CS 580: Algorithm Design and Analysis

Jeremiah Blocki
Purdue University
Spring 2018

Announcement: Homework 3 due February 15th at 11:59PM

Recap: Divide and Conquer

Framework: Divide, Conquer and Merge

Example 1: Counting Inversions in O(n log n) time.
- **Subroutine:** Sort-And-Count (divide & conquer)
- Count Left-Right inversions (merge) in time O(n) when input is already sorted

Example 2: Closest Pair of Points in O(n log n) time.
- **Split input in half** by x coordinate and find closest point on left and right half ($\delta = \min(\delta_1, \delta_2)$)
- **Merge**: Exploits structural properties of problems
  - Remove elements at distance > δ from dividing line L
  - Sort remaining points by y coordinate to obtain p_1, p_2 …
  - **Claim:** $|p_i - p_j| < \delta \Longrightarrow |i-j| \le 12$

Example 3: Integer Multiplication in time $O(n^{1.585})$
- Divide each n-bit number into two n/2-bit numbers
- **Key Trick:** Only need a=3 multiplications of n/2-bit numbers!

2

Fast Integer Division Too (!)

Integer division. Given two n-bit (or less) integers s and t, compute quotient $q = \lfloor s/t \rfloor$ and remainder $r = s \bmod t$ (such that s=qt+r).

Fact. Complexity of integer division is (almost) same as integer multiplication.

To compute quotient q: $x_{i+1} = 2x_i - t x_i^2$ ← using fast multiplication
- Approximate $x = 1/t$ using Newton's method:
- After i=$\log n$ iterations, either $q = \lfloor s\, x_i \rfloor$ or $q = \lceil s\, x_i \rceil$.
  - If $\lfloor s\, x \rfloor$ t > s then $q = \lceil s\, x \rceil$ (1 multiplication)
  - Otherwise $q = \lfloor s\, x \rfloor$
  - r=s-qt (1 multiplication)
- **Total**: $O(\log n)$ multiplications and subtractions

3

Toom-3 Generalization

Split into 3 parts → $a = 2^{2n/3} \cdot a_2 + 2^{\frac{n}{3}} \cdot a_1 + a_0$

$b = 2^{2n/3} \cdot b_2 + 2^{\frac{n}{3}} \cdot b_1 + b_0$

Requires: 5 multiplications of n/3 bit numbers and O(1) additions, shifts

$$T(n) = 5 \cdot T\left(\frac{n}{3}\right) + O(n) \Rightarrow T(n) \in O\left(n^{\log_3 5}\right)$$

$\log_3 5 \approx 1.465$

Toom-Cook Generalization (split into k parts): (2k-1) multiplications of n/k bit numbers.

$$T(n) = (2k-1) \cdot T\left(\frac{n}{k}\right) + O(n) \Rightarrow T(n) \in O\left(n^{\log_k(2k-1)}\right)$$

$$\lim_{k \to \infty} (\log_k(2k-1)) = 1$$

$T(n) \in O\left(n^{1.0000001}\right)$ **for large enough k**

Caveat: Hidden constants increase with k

4

Schönhage–Strassen algorithm

$$T(n) \in O(n \log n \log\log n)$$

Only used for really big numbers: $a > 2^{2^{15}}$

State of the Art Integer Multiplication (Theory): $O(n \log n\ g(n))$ for increasing small

$$g(n) \ll \log\log n$$

Integer Division:
- **Input:** x,y (positive n bit integers)
- **Output:** positive integers q (quotient) and remainder r s.t.

  $x = qy + r$ and $r < y$
- Algorithm to compute quotient q and remainder r requires O(log n) multiplications using Newton's method (approximates roots of a real-valued polynomial).

5

Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969] $\Theta(n^{\log_2 7}) = O(n^{2.807})$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, …]
- Two 20-by-20 matrices with 4,460 scalar multiplications. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar multiplications. $O(n^{2.7801})$
- A year later. $O(n^{2.7799})$
- December, 1979. $O(n^{2.521813})$
- January, 1980. $O(n^{2.521801})$

6

Fast Matrix Multiplication: Theory

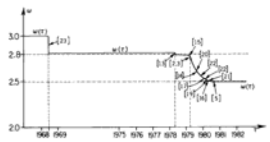


FIG. 1. ω(t) is the best exponent announced by time t.

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

7

Fast Matrix Multiplication: Theory

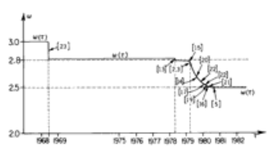


FIG. 1. ω(t) is the best exponent announced by time t.

