Analysis & Design of Algorithms

Angel Napa

January 1, 2026

1 Fibonacci Numbers

Consider the recurrence:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

The numbers $F(0), F(1), F(2), \ldots, F(n)$ are called *Fibonacci Numbers.* $(0, 1, 1, 2, 3, 5, 8, 13, 21, 42, ...)$

Problem: given an integer n, compute $F(n)$. Consider the following recursive algorithm:

Input: A number n
Output: $F(n)$
RECURSIVE-FIBONACCI(n)

```
1: if n == 0
2:    return 0
3: if n == 1
4:    return 1
5: return RECURSIVE-FIBONACCI(n − 1)+ RECURSIVE-FIBONACCI(n − 2)
```

It is clear that the running time of the algorithm (assuming all constants are 1) is given by

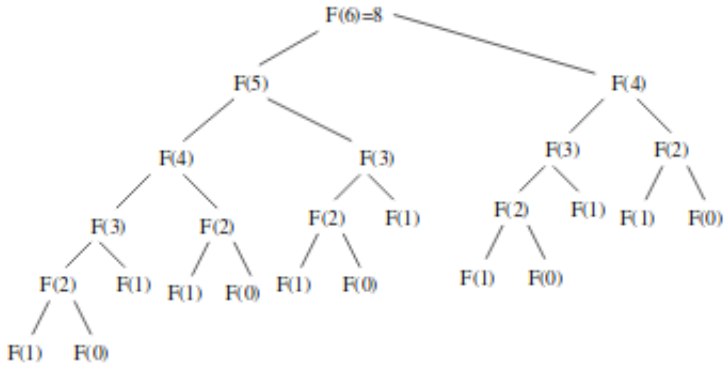
$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 1 & \text{if } n > 1 \end{cases}$$

We will prove by induction that $T(n) \geq \frac{2}{3}(3/2)^n$ for $n \geq 1$. When $n = 0$ we have that $T(0) = 1 \geq \frac{2}{3}(3/2)^0 = 2/3$. When $n = 1$ we have that $T(1) = 1 \geq$

$\frac{2}{3}(3/2)^1 = 1$. Now suppose $n > 1$. Then,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq \frac{2}{3}(3/2)^{n-1} + \frac{2}{3}(3/2)^{n-2} + 2 \\ &= \frac{2}{3}(3/2)^n(\frac{2}{3} + \frac{4}{9}) + 2 \\ &\geq \frac{2}{3}(3/2)^n \end{aligned}$$

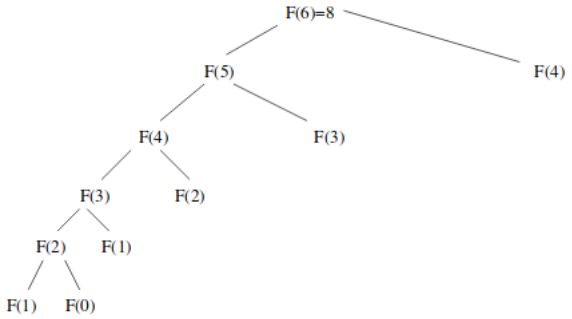
Therefore, the running time of the RECURSIVE-FIBONACCI algorithm is $\Omega((\frac{3}{2})^n)$.



Consider the following improvement.

Input: A number n
Output: $F(n)$
MEMOIZED-FIBONACCI(n)
1: **if** $M[n] \neq -1$
2: **return** $M[n]$
3: **else**
4: $M[n] =$ MEMOIZED-FIBONACCI($n-1$)+MEMOIZED-FIBONACCI($n-2$)
5: **return** $M[n]$
MAIN(n)
1: $M[0] = 0$
2: $M[1] = 1$
3: **for** $i = 2$ **to** n
4: $M[i] = -1$
5: **return** MEMOIZED-FIBONACCI(n)

The previous algorithm is a *memoized* version of the recursive version. What we are doing is storing results that have already been computed in memory.



Let $T(n)$ be the running time of the MEMOIZED-FIBONACCI routine. Note that the running time of the recursive call with parameter $n-2$ will always be constant, since $M[n-2]$ will have been previously computed in the call with parameter $n-1$. Therefore,

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1)+1 & \text{if } n > 1 \end{cases}$$

which implies that $T(n) = \Theta(n)$ (another way to arrive at this result is to observe that line 4 of the algorithm is executed at most n times). It is clear that a bit more memory is used compared to the previous case; however, the memory used is also linear in n. We conclude that the efficiency of MEMOIZED-FIBONACCI is much greater than the efficiency of RECURSIVE-FIBONACCI.

We can further improve MEMOIZED-FIBONACCI by eliminating the recursive calls while maintaining the execution time.

