

CS 101 Homework 1

Background (Order and Recurrence Relations), Speeding up algorithms with restructuring and Data Structures.
Answers to Selected Questions

Order (10 points) Is it always the case that $f(2n) \in O(f(n))$? If so, give a proof; if not, give a counter-example.

No. As a countering example, let $f(n) = 2^n$. Then $f(2n) = 2^{2n} = (2^2)^n = 4^n$. Assume 4^n were in $O(2^n)$. Then $4^n \leq c2^n$ for some $c > 0$ and all $n \geq n_0$. Then $c \geq 4^n/2^n = 2^n$ for all $n \geq n_0$, which cannot happen for $n > \log c$. From this contradiction, $f(2n) \notin O(f(n))$.

Levelling a DAG. (20 points) The next two problems concern the following computational problem and algorithm. The input is a directed, acyclic graph (DAG) $G = (V, E)$. A *levelling* of a DAG assigns each $u \in V$ a positive integer value $level(u)$ meeting the following constraint: If $(u, v) \in E$, then $level(u) < level(v)$. (One application that might be helpful is to think of nodes as jobs, and an edge (u, v) represents that the output of job u is an input to job v . A levelling assigns each job a time so that all jobs at that time can be done concurrently.) The following algorithm strategy computes a levelling of the DAG that minimizes the maximum assigned level.

Level(G=(V,E): DAG): array of integers indexed by V

1. Initialize $level(x)$ to NIL for all $x \in V$.
2. Repeat $n = |V|$ times:
3. Find a node x so that there are no edges to x from unassigned nodes, i.e., no edges (y, x) where $level(y) = NIL$.
4. If there are no edges $(y, x) \in E$ THEN $level(x) \leftarrow 1$.
5. ELSE $level(x) \leftarrow 1 + \max_{y, (y,x) \in E} level(y)$.
6. Return the array $level$.

[Part A: Correctness proofs– 10 points] The following is a proof that the above algorithm finds the levelling that uses the smallest number of levels. The proof is missing some phrases, denoted by Roman numerals. Fill in the missing phrase to complete the proof. (I put the filled in phrases in **boldface**).

We need to show that the array by the above algorithm $level$ is a levelling and that no other levelling has a smaller maximum level assigned.

To prove that $level$ is a levelling of G , we need to show that for any u, v with $(u, v) \in E$, $level(u) < level(v)$. Consider the iteration in which $level(v)$ was defined. Since $(u, v) \in E$, the IF condition in line 4

was **false**, so we assigned $level(v)$ the value $1 + \max_{\{y, (y,v) \in E\}} level(y)$. Now $\max_{y, (y,x) \in E} level(y) \geq level(u)$ since $(u,v) \in E$, so $level(v) \geq 1 + level(u)$, so **level(v) \geq level(u)**, which is what we needed to prove.

Now we need to show that no other levelling L has a smaller maximum level assigned. Let v_t be the vertex assigned a level in the t 'th iteration of our algorithm. We will prove by strong **induction** that for all t , $level(v_t) \leq L(v_t)$.

Assume the claim holds for all $1 \leq i < t$, i.e., that **level(v_i) \leq L(v_i)** for all nodes v_i previously assigned a level. We need to show that the claim also is true at t , i.e., that **level(v_t) \leq L(v_t)**.

There are two cases, depending on the branch in line **four**. In the first case, there are no edges of the form (\mathbf{u}, v_t) . In this case, we assign $level(v_t)$ value **one** in line **four**. Since $L(v_t)$ is a **positive integer** from the definition of leveling, we have $level(v_t) = 1 \leq L(v_t)$ as needed.

