Michael A. Forbes
miforbes@illinois.edu
2024-01-23-1

cs473 Algorithms: Lecture 3 (2024-01-23)

logistics:
- pset 0 due F17 ⟦gradescope⟧
- office hrs
  - miforbes T 15:30 Siebel 3232 – 3304 ⟦whiteboards⟧
  - Christian W 14:00 —"—
  - Shubhang R 13:00 —"—

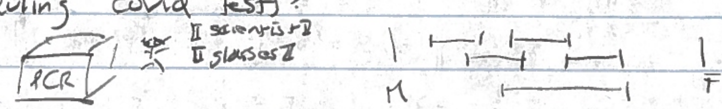
last lecture: divide and conquer: two dimensional closest pair

today: dynamic programming ⟦divide and conquer + memoization⟧

reading: KT 6.0 - 6.2

⟦move to front⟧
⟦laptops' policy⟧

Q: scheduling covid tests? ⟦scientist⟧ ⟦glasses⟧



⟦different PCR tests, varying inoculation times⟧
⟦cannot overlap⟧

def: sets of intervals $[s_1, f_1], \ldots, [s_n, f_n] \subseteq \{1, \ldots, m\} = [m]$, $s_i \leq f_i$ ⟦start⟧ ⟦finish⟧

$[s_i, f_i]$ **compatible** w/ $[s_j, f_j]$ if
- $f_i \leq s_j$ ⟦obvious⟧ ⟦$i \prec j$⟧
- $f_j \leq s_i$ ⟦$j \prec i$⟧

$S \subseteq [n]$ ⟦subset of intervals⟧ is **feasible** if $\forall i \neq j \in S$
$[s_i, f_i]$ compatible w/ $[s_j, f_j]$ ⟦last lecture⟧

convention: problem defined by $2n$ integers $\Rightarrow$ all are $O(\lg n)$ bits, arithmetic at unit cost

Q: make the most money? ⟦ppl pay for covid tests⟧ $\Rightarrow \lg m \leq O(\lg n) \equiv m \leq n^{O(1)}$ ⟦small integers⟧

def: the weighted interval scheduling problem is to, given intervals $([s_i, f_i])_{i=1}^{m}$ and weights $v_1, \ldots, v_n \in \mathbb{Z}$, ⟦convention on bit-sizes⟧ $m \leq 2n$ wlog

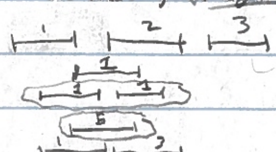
compute $\widehat{\text{arg}} \max_{S \subseteq [n],\ S \text{ feasible}} \sum_{i \in S} v_i \;//\; = \mathrm{OPT}$ ⟦sum of weights⟧ ⟦value vs soln⟧

convention: $\sum_{i \in \emptyset} v_i = 0$

lem: wlog, $v_i \geq 0$ all i

why
0 vs −1
∅ vs S

sketch: omitting intervals preserves feasibility
omitting $\boxed{\text{negative}}$ weight intervals does not decrease solution value ∎



eg: ⟦disjoint⟧ ⟦all feasible⟧
eg: ⟦not disjoint⟧ ⟦unweighted⟧
eg: ⟦weights make problem harder⟧

assumption: $f_1 \leq \cdots \leq f_n$ ⟦sorted by finish time⟧ ⟦sorting costs $O(n \lg n)$⟧ ⟦we want to look at $\boxed{\text{additional}}$ work⟧

⟦is there an algo?⟧

assumption wlog?

prop: weighted interval scheduling in $O(n \cdot 2^n)$ time ⟦brute force⟧

sketch: **algo**: – output $\max_{\substack{S \subseteq [n] \\ S \text{ feasible}}} \sum_{i \in S} v_i$

$O(2^n)$ $O(n)$ $O(n)$

fine grained

correctness: clear

complexity: $O(2^n)(O(n) + O(n)) = O(n \cdot 2^n)$ ∎

custom space

Michael A. Forbes
mforbes@illinois.edu
2024-01-23 .2

Q: do better? [2^n is bad] divide and conquer?