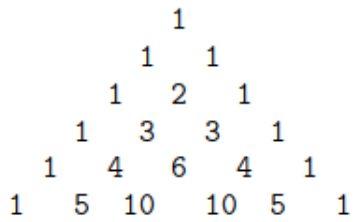
Input: A number n
Output: $F(n)$
DYN-PROG-FIBONACCI(n)
1: $M[0] = 0$
2: $M[1] = 1$
3: **for** $i = 2$ **to** n
4: $\quad M[i] = M[i-1] + M[i-2]$
5: **return** $M[n]$

The previous technique is called *dynamic programming*.

2 Binomial Coefficients

For every pair of natural numbers n, k with $n \geq k$, consider the following formula: $C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$. (This formula counts the number of ways to choose k elements from n possible ones).

If we want to compute the function $C(n,k)$, we can take advantage of a property given by Pascal's triangle:



Note that $C(n,k) = C(n-1,k-1)+C(n-1,k)$. Why is this true? We want to choose k items out of n. If item number n is not chosen, then the answer is $C(n-1,k)$. If item number n is chosen, then we must choose $k-1$ items out of the remaining $n-1$, and the answer is $C(n-1,k-1)$.

Therefore, we have the recurrence:

$$C(n,k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ C(n-1,k-1) + C(n-1,k) & \text{otherwise} \end{cases}$$

Consider the following recursive algorithm.

Input: Two numbers n and k
Output: $C(n,k)$
RECURSIVE-BINOMIAL(n,k)
1: **if** $k == 0$ **or** $k == n$
2: **return** 1
3: **return** RECURSIVE-BINOMIAL$(n-1,k-1)$ + RECURSIVE-BINOMIAL$(n-1,k)$

Similarly to the recursive version of Fibonacci, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: Two numbers n and k
Output: $C(n,k)$
MEMOIZED-BINOMIAL(n,k)
1: **if** $M[n][k] \neq -1$
2: **return** $M[n][k]$
3: **if** $k == 0$ **or** $k == n$
4: **return** 1
5: $M[n][k]$ = MEMOIZED-BINOMIAL$(n-1,k-1)$ + MEMOIZED-BINOMIAL$(n-1,k)$
6: **return** $M[n][k]$

MAIN(n,k)
1: **for** $i = 0$ **to** n
2: **for** $j = 0$ **to** k
3: $M[i][j] = -1$
4: **return** MEMOIZED-BINOMIAL(n,k)

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: Two numbers n and k
Output: $C(n, k)$
DYN-PROG-BINOMIAL(n, k)
1: **for** $i = 0$ **to** n
2: **for** $j = 0$ **to** k
3: **if** $j == 0$ **or** $j == i$
4: $M[i][j] = 1$
5: **else**
6: $M[i][j] = M[i-1][j-1] + M[i-1][j]$
7: **return** $M[n][k]$

3 Integer Partition

Given a positive integer n, we want to determine the number of ways in which n can be written as a sum of positive integers. We denote this number by $p(n)$. For example, $p(4) = 5$ because:

$$\begin{aligned} 4 &= 1+1+1+1 \\ &= 1+1+2 \\ &= 1+3 \\ &= 2+2 \\ &= 4 \end{aligned}$$

We can obtain the recurrence $p(n) = p(n-1)+p(n-2)+p(n-3)+\cdots+p(1)+1$. We add 1 to take into account the sum of n itself. We can construct the following recursive algorithm.

Input: A number n
Output: $p(n)$
RECURSIVE-PARTITION(n)
1: **if** $n == 0$
2: **return** 1
3: sum=0
4: **for** $i = 1$ **to** n
5: sum = sum + RECURSIVE-PARTITION$(n - i)$
6: **return** sum

Similarly to the recursive versions of Fibonacci and Binomial coefficients, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: A number n
Output: $p(n)$

MEMOIZED-PARTITION(n)
1: **if** $M[n] \neq -1$
2: **return** $M[n]$
3: **if** $n == 0$
4: **return** 1
5: sum=0
6: **for** $i = 1$ **to** n
7: sum = sum + MEMOIZED-PARTITION($n - i$)
8: $M[n] = sum$
9: **return** sum

MAIN(n)
1: **for** $i = 0$ **to** n
2: $M[i] = -1$
3: **return** MEMOIZED-PARTITION(n)

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: A number n
Output: $p(n)$
DYN-PROG-PARTITION(n)
1: $M[0] = 1$
2: **for** $i = 1$ **to** n
3: sum=0
4: **for** $j = 1$ **to** i
5: sum = sum + $M[i - j]$
6: $M[i] = sum$
7: **return** $M[n]$

4 Longest Common Subsequence (LCS)

Given two sequences $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_m)$, we want to find the longest sequence Z that is a subsequence of both X and Y. A *subsequence* of a sequence X is obtained by removing some or none of the elements of X without changing the order of the remaining elements. We denote by $L(i, j)$ the length of the LCS of the sequences $(x_1, x_2, ..., x_i)$ and $(y_1, y_2, ..., y_j)$. We can obtain the following recurrence:

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{L(i, j-1), L(i-1, j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Consider the following recursive algorithm.

