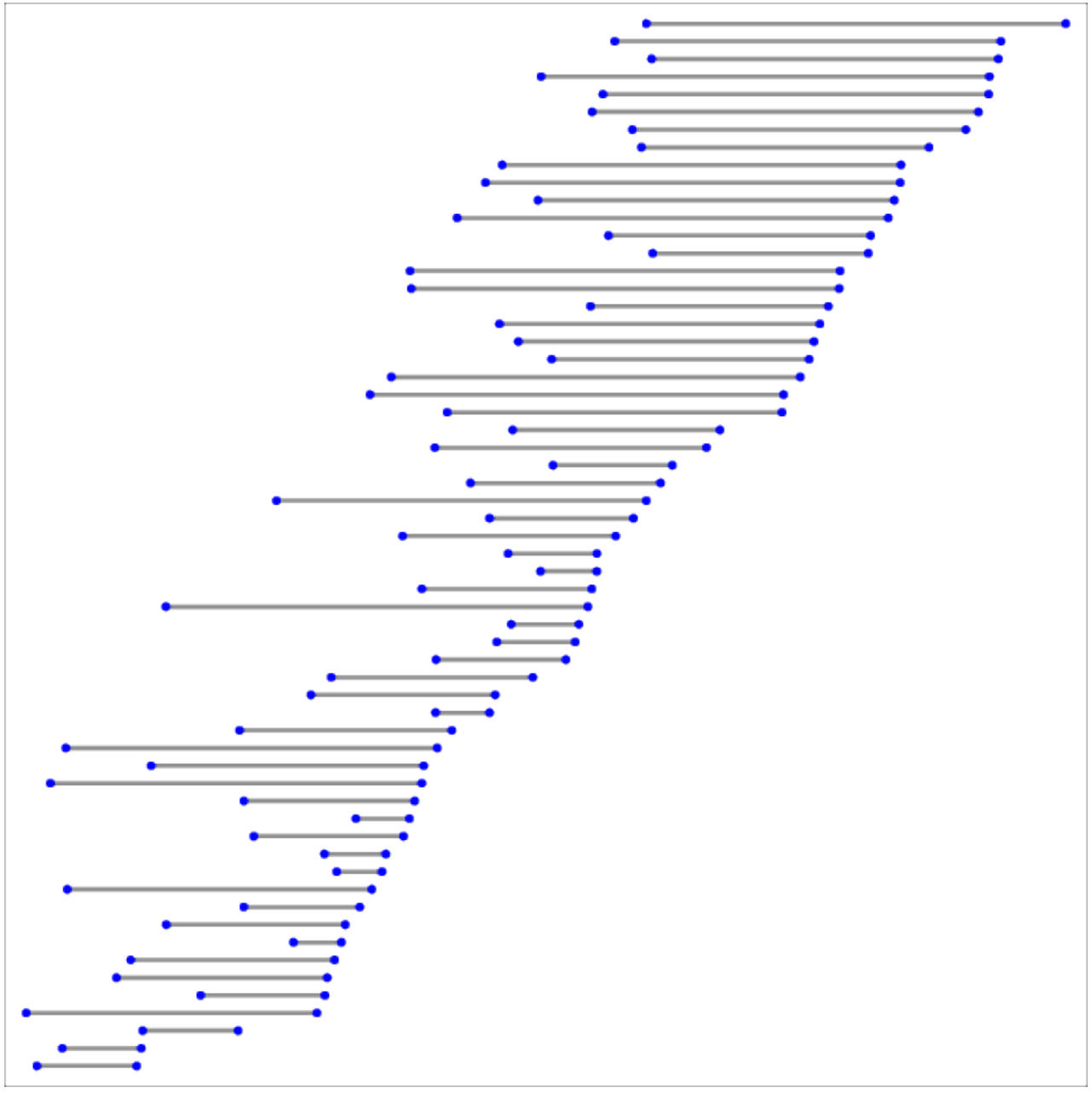
There are two opposing requirements:

