1 Homework 8: Due on Dec 9, 2023

This is another unusual homework – it won’t be graded. It is really a set of study questions to help you think about topics that I might test you on in the final exam. Try to prove every statement – ask me if any assumption are unclear.

Q1. Consider the single-pair shortest path problem (§XIV.4) with input $G = (V, E; C, s, t)$ where $C$ is non-negative. Given any heuristic function $h : V \to \mathbb{R}$, we have the following algorithm, denoted “$A^*[h]”$: it amounts to running Dijkstra’s algorithm using the modified cost function $C^h(i, j) = C(i, j) − h(i) + h(j)$. Moreover, we assume that the main loop of Dijkstra’s algorithm terminates once $t$ has been removed from the priority queue. REMARK: In Class, we call $A^*[h]$ the “Simplified A* Algorithm” because the algorithm in the solution to part(ii) will be called the “A* Algorithm”.

- (i) What is an “ideal” heuristic function for $A^*$? What happens to $A^*$ with such a function?

   The ideal heuristic function is $h(v) = \delta_v(t)$ where $t$ is the target or goal. The sequence of nodes we add to $S$ would form a shortest path from $s$ to $t$.

- (ii) Suppose $h$ is arbitrary; in particular, it may not be admissible. Can you implement $A^*$ so that it always terminate correctly? i.e., on termination, $d[t] = \delta(s, t)$? Justify your answer.

   The main observation is this: the modified cost function
   \[ C^h(i, j) = C(i, j) − h(i) + h(j) \]
   may now be negative, but it cannot create any new negative cycles. Therefore we can indeed modify Dijkstra’s algorithm to make it work: In Dijkstra, each node is deleted from the queue at most once. Now we may have to send a node back to the queue after it is deleted. The solution is this: if $u^*$ has just been removed from the queue, as usual, we relax the constraints of edges emanating from $u^*$, i.e., for all $v$ adjacent to $u^*$, we relax $d[v] \leftarrow \min\{d[v], d[u^*] + C(u^*, v)\}$. The difference is that, in the original Dijkstra, we do not have to do this relaxation if $v$ is already in $S$. Moreover, if $d[v]$ is decreased by this relaxation, there are 2 cases:
   - If $v$ is not in the queue, we must (re)insert $v$ into the queue with priority equal to $d[v] + h(v)$. (In this case, we say $v$ is “ejected” from $S$.)
   - If $v$ is already in the queue, we must do DecreaseKey$(v, d[v] + h(v))$.

   You should be able to show termination in this modification. It is not true (see below) that upon termination, the algorithm is correct. But if $h$ is feasible, i.e., $h(i) ≤ \delta(i, t)$, then this algorithm is correct. REMARK: Henceforth, let “A* Algorithm” refer to the algorithm described here. Food for thought: can you modify this A* Algorithm so that it will be correct even for non-admissible $h$?

- (iii) Suppose $h$ satisfies $h(t) = 0$ and $h$ is feasible (i.e., $C^h$ is non-negative). Prove that $h(i)$ is a lower bound on $\delta(i, t)$ (i.e., admissible).

   Since $h$ is feasible, we have $C^h(i, j) = C(i, j) − h(i) + h(j) ≥ 0$. In particular, $\delta^h(v, t) ≥ 0$ since $\delta^h(v, t)$ is a sum of such $C^h(i, j)$’s. But $\delta^h(v, t) = \delta(v, t) − h(v) + h(t) = \delta(v, t) − h(v)$ because $h(t) = 0$. Thus $\delta(v, t) ≥ h(v)$, proving that $h$ is admissible.

- (iv) Does the feasibility\(^1\) of $h$ imply that $A^*[h]$ is correct?

   Recall $A^*[h]$ refers to the Simplified A* algorithm. Unfortunately, feasibility of $h$ does not imply the correctness of $A^*[h]$. You should be able to construct a simple counter example.

