CSE 6140/ CX 4140
Computational Science and Engineering
ALGORITHMS

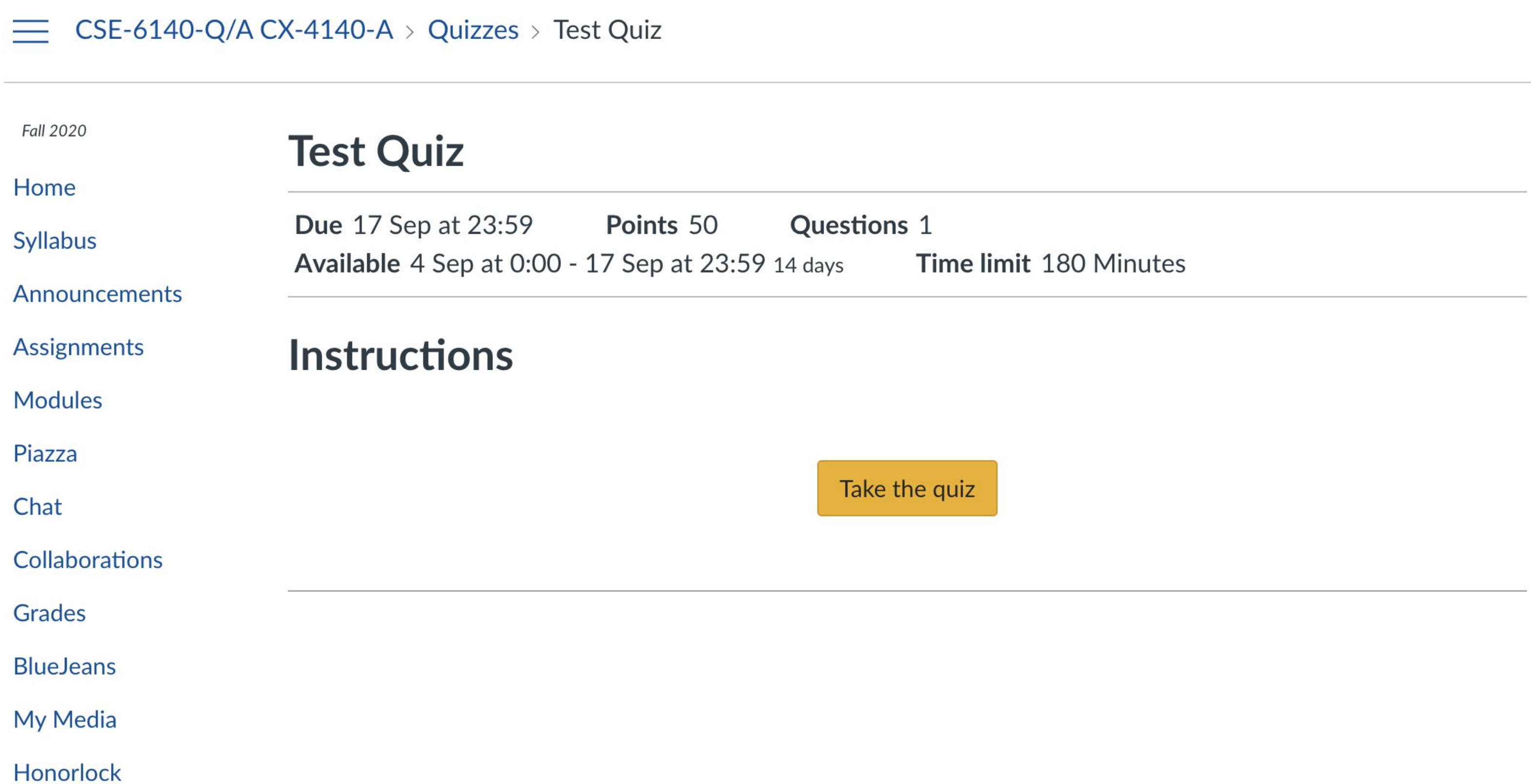
Divide and Conquer, Dynamic Programming

Instructor: Xiuwei Zhang

School of Computational Science and Engineering

Course logistics

- Test 1: Sep. 18th, 9am - 11:59pm EDT
- Duration: 3 hours
- Please take the Test Quiz to test the system
- Answers failed to be submitted to Canvas are not accepted



Course logistics

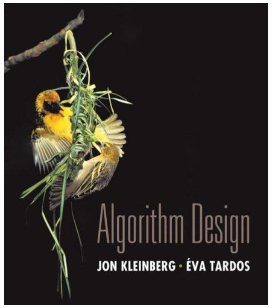
Homework 1:

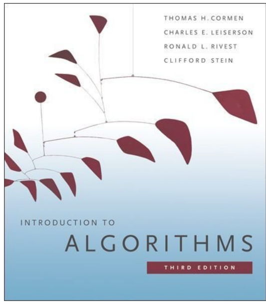
- Grades posted on Canvas
- Solutions released
- Regrading
  Contact the respective TA before the deadline for regrading request: Sep. 21, 11:59pm EDT

MATRIX MULTIPLICATION [CLRS 4.2]

Adapted from Slides by
Kevin Wayne.