Input: Two sequences X and Y
Output: $L(i, j)$
RECURSIVE-LCS(X, Y, i, j)

```
1: if i == 0 or j == 0
2:    return 0
3: if x_i == y_j
4:    return RECURSIVE-LCS(X, Y, i − 1, j − 1) + 1
5: else
6:    return max{RECURSIVE-LCS(X, Y, i, j − 1), RECURSIVE-LCS(X, Y, i − 1, j)}
```

Similarly to the recursive versions of Fibonacci, Binomial coefficients, and Integer Partition, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: Two sequences X and Y
Output: $L(i, j)$

```
MEMOIZED-LCS(X, Y, i, j)
1: if M[i][j] ≠ −1
2:    return M[i][j]
3: if i == 0 or j == 0
4:    return 0
5: if x_i == y_j
6:    return MEMOIZED-LCS(X, Y, i − 1, j − 1) + 1
7: else
8:    return max{MEMOIZED-LCS(X, Y, i, j − 1), MEMOIZED-LCS(X, Y, i − 1, j)}
9: M[i][j] = MEMOIZED-LCS(X, Y, i, j)
10: return M[i][j]
```

```
MAIN(X, Y)
1: for i = 0 to n
2:    for j = 0 to m
3:       M[i][j] = −1
4: return MEMOIZED-LCS(X, Y, n, m)
```

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: Two sequences X and Y
Output: $L(i, j)$

```
DYN-PROG-LCS(X, Y)
1: for i = 0 to n
2:    for j = 0 to m
3:       if i == 0 or j == 0
4:          M[i][j] = 0
5:       else
6:          if x_i == y_j
7:             M[i][j] = M[i − 1][j − 1] + 1
8:          else
9:             M[i][j] = max(M[i][j − 1], M[i − 1][j])
10: return M[n][m]
```

5 Intervalos disjuntos

Sean $[s_1, f_1]$, $[s_2, f_2]$, ..., $[s_n, f_n]$ una secuencia de intervalos cerrados en la recta. A cada intervalo $[s_i, t_i]$ le asignamos un *valor* v_i. Dos intervalos son *compatibles* si no se intersectan.

6 Disjoint Intervals

Let $[s_1, f_1]$, $[s_2, f_2]$, ..., $[s_n, f_n]$ be a sequence of closed intervals on the line. To each interval $[s_i, t_i]$, we assign a *value* v_i. Two intervals are *compatible* if they do not intersect.

Problem Max-Disjoint-Intervals. Given a sequence of closed intervals on the line, find a subset of pairwise compatible intervals with the maximum possible value.

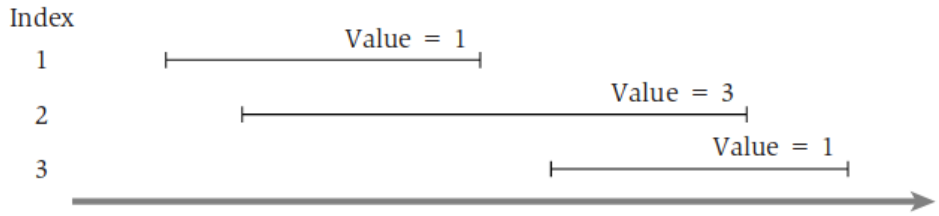


Figure 1: Taken from the book Kleinberg, Algorithm Design

We can assume, without loss of generality, that the intervals are sorted according to their finishing point. That is, $f_1 \leq f_2 \leq \cdots \leq f_n$.

According to this order, we will have the first interval, second interval, etc.

We say that an interval i is *before* an interval j if $f_i \leq f_j$. For each interval j, we define $p(j)$ as the largest interval index $i < j$ such that i and j are compatible.

If no such interval exists, then we define $p(j) = 0$.