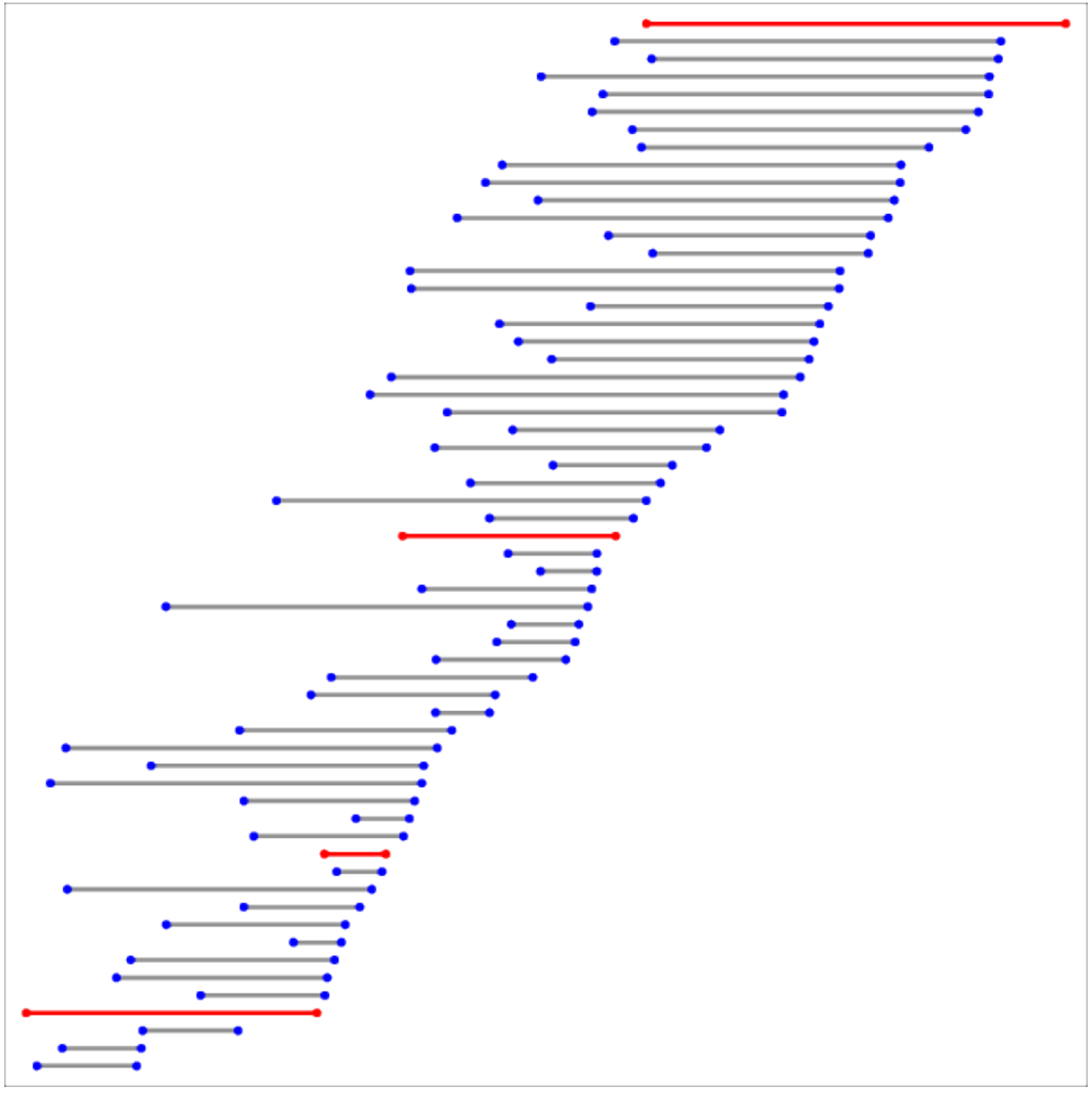
- The resource is supposed to be as busy as possible, but
- no overlapping requests can be scheduled.

By overlap we mean that the two intervals have an interior point in common, so $(1, 5), (5, 7)$ would be admissible whereas $(1, 5), (4, 6)$ would not. So we want to find a non-overlapping schedule that maximizes the use of the resource.



Intervals are sorted by right endpoint.
In this case, using (human) geometric intuition, it's easy to find an optimal schedule that fills the whole interval by visual inspection.





The input for the scheduling problem is given as a list of intervals $I_1, I_2, \ldots, I_n$ where

$$I_i = (a_i, b_i) \qquad\qquad a_i < b_i \in \mathbb{N}.$$

Let's write $\ell_k = b_k - a_k$ for the length of interval I_k. We'll be sloppy and also speak of "interval k".

Definition

A schedule is a set $S \subseteq [n] = \{1, 2, \ldots, n\}$ of intervals subject to the constraint that for $i \neq j \in S$ intervals I_i and I_j do not overlap.

We need to maximize

$$\text{val}(S) = \sum_{i \in S} \ell_i$$

over all (exponentially many) schedules S.

Again, this is an optimization problem and val is the objective function.

Again, as usual, we will first focus on computing the optimal value val(S) of the solution, then handle the actual solution S in a second phase.