And Bistra Dilkina, Anne Benoit, Ümit V. Çatalyürek





Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Naive method. $\Theta(n^3)$ arithmetic operations.

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

Q. Is the naive matrix multiplication algorithm optimal?

Block Matrix Multiplication

C_{11} A_{11} A_{12} B_{11}

$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

B_{21}

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Number of "block operations" to calculate C:

- 8 multiplication
- 4 addition

Block matrix multiplication: warmup

- To multiply two n-by-n matrices A and B:
  - Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
  - Conquer: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
  - Combine: add appropriate products using 4 matrix additions.

n-by-n matrices

$$C = A \times B$$

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

½n-by-½n matrices

8 matrix multiplications (of ½n-by-½n matrices)

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

4 matrix additions (of ½n-by-½n matrices)

Block matrix multiplication: running time

Running time. Apply master theorem.

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}}$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$$T(n) = \Theta(n^3)$$

Fast Matrix Multiplication

Key idea. multiply 2-by-2 blocks with only 7 multiplications.

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

$$\begin{aligned}
P_1 &= A_{11} \times (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12}) \times B_{22} \\
P_3 &= (A_{21} + A_{22}) \times B_{11} \\
P_4 &= A_{22} \times (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12})
\end{aligned}$$

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

To multiply two n-by-n matrices A and B: [Strassen 1969]

- Divide: partition A and B into ½n-by-½n blocks.
  Compute: 14 ½n-by-½n matrices via 10 matrix additions/subtractions.
- Conquer: multiply 7 pairs of ½n-by-½n matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions/subtractions.

Fast Matrix Multiplication: Strassen

To multiply two n-by-n matrices A and B: [Strassen 1969]

- Divide: partition A and B into ½n-by-½n blocks.
- Compute: 14 ½n-by-½n matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of ½n-by-½n matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

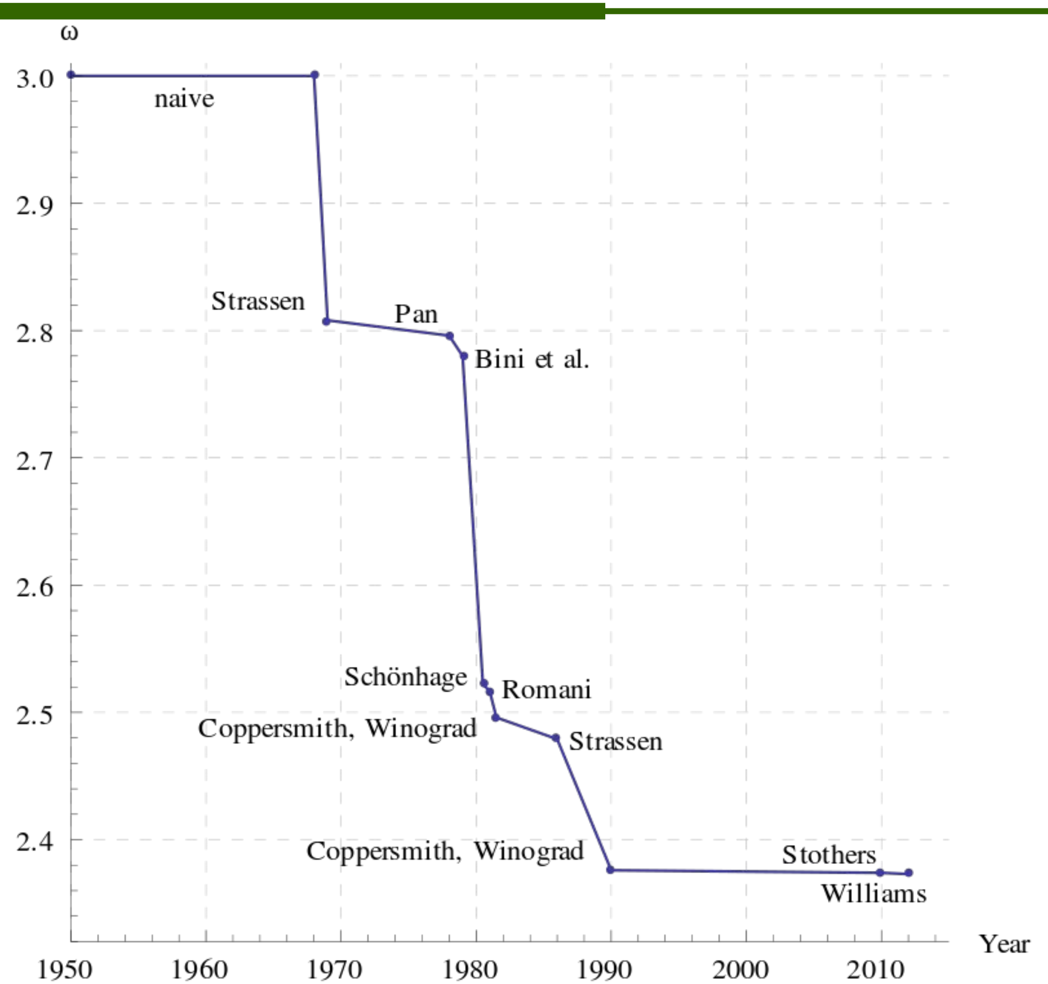
- Assume n is a power of 2.
- $T(n)$ = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Common misperception. *"Strassen is only a theoretical curiosity."*

- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2{,}500$.
- Range of instances where it's useful is a subject of controversy.

Fast Matrix Multiplication: Theory



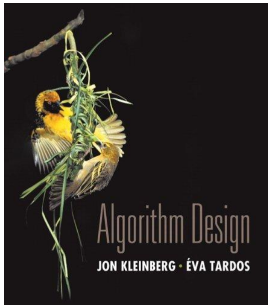
Best known. $O(n^{2.373})$

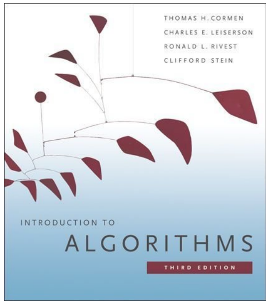
Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

DYNAMIC PROGRAMMING [KT6, CLRS15, BRV4]

Adapted from Slides by
Kevin Wayne.

And Bistra Dilkina, Anne Benoit, Ümit V. Çatalyürek





Algorithmic Paradigms

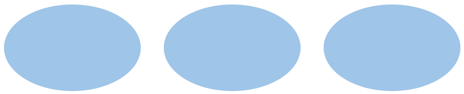
- Greedy. Build up a solution incrementally, myopically optimizing some local criterion. (not trying all options but can prove that greedy choice results optimal solution at the end)

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

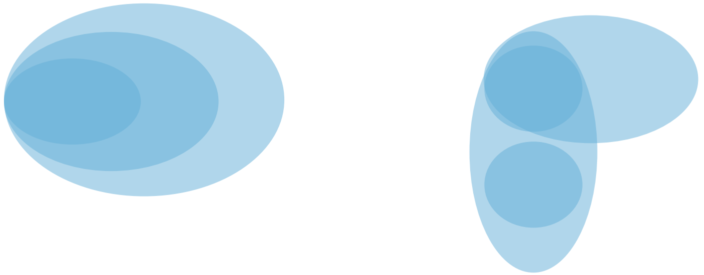
- **Dynamic programming.** Break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger sub-problems from smaller subproblems, (*reusing* solutions of encountered subproblems as much as possible).

Subproblems

Divide and conquer



Greedy or Dynamic Programming



Dynamic programming: algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

Greedy vs Dynamic Programming

	Greedy	Dynamic programming
Optimal substructure	the optimal solution can be constructed from optimal solutions to subproblems	
Optimality	Does not guarantee optimality	Guarantees optimality; equivalent to exhaustive search; efficient because of the reuse of subproblems
	Makes decisions based on local subproblem; once a choice is made, it is not changed	Makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution

Dynamic Programming Applications

Areas.

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

Some famous dynamic programming algorithms.

- Shortest paths with negative weights - Bellman-Ford
- Comparing two files - Unix diff
- Hidden Markov models - Viterbi
- Genetic sequence alignment - Smith-Waterman
- Parsing context free grammars - Cocke-Kasami-Younger

Dynamic Programming

1) Show problem has optimal substructure: the optimal solution can be constructed from optimal solutions to subproblems (recurrence relation).

2) Show subproblems are overlapping, i.e., subproblems may be encountered many times but the total number of distinct subproblems is polynomial

3) Construct an algorithm that computes the optimal solution to each subproblem only once, and reuses the stored result all other times

4) Show that time and space complexity is polynomial

Coin-changing problem [BRV4.1]

The problem: We want to make change for S cents, and we have infinite supply of each coin in the set Coins=$\{v_1, v_2, \ldots, v_n\}$, where v_i is the value of the i-th coin. What is the minimum number of coins required to reach value S?

Greedy algorithm:

- sort coins by non-increasing values $v_1 > v_2 > \ldots > v_n$
- R <- S (remaining sum to reach)
- For i=1 to n, { $c_i = \lfloor R/v_i \rfloor$; R <- R – c_i x v_i }

  (returns c_i coins of value v_i)

Is this optimal?

Set: {6,4,1}, S=8

Coin changing problem: DP algorithm

- Optimal algorithm. Find $z(S,n)$: reach sum S with coins of value $\{v_1, \ldots, v_n\}$. Greedy may fail: try to solve more subproblems so that we do not take a bad greedy choice. Must be able to come back to a choice already made and try another set of coins.

