Algorithms Exam [1]

Oct 20 2009
kl 1400 – 1800
"vag och vatten"
salar

Ansvarig:
Devdatt Dubhashi Tel. 772 1062 (Willard Rafnsson) Rum 6479 EDIT

Points:	60	
Passing criteria:	Chalmers	5:48, 4:36, 3:24
	GU	VG:48, G:28
	Doktorander	G:36
Helping material:	Textbook, notes, course page stuff.	

- Recommended: First look through all questions and make sure that you understand them properly. In case of doubt, do not hesitate to ask.
- **Answer all questions in the given space on the question paper (the stapled sheets of paper you are looking at). The question paper will be collected from you after the exam. Only the solutions written in the space provided on the question paper will count for your points.**
- Use extra sheets only for your own rough work and then write the final answers on the question paper.
- Answer concisely and to the point. (English if you can and Swedish if you must!)
- Code strictly forbidden! Motivated pseudocode or plain but clear English/Swedish description is fine.

Lycka till! Good Luck!

[1] 2009 LP 1, INN200 (GU) / TIN090 (CTH).

Problem 1 Sorting Revisited [10] A

(a) Give a linear time algorithm to sort an array that is known to consist only of 0s and 1s.
1. Count nr. of 0s in A, k. $\mathcal{O}(n)$; $|A|=n$
2. Write 0s in first k fields of A, $\mathcal{O}(n)$ and 1s in the rest

total running time: $\mathcal{O}(n)$

(b) Suppose it is known that an array $A[1 \ldots n]$ contains at most k distinct values (which are not known). Describe a data structure and algoritahm to sort the array in time $O(nk)$. For what range of values of k is this an asymptotically better algorithm than Mergesort?

1. Create arrays v[1..k] and c[1..k]. $\mathcal{O}(k)$
2. Scan A k times to find the k different elements. Write those elements into v in sorted order. $\mathcal{O}(nk)$
3. For i from 1 to k, write nr. of occurrences of v[i] in A into c[i]. $\mathcal{O}(nk)$
4. From start of A, for i from 1 to k, write v[i] into next c[i] fields of A. $\mathcal{O}(nk)$

total running time: $\mathcal{O}(nk)$

Mergesort runs in $\mathcal{O}(n \log n)$.
For $\mathcal{O}(nk) \subsetneq \mathcal{O}(n \log n)$ to hold, then in a sense $k \ll \log n$ must hold.
Formally, $|\log n| \cdot c$ must dominate $|k|$, for <u>any</u> C.
This is denoted $k \in o(\log n)$.

(c) For the same situation as in (b), describe a data structure and algorithm to sort in time $O(n \log k)$. For what range of values of k is this an asymptotically bettre algorithm than Mergesort?

Use priority queue Q to store (v, c) pairs.
Key: value in A
value: occurrences of v in A

1. For i from 1 to n, check if Q contains (A[i], c) for some c. If so, update c to c+1. if not, insert (A[i], 1) into Q. $\mathcal{O}(n \log k)$

Insert, search, & removal of the pair w. minimum key takes at most $\mathcal{O}(\log k)$ time.

2. From start of A, until Q is empty, remove (v, c) w. minimum v in Q, and write v into next c fields of A. $\mathcal{O}(n \log k)$

total running time: $\mathcal{O}(n \log k)$

This algorithm is asymptotically better than mergesort when $\log k \in o(\log n)$.

(d) Show that the algorithm in (c) is asymptotically best possible.

A has $n!$ distinct permutations P, when $k=n$.
Sorting A amounts to finding the sorted $\sigma \in P$.
Best we can do: Algorithm finding path to σ in a binary decision tree with all $p \in P$ as leaves. Running time = height of tree =

$$\log n! \in \Theta(n \log n) \quad ⊛$$

If an element in A occurs $b>1$ times, $|P|$ shrinks to $\frac{n!}{b!}$, A has k distinct values.
If n_i is nr. of occurrences of value i in A, then

$$|P| = \frac{n!}{n_1! n_2! \cdots n_k!} \leq \frac{n!}{(\frac{n}{k}!)^k}$$

Running time = height of tree =

$$\log\left(\frac{n!}{(\frac{n}{k}!)^k}\right) \in \Theta(n \log k) \quad \leftarrow \text{follows from } ⊛$$