In the other case, there are edges of the form (\mathbf{u}, v_t) in E , and we assign $level(v_t)$ value $1 + \max_{\{y, (y,v_t) \in E\}} level(y)$. in line **five**. Pick y_0 so that $(y_0, v_t) \in E$ and $level(y_0) = \max_{y, (y,v_t) \in E} level(y)$. Then since at time t , there are no edges of the form (\mathbf{y}, v_t) where **level(y) = NIL**, $y_0 = v_i$ for some $i < t$. Thus, we can apply the **induction hypothesis** to y_0 to see that **level(y₀) \leq L(y₀)**. Since L is a levelling, and $(y_0, v_t) \in E$, $L(v_t) > L(y_0)$ and since both are integers, $L(v_t) \geq L(y_0) + 1$. Thus, we have $level(v_t) = level(y_0) + 1 \leq L(y_0) + 1 \leq L(v_t)$, which is what we needed to prove.

Therefore, by strong induction on t , we have $level(v) \leq L(v)$ for all $v \in V$. It follows that $\max_v level(v) \leq \max_v L(v)$, so our algorithm uses the minimum possible number of levels.

Part B: Data structures and efficient versions of algorithms, 10 points.

For the above algorithm (levelling a DAG) give an efficient implementation, specifying pre-processing and data structures. Give a time analysis. Assume that the directed, acyclic graph G is given in adjacency list format, with the list at x being nodes y where $(x, y) \in E$. Let $n = |V|, m = |E|$, and give your time analysis in terms of both n and m .

In adjacency list format, for each node v we are given the list of v 's successors in G . To implement the strategy, we need to repeatedly

1. Find a node v_t with no unassigned predecessors
2. Run through all of its predecessors, and examine their assigned levels.

To handle the second efficiently, let's invert the adjacency list to also get a list of predecessors for each node. We can do this by running through

the successor lists, and for each edge (x, y) we see in x 's successor list, add x to y 's predecessor list.

Note that it doesn't matter which of the available nodes we pick as v_t , and a node doesn't change from being available (not having unassigned predecessors) to being unavailable. So we can use a stack or queue L to keep tabs of the set of nodes that are available, and not yet assigned levels.

Let's keep the number of unassigned predecessors of each node in a separate array, so that we can tell if a node should be added to L . Then the above pattern becomes:

1. Find any member x of L
2. delete x from L
3. Run through predecessors y of x , keeping maximum *level*.
4. Run through successors y of x , and decrement count of unassigned predecessors for each such y .
5. For each such y , add y to L if this count reaches 0.

This gives the algorithm in the following pseudocode, using a list for L .

```

0   Initialize InD(1..n), Pred(1..n);
1   for i = 1..n;
2       for each z in N(i) do: InD(z)++; Append i to Pred(z);
3   Initialize L : linked list of nodes;
4   for i = 1..n do:
10      if InD[i]=0
11          insert(L, i)
12   Initialize level(1..n)
13   while L not empty
14       x <- head(L); In(x) <- F
15       L <- tail(L)
16       predlevel ← 0;
17       For each y in Pred(x) do:
18           IF level(y) > predlevel THEN predlevel <- level(y)
19       level(x) <- 1+predlevel
20       For each y in N(x) do:
21           InD[y] <- InD[y] - 1
22           IF InD[y] = 0 THEN insert(L, y)
23   Return level

```

Each node is only inserted into L once, the first time $\text{InD}(x) = 0$, and so it is removed from L at most once. Note that lines 2 and 20-22 take time proportional to the out-degree of x , since we run through all successors y

of x . So the total time over all iterations for these lines will be $\sum_x out - deg(x) = m$, since each edge is counted as the out-edge for exactly one x . Similarly, lines 17-18 take time proportional to the in-degree of x , for another $O(m)$ total time. The other lines are done once per node and take constant time, so are altogether $O(n)$. So this version has time complexity $O(n + m)$.

Merge lists (20 points) You are given an array of k sorted, non-empty lists, $L[1..k]$, where each $L[I] = a[I, 1] < a[I, 2] < \dots < a[I, n_I]$. Let n be the total sizes of all the lists (so in particular, $n > k$). Give an $O(n \log k)$ time algorithm that returns a sorted list containing exactly the elements in the union of the k lists.

