Chapter 10
Algorithm Design Techniques

Dynamic Programming

- dynamic programming
  - very general approach to finding an optimal path through the state space of feasible states
  - state space may be represented as a directed graph with feasible states as nodes and feasible decisions as arcs
    - forms the decision network

Dynamic Programming

- we apply dynamic programming to a problem when
  - the problem can be divided into stages with a decision made at each stage
  - each stage has a number of possible states associated with it
  - the decision made at each stage describes how the state at the current stage leads to the state at the next stage
  - given the current state, the optimal choice for the remaining stages does not depend on previous decisions or their associated states

Dynamic Programming

– formulating dynamic programming recursions

The dynamic programming recursion (for minimization) is

$$f_t(i) = \min\{\text{cost during stage } t + f_{t+1}(\text{new state at stage } t+1)\}.$$

To derive the recursion we need to identify:

- the set of feasible decisions for the given state and stage;
- how the cost during the current stage t depends on t, the current state, and the decision chosen at stage t; and
- how the state at stage $t+1$ depends on t, the state at stage t, and the decision chosen at stage t.

Dynamic Programming

– the knapsack problem

We have a knapsack that can hold a weight of at most W.

We can choose from T different types of articles to pack.

Each article of type t has (integer) weight w_t and (integer) value of v_t.

We wish to load a knapsack to maximize the total value of the articles included, subject to the capacity constraint.

Let

$$x_t = \text{number of articles of type } t \text{ included.}$$

An integer programming formulation is

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_t v_t x_t \\ \text{subject to} \quad & \textstyle\sum_t w_t x_t \le W \\ & x_t \ge 0 \text{ integer for all } t. \end{aligned}$$

Dynamic Programming

– the knapsack problem (cont.)

The only resource here is weight.

Let $f_t(d)$ be the maximum value of items $t, t+1, \ldots, T$ if their combined weight is $\leq d$.

The dynamic programming recursion is

$$f_{T+1}(d) = 0;$$
$$f_t(d) = \max_{x_t \in \mathbb{Z}_+} \{ v_t x_t + f_{t+1}(d - w_t x_t) \mid w_t x_t \leq d \}.$$

The optimal solution we seek corresponds to $f_1(W)$.

We start by computing f_T for all possible states and work our way back to $f_1(W)$.

Dynamic Programming

– the knapsack problem (cont.)

To illustrate this approach, suppose

item type	weight	value
1	4	11
2	3	7
3	5	12

and that $W = 10$.

Observe that we can work forwards from $t = 1$ and $d = 10$ to eliminate some states from consideration.

Dynamic Programming

– the knapsack problem (cont.)

The preceding figure makes clear that we need only compute

$$f_3(0),\ f_3(1),\ f_3(2),\ f_3(3),\ f_3(4),\ f_3(6),\ f_3(7),\ f_3(10)$$

and

$$f_2(2),\ f_2(6),\ f_2(10),$$

and, of course, $f_1(10)$.

Dynamic Programming

– the knapsack problem (cont.)

If our weight allowance d is 0 when we must decide how many of Item 3 to pack, then all we can do is choose 0 of the Item 3, for a net value of 0:

$$f_3(0) = 0.$$

Similarly,

$$f_3(1) = f_3(2) = f_3(3) = f_3(4) = 0.$$

because Item 3 weighs 5 pounds.

On the other hand,

$$f_3(5) = f_3(6) = f_3(7) = f_3(8) = f_3(9) = 12,$$

since in these cases we have 5–9 pounds of capacity available to us, so we can fit one of the 5 lb Item 3 into the knapsack.

Finally,

$$f_3(10) = 24,$$

since in this case we can space for 2 of Item 3.

– the knapsack problem (cont.)

In the next stage we compute

$$f_2(2),\ f_2(6),\ f_2(10).$$

We have

$$f_2(d) = \max_{x_2 \in \mathbb{Z}_+} \{\, 7x_2 + f_3(d - 3x_2) \mid 3x_2 \le d \,\}.$$

If $d = 2$ our only option is to choose none of Item 2, so

$$f_2(2) = \max_{x_2 \in \mathbb{Z}_+} \{\, 7x_2 + f_3(2 - 3x_2) \mid 3x_2 \le 2 \,\} = 0 + f_3(2) = 0.$$

Dynamic Programming

– the knapsack problem (cont.)