---

\(^1\) Typo: in the original question, I said “admissibility” instead of feasibility. It is clearly a typo because we know that feasibility is exactly what Dijkstra need: $C^h(i, j) ≥ 0$. So trivially, $A^*[h]$ is correct.
• (v) In what sense can we assume “without loss of generality” that \( h(s) = h(t) = 0 \) in an \( A^*[h] \) algorithm?

Given any \( h \), we first modify it to \( h'(i) := h(i) - h(s) \) for all \( i \). The algorithms \( A[h'] \) and \( A[h] \) operate in exactly the same way, and moreover, we have \( h'(s) = 0 \). Next, define \( h''(i) = h'(i) \) if \( i \neq t \) and \( h''(t) = 0 \). Note that \( A^*[h''] \) does not do any worst than \( A^*[h'] \). Therefore, the properties \( h(s) = h(t) = 0 \) is “without loss of generality”.

• (vi) Suppose \( h, h' \) are two feasible heuristic functions satisfying the property in (v). Suppose \( h(i) \geq h'(i) \) for all \( i \). Show that the stable set \( S \) visited by \( A^*[h] \) is a subset of the stable set \( S' \) visited by \( A^*[h'] \). Explain why it is necessary to assume that in our priority queue, ties are broken consistently.

Let \( S_r := \{ i \in V : \delta(s, i) + h(i) \leq r \} \) and \( S'_r := \{ i \in V : \delta(s, i) + h'(i) \leq r \} \).

Think of \( S_r \) (\( S'_r \)) as the ball of radius \( r \) centered at \( s \). Basically, Dijkstra’s algorithm is searching in balls with increasing radii.

From the definition we see that \( S_r \subseteq S'_r \). Suppose \( r = \delta(s, t) \). At termination, the stable set in \( A^*[h] \) is \( S_r \) and the stable set in \( A^*[h'] \) is \( S'_r \). Here we are exploiting the fact that \( h(t) = h'(t) = 0 \) (as suggested by part (v)).

This almost seems to be what we need to prove. But what can go wrong? Suppose there are vertices \( i \) different from \( t \) that satisfies \( \delta(s, i) = \delta(s, t) \). But if ties are broken consistently, then what we claim is true. Food for thought: how should we choose the tie breaking rule?

Q2. Please refer to the special handout on the problem of root isolation.

• (i) Do Exercises L1, L2, L3

Remark: L3 shows that two real roots of an integer polynomial can be as close as \( 2^{-\tau} \) where the polynomial coefficients has at most \( \tau \) binary bits. Using the theory of resultants, one can show that this bound is tight.

L1: Example where EVAL terminate on a a function with multiple roots in \( I_0 = [-1, 1] \): Assume that \( \square f(I) \) is the “ideal function”, namely, \( \square f(I) = f(I) \). This is wildly unrealistic, of course. Moreover, we assume that \( I \) can be open or half-open, and when we detected that \( \{ m \} \) is a root, we will put the half-open intervals \([a, m] \) and \((m, b]\) into the queue. If the original interval \( I \) is not a closed interval (say \( I = [a, b) \) or \( I = (a, b] \), etc), we modify the \([a, m] \) and \((m, b]\) accordingly. It is now clear that this ideal EVAL algorithm halts if we choose \( f(x) = x^2 \).

The multiple root \( x = 0 \) will be detected at the first split of \( I_0 = [-1, 1] \), and the ideal function will see that there are no roots in the subintervals \([-1, 0) \) and \((0, 1] \). For the second example of non-termination: any function with multiple roots in \( I_0 \) will do. Assume the box functions are only evaluated on closed intervals, and our subdivision always produce closed intervals. Clearly, the \( 0 \notin \square f(I) \) and \( C_1 \) will never never halt and \( 0 \notin \square f'(I) \) will never hold for any interval that contains 0. The box functions \( \square f(I) \) and \( \square f'(I) \). For instance,