Similarly to the top k in a heap, we'll use a supplementary min-heap, each element keyed by a value but having another field with a pointer to the place that element occurred in one of the lists. We initialize the heap to contain the heads of each list. Each iteration, we place the top value of the heap in the output array, and replace it with the next element in the list it came from (by following the pointer to the list, and then following the successor pointer in the list).

We maintain the invariant that the heap elements are the first elements from each list not already in the output array. Since the original lists are sorted, this means that the next largest element has to be the smallest element in the heap.

Note that the heap always has size at most k , since there is at most one element from each list, so heap operations are $O(\log k)$. Thus, the algorithm's time is $O(n \log k)$, since we go through n iterations, and in each iteration, we delete the root of the heap, and then insert one new element into the heap, both of which take $O(\log k)$ time.

Maximum sum of four symmetric entries, 10 points Give an efficient algorithm for the following problem: Given an $n \times n$, (where $n \geq 2$) matrix of integers $A[I, J]$, find four entries of the form $A[I_1, J_1]$, $A[I_2, J_1]$, $A[I_1, J_2]$ and $A[I_2, J_2]$ where $1 \leq I_1 < I_2 \leq n$ and $1 \leq J_1 < J_2 \leq n$ whose sum is maximum. Try to be strictly faster than $O(n^4)$.

For each $I_1 \neq I_2$, we can find the best values of J_1 and J_2 (given the choices for I 's) as those two J that give the largest and second largest values for $A[I_1, J] + A[I_2, J]$. Let M_1 be a J that gives the largest $A[I_1, J] + A[I_2, J]$, and M_2 that that gives the second largest such value. (If there's a tie for the largest value, pick M_1 with the smallest index of those columns that give the largest value and let M_2 be the second smallest such index.)

It is fairly obvious that these are the best choices, but for completeness I'll prove it. Let J_1, J_2 be any other choice of two distinct rows, and assume without loss of generality, that $A[I_1, J_1] + A[I_2, J_1] \geq A[I_1, J_2] +$

$A[I_2, J_2]$, and if the two are equal, assume that $J_1 < J_2$. Then J_2 is not M_1 , since M_1 is the first column with the maximum sum and J_2 is not, so since M_2 has the best sum of any column other than M_1 , $A[I_1, M_2] + A[I_2, M_2] \geq A[I_1, J_2] + A[I_2, J_2]$. Since M_1 has the maximum sum of any column, $A[I_1, M_1] + A[I_2, M_1] \geq A[I_1, J_1] + A[I_2, J_1]$. Therefore, $A[I_1, M_1] + A[I_2, M_1] + A[I_1, M_2] + A[I_2, M_2] \geq A[I_1, J_1] + A[I_2, J_1] + A[I_1, J_2] + A[I_2, J_2]$. for any choice of J_1, J_2 , so choosing M_1 and M_2 gives us the largest symmetric sum for that I_1 and I_2 .

We can carry this strategy out in $O(n^3)$ time using the following algorithm:

1. $BestSumof4 \leftarrow -infinity$
2. FOR $I1=1$ to $n-1$ do:
3. FOR $I2=I1+1$ to n do:
4. $BestfortheseRows \leftarrow -infinity$
5. $SecondBestfortheseRows \leftarrow -infinity$
6. FOR $J=1$ to n do:
7. $ThisSum \leftarrow A[I1, J] + A[I2, J]$
8. IF $ThisSum > BestfortheseRows$
9. THEN $SecondBestfortheseRows \leftarrow BestfortheseRows;$
 $BestfortheseRows \leftarrow ThisSum$
10. ELSE IF $ThisSum > SecondBestfortheseRows$ THEN
 $SecondBestfortheseRows \leftarrow ThisSum$
11. $Sumof4fortheserows \leftarrow BestfortheseRows + SecondBestfortheseRows$
12. IF $Sumof4fortheserows > BestSumof4$ THEN $BestSumof4 \leftarrow$
 $Sumof4fortheserows$
13. Return $Bestsumoffour$

The loop structure here has three nested loops each iterating at most n times, with constant time operations inside, so the total time is $O(n^3)$.