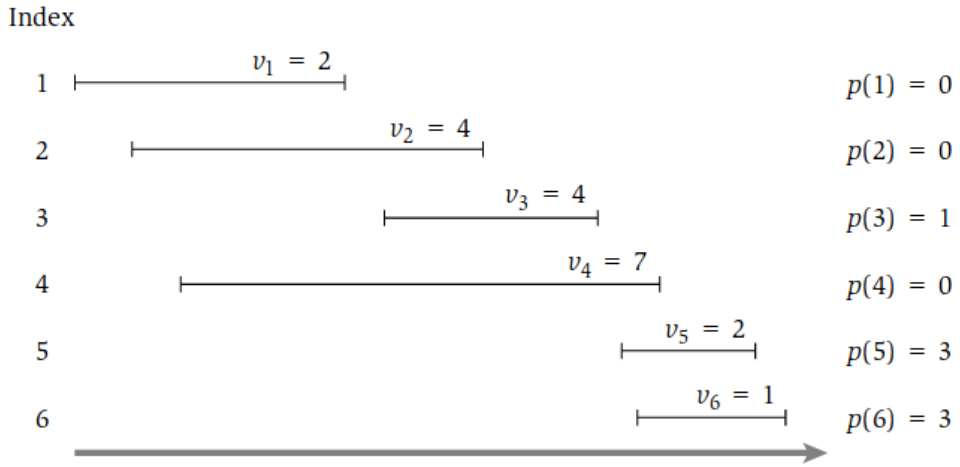


Figure 2: Taken from the book Kleinberg, Algorithm Design

We will start to formally define the problem. Consider the following variables:

- $\mathcal{I}$: List of intervals $[I_1, I_2, \ldots, I_n]$, sorted by finishing point.
- $\mathcal{I}_\rangle$: List of the first i intervals of $\mathcal{I}$.
- $v[i]$: the value contributed by interval i.
- $val(S) = \sum_{I_i \in S} v[i]$
- $p(i)$: the maximum index $j < i$ such that $I_i \cap I_j = \emptyset$.
- $dp(i)$: The maximum value of a subset of intervals of $S \subset I_i$ that are pairwise disjoint.

For convenience, we will say that X is valid if no pair of its elements intersects. The problem can be formulated as computing $dp(n)$:

$$dp(n) = \max\{val(X) : X \subset I_n \text{ and X is valid}\}$$

Lemma 6.1.

$$dp(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\{v[i] + dp(p(i)), dp(i-1)\} & \text{otherwise} \end{cases}$$

Proof. Let X be an arbitrary solution to the problem with input I_i. In particular, X is valid.

There are two types of solutions, from the perspective of I_i.

Case 1: $I_i \notin X$. In this case, consider all X that satisfy the following conditions:

- $X \in \mathcal{I}_{i-1}$
- X is valid

The value of the best candidate is $dp(i-1)$.

Case 2: $I_i \in X$.
We will construct $X' = X - \{I_i\}$. Originally we have this:

- $X' \in \mathcal{I}_{i-1}$
- X' is valid

But not only is X' valid. It also holds that $I_j \cap I_i = \emptyset$, for all $I_j \in X'$. By definition of the function p, this implies that all indices of the intervals in X' are at most $p(i)$.

The conditions for X' are these:

- $X' \in \mathcal{I}_{p(i)}$
- X' is valid.

The value of the best candidate is $dp(p(i))$.

Each case has its best candidate. If we want the best of either case, the answer is the maximum of both candidates. This proves the lemma. □

The previous analysis allows us to design a recursive algorithm for the problem.

Input: A sequence of intervals $\mathcal{I} = \{[s_i, f_i] : i \in \{1 \dots n\}\}$ with weights v_i sorted by finishing point. An integer j.

Output: The value of a subset of compatible intervals with maximum weight among the first j intervals.

RECURSIVE-INTERVALS$(\mathcal{I}, j)$
1: **if** $j = 0$
2: **return** 0
3: OPT_1= RECURSIVE-INTERVALS$(\mathcal{I}, p(j))$
4: OPT_2= RECURSIVE-INTERVALS$(\mathcal{I}, j-1)$
5: **return** $\max\{v_j + \text{OPT}_1, \text{OPT}_2\}$

The proof of correctness of the algorithm is direct from the case analysis. It suffices to prove by induction.

Theorem 6.1. *The algorithm* RECURSIVE-INTERVALS *performs as requested.*

Proof. By induction on j. When $j = 0$ then there are no intervals and any subset will have value 0. Suppose now that $j > 0$. By the induction hypothesis, lines 4 and 5 return values of optimal solutions for the subproblems with intervals $1, \dots, p(j)$ and $1 \dots j-1$, respectively. Then, $\text{OPT}_1 = OPT(p(j))$ and $\text{OPT}_2 = OPT(j-1)$. The proof follows from Lemma 3.1. □

Note that the running time of the algorithm is exponential.