- Subproblem:
  Find $z(T,i)$, min number of coins to reach $T \le S$ with first i coins;

- now we solve S x n problems, but we have a recurrence relation:

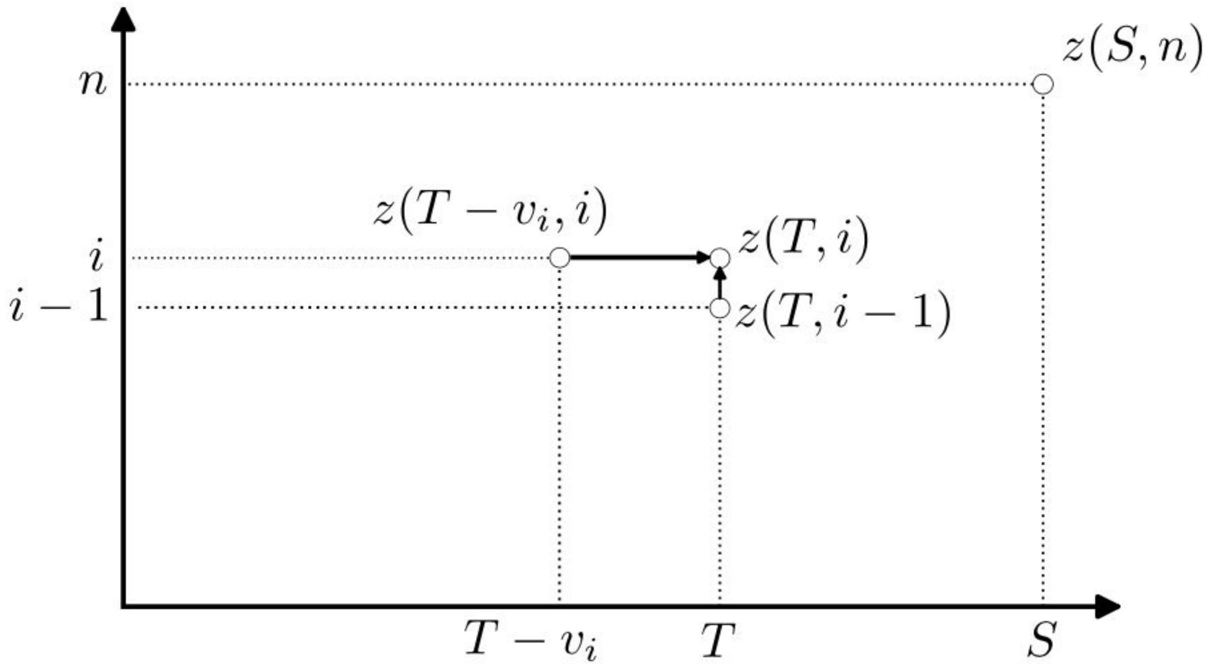
- $z(T,i) = \min\{\, z(T, i-1)$ (i-th coin not used),
  $z(T-v_i, i) + 1$ (i-th coin used at least once) $\}$

- Need to initialize the recurrence properly:
  $z(T,0) = +\infty$ if $T > 0$ (no more coins)
  $z(0,i) = 0$ (we are done)
  $z(T,i) = +\infty$ if $T < 0$ (too much change given)

Coin changing problem: implementation

- Recursive algorithm: exponential number of computations!

- We make « memo » of values already computed, hence using memoization, or use an iterative algorithm so that we always have the values required to compute $z(T, i)$. Check precedence constraints!



Coin changing problem: the algorithm

for $T = 1$ to S **do**
- $z(T, 0) \leftarrow +\infty$ { *Initialization: case* $i = 0$ }

for $i = 0$ to n **do**
- $z(0, i) \leftarrow 0$ { *Initialization: case* $T = 0$ }

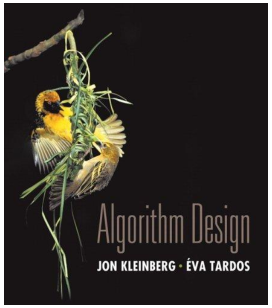
for $i = 1$ to n **do**
- **for** $T = 1$ to S **do**
  - $z(T, i) \leftarrow z(T, i-1)$
    { $z(T, i-1)$ *computed at previous iteration, or case* $i = 0$ }
  - **if** $T - v_i \geqslant 0$ **then**
    - $z(T, i) \leftarrow \min(z(T, i), z(T - v_i, i) +1)$
      { $z(T - v_i, i)$ *computed earlier in this loop, or case* $T = 0$ }

Complexity of DP algorithm: $O\ (n\ x\ S)$
Greedy algorithm: $O\ (n \log n)$

WEIGHTED INTERVAL SCHEDULING [KT 6.1]

