Optimization Problems

Note Title 11/20/2007

Problems we've seen so far:
- selection
- sorting
- matrix multiplication

only one valid answer

new kind of problem:
optimization problem
- many valid answers but some of those answers are <u>better</u> than other answers
- we want the <u>optimal answer</u>

usually optimality is measured by minimizing or maximizing something.
↳ a "metric"

example: room reservation.

a valid answer consists of a set of reservation requests that don't overlap in time.

say there is a <u>fee</u> for making a reservation.
an optimal answer is valid and gives the <u>max</u> amount of fees.

alternative optimization criterion:
minimize cleaning costs
then an optimal answer is to accept no reservation requests
— still valid and there will be nothing to clean!

Which answer is optimal depends on the <u>optimization criterion</u>.

types of

algorithms for optimization problems

- dynamic programming
- greedy algorithms

we want to find an optimal solution
quickly.

That means we may try to
make a series of choices
each based on only a little info
(looking at all info may be slow)
to lead to a hopefully optimal solution.

Examples of Optimization Problems

SHORTEST PATH
- find an optimal route
  - maps.google.com
  - minimize distance
  - minimize time

  POSSIBLE OPTIMIZATION CRITERIA

- find an optimal flight itinerary
  - kayak.com, expedia
  - minimize time
  - minimize cost.

  POSSIBLE OPTIMIZATION CRITERIA

MINIMUM SPANNING TREE
- given a graph, find the cheapest tree that touches all nodes

BIN PACKING
- items of different sizes and some containers.
- goal: minimize # of containers used

SCHEDULING
- room reservations
- doctor appointment requests

Activity Scheduling

single resource. cannot be shared.

n requests to use resource.

↳ each request has form

request$_i$ $[s, f)$

output: list of approved request
such that you max the #
of requests serviced.

example: doctor appointments
patient A requests 2–3:30 PM
patient B requests 2:15-2:45 PM
patient C requests 3:00–3:15 PM
patient D requests 1:45–2:20 PM

max # of patients that can be
seen if you only see one at a
time.

one way to do this: (BRUTE-FORCE)

① Exhaustively enumerate all possible answers (valid and invalid):

$$\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \cdots \binom{n}{n}$$

$$= \sum_{i=1}^{n} \binom{n}{i}$$ hard to simplify!

so try Truth Table Approach...

request #

in or not in returned list

1	2	3	4	5	...
0	0	0	0	0	.....
1	0	0	0	0	....
0	1	0	0	0	....
⋮					
1	1	1	1	1	......

How many rows? $\underline{\underline{2^n}}$

BRUTE FORCE

algorithm for activity scheduling

① so, exhaustively enumerate all possible lists.

(2^n lists)

② Identify if each answer is valid. (no requests overlap)

- compare every request in a list to all other requests.

  $n^2 \cdot \#\text{lists} = n^2 \cdot 2^n$

OR

- Sort each list, look for overlap

  counting sort

  $O(n) \cdot \#\text{lists} = n \cdot 2^n$

OR

- compare all requests to each other, store in table. Then examine lists.

  $n^2 + \underbrace{2^n(n-1)}_{\text{lookups}} = O(n \cdot 2^n)$

③ Find valid list with max # of request in it: $2^n - 1$ to do max

TOTAL runtime: $O(n 2^n)$

a faster way?

note worst case: every request conflicts with some other request

in many optimization problems, the exponential approach is the straightforward way, but we want to try to find polynomial approaches.

Let's try divide and conquer.

- how to divide the problem into subproblems??

Given list of requests REQUESTS,

REQUESTS_{ij} = set of all requests that can fit between request_i and request_j

$= \{ \text{request}_k \mid \text{finish}_i < \text{start}_k < \text{finish}_k < \text{start}_j \}$

faster? activity scheduling algorithm.

1. sort all requests by finish time
2. Add fake requests
   $\left.\begin{array}{l} \text{request}_{\emptyset} \\ \text{request}_{n+1} \end{array}\right\}$ boundary markers
3. The largest <u>non-conflicting</u> subset of $\text{REQUESTS}_{\emptyset, n+1}$ is our answer.

How to do step #3?
- divide and conquer!

Finding The Largest Non-Conflicting Sublist of REQUESTS_{ij}

request$_i$ request$_j$
$i<j$

① if $i \geq j$ (we have sorted requests by finish time), then REQUESTS_{ij} is empty.

② If REQUESTS_{ij} is empty for any reason, then there is no conflicts to remove.

③ If REQUESTS_{ij} is not empty, then that means there must exist some request$_k$ where $i<k<j$ that will appear in the optimal solution.

If only one request$_k$, that request will appear in optimal solution.

if many requests fit between request$_i$ and request$_j$, we have to find the largest non conflicting subset.

How to find it?

Divide and conquer!