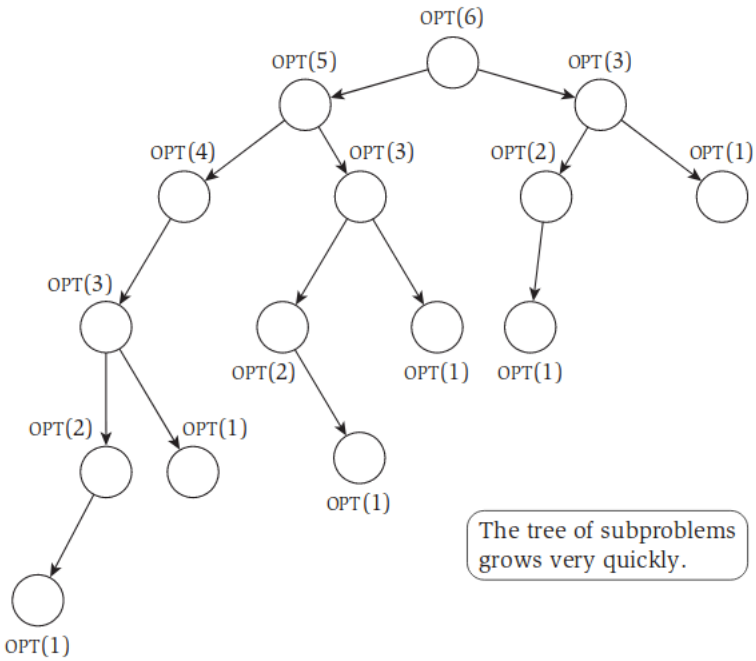


Figure 3: Taken from the book Kleinberg, Algorithm Design

In a worst case we would have the following recurrence $T(n) = T(n-1) + T(n-2)$:

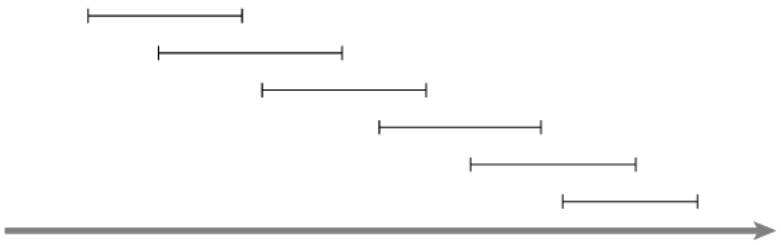


Figure 4: Taken from the book Kleinberg, Algorithm Design

It is clear that in this case $T(n) = \Omega(2^n)$.

Memoized Version

Note that there are only n possible subproblems. This gives us an idea to develop a polynomial algorithm in n.

The issue with the recursive algorithm is that we are solving the same problem more than once. To store previous solutions we use the memoization technique.

Input: A sequence of intervals $\mathcal{I} = [s_i, f_i] : i = 1 \dots n$ with weights v_i sorted by finishing point. An integer j.

Output: The maximum weight of a subset of intervals with maximum weight among the first j intervals.

MEMOIZED-INTERVALS($\mathcal{I}, j$)

1: **if** $j = 0$
2: **return** 0
3: **if** $M[j] \neq -1$
4: **return** $M[j]$
5: OPT_1= MEMOIZED-INTERVALS($\mathcal{I}, p(j)$)
6: OPT_2= MEMOIZED-INTERVALS($\mathcal{I}, j-1$)
7: $M[j] = \max\{v_j + \text{OPT}_1, \text{OPT}_2\}$
8: **return** $M[j]$

Note that the running time of MEMOIZED-INTERVALS is determined by the number of calls to line 8 of the algorithm and line 4 of the algorithm. Observe that the maximum number of calls to line 8 is n, and the maximum number of calls to line 4 is also n, therefore the running time is $O(n)$.

Observe that the previous algorithm provides *the optimal value of the solution*, however it does not return *an optimal solution* (the set of intervals). Knowing the result of the previous algorithm, we can modify this algorithm to find such a solution.

Input: A sequence of intervals $\mathcal{I} = [s_i, f_i] : i = 1 \dots n$ with weights v_i sorted by finishing point. An integer j. An array M resulting from the algorithm Memoized-Intervals.
Output: A subset of intervals with maximum weight among the first j intervals.
Memoized-Intervals-Solution($\mathcal{I}, j$)
1: **if** $j = 0$
2: **return** $\emptyset$
3: **if** $v_j + M[p(j)] \geq M[j-1]$
4: **return** $\{j\} \cup$ Memoized-Intervals-Solution($\mathcal{I}, p(j)$)
5: **else**
6: **return** Memoized-Intervals-Solution($\mathcal{I}, j-1$)

The running time of Memoized-Intervals-Solution is $O(n)$, since only one recursive call is made each time.

Dynamic Programming (Iterative Version)

We can create an iterative version for the previous problem, since we are only interested in what is stored in M.