L2: I think the descriptions and proofs preceding L2 are too compressed for you to be able to make much sense of the arguments. If you are interested, ask me.
L3: $f(x) = x^n - (ax - 1)^2$ is actually the famous Mignotte polynomials that contain two real roots very close to $1/a$. We will assume $a \geq 4$ and $n \geq 4$ in the following. Following the hint to use Rouché’s theorem, with $f(x) = g(x) + h(x)$ where $g(x) = -(ax - 1)^2$ and $h(x) = x^n$. Let us consider the disc $\Delta$ of radius $\epsilon$ centered at $1/a$. Clearly, $g(x)$ has a double root inside $\Delta$. To prove that $f(x)$ has the same number of roots as $g(x)$ inside $\Delta$, Rouché says we must verify that

$$|g(z)| > |f(z) - g(z)| = h(z)$$

for every point $z$ on the boundary of $\Delta$. These boundary points have the form

$$z = (1/a) + \epsilon \cdot e^{i\theta}$$

for $\theta \in [0, 2\pi)$ where (recall) $e^{i\theta} = \cos \theta + i \sin \theta$ (and $i = \sqrt{-1}$). We will choose $\epsilon$ very small – again following the hint in the text, we want $\log \epsilon = -\Omega(n \log a)$. You might try begin with $\epsilon = (1/a)^n$. You will fail below. After some trial and error, you decide $\epsilon = 4/a^{1+(n/2)}$ is good. Let us check that

$$|h(z)| = \left| (1/a) + \epsilon e^{i\theta} \right|^n \leq |(1/a) + \epsilon|^n = \left( (1/a) + (4/a^{1+(n/2)}) \right)^n \leq \frac{1}{a^n} \left( 1 + \frac{4}{a^{n/2}} \right)^n \leq \frac{1}{a^n} e^{4n/a^{n/2}}.$$  

Thus $|h(z)| < |g(z)|$.

Are we there yet? Now, we check that

$$f(1/a + \epsilon) < 0, \quad f(1/a) > 0, \quad f(1/a - \epsilon) < 0.$$  

Thus the two roots of $f$ in $\Delta$ are real.

• (ii) (Continuous Amortization) Let $C(\cdot)$ be an interval predicate, i.e., for any real interval $I = [a, b]$, $C(I)$ that returns either true or false. E.g., $C(I)$ return true iff $I$ contains at most one real root of some function $f : \mathbb{R} \to \mathbb{R}$. Such predicates can be constructed from Sturm sequences, Descartes’ method or EVAL type algorithms. Consider a subdivision algorithm called $SubDiv$ (or $SubDiv_C$) that, for any input $I_0$, will keep subdividing $I_0$ and its children until every interval satisfies $C(I)$. The subdivision is based on bisection, so these intervals form a full binary tree $T(I_0)$ rooted at $I_0$, and the set of intervals at the leaves forms a “subdivision of $I_0$”. We want to bound the size of $T(I_0)$.

Let $G : \mathbb{R} \to \mathbb{R}_{\geq 0}$ be any function. Recall that an interval $I = [a, b]$ is $G$-small if there exists an $x' \in I$ such that $w(I)G(x) \leq 1$. Note that $w(I) := b - a$ is the width of $I$. We call $G(x)$ a stopping function for $C$ if for all $I$, if $I$ is $G$-small, then $C(I)$ holds. Prove that if $G$ is a stopping function for $C$, then the size of $T(I_0)$ is bounded by $1 + \int_{I_0} 4G(x)dx$. 