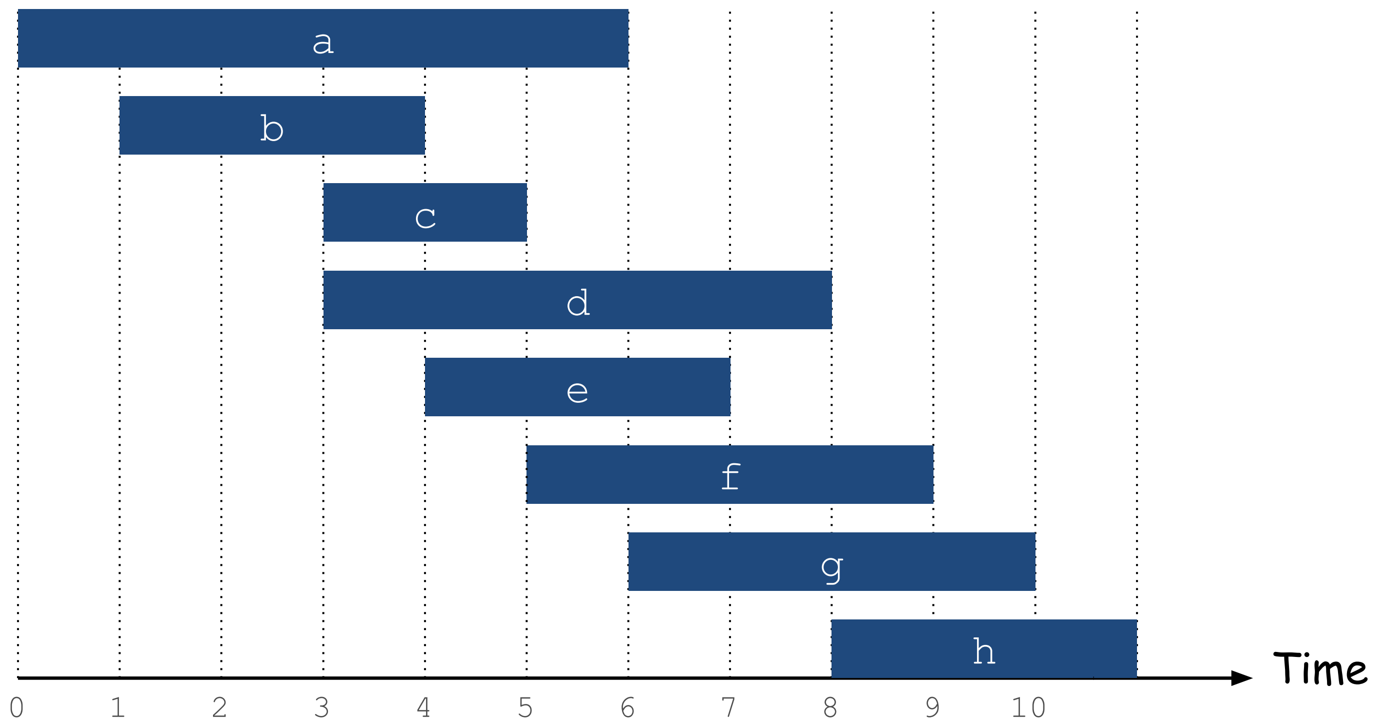
Adapted from Slides by
Kevin Wayne.

And Bistra Dilkina, Anne Benoit, Ümit V. Çatalyürek



Weighted Interval Scheduling

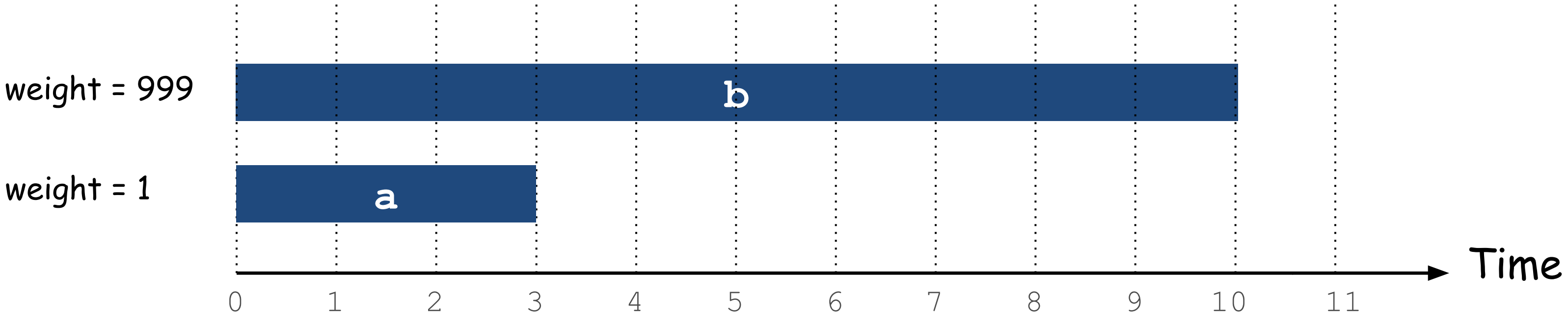
- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

- **Recall**. Greedy algorithm works if all weights are 1.
  - Consider jobs in ascending order of finish time.
  - Add job to subset if it is compatible with previously chosen jobs.