Problem 2 Greedy Tape Storage [10] Let $P_1, P_2, \ldots, P_n$ be n programs to be stored on a tape. Program P_i requires s_i kilobytes of storage; the tape has enough capacity to store all programs. We also know how often each program is used: a fraction π_i of requests concern program P_i (thus $\sum_i \pi_i = 1$). Information is recorded along the tape at constant density and the speed of the tape drive is also constant. After a program is loaded, the tape is rewound to the beginning. So, if the programs are stored in the order $i_1, i_2, \ldots, i_n$, the time needed to load program P_j when it is requested is $\sum_{1 \leq k \leq j} s_{i_k}$ and the average time to load a program (computed over all program requests) is thus:

$$\overline{T} = c \sum_{1 \leq j \leq n} \left[\pi_{i_j} \sum_{1 \leq k \leq j} s_{i_k} \right],$$

where the constant c depends on the recording density and the speed of the drive. We want to minimise $\overline{T}$ using a greedy algorithm. Prove, or give a counter-example for each of the following: to minimize $\overline{T}$, we can select the programs

(a) in order of non-decreasing s_i, no. counterexample:
P_1 : $s_1 = 3$ $\pi_1 = 0.1$ P_2 : $s_2 = 7$ $\pi_2 = 0.9$
This approach: Storeorder P_1P_2. $\overline{T} = c(0.1 \cdot 3 + 0.9(3+7))$
$= c\,9.3$
Better approach: Storeorder $\underline{P_2P_1}$. $\overline{T} = c(0.9 \cdot 7 + 0.1(7+3))$
$= c\,7.3$

(b) in order of non-increasing π_i, no. counterexample:
P_1 : $s_1 = 1$ $\pi_1 = 0.3$ P_2 : $s_2 = 9$ $\pi_2 = 0.7$
This approach: storeorder P_2P_1. $\overline{T} = c(0.7 \cdot 9 + 0.3(9+1))$
$= c\,9.3$
Better approach: store order $\underline{P_1P_2}$. $\overline{T} = c(0.3 \cdot 1 + 0.7(1+9))$
$= c\,7.3$

(c) in order of non-increasing $\frac{\pi_i}{s_i}$. yes. Given (arbitrary) $P_1, \dots, P_n$,
S : proposed ordering of the P_i r_i
O : any ordering of P_i.
We show (by exchange argument) that O cannot both be i) optimal, and ii) different from S.
Re-index P_i, s_i, π_i according to ordering O.
Assume $O \neq S$ and O optimal. Then there is some i s.t. $\frac{\pi_i}{s_i} < \frac{\pi_{i+1}}{s_{i+1}}$. ⊛ For some T,
$\frac{1}{c}\overline{T} = T + \pi_i s_i + \pi_{i+1} s_i + \pi_{i+1} s_{i+1}$. Swapping P_i, P_{i+1} yields
$\frac{1}{c}\overline{T}' = T + \pi_{i+1} s_{i+1} + \pi_i s_{i+1} + \pi_i s_i$.
To prove $\overline{T}' < \overline{T}$, we only need to prove $\pi_i s_{i+1} < \pi_{i+1} s_i$.
This follows from ⊛, since
$\frac{\pi_i}{s_i} < \frac{\pi_{i+1}}{s_{i+1}} \iff \pi_i s_{i+1} < \pi_{i+1} s_i$.
i) is contradicted. If no i exists satisfying ⊛, then ii) is contradicted. Thus S is optimal.

(For a counter-example, you have to specify program sizes and frequencies and show that the selected order does not give the optimum.)

Problem 3 Buying Stocks [10] You're consulting for a small computation-intensive investment company and they have the following type of problem that they want to solve over and over every day. They're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $1, 2, \dots, n$. For each day i, they have a price $p(i)$ per share for the stock on that day. Suppose that during this time period, they want to buy 1000 shares on some day and sell all those shares on some later date. They want to know: when should they have bought and when should they have sold to make as much money as possible?
For example, suppose $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then you should "*buy on day 2 and sell on day 3*". This would make a profit of SEK 4 on each share, the maximum in that period. (This was
This was Solved Exercise 2 in [KT, Chapter 5] where a Divide-and-Conquer strategy was used to develop a $O(n \log n)$ algorithm. Here your goal is to design a faster algorithm. Let $OPT(i)$ denote the optimal solution considering transactions are possible only on days $1 \cdots i$.

(a) What is the value we are trying to compute in terms of this notation?
$OPT(n)$.

(b) What is the value $OPT(1)$?
0.

(c) Write a recurrence for $OPT(i)$ based on what is done on day i.
In the optimal solution, we either i) sell on last day or ii) not. If i), we buy on one of previous $i-1$ days which has lowest purchase price, $b(i)$. So,
$OPT(i) = \max(OPT(i-1), p(i) - b(i))$.