Input: A sequence of intervals $\mathcal{I} = \{[s_i, f_i] : i = 1 \dots n\}$ with weights v_i sorted by finishing point in non-decreasing order.
Output: The value of a subset of compatible intervals with maximum weight.
Dynamic-Prog-Intervals($\mathcal{I}$)
1: $M[0] = 0$
2: **for** $j = 1$ **to** n
3: $M[j] = \max\{v_j + M[p(j)], M[j-1]\}$
4: **return** $M[n]$

A similar analysis to the previous cases allows us to demonstrate that the algorithm performs as requested. Furthermore, it is clear that the running time of the algorithm is $\Theta(n)$.

Observation 6.1. *In the analysis of the previous algorithms, the execution time for preprocessing is not considered. For example, to sort the intervals by finishing time we must consume $O(n \lg n)$ using some known sorting algorithm. Furthermore, to compute the values of p, we must also perform an $O(n^2)$ preprocessing which can be improved to $O(n \lg n)$ using binary search.*

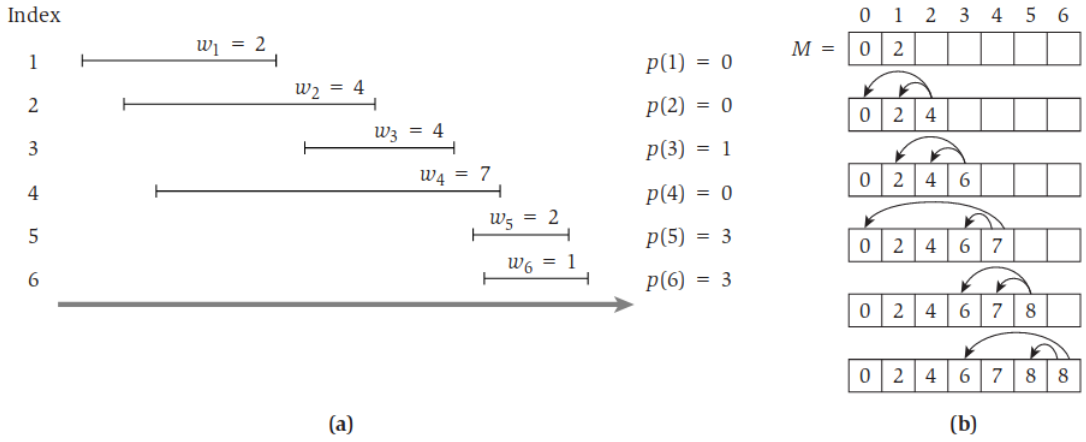


Figure 5: Taken from the book Kleinberg, Algorithm Design

7 Longest Increasing Subsequence

Given an array $A[1..n]$ of numbers, an increasing subsequence is a sequence of indices $(i_1, i_2, \cdots, i_k)$ such that $i_1 < i_2 < \cdots < i_k$ and $A[i_1] < A[i_2] < \cdots < A[i_k]$.

Problem Longest-Increasing-Subsequence. Given an array $A[1..n]$, find an increasing subsequence of maximum size in A.

For example, $(1, 2, 4, 6, 8)$ is a maximum increasing subsequence, with size 5, in the array $[2, 4, 3, 5, 1, 7, 6, 9, 8, 5]$. For each i, let X_i be a maximum subsequence that *ends at* i. For example, for the previous array, we have

$$X_1 = (1),$$

$$X_2 = (1, 2),$$

$$X_3 = (1, 3),$$

$$X_4 = (1, 2, 4),$$

$$X_5 = (5),$$

$$X_6 = (1, 2, 4, 6),$$

$$X_7 = (1, 2, 4, 7),$$

$$X_8 = (1, 2, 4, 6, 8),$$

$$X_9 = (1, 2, 4, 6, 9),$$

$$X_{10} = (1, 2, 10).$$

We reformulate the problem Longest-Increasing-Subsequence as

Problem Longest-Increasing-Subsequence-i. Given an array $A[1..n]$, find an increasing subsequence of maximum size in A that ends at i.

Note then that if $(i_1, i_2, \ldots, i_{k-1}, i_k)$ is an increasing subsequence ending at i_k, then $(i_1, i_2, \ldots, i_{k-1})$ is an increasing subsequence ending at i_{k-1}. Why? If this were not the case, there would exist another sequence ending at i_{k-1} of greater size; by appending i_k to this sequence, we would obtain a larger sequence for the original problem, a contradiction.

