DESIGN AND ANALYSIS OF ALGORITHMS 2017

MOISE GABRIEL

QUESTION 5 (30 mins)

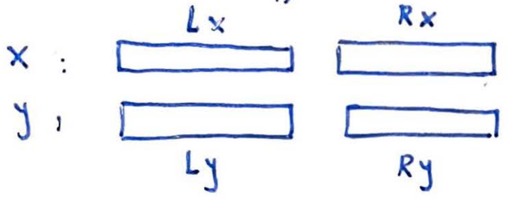
(a) A divide-and-conquer algorithm usually solves a problem by:

- breaking it into smaller instances of it
- solving them recursively, using brute-force when reaching base cases
- combining their answers appropriately

Let $T(n)$ be the time to solve a problem of size n, $D(n)$ the time needed to divide the problem into subproblems, $C(n)$ the time needed to combine their results, we reduce the problem into a subproblems of size n/b. Therefore, the total time is given by

$$T(n) \leq a\,T(\lceil n/b \rceil) + C(n) + D(n)$$

We'll use this approach to multiply two n-bit numbers: let x and y be



$\Rightarrow x = Lx \cdot 2^{\frac{n}{2}} + Rx$

$\Rightarrow y = Ly \cdot 2^{\frac{n}{2}} + Ry$

To multiply x with y we need $(Lx \cdot 2^{\frac{n}{2}} + Rx)(Ly \cdot 2^{\frac{n}{2}} + Ry) = Lx \cdot Ly \cdot 2^n + 2^{\frac{n}{2}}(LxRy + LyRx) + RxRy$. We'll use the following trick: by multiplying Lx with Ly, Rx with Ry and $(Lx+Rx)$ with $(Ly+Ry)$ we can obtain $LxLy$, $RxRy$, $LxRy + RxLy$.

$$LxRy + LyRx = (Lx+Rx)(Ly+Ry) - LxLy - RxRy$$

So, to summarize:

- we break the problem into three smaller subproblems of size $T(\lceil n/2 \rceil)$
- to combine them we need $O(1)$ and to divide also $O(1)$

Therefore, we have:

$$T(n) \leq 3\,T(\lceil n/2 \rceil) + O(1)$$

We'll use the Master Theorem, where $a=3$, $b=2$, $d=0$ $(\log_b a > d) \Rightarrow$

$$\Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 3}).$$

(b) in a "circularly shifted array", the maximum is the only value that is bigger than its neighbours. At each step, we'll split the interval we are working with in half, and there are 3 cases:

- $A[mid-1] \leq A[mid]$ and $A[mid] \geq A[mid+1] \Rightarrow A[mid]$ is the maximum
- $A[mid-1] \geq A[mid]$ and $A[mid] \leq A[mid+1] \Rightarrow A[mid-1]$ is the maximum

• $A[mid-1] \leq A[mid] \leq A[mid+1] \Rightarrow$ the maximum is in $[left..(mid-1)]$ (it wrapped round, so it can't be in the right half

Algorithm:

```
SEARCH (A, left, right)        // Search in A[left..right)
1. if (right - left == 1) return A[left]
2. mid = (left + right) / 2
3. if (A[mid] ≥ A[mid+1]) && (A[mid] ≥ A[mid-1]) return A[mid]
4. if (A[mid] ≤ A[mid+1]) && (A[mid] ≤ A[mid-1]) return A[mid-1]
5. return SEARCH(A, left, mid)
```

(c) First, the Master Theorum:

$$T(n) \leq a\,T(\lceil n/b \rceil) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d) & , \text{if } d > \log_b a \quad (\text{Case } 1) \\ O(n^d \log_b n) & , \text{if } d = \log_b a \quad (\text{Case } 2) \\ O(n^{\log_b a}) & , \text{if } d < \log_b a \quad (\text{Case } 3) \end{cases}$$

• Algorithm A: $T_A(n) \leq 3\,T_A(\lceil 2n/3 \rceil) + O(1)$

$$\left.\begin{matrix} a = 3 \\ b = \frac{3}{2} \\ d = 1 \end{matrix}\right| \Rightarrow \log_b a = \log_{\frac{3}{2}} 3 \; \Big| \Rightarrow \log_b a > d \overset{(3)}{\Rightarrow} T_A(n) = O(n^{\log_{\frac{3}{2}} 3})$$