def: [notation] $0 \le k \le n$ $\quad OPT_k = \max_{S \subseteq [k],\ S \text{ feasible}} \sum_{i \in S} v_i$ (^ arg) [only first k intervals] [not originally of interest] [soln vs value]

$1 \le i \le n$, define $prev(i) = \max\{j : j \le i\ \ [s_j, f_j] \text{ compatible w/ } [s_i, f_i]\}$ $\overset{[\text{sorted}]}{\iff}$ $f_j \le s_i$

convention: $prev(i) = 0$ if no such

eg: 1, 2, 3 [sorted] $prev(1) = 0$, $prev(2) = 0$, $prev(3) = 1$

fact: $prev(1), \ldots, prev(n)$ computable in $O(n \lg n)$ time, $\boxed{\text{given}}$ the intervals are sorted $\implies$ will assume $prev[\cdot]$ precomputed

prop: [main] $S \subseteq [n]$ feasible iff either (a) $S = T \subseteq [n-1]$ feasible (b) $S = \{n\} \cup T$, $T \subseteq [prev(n)]$ feasible

pf: $\Leftarrow$: [easier] (a) $S = T$ feasible
(b) $T = \{i_1, \ldots, i_k\} \subseteq [prev(n)]$ w/ $f_{i_1} \le \cdots \le f_{i_k} \le s_n \implies$ all $j \in T$ have $[s_j, f_j]$ compat w/ $[s_n, f_n]$ $\overset{T \text{ feasible}}{\implies}$ $S = \{n\} \cup T$ feasible

$\implies$: (a) $S \subseteq [n-1]$ feasible $\implies S = T \subseteq [n-1]$ feasible
(b) $\not\subseteq$ — " — $\to S = \{n\} \cup T$ $\quad T \subseteq [n-1]$ $\quad T$ feasible
$i \in T$ $[s_i, f_i]$ compat $[s_n, f_n] \overset{[\text{sorted}]}{\implies} f_i \le s_n \implies i \in [prev(n)]$ $\implies T \subseteq [prev(n)]$ ∎

cor: $\underset{[\text{all } n \text{ intervals}]}{\overset{[\text{value}]}{OPT_n}} = \max\{\underset{[\text{first } n-1]}{OPT_{n-1}}, OPT_{prev(n)} + v_n\}$

pf: $OPT_n = \max_{S \subseteq [n],\ S \text{ feasible}} \sum_{i \in S} v_i \overset{[\text{prop}]}{=} \max\left\{ \max_{S \subseteq [n-1],\ S \text{ feasible}} \sum_{i \in S} v_i,\ \max_{S = \{n\} \cup T,\ T \subseteq [prev(n)],\ T \text{ feasible}} \sum_{i \in S} v_i \right\}$ $\quad \sum_{i \in S} v_i = \left(\sum_{i \in T} v_i\right) + v_n$

in lecture had issue?

algo: SOLVE(k): - if $k = 0$, output 0 [no interval]
- output $\max\{solve(k-1), solve(prev(k)) + v_k\}$ [recursion]

pseudo
output vs return
named recursion

prop: SOLVE(n) computes OPT_n in $O(2^n)$ time [$\ll n 2^n$, but not by much]

pf correctness: clear

complexity: $T(k) = \max_{j \le k}\{\text{runtime of SOLVE}(j)\}$

$T(k) \le T(k-1) + T(\underbrace{prev(k)}_{\le k-1}) + O(1)$ $\overset{[\text{arithmetic}]}{\le} 2T(k-1) + O(1) \le \cdots \le O(2^k)$ ∎

why may missing a step

Q: do better?