Q3. Consider universal hashing and the FKS Scheme. Let
\[ H \]
\[ \mu \]
\[ If \]
\[ \rho \]
\[ For \]
\[ \rho \]
\[ H \]
\[ We also say \]
\[ \{g \circ h : h \in H\} \] denote the weighted set
\[ \{g \circ h : h \in H\} \] of all \( x \in I \). Thus integral \( \int_I G(x)dx \geq \int_I \frac{1}{2w(I)} \geq 1/2 \). Therefore
\[ \int_{I_0} G(x)dx = \sum_i \int_I G(x)dx \geq \sum_i 1/2 \]
where \( I \) ranges over all leaves in \( T(I_0) \). Thus \( \int_{I_0} G(x) \) is at least half of the number of leaves in \( T(I_0) \). Thus \( 2\int_{I_0} G(x) \) is at the number of leaves. Thus \( 4\int_{I_0} G(x) \) is an upper bound on the size of \( T(I_0) \).

- (iii) Let \( f \) be the polynomial \( f(z) = a_0 \prod_{i=1}^{n} (z-a_i) \) where \( a_i \in \mathbb{C} \) are complex numbers. We assume they are distinct for this discussion. If \( I = [a, b] \) let \( m_I = m(I) := (a+b)/2 \) and \( r_I = r(I) = (b-a)/2 \). Define \( C(I) \) as follows:
\[ C(I) = |f(m_I)| > \sum_{i=1}^{n} \left| f^{(i)}(m_I) \right| \]
where \( f^{(i)} \) is the \( i \)-th derivative of \( f \). Prove that if \( C(I) \) holds, then \( f \) has no roots in \( I \). HINT: use Taylor expansion of \( f(x) \) at \( x = m_I \).

- (iv) Then \( f'(z)/f(z) = \sum_{i=1}^{n} \frac{1}{z-a_i} \). Show that the function
\[ G(x) = \sum_{i=1}^{n} \frac{3}{2|x-a_i|} \]
is a stopping function for \( C \) from (ii). REMARK: we can bound \( \int_{I_0} G(x)dx = O(n(\tau + \log n)) \) if \( I_0 = [-2^\tau, 2^\tau] \) (which contains all real roots of \( f \).

Q3. Consider universal hashing and the FKS Scheme. Let \( H \subseteq [U \to Z_v] \) be any set of functions. View \( H \) as a weighted set where \( \mu(S) \geq 0 \) denotes the weight of any \( S \subseteq H \). Suppose \( g : Z_v \to Z_m \) where \( g(x) = x \mod m \). Then we let
\[ (g \circ H) \subseteq [U \to Z_m] \]
denote the weighted set \( \{g \circ h : h \in H\} \) where \( g \circ h \) denotes function composition. The weight of \( f \in g \circ H \) is defined by \( \mu(f) = \mu \{h \in H : g \circ h = f\} \).

For \( \rho \geq 1 \), we say \( H \) is \( \rho \)-universal if for all \( \{x, y\} \in \binom{V}{2} \) we have
\[ \mu \{h \in H : h(x_1) = h(x_2)\} \leq \rho \frac{\mu(H)}{|V|}. \]
We also say \( H \) is \( \rho \)-independent if for all \( \{x, y\} \in \binom{V}{2} \) and all \( y_1, y_2 \in V = Z_v \)
\[ \mu \{h \in H : h(x_1, x_2) = (y_1, y_2)\} \leq \rho \frac{\mu(H)}{|V|^2}. \]
If \( \rho = 1 \), we simply say “universal” or “pairwise independent”.

- (i) Prove that if \( H \) is pairwise \( \rho \)-independent then it is \( \rho \)-universal.