• Algorithm B: $T_B(n) \leq 9\,T_B(\lceil n/3 \rceil) + O(n^2) + O(n) = 9\,T_B(\lceil n/3 \rceil) + O(n^2)$

$$\left.\begin{matrix} a = 9 \\ b = 3 \\ d = 2 \end{matrix}\right| \Rightarrow \log_b a = \log_3 9 = 2 \; \Big| \Rightarrow \log_b a = d \overset{(2)}{\Rightarrow} T_B(n) = O(n^2 \log_3 n)$$

• Algorithm C: $T_C(n) \leq 3\,T_C(n-1) + O(1)$

Since we can arrange this as

$$T_C(n) \leq 3T_C(n-1) \leq 3^2 T_C(n-2) \leq \dots \leq 3^n \cdot k \Rightarrow T_C(n) = O(3^n)$$

Since algorithm C is the only exponential algorithm, it is the slowest of the three and since $\log_{\frac{3}{2}} 3 = \log_{1.5} 3$ we can compare the complexities of A and B:

$$\frac{n^{\log_{\frac{3}{2}} 3}}{n^2 \log_3 n} = \frac{n^{\log_{1.5} 3}}{n^{\log_{1.5} 2.25} \log_3 n} = \frac{n^{\log_{1.5} \frac{4}{3}}}{\log_3 n} > \frac{n^{\frac{1}{2}}}{\log_3 n} = \frac{\sqrt{n}}{\log_3 n} > 1 \quad \left(\log_{1.5} \tfrac{4}{3} > \tfrac{1}{2} \text{ since } \tfrac{4}{3} > (1.5)^{\frac{1}{2}},\ \tfrac{16}{9} > \tfrac{3}{2}\right)$$

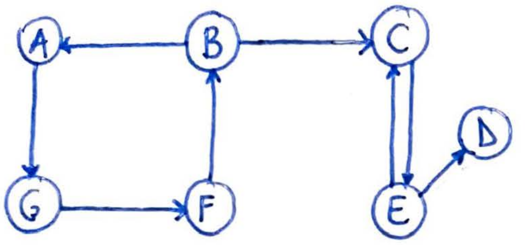
So, we get that $T_A(n) > T_B(n) \Rightarrow B$ is the fastest algorithm.

QUESTION 6 (40 mins)

(a) Let $G = (V, E)$ be a directed graph. A strongly connected component (SCC) of G is a subgraph of G that is strongly connected i.e. there is a path with edges in the subgraph between any to vertices from it.

The SCC graph of G is $G^{scc} = (V^{scc}, E^{scc})$, where V^{scc} is represented by a vertex for each scc of G and $(v_c, v_{c'}) \in E^{scc}$ if there is an edge in G between c and c'

Example: $C_1 = \{A, B, F, G\}$, $C_2 = \{C, E\}$, $C_3 = \{D\}$



The SCC graph is

$$\{A, B, F, G\} \rightarrow \{C, E\} \rightarrow \{D\}$$

(b) To find the SCCs of a graph $G = \{V, E\}$ we do the following

```
SCC(G)
1. Call DFS(G) to compute the finishing times f[v] for all v ∈ V
2. Compute G^T
3. Call DFS(G^T) in order of decreasing finishing times from 1.
4. Output the trees obtained in the DFS forest in the second as separate SCCs
```

The total time need is $\Theta(|V| + |E|)$

- $G = (V, E)$ directed graph, $s \in V$ source vertex, V finite
- "infinite trace" $= \rho = v_0 v_1 v_2 \dots$, $v_0 = s$, $(\forall) i \geq 0$ $(v_i, v_{i+1}) \in E$
- $Inf(\rho) \subseteq V$ = the set of vertices that occur infinitely often in ρ

(c) Let's suppose that $Inf(\rho)$ is not a subset of a single SCC in G. Therefore, there must exist two nodes in $Inf(\rho)$, u, v s.t. there is no path from u to v. Since we arrive at some point to v, we know now that we cannot get to u anymore from then on. However, that contradicts the fact that u must be visited infinitely many times, so we must have a path between u and v. Therefore, $Inf(\rho)$ must be a subset of a single SCC of G.

(d) From (c) we know that

$(\exists)$ ρ s.t. $\inf(\rho)$ implies that $\inf(\rho) \subseteq$ SCC, therefore if there are no SCCs in G, there cannot be any infinite trace in G.