(d) Implement the recurrence efficiently in pseudo-code.

First compute all $b(i)$. Let $b[1..n]$ and $b[1] = p(1)$.

```
1. for i from 2 to n,                                  }
   1.1 if $p(i) > b[i-1]$, then $b[i] = p(i)$.         }  $O(n)$
       else $b[i] = b[i-1]$.                            }
```

Then compute all $OPT(i)$. Let $OPT[1..n]$ and $OPT[1] = 0$.

```
2. for i from 2 to n,                                                      }
   2.1 if $p(i) - b[i] > OPT[i-1]$ then $OPT[i] = p(i) - b(i)$              }  $O(n)$
       else $OPT[i] = OPT[i-1]$                                             }
```

We then have $OPT(n)$.

```
3. return $OPT[n]$.          } $O(1)$      total: $O(n)$
```

(e) What is the time and space complexity of your algorithm?

Time: $O(n)$.
Space: $O(n)$. (used two arrays of size n)

(f) Can your algorithm also tell you which day to buy and sell and if so, how?

yes. We find the sell-day d_s by analyzing $OPT[\,]$.

```
1. $d_s = n$
2. for i from n-1 to 1,
   2.1 if $OPT[i] \neq OPT[i+1]$, then $d_s = i+1$ and break.
```

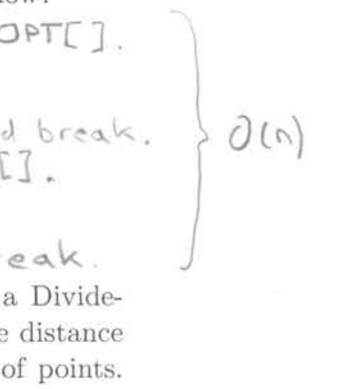
we now find buy-day d_b by analyzing $b[\,]$.

```
3. $d_b = n$
4. for i from n-1 to 1,
   4.1 if $b[i] \neq b[i+1]$, then $d_b = i+1$ and break.
```

$\Big\}\ O(n)$

Problem 4 Closest Points [10] Given n points in the plane, develop a Divide-and-Conquer algorithm to count the number of pairs of points such that the distance between the points is at most twice the distance between the closest pair of points. Explain the conquer step clearly and briefly justify it. (HINT: First compute the minimum distance between any two points.) Follow the hint using the closest pair algorithm from class & book. Yields minimum distance δ in time $O(n \log n)$.

To count pairs of points within distance 2δ of each other: Divide plane in two parts vertically s.t. ca. the same nr. of points are in each part. Apply algorithm recursively on left & right part. Yields n_l, n_r ; nr. of close pairs in left and right part, resp.. In conquer step, consider points within distance 2δ of the vertical line, sorted by y-coordinate. Compare each point to the following 40 points to see if their distance is $\leq 2\delta$. If so, and the points are not both on same side of vertical line, then increment n_c. Finally, return $n_l + n_r + n_c$.

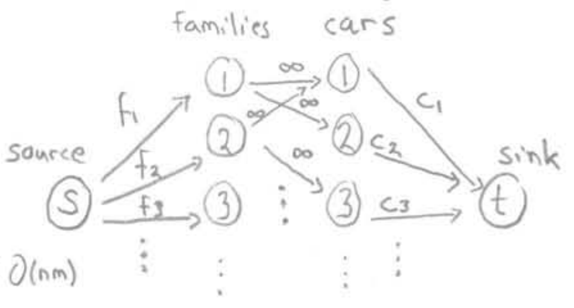


(minor detail: if plain has ≤ 1 points, return 0)

Running time satisfies recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$.
Thus, running time is $O(n \log n)$.

Justification of conquer step: Consider tesselation of vertical split. For same reason as in book, each tile has at most 1 point. If a point p is in a given tile (For instance ⊛), then each other point above p within dist. $\leq 2\delta$ must be in p's row, or in one of the 4 rows above p. Each row has 8 tiles. Total: $8 \cdot 5 = 40$ tiles. Worst case, each tile is populated. Thus at most 40 comparisons needed.

Problem 5 Bilkoperative Picnic [10] Majorna's Bilkoperative (car pool society) is organising a picnic. There are n families with family i having f_i members, and there are m cars, with car$j$ having capacity to seat c_j people. Also family i has a list L_i of cars which they are willing to to travel in (for reasons of safety). Not all members of a family need be in the same car. Give an efficient algorithm to determine if everyone can be packed into the available cars or whether they will have to rent more cars. Analyse the running time of your algorithm.