- (ii) If \( H \) is pairwise \( \rho \)-independent, then \( g \circ H \) is pairwise \( (\rho(1 + \frac{m}{v})^2) \)-independent. Write \( \sigma_m \) for \( \rho(1 + \frac{m}{v})^2 \). Note that \( m \) is expected to vary, but \( \rho \) and \( v \) are fixed in the FKS Scheme.
\( \bullet \) (iii) Suppose \( H \) is \( \rho \)-universal, and \( K \subseteq U \), \( n = |K| \). Write \( b_h(i; K) \) for the size of the (unweighted) set \( h^{-1}(i) \cap K \). Then
\[
\sum_{h \in H} \mu(h) \sum_{i \in V} \left( \frac{b_h(i; K)}{2} \right) \leq \left( \frac{n}{2} \right) \frac{\mu(H)}{v} \rho. \tag{2}
\]
NOTE: this is just the weighted version of a lemma proved in the Lectures. Call the quantity
\[
X(h) := \sum_{i \in V} \left( \frac{b_h(i; K)}{2} \right)
\]
the “total conflict” of \( K \) under \( h \). If \( h \) is a uniformly random function in \( H \), then \( X(h) \) is a random variable. Then the expected value \( E[X(h)] \) is at most \( \left( \frac{n}{2} \right) \frac{1}{m} \sigma_m \).

\( \bullet \) (iv) If \( m \leq v \) then there exists \( h \in g \circ H \) such that
\[
X(h) \leq \left( \frac{n}{2} \right) \frac{1}{m} \sigma_m
\]
where \( \sigma_m \) is the constant in (iii).

\( \bullet \) (v) In the original FKS, the universe is \( U = \mathbb{Z}_p \). E.g., in homework 7, with keys from our dataset that are ASCII strings of length \( r = 20 \), you need \( p > 2^{8r} \). To avoid such large primes, we want to use \( U = \mathbb{Z}_p^r \) and employ functions of the form:
\[
h_a(x) = (a_0 + \sum_{i=1}^r a_i x_i) \mod p
\]
where \( x = (x_1, \ldots, x_r) \in \mathbb{Z}_p^r \) and \( a = (a_0, a_1, \ldots, a_r) \in \mathbb{Z}_p^{r+1} \). Show that the set
\[
H^r_p = \{ h_a : a \in \mathbb{Z}_p^{r+1} \} \subseteq [\mathbb{Z}_p^r \rightarrow \mathbb{Z}_p] \tag{3}
\]
is pairwise 1-independent. Conclude that you could use the family (3) in the FKS construction.

\( \bullet \) (vi) Suppose \( U \subseteq 2^N \) is given (e.g., \( N = 8r \) as in homework 7). Given \( K \subseteq U \), how do you choose \( p \) and \( r \) in the (3) so that we can use \( H^r_p \) in the FKS construction? Assume that you would like to use “byte size” data whenever possible (a byte is 8-bits). Illustrate this from our problem in Homework 7, where the set of keys \( K \) in the file `small.txt` has size 1000, and \( N = 8r = 160 \).

Choose \( p \) to be a prime not much larger than \( |K| \). Choose the smallest \( k \) such that \( p \) fits into \( k \) bytes (\( p \leq 2^{8k} \)). Then \( r = \lceil N/8k \rceil \).

In our example, \( U = 2^N \) where \( N = 160 \), and \( |K| = 1000 \). So we can pick \( p = 1009 \). Clearly \( p \) fits into \( 2^{10} \) bits, and thus \( k = 2 \). Then \( r = \lceil N/8k \rceil = \lceil 160/16 \rceil = 10 \).
2 Homework 7: Due on Thu Dec 6

This is an unusual homework set – you need to do some programming in your programming language of choice. Note that scripting languages are fine. When you submit your programs, you need to include a Makefile in which I can type ”make” to compile your programs. If I type ”make help”, it should show me what to do to run the various programs described below. You can work in groups of two or three. I will be available for consultation throughout.

Q1. (3+3+1 Points) Consider arithmetic in $GF(2^n)$, viewed as arithmetic on polynomials in $GF(2)[X]$ modulo some irreducible polynomial $I(X)$ of degree $n$. View a vector $(a_k, a_{k−1}, \ldots, a_0)$ as the polynomial $\sum_{i=0}^{k} a_i X^i$. NOTE: All the calculation below are to be done by hand (we want you to internalize these algorithms). So it is important to explain and organize your hand-calculation systematically. (You may have to do these calculations in our exams.) Also, to save time, I do not recommend using latex – just write neatly with a pen and scan it – e.g., Bobst Library basement has scanners.