Best Party Time-20 points You have n friends, of whom you want to invite at most $1 \leq k \leq n$ to a two-hour party. Each friend F_i told you the earliest time a_i that they can arrive, and the latest time d_i that they would need to leave. You have also rated each friend with a “fun factor” f_i saying how much fun they add to the party. You want to pick a time t to start the party and invite up to k friends i that can attend the whole party ($a_i \leq t < t + 2 < d_i$). Given this constraint, you wish to maximize the sum of the fun factors of invited guests. Give an efficient algorithm to solve this problem. My best algorithm is $O(n \log n)$.

There is always an optimal party that starts at some a_i , because for any party, if we move the start time to the latest a_i for an invited guest, we

can invite the same set of people, and so get the same sum of fun factors. Therefore, if we compute the best party starting at $t = a_i$ for each of the a_i , the best of these gives us the best overall party.

Fix $t = a_i$. Since the party goes on for two hours, the set of available friends are those F_i with $a_i \leq t < t + 2 \leq d_i$. Call this set $Avail_t$. If $|Avail_t| \leq k$ we can invite everyone in this set, which will give us the largest total fun factors. Otherwise, the best total fun factors will be obtained by inviting the k elements of $Avail_t$ with the largest fun factors. Call this set $Invite_t$. Call the set of the other people in $Avail_t$, $(Avail_t - Invite_t) Uninvite_t$.

Say that the F_i are sorted by a_i . Then between a_i and a_{i+1} , some people might have become available, and others become unavailable. If a person becomes available, and we have k or more available friends already, we have to decide whether to invite them. We'll want to invite them if their fun factor is greater than the smallest fun factor of a person in $Invite$, since then they must be in the top k people. If so, we need to move that smallest fun factor person to the $Uninvite$ set. If a person in $Invite$ becomes unavailable, then we want to replace them, if possible, with a person in $Uninvite$; the k th largest fun factor of an available person will be the largest in $Uninvite$.

Thus, we need to be able to support the following operations:

1. Go through the a_i in increasing order
2. Find all F_j that arrive at time a_i .
3. Compare f_j to the smallest fun factor in $Invite$
4. Add f_j to either $Invite$ or $Uninvite$
5. Possibly delete the smallest fun factor in $Invite$
6. Find all F_j that leave between $a_{i-1} + 2$ and $a_i + 2$.
7. Test whether F_j is in $Invite$ or $Uninvite$.
8. Delete F_j from $Invite/Uninvite$ as appropriate.
9. If we delete F_j from $Invite$, move the largest fun factor friend from $Uninvite$ to $Invite$

Thus, the operations that our data structures for the sets $Invite$ and $Uninvite$ need to support are:

1. Find the smallest element of $Invite$ /largest in $Uninvite$.
2. Delete the largest/smallest elements.
3. Insert a new element
4. Given the *name* of a friend, decide which set the friend is in and delete it from that set

5. Tell whether Invite has k elements

The NamedHeap data structure from class (see skyline problem) supports these operations. Using a max-NamedHeap for Uninvite (of size up to n), and a min-NamedHeap for Invite (of size up to k), we can perform FindMin/FindMax in $O(1)$ time, DeleteMin in $O(\log k)$ time, DeleteMax in $O(\log k)$ time, IsFriendInvited? in $O(1)$ time (by looking at the pointer array for the Invited NamedHeap; a NIL pointer for F means F is not in Invited), and IsInvitedFull? in $O(1)$ time (by checking whether the size of the Invited heap is k , since we keep a size counter for any heap in order to handle inserts.) We'll also add one more operation to Invited: Total is the sum of the fields of all fields in the heap. We can maintain this in $O(1)$ time by adding any new element's field to Total when we insert, and subtracting it when we delete. (That only adds a constant time to the Insert and Delete operations)

In order to quickly identify which friends to add or delete from Available, we create two sorted lists, one of pairs (a_i, i) and the other of pairs (d_i, i) as a pre-processing step.