Best known. $O(n^{2.373})$ [Williams, 2014]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

8

Fast Matrix Multiplication: Theory

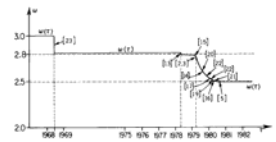


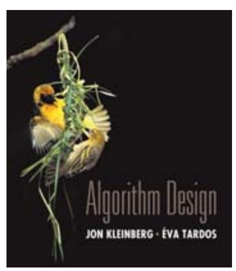
FIG. 1. ω(t) is the best exponent announced by time t.

Best known. $O(n^{2.3729})$ [Le Gall, 2014]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

9



Dynamic Programming

10

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

11

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

> "it's impossible to use dynamic in a pejorative sense"
> "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

12

Dynamic Programming Applications

Areas.
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

13

Computing Fibonacci numbers

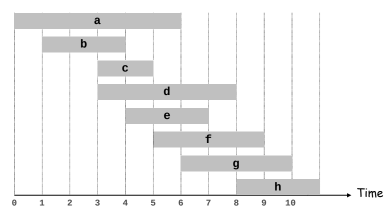
On the board.

14

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.
- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

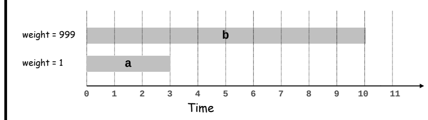


16

Unweighted Interval Scheduling (will cover in Greedy paradigms)

Previously Showed: Greedy algorithm works if all weights are 1.
- **Solution:** Sort requests by finish time (ascending order)

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

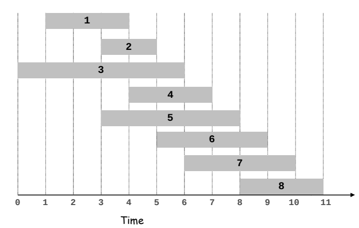


17

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \le f_2 \le . . . \le f_n$.
Def. p(j) = largest index i < j such that job i is compatible with j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



18

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - collect profit v_j
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\, v_j + OPT(p(j)),\ OPT(j-1) \,\} & \text{otherwise} \end{cases}$$

19

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

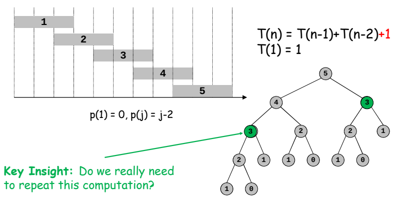
T(n) = T(n-1)+T(p(n))+O(1)
T(1) = 1

20

Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence ($F_n > 1.6^n$).



T(n) = T(n-1)+T(n-2)+1
T(1) = 1

21

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

global array

22

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.
- Sort by finish time: O(n log n).
- Computing p(·): O(n log n) via sorting by start time.

- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially Φ = 0, throughout Φ ≤ n.
  - (ii) increases Φ by 1 ⇒ at most 2n recursive calls.

- Overall running time of `M-Compute-Opt(n)` is O(n). ▪

Remark. O(n) if jobs are pre-sorted by start and finish times.

23

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (vj + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- # of recursive calls ≤ n ⇒ O(n).

24

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

25

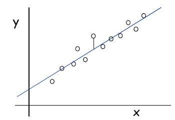
6.3 Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solution. Calculus $\Rightarrow$ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

27

Segmented Least Squares

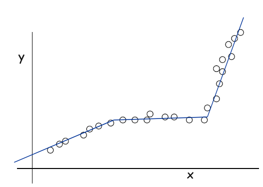
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What's a reasonable choice for f(x) to balance accuracy and parsimony?

goodness of fit

number of lines

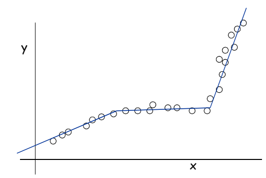


28

Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors E in each segment
  - the number of lines L
- Tradeoff function: E + c L, for some constant c > 0.



29

Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- e(i, j) = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute OPT(j):

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
- Cost = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

30

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

Running time. $O(n^3)$. can be improved to $O(n^2)$ by pre-computing various statistics

- Bottleneck = computing $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ per pair using previous formula.

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.
- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }
- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ \ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

37

Knapsack Problem: Running Time

Running time. Θ(n W).
- Not polynomial in input size!
  - Only need $\log_2 W$ bits to encode each weight
  - Problem can be encoded with $O(n \log_2 W)$ bits
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

38