(a) Construct the multiplication table for $GF(8)$, using the irreducible polynomial $I(X) = X^3 + X + 1$.

(b) Compute the inverses of non-zero elements of $GF(8)$ modulo $I(X)$. Note that we want you to organize your hand calculations and display them for us to check. The point is to internalize the algorithm. Of course, you should check that what you compute agrees with your table in part (a).

NOTE: We described the extended Euclidean algorithm in class for computing inverses: I will briefly review it here as some of you missed that lecture. To compute the inverse of a polynomial $P(X)$, the idea is to compute a sequence of triples

$$(T_0, T_1, \ldots, T_k)$$

where $T_0 := (I(X), 1, 0)$, $T_1 := (P(X), 0, 1)$, and for $i \geq 1$,

$$T_{i+1} = (a_{i+1}, s_{i+1}, t_{i+1}) := T_{i−1} − q_i T_i.$$  \hspace{1cm} (4)

Here, $q_i$ is the quotient of $a_{i−1}$ divided by $a_i$ (remember your High-School polynomial division), and the vector operations in (4) are defined via the corresponding component-wise operations. The sequence terminates at $k \geq 1$ where $a_k = 1$ (the GCD of $P(X)$ and $I(X)$). The following invariant is easy to verify:

$$a_i = s_i I(X) + t_i P(X).$$

This shows that the inverse of $P(X)$ mod $I(X)$ is given by $t_k$.

(c) Combine the results from (a) and (b) to produce a division table for $GF(8)$. Open ended: outline a more direct method to produce the division table.

SOLUTION: (a,b)

Table 1: Multiplication and Inverse Tables for $GF(8)$ modulo $X^3 + X + 1$

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>010</td>
<td>100</td>
<td>110</td>
<td>010</td>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>011</td>
<td>110</td>
<td>101</td>
<td>111</td>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>100</td>
<td>011</td>
<td>111</td>
<td>100</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>101</td>
<td>010</td>
<td>100</td>
<td>101</td>
<td>010</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td>110</td>
<td>111</td>
<td>001</td>
<td>110</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td>111</td>
<td>101</td>
<td>010</td>
<td>111</td>
<td>100</td>
<td>001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x · y</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>010</td>
<td>100</td>
<td>110</td>
<td>010</td>
<td>011</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>011</td>
<td>110</td>
<td>101</td>
<td>111</td>
<td>001</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>x⁻¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>—</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>100</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>011</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

SOLUTION: (c) Let $M[x][y]$ be the entry giving the product of $x, y \in GF(8)$, and $Inv[x]$ gives the multiplicative inverse of $x$. Thus, we can obtain $x/y$ as $M[x][Inv[y]]$. So the division table can simply obtained by permuting the columns of $M$ according to $Inv$. 

Comments:
Q2. (10 Points) Let \( n, d \) be given. We want to construct a sample space \( \Omega \) of size \( |\Omega| = 2^N \) with the uniform discrete probability function satisfying:

(A) \( \Omega \) is small in the sense that \( N = O(d \log n) \).

(B) There is Boolean matrix \( X \) of size \( |\Omega| \times n \) such that for any \( a \ (1 \leq a \leq d) \), if \( J \subseteq \{1, \ldots, n\} \) has size \( a \) then the submatrix \( X_J \) comprising the columns \( \{X_j : j \in J\} \) has this property:

For any \( r \in \{0, 1\}^a \), there are exactly \( 2^{N-a} \) rows of \( X_J \) that are equal to \( r \).  

