CSE 417 Algorithms

Lecture 17, Winter 2020
Divide and Conquer
Dynamic Programming

Announcements

-

Divide and Conquer Algorithms

- Mergesort, Quicksort
- Strassen's Algorithm
- Median
- Inversion counting
- Closest Pair Algorithm (2d)
- Integer Multiplication (Karatsuba's Algorithm)

Integer Arithmetic

```
    9715480283945084383094856701043643845790217965702956767
+   1242431098234099057329075097179898430928779579277597977
```

Runtime for standard algorithm to add two n digit numbers:

```
    2095067093034680994318596846868779409766717133476767930
X   5920175091777634709677679342929097012308956679993010921
```

Runtime for standard algorithm to multiply two n digit numbers:

Recursive Multiplication Algorithm (First attempt)

$x = x_1 2^{n/2} + x_0$

$y = y_1 2^{n/2} + y_0$

$xy = (x_1 2^{n/2} + x_0) (y_1 2^{n/2} + y_0)$

$\quad = x_1y_1 2^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0$

Recurrence:

Run time:

Simple algebra

$x = x_1 2^{n/2} + x_0$

$y = y_1 2^{n/2} + y_0$

$xy = x_1y_1 2^n + (x_1y_0 + x_0y_1) 2^{n/2} + x_0y_0$

$p = (x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$

Karatsuba's Algorithm

Multiply n-digit integers x and y

Let $x = x_1 2^{n/2} + x_0$ and $y = y_1 2^{n/2} + y_0$
Recursively compute
$a = x_1 y_1$
$b = x_0 y_0$
$p = (x_1 + x_0)(y_1 + y_0)$
Return $a2^n + (p - a - b)2^{n/2} + b$

Recurrence: $T(n) = 3T(n/2) + cn$

$\log_2 3 = 1.58496250073\ldots$

Fast Integer Multiplication

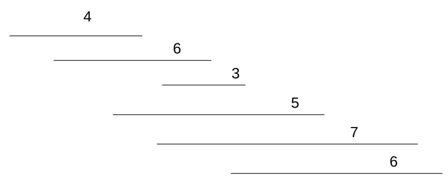
- Grade School $O(n^2)$
- Karatsuba $O(n^{1.58})$
- Toom-Cook $O(n^{1.46})$ [For 3 pieces]
  - $O(n^{1+eps})$ [For k pieces]
- Schonhage-Strassen
  - Fast Fourier Transform based algorithm
  - O(n log n loglog n)
  - Becomes practical for ~25,000 digits

Dynamic Programming

Intervals sorted by end time

Dynamic Programming

- Weighted Interval Scheduling
- Given a collection of intervals $I_1,\ldots,I_n$ with weights $w_1,\ldots,w_n$, choose a maximum weight set of non-overlapping intervals



Intervals sorted by end time

Optimality Condition

- Opt[j] is the maximum weight independent set of intervals $I_1, I_2, \ldots, I_j$
- Opt[j] = max(Opt[j – 1], w_j + Opt[p[j]])
  - Where p[j] is the index of the last interval which finishes before I_j starts

Algorithm

MaxValue(j) =
if j = 0 return 0
else
return max(MaxValue(j-1),
w_j + MaxValue(p[j]))

Worst case run time: 2^n

A better algorithm

M[j] initialized to -1 before the first recursive call for all j

MaxValue(j) =
  if j = 0 return 0;
  else if M[j] != -1 return M[j];
  else
    M[j] = max(MaxValue(j-1), w_j + MaxValue(p[j]));
    return M[j];

Iterative Algorithm

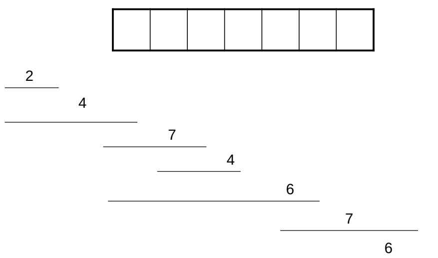
Express the MaxValue algorithm as an iterative algorithm

MaxValue {

}

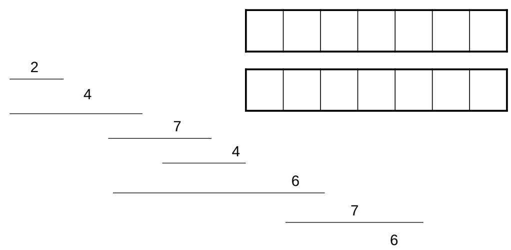
Fill in the array with the Opt values

Opt[j] = max (Opt[j – 1], w_j + Opt[p[j]])



Computing the solution

Opt[j] = max (Opt[j – 1], w_j + Opt[p[j]])
Record which case is used in Opt computation



Dynamic Programming

- The most important algorithmic technique covered in CSE 421
- Key ideas
  - Express solution in terms of a polynomial number of sub problems
  - Order sub problems to avoid recomputation