Let request$_k$ = a request in the optimal solution between request$_i$ and request$_j$

$$\boxed{\left|\text{OPTIMAL REQUESTS}_{ij}\right| = \left|\text{optimal REQUESTS}_{ik}\right| + 1 + \left|\text{optimal REQUESTS}_{kj}\right|}$$

Let OPTIMAL-COUNT[i, j]

= number of requests in the optimal $REQUESTS_{ij}$

(max # of events possible between $request_i$ and $request_j$)

algorithm:

OPTIMAL-COUNT[i, j]

= max(OPTIMAL-COUNT[i, k]

+ 1 + OPTIMAL-COUNT[k, j]).

or ∅ if no $request_k$

(compute sum for all possible values of k and find max)

algorithm:

compute OPTIMAL-COUNT[∅, n+1]

Dynamic Programming

Once you compute a
value in OPTIMAL-COUNT[i,j],

Store it and reuse if you need
it later.

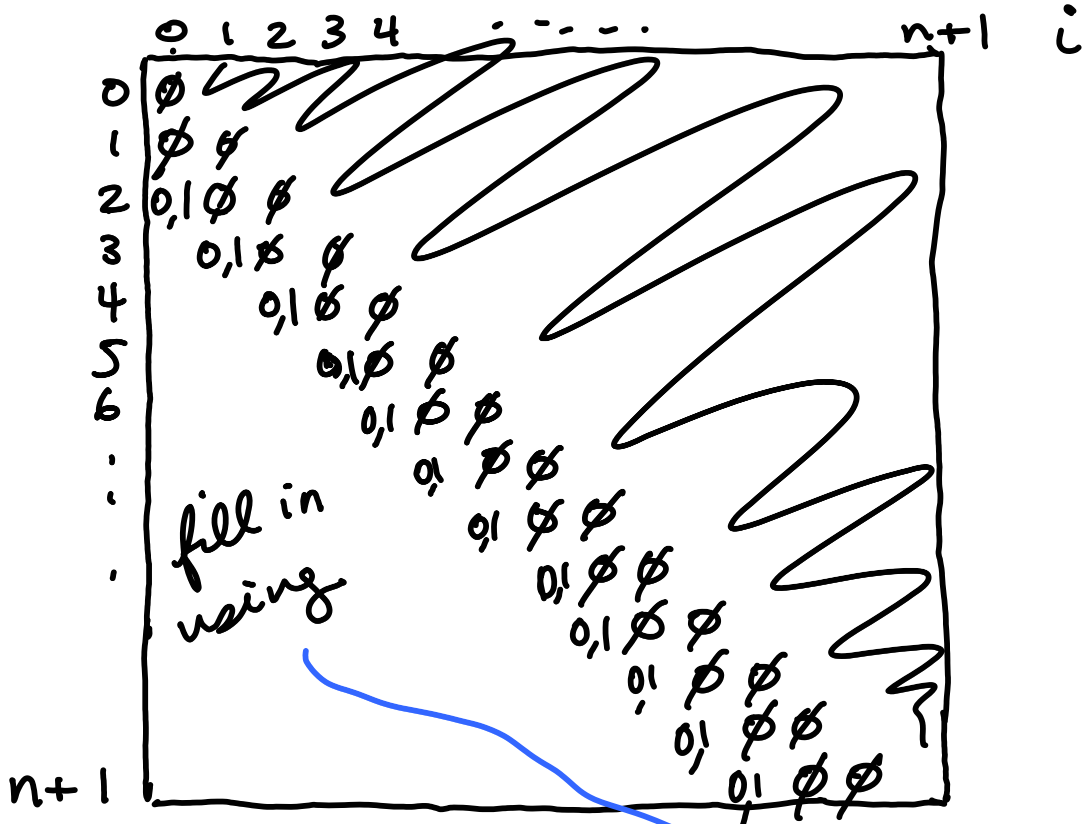
To get max reuse,

start with small subproblems and
use answers to solve bigger subproblems
("opposite" of recursion)
↓
start at bottom of recursion tree.

Like one our Fibonacci solutions...
memoization

OPTIMAL-COUNT MATRIX

or "C" for short



c[i,j] where $i \geq j = \emptyset$

$$C[i,j] \text{ where } j \geq i+3$$
$$= \max\left[C[i,k] + 1 + C[k,j] \right]$$

you can fill in the table using previously computed values in the table!

OPTIMAL-COUNT[i, j]. or C[i, j] for short

① if $i=j$, then $C[i,j] = \emptyset$.

② if $i>j$, then $C[i,j]$ is invalid.

③ if $j = i+1$

request i request $i+1$

$C[i,j] = \emptyset$

④ if $j = i+2$ for ($i = \emptyset$ to $n-1$)

compute $C[i, i+2]$

⑤ if $j = i+3$ for ($i = \emptyset$ to $n-2$)

$j = i+3$;

compute $C[i,j]$

and so on . . .

Pseudocode (use on homework 7)

```
Initialize the matrix c to all zeros.
Assume we have an array r of request
  records.
for d=1 to n+1
   for i=0 to n-d+1
      j=i+d
      if (r[i].f<=r[j].s)
         for k=i+1 to j-1
          if (
             ((r[i].f<=r[k].s)
                  &&
              (r[k].f<=r[j].s))
            &&
              (c[i,k]+1+c[k,j]>c[i,j])
          )
         then c[i,j]= c[i,k]+1+c[k,j];
```