(a) Show how the ability to construct such a sample space can be used to derandomize (i.e., make deterministic) the randomized algorithm such as our Monte Carlo algorithm for finding “good” \( 2 \)-colorings of \( K_m \). Here, “good” means there are at most \( \frac{d}{4} \binom{n}{3} \) monochromatic triangles. What is the complexity of the deterministic algorithm? REMARK: in retrospect, I should have asked defined “good”  

(b) The construction of \( X \) goes as follows: wlog, let \( n + 1 \) be a power of 2 (say \( k = \lg(n + 1) \)) and \( d \) is odd (say \( d = 2t + 1 \)). Let \( x_1, \ldots, x_n \) be the non-zero elements of \( GF(2^k) \). Consider the matrix

\[
H := \begin{bmatrix}
1 & 1 & \cdots & 1 \\
x_1 & x_2 & \cdots & x_n \\
x_1^3 & x_2^3 & \cdots & x_n^3 \\
\vdots & \vdots & \ddots & \vdots \\
x_1^{2t-1} & x_2^{2t-1} & \cdots & x_n^{2t-1}
\end{bmatrix}
\]  

Notice the powers \( x_i^j \) of the \( x_i \)'s are odd \((i = 1, 3, 5, \ldots, 2t-1)\). This looks like a \((1+t) \times n\) matrix with entries in \( GF(2^k) \). But we actually think of \( H \) as Boolean matrix of size \((1+t)k \times n\), by interpreting each \( x_i^j \) as a column of \( k \) bits. Let \( \Omega = \{1, \ldots, 2^{1+tk}\} \) where each \( \omega \in \Omega \) is regarded as selecting a subset of the rows of \( H \). The rows of the matrix \( X \) are indexed by \( \omega \in \Omega \). Let \( X[\omega] \) denote the \( \omega \)-th row of \( X \). Then \( X[\omega] \) is defined as the componentwise sum (over \( GF(2) \)) of the rows of \( H \) that are selected by \( \omega \). E.g., if \( \omega \) selects the 2nd, 7th and 21st rows of \( H \), then \( X[\omega] = H[2] + H[7] + H[21] \) (componentwise addition in \( GF(2) \)). FACT: this matrix \( X \) has the desired properties (A) and (B) above.

We want you to program up such a sample for \( n = 200 \) and \( d = 3 \). Use your favorite programming language, and work in pairs. The program does not have to be efficient, just has to be correct.

(c1) Use your program in part(b) to implement a Las Vegas algorithm for computing a good coloring. Run your algorithm for the graphs \( K_m \) where \( m = 5, \ldots, 20 \). Report on the running times for each \( m \), and report the number of monochromatic triangles in the 2-coloring you found.

(c2) Same as (c1) but we want a Monte Carlo algorithm. The probability of success should be at least 0.01.

(c3) Same as (c1) but now do a deterministic search through the sample space \( \Omega \) to find the best 2-coloring of \( K_m \) (\( m = 5, \ldots, 20 \)).

SOLUTION: (a) The \( j \)-th column \( X_j \) of \( X \) can be viewed as a Bernoulli random variable. If the rows are labeled by elements \( \omega \in \Omega \), then \( X_j(\omega) \) is just the \((\omega,j)\)-th entry of \( X \). We can now search through each sample point \( \omega \). For a 2-coloring of \( K_m \), we must associate each edge of \( K_m \) with some r.v. \( X_j \) (we need at least \( \binom{m}{2} \) r.v.'s, so \( n > \binom{m}{2} \)). For each \( \omega \), the values of r.v.'s are known, so the coloring of \( K_m \) is fixed. It is easy to decide if this leads to a good coloring. We try all possible choices of \( \omega \). But what is \( d \) in our construction of \( X \)? For monochromatic triangles, you only need \( d = 3 \) for 3-wise independent r.v.s.  

(b) Since \( d = 3 \) and \( d = 2t - 1 \), this means that our matrix \( H \) for this problem has only two rows! So this construction is really easy.

(c) Note that we said \( n = 200 \) because when \( m = 20 \), you need \( \binom{20}{2} = 190 \) r.v.s. Since \( n + 1 \) is a power of 2 for our construction, it means you should pick \( n = 255 \).