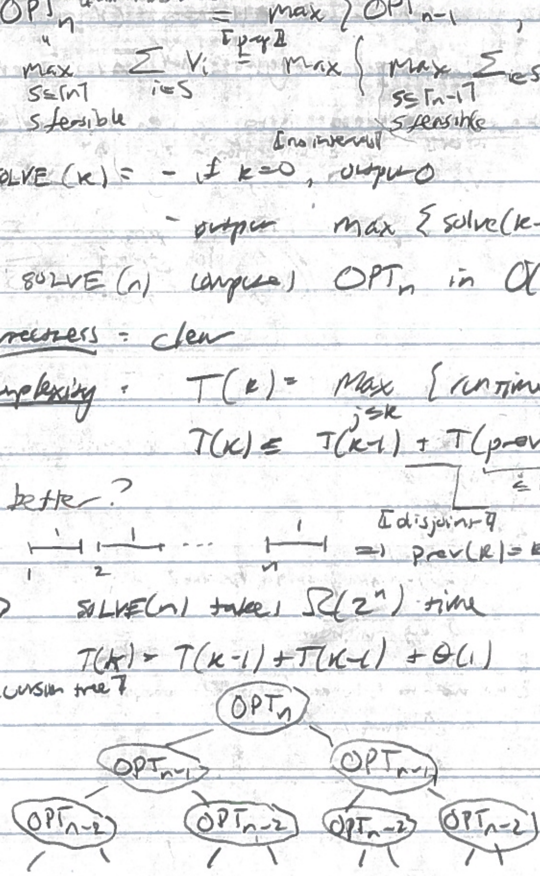
eg: [disjoint] 1, 2, …, n $\implies prev(k) = k-1$ all k

thumbs up?

$\implies$ SOLVE(n) takes $\Omega(2^n)$ time

$T(k) = T(k-1) + T(k-1) + \Theta(1)$

[recursion tree]

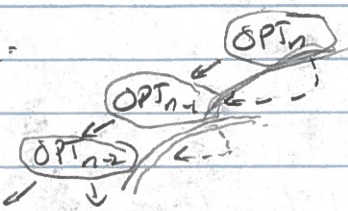


problem: re-solving same subproblem $\boxed{\text{multiple}}$ times

idea (dynamic programming): solve each subproblem at $\boxed{\text{most}}$ once, by $\boxed{\text{storing}}$ the solutions
memoization

Michael A. Forbes
mforbes@illinois.edu
2024-01-23.3

eg:



algo: global array $M[\cdot]$ on $\{1,\ldots,n\}$

SOLVE-DP(k) = - if $k=0$ output 0

- if $M[k]$ empty, [subproblem not yet solved] $M[k] = \max\{$SOLVE-DP($k-1$), SOLVE-DP(prev(k)) $+ v_k\}$

- output $M[k]$

prop: SOLVE-DP(n) computes OPT_n in $O(n)$ time [efficient] [$\ll 2^n$]

[same algo, but avoiding waste]

pf: correctness - clear [inductive] clear!]

complexity: clm. runtime is $O(1)\cdot$(# recursive calls) [charge $O(1)$ work to each call]

clm. # recursive calls $\le 2n$ [implies result]

pf. subclaim - throughout algo, 2(# empty cells in M) + (# recursive calls) = $2n$ ↳ invariant

pf: induction

initially: $2\cdot(n) + (0) = 2n$

		Δ # empty cell	Δ # calls	
in algo -	$k=0$ -	0	0	[no change]
	$M[k]$ nonempty -	0	0	[—]
	$M[k]$ empty -	-1	2	[invariant holds]

Q: find solution? [not just value]

prop: [above] $S \subseteq [n]$ feasible iff either (a) $S = T \subseteq [n-1]$ feasible (b) $S = \{n\} \cup T$, $T \subseteq [\text{prev}(n)]$ feasible

cor: exists optimal soln containing $[s_n, f_n]$ iff $\underbrace{OPT_{\text{prev}(n)} + v_n \ge OPT_{n-1}}$

pf: [above] $OPT_n = \max\{\underbrace{OPT_{n-1}}_{\text{type (a)}}, \underbrace{OPT_{\text{prev}(n)} + v_n}_{\text{type (b)}}\}$