Sub-instances are easy: consider only the first k intervals, $k \leq n$. For simplicity we write val(k) for the value of an optimal solution using only the intervals $I_1, I_2, \ldots, I_k$. So we are only looking at $S \subseteq \{1, 2, \ldots, k\}$.

Furthermore, let us assume that the $I_i = (a_i, b_i)$ are already sorted by right endpoint:

$$b_1 \leq b_2 \leq \ldots\ldots \leq b_{n-1} \leq b_n.$$

If not, this can be ensured by a simple pre-processing step (linear time in the RealWorld$^{\text{TM}}$).

For the recursion we consider two cases.

Case Out

The easy case is when I_k is not part of the optimal solution for k: then $\text{val}(k) = \text{val}(k-1)$.

Case In

In this case $\text{val}(k-1)$ won't be of much help in general since there may be overlap between some of these intervals and I_k.

Recall our convention: intervals are sorted by right endpoint. So, we need to consider $\text{val}(k')$ for some $k' < k$: k' has to be small enough so that no collisions can occur between the intervals in $I_1, \ldots, I_{k'}$ and I_k.

To make sure we don't miss out on any possible solutions we will choose k' maximal.

To this end we compute the constraint function $c(k)$:

$$c(k) = \max\big(i < k \mid I_i \text{ and } I_k \text{ do not overlap} \big)$$

Now we can compute $\mathsf{val}(k)$ as follows:

$$\mathsf{val}(k) = \max\Big(\mathsf{val}(k-1), \mathsf{val}(c(k)) + \ell_k \Big)$$

Exercise

Explain how this really implements our informal strategy.

How do we compute the c-array?

Recall that we have the intervals sorted by their right endpoints (b_i). We try to avoid collision with I_k, so we need to worry about the left endpoint a_k. So we can perform binary search to find

$$\max(i < k \mid b_i \leq a_k).$$

The total cost of this is $O(n \log n)$.

Exercise

Find a linear time method to compute the constraint function assuming that we have interval lists sorted by left endpoints, and also sorted by right endpoints.

Given the constraint function one can easily construct a table

$$\mathsf{val}(1), \mathsf{val}(2), \ldots, \mathsf{val}(n-1), \mathsf{val}(n)$$

using a single pass from left to right in time $\Theta(n)$.

So time and space complexity are both $\Theta(n)$.

Just to be clear: unlike with the Fibonacci example, we cannot simply dump the array and use an iterative method.

Once the table has been constructed we can now trace back the final answer $\mathsf{val}(n)$ to obtain the corresponding actual solution S. We must have

$$\mathsf{val}(k) = \mathsf{val}(k-1)$$

or

$$\mathsf{val}(k) = \mathsf{val}(c(k)) + \ell_k$$

The second pass is also linear time.

Exercise

Figure out exactly how to construct an actual optimal solution.

Suppose we have a list of sizes $s_1, s_2, \ldots, s_n$ and a list of values $v_1, v_2, \ldots, v_n$ for n items. We are also given an upper bound B for the allowable total size of chosen items. Assume everything is positive integers.

The goal is to choose a subset $I \subseteq [n]$ such that

$$\sum_{i \in I} s_i \leq B$$

$$\sum_{i \in I} v_i \text{ is maximal}$$

The decision version of this problem has an additional lower bound for value and is one of the classical problems shown to be $\mathbb{NP}$-complete in Karp's seminal paper. Without getting involved in a philosophical discussion, this probably means that, in general, there are no good algorithms.

The right sub-instances are again fairly natural: constrain the selected set I to $[k]$ and march from k to $k+1$.

Define $\mathsf{val}(k, b) = \sum_{i \in I} v_i$ to be the maximal value obtainable with

