CS330 Introduction to Analysis of Algorithms

Summer 2023
by Tiago Januario

Last day!!!

Please fill in the course evaluation!
bu.campuslabs.com/courseeval

Rate your Professor!
https://www.ratemyprofessors.com/professor/2880902
Info about final exam

date: Tuesday, June 29, 10-12 pm in CAS 224

format: similar to midterm and practice final

content:
• cumulative over the entire semester, includes lecture, lab, hw
• everything that we covered, including proofs

tools:
• closed book
• 2 cheat sheets: one from midterm, one for the second half
• double sided, hand-written. Can’t use typed pseudocode
• no electronics
• must be handed back at the end of the exam

No collaboration!!!!
How to study

• Know the definitions
  - be able to identify them on examples
• Trace all algorithms on specific input
• Be able to compute their running times
  - be able to put running times in order of growth
• Review lab problems & problems from lecture
  - make a list of tricks we used
• Review format of solution
  - DP and network flow applications
• Review proofs
Divide and Conquer

- Break down problem into smaller instances
- Find solution to large instance by first recursively solving the smaller ones
  - instance means solving the same problem with a different (here shorter) input
- Note that instances of the same size have different inputs
  - e.g. in mergeSort we make recursive calls on arrays of length n/2 but each n/2 array holds different numbers
- We use recurrence equations to compute the running time (= number of computational steps) of the algorithm
- Goal is to find a closed-form math formula
  - in general we consider one mathematical operation or one read/write operation a computational step. The exception being the lecture we spent on integer multiplication
Divide and Conquer - write recurrences

What is the recurrence corresponding to the following algorithm description?

Some algorithm takes as input an array of length n. It divides the data into 5 parts, and makes recursive calls to 4 of those parts. It combines the results in $O(n)$ steps.

A. $T(n) = 4T\left(\frac{n}{5}\right) + O(n)$

B. $T(n) = O\left(n^{\log_4 5}\right)$

C. $T(n) = 5T\left(\frac{n}{4}\right) + O(n)$

D. $T(n) = 5T\left(\frac{n}{5}\right) + O(1)$
D & C - some common recurrences and rules

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n) \quad \text{- mergeSort} \]

\[ T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n) \quad \text{- binary search} \]

If the formula is of the form \( T(n) = qT\left(\frac{n}{2}\right) + cn \) where \( q > 2 \) then we always have \( O\left(n^{\log_2 q}\right) \)

Apply this to \( T(n) = 8T\left(\frac{n}{2}\right) + cn = O(?) \)

A. \( O(\log n) \)
B. \( O(n) \)
C. \( O(n \log n) \)
D. \( O(n^2) \)
E. \( O(n^3) \)
D & C - telescoping method

\[ T(n) = 9T\left(\frac{n}{3}\right) + O(n) \]

trick: when solving the recurrence use \( cn \) instead of \( O(n) \)
D & C - recurrence tree method

\[ T(n) = 4T\left(\frac{n}{2}\right) + O(n) \]
DP - structure, layout of solution

Subproblems correspond to the solution on a smaller input
• “smaller input” is a fixed subset of elements
• may be described by 1, 2, … variables
• always make clear what a solution to a subproblem means

Recursive formula
• find the “last” choice (corresponding to the highest index) to get the opt solution
• may be a max/min over multiple cases
• recursive call is always made to (at least one) lower index

Implementation
• use memoization table
• running time: (size of table)* (operations to compute one entry of the table)

backtrack solution:
• find the actual elements in min/max solution
• find what choice was made, e.g. what was the min/max
• recurse on the index corresponding to the choice
• can store info on choice in an extra table while computing opt, not required
knapsack with infinite resources

We are given the info on $n$ items, each with value $v_i$ and weight $w_i$. We are also given a maximum capacity $W$. We have an infinite supply of each item available.

Decide how many of each item to select to maximize the total value while not exceeding the weight limit $W$.

$$OPT(i,w) =$$

What is the size of the corresponding DP table?

Select the best recursive formula:

A. $OPT(i, w) = \max\{OPT(i - 1, w); v_i + OPT(i - 1, w - w_i)\}$

B. $OPT(i, w) = \max\{OPT(i - 1, w); v_i + OPT(i, w - w_i)\}$

C. $OPT(i, w) = \max\{OPT(i - 1), v_i + OPT(i)\}$
Flow

Input: weighted directed graph G, source s, sink t, capacity on each edge c(u,v)

Flow: a value \( f(u,v) \) assigned to each edge

- flow constraints
  - when updating an existing flow we have to make sure that the constraints are still met
  - a flow is given as an input, if we know its value on the edges

Value of flow: total flow from s to t

- max flow: maximum achievable
  - the value of the max flow is a unique number
  - the amount of flow on each edge in a max flow is not unique

FF: be comfortable running it

Residual graph:

- be able to draw it, when you are given the flow on each edge (may not be max)
Flow - MFMC

Value of flow: total flow from s to t
• max flow: maximum achievable

st-cut: subset of nodes connected to s that doesn’t contain t
edges in cut: edges directed from nodes in A to V-A

capacity of cut: total capacity of edges in the cut
• not flow values
• min-cut: the cut with the minimum sum
  • not always unique
  • be able to find min-cut given the max flow
  • be able to verify whether a flow is maximum

MFMC: The value of the max flow = capacity of the min-cut
• intuition: edges in the cut are a bottleneck to the max flow
How many min-cuts do you see in this graph?