↳ iff type (b) soln achieves opt value

iff

return vs output
↳ finished vs continue

algo: global array $M[\cdot]$, $N[\cdot]$ [array]

soln-DP(k) = - if $k=0$, output $(\emptyset, 0)$

- if $M[k]$ empty

- if $\underbrace{\text{soln-DP(prev}(k))[1] + v_k}$ $\ge$ $\underbrace{\text{soln-DP}(k-1)[1]}$

$M[k] =$

$N[k] = \{k\} \cup$ soln-DP(prev(k))[0] || $O(n)$ copying

- else $M[k] =$

$N[k] =$ soln-DP($k-1$)[0] || $O(n)$ copying

- output $(M[k], N[k])$

fix order.

Michael A. Forbes
mforbes@illinois.edu
2024-01-23.4

prop - soln-DP finds optimal soln in $O(n^2)$ time

sketch = **correctness** : clear.

complexity - **clm** : # recursive calls $\le O(n)$ [as before]

clm : time $\le O(n) \cdot$ (# recursive calls)

[@ accounting] wasteful copying

Q : do better? [$O(n)$ for value, $O(n^2)$ for soln]

algo · global array $M[\cdot]$, initialized by SOLVE-IT(n) [value, no soln]

soln-DP-fast(k) = - if $k=0$, output nothing

- if $M[\mathrm{prev}(k)] + v_k \ge M[k-1]$ - output k
  - [recurse] soln-DP-fast(prev(k))
- else - soln-DP-fast ($k-1$)

prop : soln-DP-fast finds opt soln in $O(n)$ time

pf : **correctness** : clear [same]

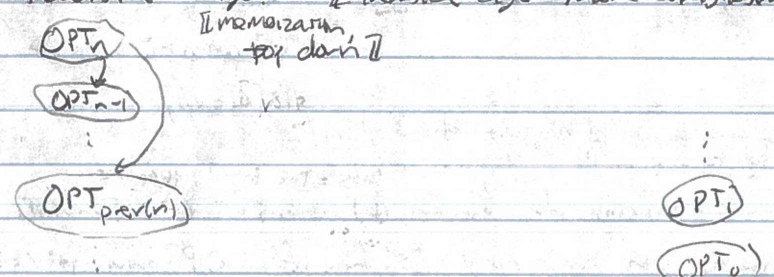
complexity : $O(n)$ [find value] + $T(k)$ [recurse]

$T(k) \le \max\{T(\mathrm{prev}(k)), T(k-1)\} + O(1)$ [$O(1)$ output]

$\le \cdots \le O(k)$

Q : non-recursive algo? [memoization can be hard to analyze the complexity] [iterative algo more straightforward]

[memoization top down] [bottom up]



this course : $\boxed{\text{strongly}}$ prefer iterative algo

algo · SOLVE-ITER(n) = - $M[0] = 0$

- ~~for~~ $k = 1, \ldots, n$
  - $M[k] = \max\{M[k-1], M[\mathrm{prev}(k)] + v_k\}$
- output $M[n]$

prop - SOLVE-ITER(n) computes OPT$_n$ in $O(n)$ time

pf - **correctness** - clear [as before]

complexity - $O(n \cdot O(1))$ [clear]

today - dynamic programming: weighted interval scheduling

- brute force $O(n \cdot 2^n)$
- recursive $O(2^n)$
- memoized recursive $O(n)$
- iterative $O(n)$

reading : KT 6.0 - 6.2

next lecture - dynamic programming

logistics - ~~pset~~ 0 due F17 [gradescope]

office hrs - mforbes T 15:30 Siebel 3232 - 3304 [whiteboards]
- Christian W 14:00
- Shubhang R 13:00

· header

· $O(1)$ output
· $T(n)$

· soln
compute soln
my office hrs conflicts .]