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

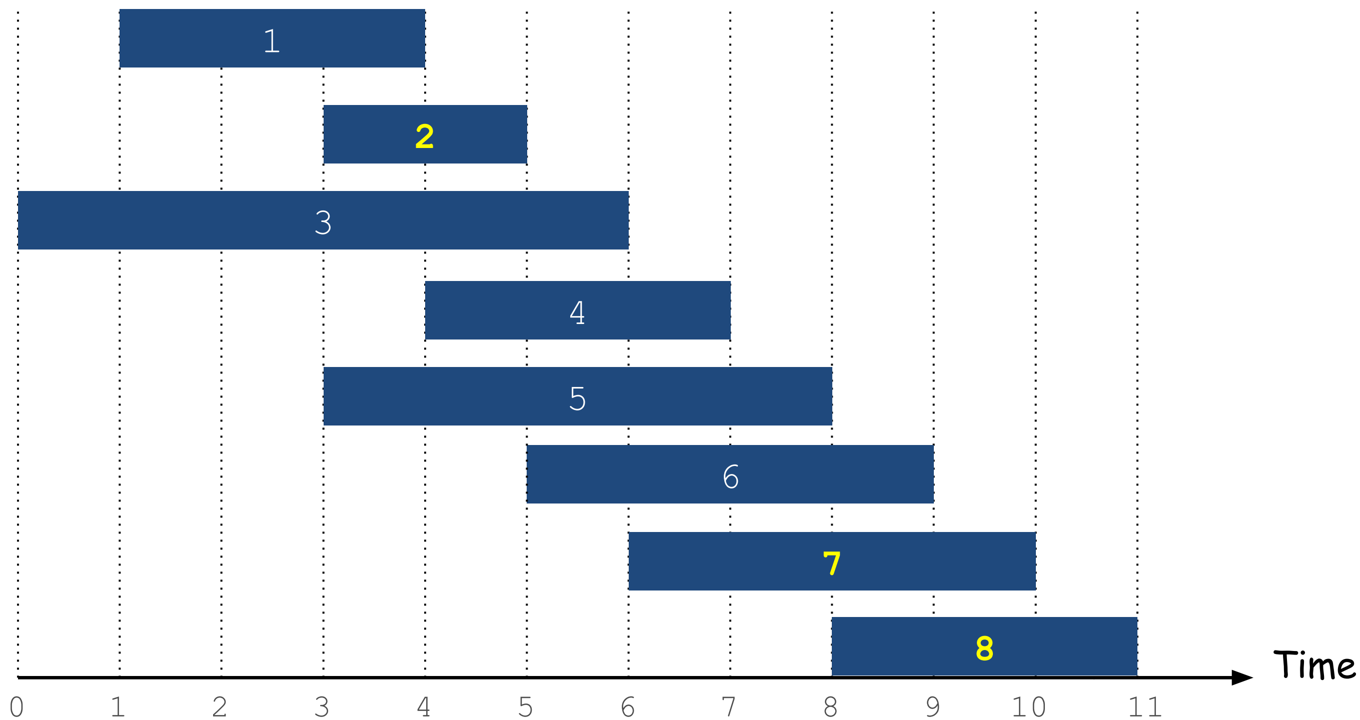


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

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.



Dynamic Programming: Binary Choice

Consider an optimal solution O for the jobs {1, . . , n}

No matter what O is, what can we say about the job n?
- Either O contains the last job n (Case 1)
- Or O does not contain the last job n (Case 2)

This covers all possible cases for O

Dynamic Programming: Binary Choice

Consider an optimal solution O for the jobs {1, . . . , n}

No matter what O is, what can we say about the job n?
- Either O contains the last job n (Case 1)
- Or O does not contain the last job n (Case 2)

Case 1: O contains job n

what can we say about the remaining part of the solution O − {n}?

- O− {n} cannot contain any job that is incompatible with n, i.e., cannot contain any job in p(n) + 1, . . . , n − 1, i.e., it only contains jobs in {1, . . . , p(n)}
- Since O is feasible, O − {n} is a feasible solution for the problem of scheduling {1, ... , p(n)}
- More importantly O − {n} must be an optimal solution for scheduling {1, ... , p(n)}. If not, then we could take the optimal solution for {1, . . ,p(n)} and safely add job n to it, and obtain an overall solution O' better than the given optimal solution O

Dynamic Programming: Binary Choice

Consider an optimal solution O for the jobs {1, . . , n}

No matter what O is, what can we say about the job n?
- Either O contains the last job n (Case 1)
- Or O does not contain the last job n (Case 2)

Case 2: O does not contain job n

- Then O is a feasible solution for scheduling {1, ... , n − 1}
- If O is not the optimal solution for {1, . . , n − 1}, we can replace it with the optimal solution for {1, ... , n − 1} and obtain a better solution also for scheduling {1, ... , n}
- O must contain the optimal solution for scheduling {1, . . , n − 1}

Finding the optimal solution for {1, . . , n} involves looking at optimal solutions for smaller problems of the form {1, ... , j}

Dynamic Programming: Binary Choice

Georgia Tech

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

  Case 1: OPT selects job j.

