Algorithms

Algorithms

Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always these are the result of strategic breakthroughs rather than tactical cleverness. Sometimes the strategic breakthrough will be a new algorithm, such as the Cooley-Tukey Fast Fourier Transform or the substitution of an $n \log n$ sort of an n^2 set of comparisons. … Representation *is* the essence of programming.
—Fred Brooks, 1974 (lead on IBM/360, Turing Award)

Algorithms: Modern Version

Smart data structures and dumb code works a lot better than the other way around.
—Guy Steele, 2002 (inventor of Scheme)

Types of Algorithms

- divide and conquer
- dynamic programming
- greedy
- backtracking
- reduction

Rod Cutting

Problem Description

Given table of profits p_i for each possible integer length i, find the best way to cut a rod of length n. Cuts are free, but must be integer lengths.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	14	15	17	20	24	30

2^{n-1} possible solutions. Can we solve it faster?

Problem Description

Given table of profits p_i for each possible integer length i, find the best way to cut a rod of length n. Cuts are free, but must be integer lengths.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	14	15	17	20	24	30

2^{n-1} possible solutions. Can we solve it faster?

Optimal Value

Step 1: write down value of optimal solution

$$
\begin{aligned}
best(n) &= \text{best profit achievable for length } n \\
&= \max_{1 \le first \le n} (p_{first} + best(n - first)) \\
best(0) &= 0
\end{aligned}
$$

What is the complexity of the naive recursive algorithm?
How can we make this efficient?

An Algorithm

Step 2: compute optimal value

CUTROD(p, n)

1: best[0] $\leftarrow$ 0
2: **for** len from 1 to n **do**
3: best[len] $\leftarrow \max\limits_{1 \le first \le len} (p[first] + best[len - first])$
4: **return** best[n]

Solution Recovery

$\text{CUTROD}(p,n)$

```
best[0] ← 0
for len from 1 to n do
    best[len] ← −∞
    for first from 1 to len do
        this ← p[first]+best[len-first]
        if this > best[len] then
            best[len] ← this
            cut[len] ← first
n0 ← n
while n0 > 0 do
    print cut[n0]
    n0 ← n0 − cut[n0]
return best[n]
```

Properties

- optimal substructure: global optimum uses optimal solutions of subproblems
- ordering of subproblems: solve 'smallest' first, build larger solutions from smaller
- 'overlapping' subproblems: polynomial number of subproblems, used multiple times
- independent subproblems: optimal solution of one subproblem doesn't affect optimality of another