This leads to the following algorithm:

BestParty ($F[1..n]$: array of triples $(f[I], a[I], d[I])$):

1. $MostFun \leftarrow 0; BestTime \leftarrow NIL$
2. Initialize Arrivals $[1..n]$ an array of pairs (time,friend) each to $(a[I], I)$.
3. Heapsort Arrivals by increasing time
4. Initialize Departures $[1..n]$ an array of pairs (time,friend), each to $(d[I], I)$
5. Heapsort Departures by increasing time
6. Initialize empty min-NamedHeap Invited of size k and empty max-NamedHeap Uninvited of size n , with possible names $1..n$.
7. $I \leftarrow 1, J \leftarrow 1, T \leftarrow a_1$.
8. While $I \leq n$ do:
 9. $T \leftarrow Arrivals[I].time$ {Consider next start time}
 10. While $I \leq n$ and $Arrivals[I].time = T$ do: {Add new arrivals}
 11. IF InvitedIsFull
 12. THEN IF Invited.FindMin $\neq f[Arrivals[I].friend]$
 13. THEN do:
 14. $K \leftarrow Invited.FindMin.friend$;
 15. Invited.DeleteMin
 16. UninvitedInsert($f[K], K$)

```

17.             InvitedInsert(f[I],I)
18.             ELSE UninvitedInsert(f[I],I)
19.             ELSE InvitedInsert(f[I],I);
20.         I++
21.     While  $J \leq n$  and  $Departures[J].time < T + 2$  do: {Delete new
departures}
22.         IF IsInvited[Departures[J].friend]
23.             THEN do:
24.                 Invited.DeleteByName[Departures[J].friend]
25.                 IF Uninvited.FindMax  $\neq NIL$ 
26.                     THEN do:
27.                          $K \leftarrow$  UnInvited.FindMax.friend
28.                         Uninvited.DeleteMin
29.                         Invited.Insert(f[K],K)
30.                 ELSE Uninvited.DeleteByName[Departures[J].friend]
31.         J++
32.     If  $Invited.Total > MostFun$  THEN  $MostFun \leftarrow Invited.Total$ ;
 $BestTime \leftarrow T$ 
33. Scan through the list of friends and construct an array of pairs
 $(f(I), I)$  where  $a[I] \leq BestTime \leq d[I] - 2$ . Sort this list by first
coordinate, and return  $T$  and the largest  $k$  elements. This is the start
time and guest list for the best party.

```

While this is a bit complicated, if we take an amortized view point, we see that each I is added to Available once (when it is the I in the first inside while loop) and is deleted from Available once (when it is the J in the second inside while loop). We can see this, since I and J are incremented each loop, and never decremented, so they can take on each value only once. So each of these inside while loops get performed at most n times. The maximum cost for any operation inside these loops is $O(\log n)$, so they both have time $O(n \log n)$. This matches the preprocessing time for the two heapsorts, and the final sort in the last step, and dominates the other initialization times. So the total time is $O(n \log n)$.

Implementation-20 points Implement bubble-sort and heap-sort. You can use heaps from a standard library to implement heap-sort. Plot their performance on random arrays of n integers with values between 1 and n , for $n = 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$. Plot their performance on a log-log scale. Is heap-sort always better than bubble-sort? Why or why not?

Heap sort will have a smaller slope than bubble sort, but for very small values of n bubble-sort may be faster. You should get lines of different slopes, but they may or may not intersect depending on implementation details. See the webpage for Ari Frank's solution (Ari is a TA from a previous year).

You should use ACTUAL times, not just plot the theoretical bounds on a log-log curve. You can use standard libraries for things like heaps, and for timing programs. If the data aren't smooth, you should probably run the algorithms many times and take an average, rather than just once on a small input.