Use network flows.

Construct flowgraph G:

- Bipartite graph of families and cars: connect family i to each car in L_i. Orient edges towards cars. Make weights infinite. } $\mathcal{O}(nm)$
- Add source s. For each family i, add edge (s,i) w. capacity f_i. } $\mathcal{O}(n+m)$
- Add sink t. For each car j, add edge (j,t) w. capacity c_j.

Total construction time: $\mathcal{O}(nm)$. Ford-Fulkerson alg. yields max flow in time $\mathcal{O}(nm\sum f_i)$. } total time: $\mathcal{O}(nm\sum f_i)$

Correctness: G has maxflow $\sum f_i \iff$ all families can be packed into cars.

Proof: $(\Rightarrow)$ Given a max flow, you can construct people-car distribution by placing members of family i into car j equal to flow on edge (i,j).

$(\Leftarrow)$ Given a people-car distribution, construct max flow: For each member in family i in car j, put 1 flow unit on edge (i,j). ▨

Problem 6 Cybercommunities [10] An interesting problem in internet algorithmics is to find a collection of densely connected web sites i.e. set of web sites which have a lot of hyperlinks between each other. Such collections are called *cybercommunities* - for example, the set of sites discussing *Harry Potter* films is one such cybercommnunity. In graph theory terms, the concept is captured by the notion of a *clique*: a clique is an undirected graph $G = (V, E)$ is a subset $U \subseteq V$ such that for any two vertices $u, v \in U$, $(u, v) \in E$. Consider the optimization problem of finding the size of the largest clique in a given graph.

(a) Formulate a decision problem corresponding to whether or not a graph has a clique of a certain size. Show that the decision problem and the optimization problem are polynomial-time equivalent i.e. if one can be solved in polynomial time, so can the other. Decision problem CLIQUE:

"Given graph G and $k\in\mathbb{N}$, does G have a clique of size $\geq k$?".

Decider D and optimizer O can be defined in terms of each other:

```
D(G,k) = "Given graph G and $k \in \mathbb{N}$,
  1. if $k \leq O(G)$, return yes.
  2. return no."

O(G) = "Given graph $G = (V,E)$,
  1. For k from $|V|$ to 1,
    1.1 if $D(G,k)$ = yes, return k.
  2. return 0"
```

D adds $\mathcal{O}(1)$ operations to a single run of O.

O invokes D $\mathcal{O}(n)$ times, adding $\mathcal{O}(1)$ operations each time.

So if O runs in polynomial-time, then so does D, and vice versa.

So decision problem and optimization problem are polynomial-time equivalent.

(b) Show that the decision problem is in $\mathcal{NP}$. We show CLIQUE $\in \mathcal{NP}$ by giving a polynomial-time certifier. Certificate: set U of vertices (the clique).

```
C(G,k,U) = "Given $G=(V,E)$, $k \in \mathbb{N}$ and $U \subseteq V$,
  1. if $|U| < k$, return no.
  2. for each $(u,v) \in U^2$,
    2.1 if $(u,v) \notin E$, return no.
  3. return yes."
```

This algorithm runs no worse than $\mathcal{O}(n^4)$. polynomial time

(c) Show that the decision problem is $\mathcal{NP}$-complete by giving a reduction from the *Independent Set* problem (which is know to be $\mathcal{NP}$-complete). Recall IS:
"Given graph G and $k \in \mathbb{N}$, does G have an independent set of size $\geq k$?"

We reduce an instance G, k of IS ($G=(V,E)$) to an instance G', k' of CLIQUE ($G'=(V',E')$) as follows.

1. $k' = k$	$\mathcal{O}(1)$
2. $V' = V$	$\mathcal{O}(1)$
3. $E' = V^2 \setminus E$	$\mathcal{O}(\|V\|^2)$

In short, G' is the complement graph of G.
Total running time: $\mathcal{O}(n^2)$ where $n = |V|$.

Correctness: If U is an independent set in some graph $G=(V,E)$, then there is no edge between any two vertices $u,v \in U$ in G ($(u,v) \notin E$). Then, in the complement graph $\overline{G}=(V,\overline{E})$, u and v are connected by an edge ($(u,v) \in \overline{E}$). Since this holds for any $(u,v) \in U^2$, then U is a clique in $\overline{G}$. Running this argument backwards, we get that if U is a clique in $\overline{G}$, then U is an independent set in G. ∎