On the other hand,

$$f_2(6) = \max_{x_2 \in \mathbb{Z}_+} \{ \, 7x_2 + f_3(6 - 3x_2) \mid 3x_2 \le 6 \, \}$$

$$= \begin{cases} 0 + f_3(6) & = & 12 & x_2 = 0 \\ 7 + f_3(3) & = & 7 & x_2 = 1 \\ 14 + f_3(0) & = & 14 & x_2 = 2 \end{cases}$$

$$f_2(10) = \max_{x_2 \in \mathbb{Z}_+} \{ \, 7x_2 + f_3(10 - 3x_2) \mid 3x_2 \le 10 \, \}$$

$$= \begin{cases} 0 + f_3(10) & = & 24 & x_2 = 0 \\ 7 + f_3(7) & = & 19 & x_2 = 1 \\ 14 + f_3(4) & = & 14 & x_2 = 2 \\ 21 + f_3(1) & = & 21 & x_2 = 3 \end{cases}$$

– the knapsack problem (cont.)

Finally,

$$f_1(10) = \max_{x_1 \in \mathbb{Z}_+} \{ 11x_1 + f_2(10 - 4x_1) \mid 4x_1 \leq 10 \}$$

$$= \begin{cases} 0 + f_2(10) & = & 24 & x_2 = 0 \\ 11 + f_2(6) & = & 25 & x_1 = 1 \\ 22 + f_2(2) & = & 22 & x_1 = 2 \end{cases}$$

The optimal strategy is

1. one of Item 1,
2. two of Item 2,
3. zero of Item 3.

Dynamic Programming

– an alternative approach

The following bottom-up approach leads to a simpler solution algorithm:

1. first determine how to fill a smaller knapsack optimally, then
2. use this knowledge to fill a larger knapsack optimally.

Let $V(w)$ denote the maximum value of a w-lb knapsack.

To fill a w-lb knapsack optimally, we must begin by packing an item. If we begin with an item of type t then the best value we can achieve is

$$v_t + \text{the best we can do with a } (w - w_t)\text{-lb knapsack.}$$

This leads to the recurrence

$$V(0) = 0$$
$$V(w) = \max_t \{ v_t + V(w - w_t) \mid w_t \le w \}, \quad w > 0.$$

Dynamic Programming

– an alternative approach (cont.)

To illustrate this approach, suppose

item type	weight	value
1	4	11
2	3	7
3	5	12

and that $W = 10$.

Clearly,

$$V(0) = V(1) = V(2) = 0,$$

since no item weighs 2 pounds or less, and

$$V(3) = 7$$

since only an item of type 2 will fit in the 3-lb knapsack.

Dynamic Programming

– an alternative approach (cont.)

Now follow the recursion to fill out the values of V:

$$V(4) = \max\begin{cases} 11 + V(0) = 11 & \text{type 1} \\ 7 + V(1) = 7 & \text{type 2} \end{cases}$$

$$V(5) = \max\begin{cases} 11 + V(1) = 11 & \text{type 1} \\ 7 + V(2) = 7 & \text{type 2} \\ 12 + V(0) = 12 & \text{type 3} \end{cases}$$

$$V(6) = \max\begin{cases} 11 + V(2) = 11 & \text{type 1} \\ 7 + V(3) = 14 & \text{type 2} \\ 12 + V(1) = 12 & \text{type 3} \end{cases}$$

$$V(7) = \max\begin{cases} 11 + V(3) = 18 & \text{type 1} \\ 7 + V(4) = 18 & \text{type 2} \\ 12 + V(2) = 12 & \text{type 3} \end{cases}$$

Dynamic Programming

– an alternative approach (cont.)

$$V(8) = \max \begin{cases} 11 + V(4) = 22 & \text{type 1} \\ 7 + V(5) = 19 & \text{type 2} \\ 12 + V(3) = 19 & \text{type 3} \end{cases}$$

$$V(9) = \max \begin{cases} 11 + V(5) = 23 & \text{type 1} \\ 7 + V(6) = 21 & \text{type 2} \\ 12 + V(4) = 23 & \text{type 3} \end{cases}$$

$$V(10) = \max \begin{cases} 11 + V(6) = 25 & \text{type 1} \\ 7 + V(7) = 25 & \text{type 2} \\ 12 + V(5) = 24 & \text{type 3} \end{cases}$$

Starting with a 10 lb knapsack, one optimal selection is given by

1. a type 1 item, leaving 10 - 4 = 6 lb;
2. a type 2 item, leaving 6 - 3 = 3 lb;
3. a type 2 item, leaving 3 - 3 = 0 lb.