Additionally, if there exists a SCC in G, by going through each vertex of it in turn (we can since there is a path between all nodes) infinitely many times, we form an infinite trace. Therefore, to solve this, we use the SCC algorithm from (b) and if there is a path from s to a SCC of G (that either has at least two vertices, or a vertex with a self-loop) we output YES, otherwise we output NO

<u>Algorithm</u>:

```
1. Call DFS-VISIT(G, s)          // Search for the subgraph reachable from s
2. Let G' = Subgraph formed from the DFS tree from 1.
3. Let SCCs = SCC(G')
4. if there exists a tree in SCCs with a vertex with self-loop / two vertices
5.      return "YES"
6. else return "NO"
```

* <u>QUESTION 7</u> (45 mins - more on last subtask) - wouldn't have chosen it in exam conditions

$G = (V, E, w)$ connected, weighted, undirected graph, $w: E \to \mathbb{R}$

(a) A "spanning tree" of G is a tree (V, T) with $T \subseteq E$ such that all the vertices of G are in the tree i.e. there is a path formed by edges from T between any two vertices. A "minimum spanning tree" (MST) of G is a spanning tree with minimum weight i.e. if T is an MST of G, then $(\forall)$ T' spanning tree of G, we have $w(T) \leq w(T')$, where $w(T)$ = sum of the weights of all the edges from T.

(b) Kruskal's algorithm is based on picking at each step, the edge with the smallest weight that does not form a cycle in the set of edges picked so far. For this we need a disjoint-set data structure that has the following operations:

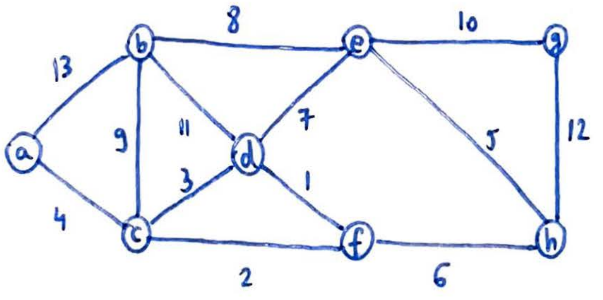
- REPRESENTANT(v) - gives the representative vertex for the vertex-set of v
- UNION(x, y) - Remove S_x and S_y from the set S of vertex-sets and adds the set $S_x \cup S_y$
- MAKE-SET(v) - Makes a new set $\{v\}$ and adds it to S

Algorithm : KRUSKAL (V, E, w)

```
1. A = ∅                          // the edge-set we'll build
2. for each v ∈ V
3.      MAKE-SET(v)
4. Sort E after w increasingly
5. for each edge (u,v) ∈ E (sorted)
6.      if REPRESENTANT(u) ≠ REPRESENTANT(v)
7.      A = A ∪ {(u,v)}
8.      UNION(u,v)
9. return A
```

(c) The key concept of a greedy algorithm is that at each step of the solution, the algorithm makes the choice that has the greatest immediate benefit (greedy choice) and it never reconsiders the choices made so far, Kruskal's algorithm having both of these properties.

(d)



$A = \{(d,f), (c,f), (a,c), (e,h), (f,h), (b,e), (e,g)\} \rightarrow w(A) = 1 + 2 + 4 + 5 + 6 + 8 + 10 = 36$

(e)

A	C ∈ S
∅	{{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}}
{(a,c), (b,e), (c,f), (d,f), (e,h), (e,g)}	{{a,c,d,f}, {b,e,g,h}}
{(a,c), (b,e), (c,f), (d,f), (e,h), (e,g), (f,h)}	{{a,b,c,d,e,f,g,h}}

w(A) = 36, therefore the algorithm gives back an MST for G.

We'll now prove that the algorithm always returns an MST for any graph (V, E, w)
We will use the invariant

$$i: A = \bigcup_{C \in S} T_C, \text{ where } T_C = \text{an MST for } C, \text{ where } C \text{ is a connected component of } S$$

Initialisation: $A = \emptyset$, $S = \{\{v\} \mid v \in V\}$, therefore each $T_C = \emptyset$, so the invariant trivially holds

Termination: $S = \{C\}$, since at each step the union of the sets in S is equal to V (can be an invariant of its own, but it is easy to prove : initially it holds, after each step we just merge the sets together and at the end it is going to be V), we have $S = V \Rightarrow$

$$\Rightarrow A = \bigcup_{C \in V} T_c = T_V = MST(G)$$, so we obtained an MST for G

Maintenance: We currently know that A is a subset of an MST (also can be an invariant of its own, but it is implied by the current one). For each strongly component C, the set $V \setminus C$ is a cut that respects A, because otherwise C would be connected with another component of $S = (V, A)$, so it would be ill-formed. Then, any light edge crossing the cut i.e the edge with the lowest weight $(u,v) \in E$ with $u \in C$ and $v \notin C$ is safe for A (by the Cut Lemma). Therefore, by adding (u,v) to A we connect two connected components of S and the new $S = (V, A)$ will have those two sets of vertices combined (union). Therefore, the invariant is maintained.

