CS38
Introduction to Algorithms

Lecture 9

April 29, 2014

Outline

- Divide and Conquer design paradigm
  - matrix multiplication
- Dynamic programming design paradigm
  - Fibonacci numbers
  - weighted interval scheduling
  - knapsack
  - matrix-chain multiplication
  - longest common subsequence

Discrete Fourier Transform (DFT)

- Given n-th root of unity ω, DFT_n is a linear map from C^n to C^n:

$$\begin{pmatrix} (\omega^0)^0 & (\omega^0)^1 & (\omega^0)^2 & \cdots & (\omega^0)^{n-1} \\ (\omega^1)^0 & (\omega^1)^1 & (\omega^1)^2 & \cdots & (\omega^1)^{n-1} \\ (\omega^2)^0 & (\omega^2)^1 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\ \vdots & & & & \\ (\omega^{n-1})^0 & (\omega^{n-1})^1 & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1} \end{pmatrix}$$

- (i,j) entry is ω^{ij}

Fast Fourier Transform (FFT)

- DFT_n has special structure (assume $n= 2^k$)
  - reorder columns: first even, then odd
  - consider exponents on ω along rows:

multiples of: | same multiples plus:

0 →	0	0	0	0	0	0	…	0	0	0	0	0	0	…	← 0
2 →	0	2	4	6	8	10	…	1	3	5	7	9	11	…	← 1
4 →	0	4	8	12	16	20	…	2	6	10	14	18	22	…	← 2
6 →	0	6	12	18	24	30	…	3	9	15	21	27	33	…	← 3
8 →	0	8	16	24	32	44	…	4	12	20	28	36	40	…	← 4

rows repeat twice since $\omega^n = 1$

Fast Fourier Transform (FFT)

- so we are actually computing:

ω^2 is (n/2)-th root of unity

$$\text{DFT}_n \cdot \begin{pmatrix} x_{\text{even}} \\ x_{\text{odd}} \end{pmatrix} = \left(\begin{array}{c|c} \text{DFT}_{n/2} & D \cdot \text{DFT}_{n/2} \\ \text{DFT}_{n/2} & \omega^{n/2} \cdot D \cdot \text{DFT}_{n/2} \end{array}\right) \cdot \begin{pmatrix} x_{\text{even}} \\ x_{\text{odd}} \end{pmatrix}$$

- so to compute $\text{DFT}_n \cdot x$

D = diagonal matrix diag(ω^0, ω^1, ω^2, …, $\omega^{n/2-1}$)

FFT(n:power of 2; x)
1. let ω be a n-th root of unity
2. compute a = FFT(n/2, x_{even})
3. compute b = FFT(n/2, x_{odd})
4. y_{even} = a + D·b and y_{odd} = a + $\omega^{n/2}$ ·D·b
5. return vector y

Fast Fourier Transform (FFT)

FFT(n:power of 2; x)
1. let ω be a n-th root of unity
2. compute a = FFT(n/2, x_{even})
3. compute b = FFT(n/2, x_{odd})
4. y_{even} = a + D·b and y_{odd} = a + $\omega^{n/2}$ ·D·b
5. return vector y

- Running time?
  - T(1) = 1
  - T(n) = 2T(n/2) + O(n)
  - solution: T(n) = O(n log n)

matrix multiplication

A X B = C

- given $n \times n$ matrices A, B
- compute $C = AB$
- standard method: $O(n^3)$ operations
- Strassen: $O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's algorithm

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

- how many product operations?
- Strassen: it is possible with 7 (!!)
  - 7 products of form: (linear combos of a entries) x (linear combos of b entries)
  - result is linear combos of these 7 products

Strassen's algorithm

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

- 7 products of form: (linear combos of a entries) x (linear combos of b entries)
- result is linear combos of these 7 products

Key: identity holds when entries above are n/2 x n/2 matrices rather than scalars

Strassen's algorithm

Strassen-matrix-mult(A, B: n x n matrices)
1. $p_1 = A_{11}(B_{12} - B_{22})$
2. $p_2 = (A_{11} + A_{12})B_{22}$
3. $p_3 = (A_{21} + A_{22})B_{11}$
4. $p_4 = A_{22}(B_{21} - B_{11})$
5. $p_5 = (A_{11} + A_{22})(B_{11} + B_{22})$
6. $p_6 = (A_{12} - A_{22})(B_{21} + B_{22})$
7. $p_7 = (A_{11} - A_{21})(B_{11} + B_{12})$
8. $C_{11} = P_5 + P_4 - P_2 + P_6$; $C_{12} = P_1 + P_2$
9. $C_{21} = P_3 + P_4$; $C_{22} = P_5 + P_1 - P_3 - P_7$
10. return C

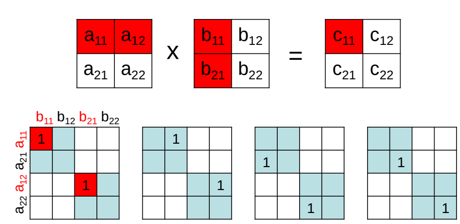
Strassen's algorithm

- 7 recursive calls
- additions/subtractions are entrywise: $O(n^2)$
- running time recurrence?