Dynamic Programming

– computational complexity

This DP approach requires we compute $V(0), \ldots, V(W)$, and each $V(w)$ requires we look at (at most) T sums.

Thus, $O(WT)$ operations are required.

However, the knapsack problem is NP-hard!

This DP solution of knapsack is a pseudo-polynomial time algorithm—the run-time is polynomial in the numeric value of the input W, not the number of bits in W (length of the input).

Suppose it takes $m > 1$ bits to represent W. This means $2^{m-1} \leq W \leq 2^m - 1$, so the DP approach is actually exponential in m.

Multiplication

– how fast can we multiply?

If we multiply two n-digit numbers in the obvious way, the time required is proportional to n^2.

$$
\begin{array}{rr}
 & 123 \\
\times & 456 \\
\hline
 & 738 \\
+ & 7150 \\
+ & 49200 \\
\hline
 & 56088
\end{array}
$$

Can we do better?

Multiplication

– standard multiplication

Given two $2n$-bit numbers $u = (u_{2n-1} \cdots u_1 u_0)_2$ and $v = (v_{2n-1} \cdots v_1 v_0)_2$, we can write

$$u = 2^n U_1 + U_0$$
$$v = 2^n V_1 + V_0,$$

where $U_1 = (u_{2n-1} \cdots u_1 u_n)_2$ consists of the n most significant bits of u, while $U_0 = (u_{n-1} \cdots u_1 u_0)_2$ are the n least significant bits, and similarly for V_1, V_0.

The obvious way of multiplication is

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1V_0 + U_0V_1) + (2^n + 1)U_0V_0.$$

Multiplications by powers of 2 are $O(n)$ left shifts and $\pm$ is also $O(n)$.

Recursion for runtime T:

$$T(2n) = 4T(n) + cn \quad \Rightarrow \quad T(n) = \Theta(n^2).$$

Multiplication

– multiplication by divide and conquer

A faster approach (A. A. Karatsuba (1962)):

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_1 - V_0) + (2^n + 1)U_0V_0.$$

This is true since

$$(U_1 - U_0)(V_1 - V_0) = U_1V_1 - U_1V_0 - U_0V_1 + U_0V_0.$$

Multiplication

– recursion for the complexity

Let $T(2n)$ be the time needed to compute the product of two $2n$-bit numbers via

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_1 - V_0) + (2^n + 1)U_0V_0.$$

How many multiplications are on the right?

There are only 3 multiplications, since the multiplications by powers of 2 are just shifts. The cost of the shifts are $\propto n$

There are also a bunch of additions, but this work is also $\propto n$.

This leads to the recursion

$$T(2n) = 3T(n) + cn$$
$$T(1) = c'.$$

Multiplication

– solution via reduction

Suppose $2n = 2^m$. From the recursion

$$T(2n) = 3T(n) + cn$$
$$T(1) = c'$$

we obtain the following:

$$T(n) = 3T(n/2) + c(n/2),$$

so

$$\begin{aligned} T(2n) &= 3(3T(n/2) + c(n/2)) + cn \\ &= 9T(n/2) + 3c(n/2)) + cn \end{aligned}$$

Multiplication

– solution via reduction (cont.)

Again applying the recursion, we obtain

$$T(n/2) = 3T(n/4) + c(n/4),$$

so

$$T(2n) = 27T(n/4) + 9c(n/4) + 3c(n/2) + cn.$$

Now a pattern has emerged: we conjecture that after k steps of this process,

$$T(2n) = 3^k T(2^{-k} 2n) + \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i cn.$$

Multiplication

– complexity

If we repeat this $k = m$ times, so $2^k = 2^m = 2n$, we obtain

$$T(2n) = 3^m T(1) + \sum_{i=0}^{m-1} \left(\frac{3}{2}\right)^i cn$$
$$= 3^m c' + cn \frac{1-(3/2)^m}{1-(3/2)} = 3^m c' + 2cn((3/2)^m - 1)$$

The dominant term is $3^m c'$, and

$$3^m = 3^{\lg n} = (2^{\lg 3})^{\lg n} = (2^{\lg n})^{\lg 3} = n^{\lg 3} = n^{1.5850...}$$

so the divide-and-conquer algorithm is $\Theta(n^{1.585})$.

Matrix Multiplication

Suppose A and B are $n \times n$ matrices.

How fast can we compute AB?