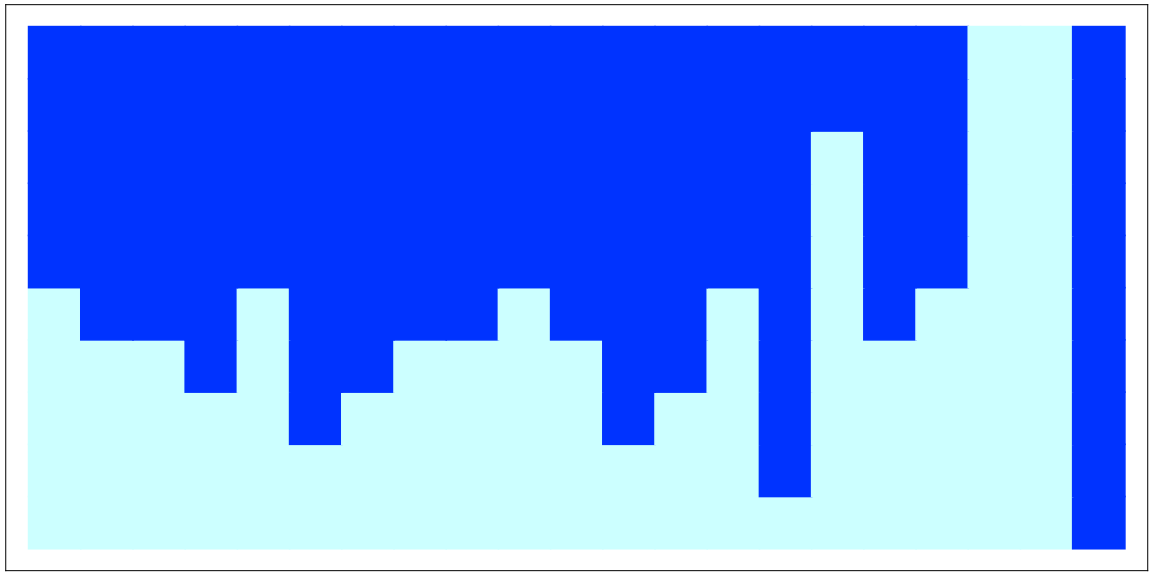
$$\begin{aligned} I &\subseteq [k] \\ \sum_{i \in I} s_i &\leq b \end{aligned}$$

Clearly, $\mathsf{val}(0, b) = 0$.

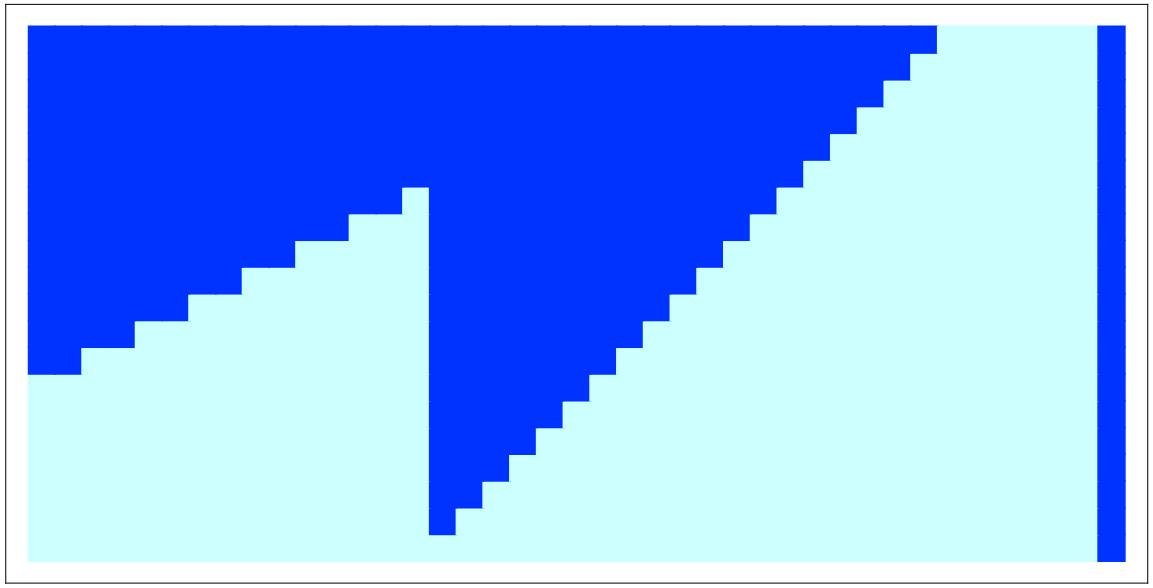
The recursion now looks like so:

$$\mathsf{val}(k, b) = \begin{cases} \mathsf{val}(k-1, b) & \text{if } s_k > b, \\ \max\big(\mathsf{val}(k-1, b), v_k + \mathsf{val}(k-1, b-s_k)\big) & \text{otherwise.} \end{cases}$$

Here is a typical call structure. Note the lovely gap on the right.



Sizes increasing, values decreasing.



The table has size $n \times B$ and each update is clearly $O(1)$, at least in the uniform cost model.

At first glance, this may look like a polynomial time algorithm, but it's most definitely not: all the numbers are given in binary, including the bound B. So its size is just $\log B$.

All we get is pseudo-polynomial time, but at least Knapsack is not strongly $\mathbb{NP}$-complete. So there is a good chance that for small "practical" instances we have a reasonable algorithm.

By contrast, the similar Bin Packing problem is strongly $\mathbb{NP}$-complete.