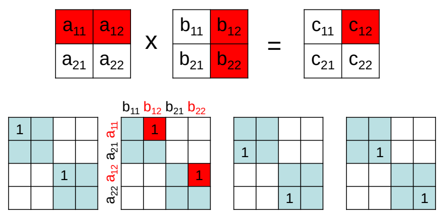
$$T(n) = 7T(n/2) + O(n^2)$$

Solution: $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

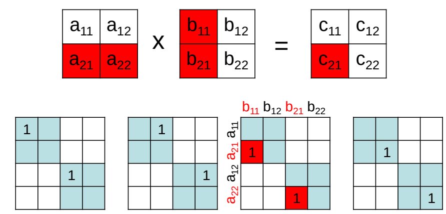
discovering Strassen



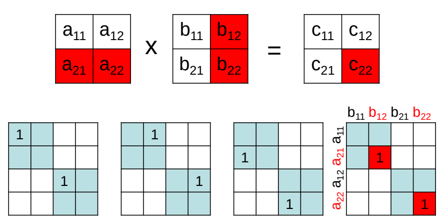
discovering Strassen



discovering Strassen

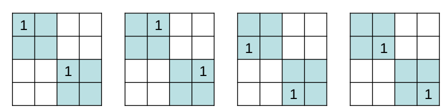


discovering Strassen

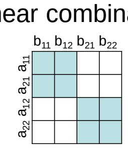


discovering Strassen

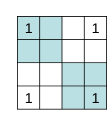
- express these



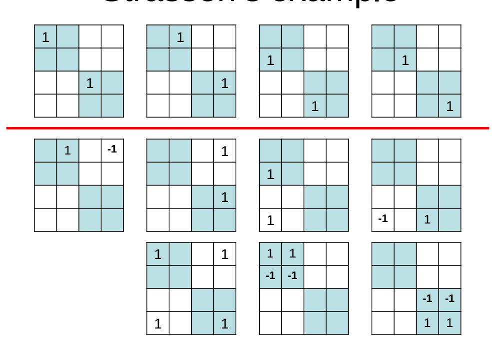
as linear combinations of rank-1 matrices



e.g.:



Strassen's example



Dynamic programming

"programming" = "planning"

"dynamic" = "over time"

- basic idea:
  - identify subproblems
  - express solution to subproblem in terms of other "smaller" subproblems
  - build solution bottom-up by filling in a table
- defining subproblem is the hardest part

Dynamic programming

- Simple example: computing Fibonacci #s
  - $f(1) = f(2) = 1$
  - $f(i) = f(i-1) + f(i-2)$
- recursive algorithm:

Fibonacci(n)
1. if n = 1 or n = 2 return(1)
2. else return(Fibonacci(n-1) + Fibonacci (n-2))

  - running time?

Dynamic programming

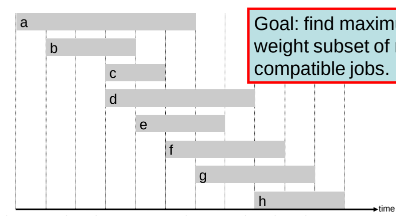
Fibonacci(n)
1. if n = 1 or n = 2 return(1)
2. else return(Fibonacci(n-1) + Fibonacci (n-2))

- better idea:
  - 1-dimensional table; entry i contains f(i)
  - build table "bottom-up"

Fibonacci-table(n)
1. T(1) = T(2) = 1
2. for i = 3 to n do T(i) = T(i-1) + T(i-2)
3. return(T(n))

Weighted interval scheduling

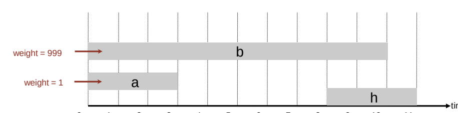
- job j starts at s_j, finishes at f_j, weight v_j
- jobs compatible if they don't overlap



Goal: find maximum weight subset of mutually compatible jobs.

Weighted interval scheduling

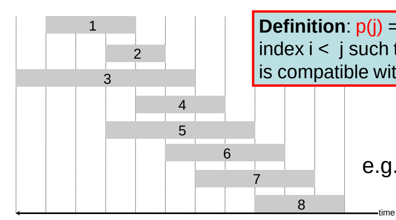
- recall: greedy by earliest finishing time worked when weights were all 1
- counterexample with general weights:



Weighted interval scheduling

- label jobs by finishing time f_j

Definition: p(j) = largest index i < j such that job i is compatible with j.



e.g. p(8) = 5

Weighted interval scheduling

- subproblem j: jobs 1…j
  - OPT(j) = value achieved by optimum schedule
- relate to smaller subproblems
  - case 1: use job j
    - can't use jobs p(j)+1, …, j-1
    - must use optimal schedule for 1… p(j) = OPT(p(j))
  - case 2: don't use job j
    - must use optimal schedule for 1… j-1 = OPT(j-1)

p(j) = largest index i such that job i is compatible with j.