Therefore,

Lemma 7.1. *If $OPT(i)$ is the maximum size of an increasing subsequence ending at i, then*

$$OPT(i) = \begin{cases} 0 & : i = 0 \\ \max\{OPT(j) + 1 : j < i, A[j] < A[i]\} & : \textit{otherwise} \end{cases}$$

Proof. Let $X = (j_1, j_2, \ldots, j_k = i)$ be a maximum increasing subsequence that ends at i. Let $X' = (j_1, j_2, \ldots, j_{k-1})$. Note that $j_{k-1} < i$ and $A[j_{k-1}] < A[i]$, then, since X' is an optimal solution for its corresponding subproblem due to the optimal substructure property (see previous paragraph),

$$|X'| \in \{|X_j| : j < i, A[j] < A[i]\},$$

where X_j is an optimal solution ending at j. Suppose for the sake of contradiction that there exists j^* in said range such that $|X_{j^*}| > |X'|$. In that case, $X_{j^*} \cup \{i\}$ would be a better solution than X for the original problem, a contradiction. This demonstrates that

$$|X'| = \max\{|X_j| : j < i, A[j] < A[i]\},$$

as we wanted to show. □

With this lemma in hand, we will design a Dynamic Programming algorithm.

Input: An array $A[1..n]$.
Output: The size of a longest increasing subsequence.

LIS-Dynamic-Prog(A)
1: $M[0] = 0$
2: **for** $i = 1$ **to** n
3: $\quad M[i] = \max\{M[j] : 0 \leq j < i, A[j] < A[i]\} + 1$
4: **return** $\max\{M[i] : 1 \leq i \leq n\}$

It is clear that the algorithm has a running time of $O(n^2)$.

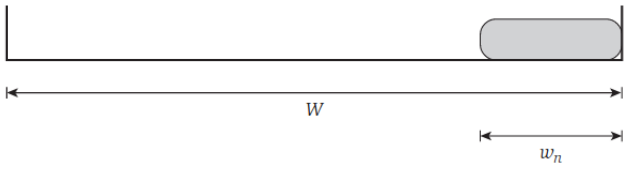
8 Subset Sum and Knapsack

Problem Max-Subset-Sum. Given a set $\{1, 2, \ldots, n\}$ of items each with a natural weight w_i, and a natural number W, find a subset of items whose sum of weights is the largest possible, but less than or equal to W.

Note that a greedy algorithm does not work: $w = [12, 10, 9]$, $W = 20$. We will design a dynamic programming algorithm.

We must find a subproblem. If X is an optimal solution to the problem and the last element is not in X, then X is also an optimal solution for the first $n-1$ elements (Why?).

If the last element is in X, then we can ensure that $X \setminus \{n\}$ is an optimal solution for the first $n-1$ elements *among those that have weight at most* $W - w_n$.



Therefore, our subproblems must consider one more associated variable: the maximum weight. Let $OPT(i, W)$ be the value of an optimal solution that uses items $\{1, 2, \ldots, i\}$ and has weight less than or equal to W. Let X be an optimal solution associated with $OPT(n, W)$. We have that

- If $n \in X$ then $\text{OPT}(n, W) = \text{OPT}(n-1, W - w_n) + w_n$
- If $n \notin X$ then $\text{OPT}(n, W) = \text{OPT}(n-1, W)$

Lemma 8.1. *Let $OPT(i, j)$ be the value of an optimal solution that uses items $\{1, 2, \ldots, i\}$ and has weight less than or equal to j. Then*

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, j) & \text{if } i > 0 \text{ and } w_i > j \\ \max\{OPT(i-1, j-w_i) + w_i, OPT(i-1, j)\} & \text{if } i > 0 \text{ and } w_i \leq j \end{cases}$$

Proof. Exercise. □

Based on the previous recurrence, we can design the following dynamic programming algorithm.

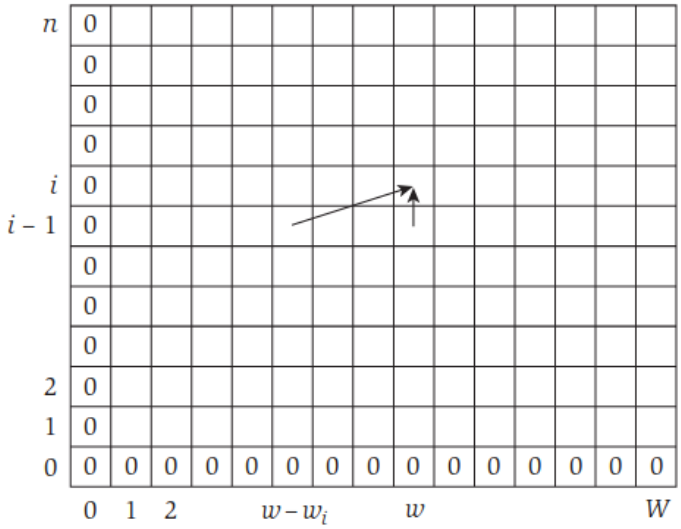
Input: An array $w[1..n]$ of natural numbers (weights) and a natural number W.
Output: An optimal solution for the Subset-Sum problem.