A. 1  
B. 2  
C. 3  
D. 4
Flow apps as reductions

examples:
• max bipartite matching
• unique paths
• advertising
• word problems

Solution structure:
1. define corresponding flow graph
   1. specify nodes, edges, capacities
2. find flow
3. relate that back to solution of original problem
Asymptotic growth

big-Oh:
intuition: $f(n) = O(g(n))$ if $g$ is an “upper bound” on $f$
  - always look for best upper bound
  • combinatoric def: $f(n) = O(g(n))$ iff there exist constant $c>1$ and $n_0$, such that for every $n > n_0$, $f(n) \leq cg(n)$ is true.
  • analytic def: $f(n) = O(g(n))$ iff $\limsup_{n \to \infty} \frac{f(n)}{g(n)} = 0$
  - the two definitions are equivalent

Write the definition for $\Omega$ and $\Theta$
  • Know the rules for ordering functions (see relevant slide)
Analysis of an iterative algorithm

**Algorithm 1: arrayIter\((A, B)\)**

```plaintext
/* A, B are arrays of ints */
/* this algorithm is nonsense ;) */
1 n ← len(A);
2 m ← len(B);
3 for i = 1 to n do
4     for j = 1 to m do
6 return A
```

**Algorithm 1: arrayIter2\((A)\)**

```plaintext
/* A are arrays of ints */
/* this algorithm is nonsense ;) */
1 n ← len(A);
2 for i = 1 to n do
3     for j = i + 1 to n do
5 return A
```
Analysis of iterative graph algorithms

Algorithm 1: deleteEdge(G)
/* G is the adjacency list if a graph */
1 for u in G do
2 for v in G[u] do
3 if u > v then
4 del G[u][v];
5 return G

Algorithm 1: reverseEdge(G)
/* G is the adjacency list if a graph */
1 for u in G do
2 for v in G[u] do
3 if u > v then
4 del G[u][v];
5 add G[v][u];
6 return G
Graph data structures

Adjacency list:
• implement as hash table of hash tables
• hash table:
  • can be used for (non)numerical indices: H[id]
  • can access index in O(1)
    • notation for nested table: G[u][v], for v in G[u]
• iterate over entire adj. list takes O(n+m)
Graph data structures

How would you solve the following using an adjacency list?

Problem: Given a directed graph $G$, fix some random order of its nodes.

- Delete all back-edges (from higher to lower index) in this graph.
  - What kind of graph is this? How can we find the source nodes?
- Instead of deleting edges, reverse the direction of the back edges (and leave forward edges unchanged). How should we update the adjacency list?
BFS and DFS and applications

Output
- tree/ parent list
- can use to backtrack paths from s
- depends on random choices

BFS:
- find paths with min number of edges
- can use to explore weighted graphs but not to find minimum weighted paths

DFS:
- doesn’t find shortest paths!
- What is it used for?
  - connectivity
    - is there a (directed) path between nodes u and v?
    - is the graphs connected

applications:
- connected components
- strong connectedness
- decide whether DAG and topological order
Scheduling and partitioning

**greedy algorithm:**
- makes choices based on local information, doesn’t look at the global structure.
- Once a decision has been made, it’s never changed.

**algorithm:**
- need to fix (and state) what order to iterate over elements
- sorting or selection order is part of the running time
- fine to order on the fly, e.g. the way the next edge is selected in Prim’s

**proof:**
- most often we assume that there is some optimal solution
- we could find it using brute force
- we prove that the output of the greedy algorithm is at least as good as the optimal.
Priority queues

Data structure containing keys and corresponding values <key, value>
• keys are priorities
• we used it in applications where lower key = higher priority

implementation:
• many implementations
• we assume binary min-heap
• running times:
  • insert, update (DECREASE-KEY), EXTRACT-MIN: O(log n)
    • if n elements are stored (edges are O(m)!!)
  • PEEK (= read key and value of minimum key): O(1)
Dijkstra’s

Find shortest paths in a weighted graph
• negative edge weights are not allowed

minimum path = minimum sum of weights
• in case of unweighted graphs we may assume that the edge weights are one.
  Then number of edges = weight of paths

uses PQ for efficiency