Weighted interval scheduling

- job j starts at s_j, finishes at f_j, weight v_j
- OPT(j) =

  max {v_j + OPT(p(j)), OPT(j-1)}

  recursive solution?

p(j) = largest index i such that job i is compatible with j.

wtd-interval-schedule ((s_1, f_1, v_1),…, (s_n, f_n, v_n))
1. a = v_j + **wtd-interval-schedule(**first p(n) jobs)
2. b = **wtd-interval-schedule(**first j-1 jobs)
3. return(max(a,b))

running time?

Weighted interval scheduling

- job j starts at s_j, finishes at f_j, weight v_j
- OPT(j) =

  max {v_j + OPT(p(j)), OPT(j-1))}

p(j) = largest index i such that job i is compatible with j.

Wtd-interval-schedule((s_1, f_1, v_1),…, (s_n, f_n, v_n)**)**
1. OPT(0) = 0
2. sort by finish times f_i; compute p(i) for all i
3. for i = 1 to n
4. OPT(i) = max {v_i + OPT(p(i)), OPT(i-1)}
5. return(OPT(n))

running time?

Weighted interval scheduling

Store extra info:
1. was job i picked?
2. which table cell has solution to resulting subproblem?

Wtd-interval-schedule((s_1, f_1, v_1),…, (s_n, f_n, v_n)**)**
1. OPT(0) = 0
2. sort by finish times f_i; compute p(i) for all i
3. for i = 1 to n
4. OPT(i) = max {v_i + OPT(p(i)), OPT(i-1))}
5. return(OPT(n))

- OPT(n) gives value of optimal schedule
  - how do we actually find schedule?

Knapsack

- item i has weight w_i and value v_i
- goal: pack knapsack of capacity W with maximum value set of items
  - greedy by weight, value, or ratio of weight/value all fail
- subproblems:
  - optimum among items 1…i-1?

Knapsack

- subproblems:
  - optimum among items 1…i-1?
  - case 1: don't use item i
    - OPT(i) = OPT(i-1)
  - case 2: do use item i
    - OPT(i) = ? [what is weight used by subproblem?]
- subproblems, second attempt:
  - optimum among items 1…i-1, with total weight w

Knapsack

- subproblems:
  - optimum among items 1…i-1, with total weight w
  - case 1: don't use item i
    - OPT(i, w) = OPT(i-1, w)
  - case 2: do use item i
    - OPT(i, w) = OPT(i-1, w – w_i)

  OPT(i, w) = OPT(i-1, w) if w_i > w else:

  max {v_i+ OPT(i-1, w-w_i), OPT(i-1, w))}
  - order to fill in the table?

Knapsack

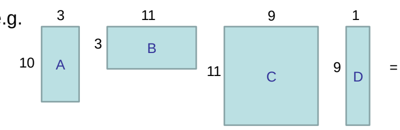
```
Knapsack(v1, w1, …, vn, wn, W)
OPT(i, 0) = 0 for all i
for i = 1 to n
  for w = 1 to W
    if wi > w then OPT(i,w) = OPT(i-1, w)
    else OPT(i,w) = {vi+ OPT(i-1, w-wi), OPT(i-1, w))}
return(OPT(n, W))
```

- Running time?
  - O(nW)
  - space: O(nW) – can improve to O(W) (how?)
  - how do we actually find items?

Matrix-chain multiplication

- Sequence of matrices to multiply

e.g.



- goal: find best parenthesization
  - e.g.: ((A·B)·C)·D) = 10·3·11 + 10·11·9 + 10·9·1 = 1410
  - e.g. (A·(B·(C·D)) = 11·9·1 + 3·11·1 + 10·3·1 = 162

Matrix-chain multiplication

- Sequence of n matrices to multiply, given by $a_1, a_2, \ldots, a_{n+1}$
- Goal: output fully parenthesized expression with minimum cost
  - fully parenthesized = single matrix: (A) *or*
  - product of two fully parenthesized: (….)(….)
- try subproblems for ranges:

  $OPT(1,n) = \min_k OPT(1,k) + OPT(k+1,n) + a_1 a_{k+1} a_{n+1}$

Matrix-chain multiplication

- Sequence of n matrices to multiply, given by $a_1, a_2, \ldots, a_{n+1}$
  - OPT(i,j) = cost to multiply matrices i … j optimally
  - OPT(i,j) = 0 if i = j
  - OPT(i,j) =

    $\min_k OPT(i,k) + OPT(k+1,j) + a_i a_{k+1} a_{j+1}$
- what order to fill in the table?

Matrix-chain multiplication

```
Matrix-Chain(a1, a2, …, an+1)
OPT(i, i) = 0 for all i
for r = 1 to n
  for i = 1 to n – r – 1; j = i + r
    OPT(i,j) = min_{i ≤ k < j} OPT(i,k) + OPT(k+1,j) + a_i a_{k+1} a_{j+1}
return(OPT(1, n))
```

- running time?
  - $O(n^3)$
- print out the optimal parenthesization?
  - store chosen k in each cell