The Cut Lemma

Let A be a subset of some MST. If $(S, V \setminus S)$ is a cut that respects A and (u,v) is a light edge crossing the cut, then (u,v) is safe for A (i.e. if we add (u,v) to A, it will form a set A' that is also a subset of some MST).

QUESTION 8 (40 mins)

- score = #matching positions − $\overset{\text{gap penalty}}{g}$ × #number of spaces
- Input: $x[0..m)$, $y[0..n)$ and g
- Output: maximum score over all alignments of x and y

(a) "Dynamic programming" is a technique used for optimization problems. It is based on reduce the problem to smaller instances of it (with a constant factor), obtaining and saving their optimal solutions and building the optimal solution to the big problem based on them

The "principle of optimality" states that whatever the choices made so far about the ini
on the intermediate states, the current choice must be itself optimal with respect to the current state.

The global alignment problem can be solved using dynamic programming since we can build a solution of any state based on the optimal solutions of the previous states: if we want to get the best score of $x[0..i)$ and $y[0..j)$, then every score of $x[0..i')$ and $y[0..j')$ must have been built optimally, otherwise we could swap the solutions with the optimal ones and get a better score.

(b) Let $s(i,j)$ = maximum score over all possible alignments of $x[0..i)$ and $y[0..j)$

Then, we get the following relation:

$$s(i+1,j+1) = \text{if } \delta(i+1,j+1) \text{ then } s(i,j)+1 \quad \longleftarrow \text{get (+1) score and go to } x[0..i), y[0..j)$$

$$\text{else } \underset{\substack{\uparrow \\ \text{get the best outcome of using a } \square \text{ on} \\ x[i+1] \text{ or } y[j+1]}}{\max\{s(i,j+1), s(i+1,j)\} - g}$$

where $\delta(i,j) = 1$ if $x[i] = y[j]$ and 0, otherwise

Also, $s(i,0) = -i \times g$, $0 \le i < m$

$s(0,j) = -j \times g$, $0 \le j < n$

We'll first complete the base cases, then we'll go through the ranges $0 \le i < m$ and $0 \le j < n$ after $k = (i+j)$, starting from 2 until $m+n$, i starting from 1 until m and $j = k - i$. (check $j > 0$)

The result will be in $s(m,n)$ and the time complexity is $O(mn)$ as we fill up a table of size $m \times n$.

(c) In addition the algorithm above, we can use a table $\Delta[i,j]$ that would indicate what solution we used at step (i,j) from the three posibilities. Then, we'll have

$$\Delta[i+1,j+1] = \begin{cases} (i,j) & \text{if } \delta(i+1,j+1) = 1 \\ (i,j+1) & \text{if } \delta(i+1,j+1) = 0 \text{ and } s(i,j+1) \ge s(i+1,j) \\ (i+1,j) & \text{if } \delta(i+1,j+1) = 0 \text{ and } s(i,j+1) < s(i+1,j) \end{cases}$$

To output the optimal alignment, we start from (m,n) and go through the table until we get to a dimension being 0 and from there we output all the remaining line or column until we get to $(0,0)$.

• local alignment problem: given $x[0..m)$, $y[0..n)$, g, output the maximum score over all alignments x', y' substrings of x and y, respectively.

(d) We can use the algorithm above to calculate all the best scores over all combinations of (x', y'), which are $\frac{m(m+1)}{2}$ and $\frac{n(n+1)}{2}$ $\Rightarrow O(m^2 n^2)$ and for each we would need $O(mn)$ time $\Rightarrow$ an upper bound for this approach would be $O(m^3 n^3)$.

(e) Let $s(i,j)$ = maximum score over all possible alignments of x', y' where x' and y' are suffixes of $x[0..i)$ and $y[0..j)$

Then, we have

$$s(i+1, j+1) = \text{if } \delta(i+1, j+1) \text{ then } 1 + s(i,j) \longleftarrow \text{if they are the same, add 1 to the previous best solution}$$

$$\text{else } \max\{\underset{\substack{\uparrow \\ \text{empty suffixes}}}{0}, \underbrace{s(i,j+1)-g, \; s(i+1,j)-g}_{\text{same as before}}\}$$

The final answer is the maximum value that appears in the table.
The total running time is $O(mn)$.