  Case 2: OPT does not select job j.

OPTIMAL SUBSTRUCTURE

- Case 1: $OPT(j)$ selects job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ... , $p(j)$ with value OPT(p(j))
  - collect profit v_j from including j
  - $OPT(j) = v(j) + OPT(p(j))$
- Case 2: $OPT(j)$ does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ... , $j-1$: $OPT(j) = OPT(j-1)$

optimal substructure

- RECURRENCE RELATION

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

Prove this algorithm is correct

- With the optimal substructure analysis we proved that:

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

- Claim. The algorithm Compute-Opt(j) computes correctly the optimal value for each j=1,..,n.
- Proof. (By induction on j)

1) True for $j = 0, OPT(0) = 0$
2) Assume true for all $i < j$
3) By induction we know $OPT(j-1)$ and $OPT(p(j))$ are computed correctly
Hence, `Compute-Opt(j) = max(v`$_j$` + Compute-Opt(p(j)), Compute-Opt(j-1)) = max(v`$_j$` + OPT(p(j)), OPT(j-1)) = OPT(j)`

Weighted Interval Scheduling: Brute Force

- Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

  Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

  Compute p(1), p(2), …, p(n)

  Call Compute-Opt(n)



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

Proof this algorithm is correct

- With the optimal substructure analysis we proved that:

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

- Claim. The algorithm Compute-Opt(j) computes correctly the optimal value for each j=1,..,n.

Proof this algorithm is correct

- With the optimal substructure analysis we proved that:

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

- Claim. The algorithm Compute-Opt(j) computes correctly the optimal value for each j=1,..,n.
- Proof. (By induction on j)

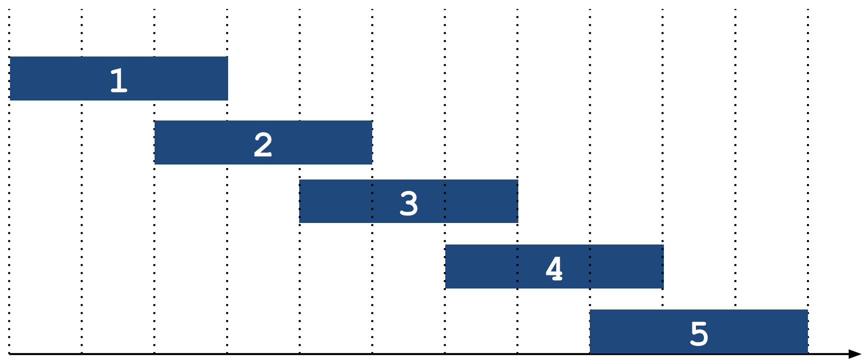
1) True for $j = 0, OPT(0) = 0$
2) Assume true for all $i < j$
3) By induction we know $OPT(j-1)$ and $OPT(p(j))$ are computed correctly
   Hence, `Compute-Opt(j) = max(v`$_j$` + Compute-Opt(p(j)), Compute-Opt(j-1)) = max(v`$_j$` + OPT(p(j)), OPT(j-1)) = OPT(j)`

Weighted Interval Scheduling: Brute Force

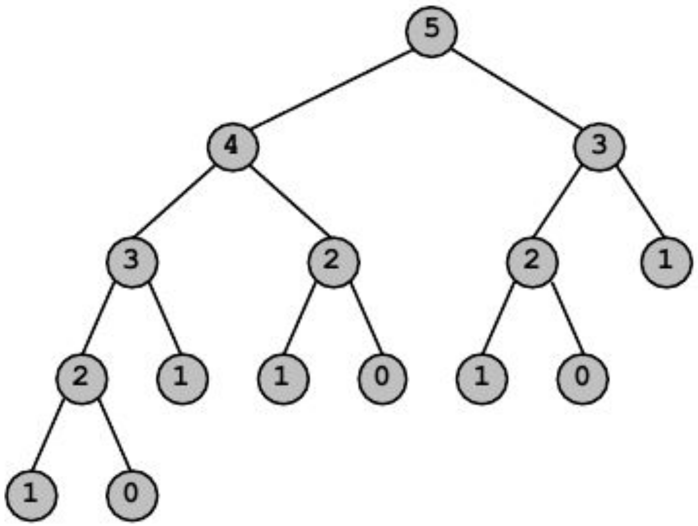
Example. Each job is incompatible with only one earlier job, i.e. p(j) = j-2.