SubsetSum-DynamicProg(w, W)

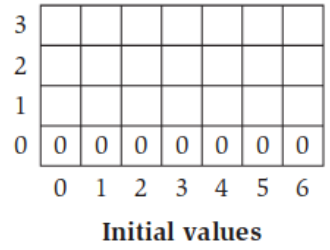
```
1: for j = 0 to W
2:     M[0, j] = 0
3: for i = 1 to n
4:     for j = 0 to W
5:         if w[i] > j
6:             M[i, j] = M[i − 1, j]
7:         else
8:             M[i, j] = max{M[i − 1, j], M[i − 1, j − w[i]] + w[i]}
```

9: **return** $M[n][W]$

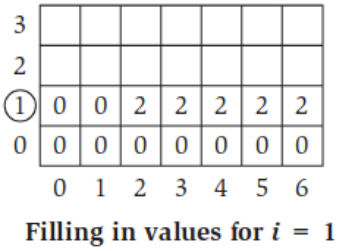
It is clear that the algorithm has a running time of $O(nW)$ (pseudo-polynomial).



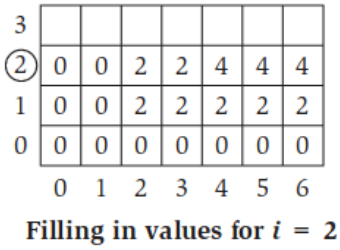
Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$



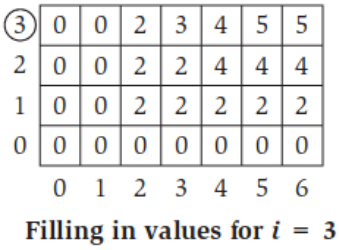
Initial values



Filling in values for $i = 1$



Filling in values for $i = 2$



Filling in values for $i = 3$

Exercise 8.1. *Design an algorithm to obtain an optimal solution (not just the value of said solution).*

Knapsack Problem

Suppose that in addition to the weight vector, we also receive a vector v of *values* for each item. The objective being to maximize the value of a solution while restricting the weight.

A similar analysis to the previous one allows us to deduce that if $OPT(i, W)$ is the value of an optimal solution that uses items $\{1, 2, \ldots, i\}$ and has weight less than or equal to W. Let X be an optimal solution for the problem $OPT(n, W)$. We have that

- If $n \in X$ then $\text{OPT}(n, W)$=$\text{OPT}(n-1, W-w_n) + v_n$
- If $n \notin X$ then $\text{OPT}(n, W)$=$\text{OPT}(n-1, W)$

Exercise 8.2. *Design an algorithm for the knapsack problem.*

9 Fair Linear Partition

Given an array $A[1..n]$ of non-negative numbers, a *linear partition* of A is a sequence of indices $P = (i_1, i_2, \ldots, i_{k+1})$ where $1 = i_1 < i_2 < \ldots < i_{k+1} = n$. We say that said partition has size k. The weight of said partition is given by

$$w(P) = \max\{\sum_{j=i_1}^{i_2} A[j], \sum_{j=i_2+1}^{i_3} A[j], \ldots, \sum_{j=i_k+1}^{i_{k+1}} A[j]\}$$

For example, if $A = [10, 20, 30, 40, 50, 60, 70, 80, 90]$, a partition could be $[1, 3, 6, 9]$ and the weight of said partition would be $\max\{10+20+30, 40+50+60, 70+80+90\} = 240$.

Problem Min-Partition. Given an array A of non-negative integers and a number k, find a partition of size k with minimum weight.

Let $OPT(n, k)$ be the value of an optimal partition.
Note that

$$OPT(n,k) = \begin{cases} \sum_{i=1}^{n} A[i] & \text{if } k = 1 \\ \min_{\ell=k}^{n} \max\{OPT(\ell-1, k-1), \sum_{j=\ell}^{n} A[j]\} & \text{otherwise} \end{cases}$$

We can design a dynamic programming algorithm.

```
MinPartition-DynamicProg(A, k)
 1: for i = 1 to n
 2:    M[i, 1] = Σ_{p=1}^{i} A[p]
 3: for i = 2 to n
 4:    for j = 2 to k
 5:       M[i, j] = ∞
 6:       for ℓ = j to i
 7:          cost = max{M[ℓ − 1][j − 1], Σ_{p=ℓ}^{i} A[p]}
 8:          if cost < M[i, j]
 9:             M[i, j] = cost
10: return  M[n][k]
```

It is clear that the running time is $O(n^2k)$.