Standard matrix multiplication:

```
C = 0 // C <- A*B
for i = 1 to n {
  for j = 1 to n {
    for k = 1 to n {
      C(i,j) += A(i,k)*B(i,k)
    }
  }
}
```

This is $\Theta(n^3)$.

Matrix Multiplication

– block matrix multiplication

Suppose $n = 2^m$ for some m. Write A and B in terms of $n/2 \times n/2$ blocks:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Standard matrix multiplication:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 $n/2 \times n/2$ matrix products and 4 $n/2 \times n/2$ matrix additions.

Matrix Multiplication

- Strassen's fast matrix multiplication (1969)

$$\begin{aligned}
\text{I} &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
\text{II} &= (A_{21} + A_{22})B_{11} \\
\text{III} &= A_{11}(B_{12} - B_{22}) \\
\text{IV} &= A_{22}(-B_{11} + B_{21}) \\
\text{V} &= (A_{11} + A_{12})B_{22} \\
\text{VI} &= (-A_{11} + A_{21})(B_{11} + B_{22}) \\
\text{VII} &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Matrix Multiplication

– Strassen's fast matrix multiplication (1969) (cont.)

$$
\begin{aligned}
C_{11} &= \mathsf{I} + \mathsf{IV} - \mathsf{V} + \mathsf{VII} \\
C_{12} &= \mathsf{III} + \mathsf{V} \\
C_{21} &= \mathsf{II} + \mathsf{IV} \\
C_{22} &= \mathsf{I} + \mathsf{III} - \mathsf{II} + \mathsf{VI}
\end{aligned}
$$

7 $n/2 \times n/2$ matrix products and 18 $n/2 \times n/2$ matrix additions.

Matrix Multiplication

– Strassen's trick

Strassen trades an $O((n/2)^3)$ matrix product for 14 $O((n/2)^2)$ matrix additions.

Now apply the algorithm recursively to compute the $n/2 \times n/2$ matrix products.

If $T(n)$ is the time it takes to compute an $n \times n$ matrix product using Strassen, then

$$T(n) = 7\,T(n/2) + 18n^2.$$

This recurrence leads to

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81}) \quad (4.7\,n^{2.81}).$$

This is better than the $O(n^3)$ complexity of standard matrix multiplication!

Matrix Multiplication

– Strassen's – practical concerns

This discussion assumes A and B are square but there exist variants for rectangular matrices.

We can compute and use the terms I–VII one at a time, so we need not store all of them.

Some extra storage is needed because of the recursion.

At some point in the recursion standard matrix multiplication becomes more efficient so we switch.

Matrix Multiplication

– Strassen's algorithm for matrix inversion

Strassen's algorithm for inversion has a complexity bounded by $5.64\, n^{\log_2 7}$.

Let

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad A^{-1} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Matrix Multiplication

– Strassen's algorithm for matrix inversion (cont.)

Then

$$
\begin{aligned}
\mathsf{I} &= A_{11}^{-1} \\
\mathsf{II} &= A_{21}\mathsf{I} \\
\mathsf{III} &= \mathsf{I}A_{12} \\
\mathsf{IV} &= A_{21}\mathsf{III} \\
\mathsf{V} &= \mathsf{IV} - A_{22} \\
\mathsf{VI} &= \mathsf{V}^{-1} \\
C_{12} &= \mathsf{III} \cdot \mathsf{VI} \\
C_{21} &= \mathsf{VI} \cdot \mathsf{II} \\
\mathsf{VII} &= \mathsf{III} \cdot C_{21} \\
C_{11} &= \mathsf{I} - \mathsf{VII} \\
C_{22} &= -\mathsf{VI}.
\end{aligned}
$$

Matrix Multiplication

- state of the art

  Winograd (1972): Variant of Strassen with 7 matrix-matrix products and 15 matrix-matrix additions $\Rightarrow \Theta(n^{\log_2 7})$ with better constant.

  Pan (1978): $\Theta(n^{2.795})$.

  Coppersmith and Winograd (1990): $\Theta(n^{2.376})$.

  Le Gall's variant of Coppersmith and Winograd (2014): $\Theta(n^{2.373})$—best known.

  Cohn, Kleinberg, Szegedy, Umans (2005): Conjectures based on group theory which, if true, implies $\Theta(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

  Clearly $\Theta(n^2)$ is a lower bound on matrix multiplication—it takes n^2 operations just to write down the answer.

  Conjecture: Matrix multiplication can be performed in $\Theta(n^{2+\varepsilon})$ for any $\varepsilon > 0$.