T(n) = T(n-1) + T(n-2) + O(1) grows like Fibonacci sequence ->

T(n) in $O(2^n)$.



$$p(1) = 0, p(j) = j - 2$$



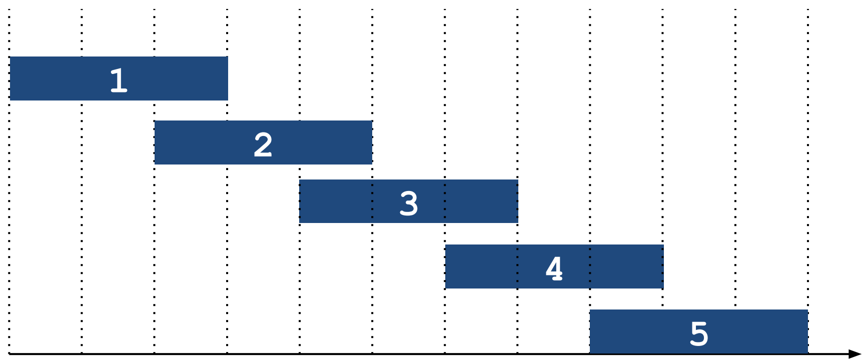
Weighted Interval Scheduling: Brute Force

Example. Each job is incompatible with only one earlier job, i.e. p(j) = j-2.

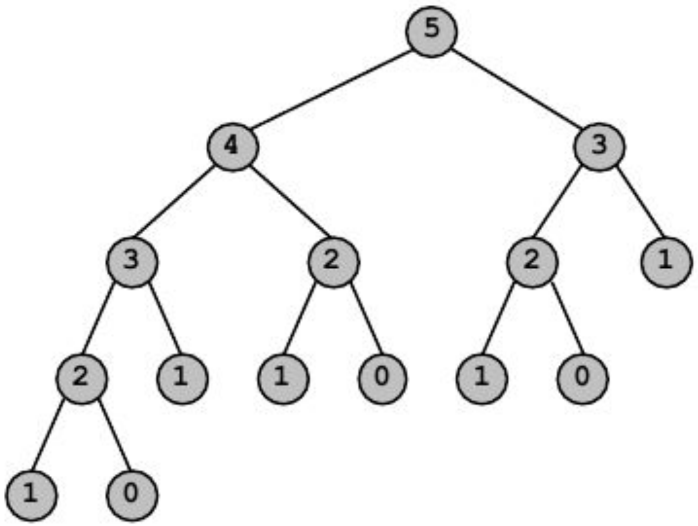
T(n) = T(n-1) + T(n-2) + O(1) grows like Fibonacci sequence ->

T(n) in $O(2^n)$.

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.



$$p(1) = 0, p(j) = j - 2$$



Weighted Interval Scheduling: Memoization

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

```
Input: n, s_1,...,s_n , f_1,...,f_n , v_1,...,v_n

 Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
 Compute p(1), p(2), …, p(n)

 for j = 1 to n
    M[j] = empty
 M[0] = 0
 M-Compute-Opt(n)


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

global array

What have we done so far?

1. We showed optimal substructure property for the problem
2. Derived a recurrence relation based on the optimal substructure (with overlapping subproblems)
3. Showed total number of distinct subproblems is polynomial and designed a DP Algorithm that implements the recurrence relation and caches explored subproblems to avoid repeated work
4. Analyze Space and Time of our algorithm

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

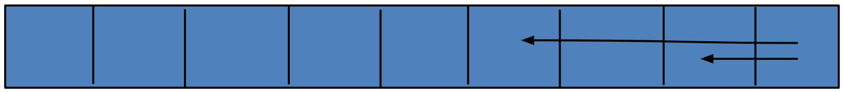
- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) via sorting by start time.
- M-Compute-Opt(j): each invocation takes O(1) time and either
  - (i) returns an existing value M[j]
  - (ii) fills in one new entry M[j] and makes two recursive calls
- The running time is bound by (a constant x the number of 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. The overall algorithm takes O(n) if jobs are pre-sorted by start and finish times when given as input.

Weighted Interval Scheduling: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

M[]



Input: $n, s_1,\ldots,s_n, f_1,\ldots,f_n, v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p(1), p(2), \ldots, p(n)$

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

Dynamic Programming

- Top-down DP = Memoization
  - Design a *recursive* algorithm
  - Store result for each subproblem when you first compute it
  - Check for existing result for a subproblem, before doing any extra work

- Bottom-up DP = Iterative DP
  - Determine dependency between a problem and its subproblems
  - Determine an order in which to compute subproblems so that you always have what you need already available
  - Fill in the table of results in the determined order (*using* **FOR** *loops*)

Weighted Interval Scheduling: Finding a Solution

- Q. Dynamic programming algorithm 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 (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- # of recursive calls $\leq$ n $\Rightarrow$ O(n).