Comments:
Q3. (12 Points) We want to implement optimal static hashing. Given a set \( K \subseteq U \), construct a hash table that uses \( O(|K|) \) space and \( O(1) \) worst case time to LookUp any \( x \in U \) (to see if \( x \in K \)). Of course, we expect you to use the FKS Scheme.

The data set is found in the Assignments page. There is a large and a small one: we suggest doing this homework on the small set. The data set is in an ASCII file, where each line contains a keyword, followed by some associated data (don’t worry what the data means). These keywords are actually from a simplified English dictionary. The problem is this: Given any key \( x \in U \), your data structure should either tell us that \( x \) is not a keyword, or else retrieve the associated data.

Write a short report on your implementation, including experimental results. Describe your design for solving this problem (it should be specific to the data set). You need to pick various parameters: what should \( U \) be, how to the prime \( p \), how to find the primary hash function and all the secondary hash functions, etc.

SOLUTION: There are important pragmatic choices you need to consider in implementing FKS. These are data-dependent—that is why we gave you a specific data set. Let us assume that \( U = \mathbb{Z}_p \) for some prime \( p \) and \( H \subseteq [U \to U] \). What should \( p \) be?

Looking at the keys in our data set, we see that each key can be of length up to 20. So we can view them as ASCII strings of length 20, and we can pick any \( p > 2^{52} \cdot 2^{20} = 2^{160} \). You can get away with \( p > 2^{120} \) by using the fact that the characters in our keys involve the 52 latin alphabet (caps and small) and a few special characters (space, period, hyphen, apostrophe). I expect you to find such a prime from the internet table.

If you want a smaller \( U \), you can try to pick one that is not much larger than \( |K| \). You can use the theory in the lecture notes, and search for a mapping from \( 2^{160} \) into \( \mathbb{Z}_p \).

Instead of using the hash functions \( h_a(x) = ax \mod p (a \in \mathbb{Z}_p \setminus \{0\}) \), you can consider other families mentioned in the lecture notes.

The rest of the implementation can follows the general theory directly. There are various optimizations to do: bins of sizes 0 or 1 can be specially encoded.

Comments:

3 Homework 6: Due on Mon Nov 19

Q1. (6 Points) (Elias) Let \( \text{bin}(n) \) denote the standard binary encoding of \( n \in \mathbb{N} \) and \( \text{len}(n) \) be the length of this encoding. E.g., \( \text{bin}(0, 1, 2, 3, 4) = (\epsilon, 1, 10, 11, 100) \) and \( \text{len}(0, 1, 2, 3, 4) = (0, 1, 2, 2, 3) \). Here, \( \epsilon \) is the empty string and we use the compact notation \( \text{bin}(n_1, n_2, \ldots, n_k) := (\text{bin}(n_1), \text{bin}(n_2), \ldots, \text{bin}(n_k)) \), etc.

We want a binary encoding of natural numbers, \( \text{rep} : \mathbb{N} \to \{0, 1\}^* \), with the following property: if \( n_1, n_2, \ldots \) is any sequence of natural numbers, the binary string \( \text{rep}(n_1)\text{rep}(n_2) \cdots \) is uniquely decodable. Such an encoding \( \text{rep}(n) \) is self-limiting in the sense that whenever we know the start of \( \text{rep}(n) \), we can also determine where it ends. Alternatively, the representation is prefix-free: if \( n \neq n' \) then \( \text{rep}(n) \) is not a prefix of \( \text{rep}(n') \).

(a) Consider the following encoding scheme \( \text{rep}_1 : \mathbb{N} \to \{0, 1\}^* \) for natural numbers: \( \text{rep}_1(0) = 1 \) and for \( n \geq 1, \text{rep}_1(n) = 0^{\text{len}(n)}\text{bin}(n) \). E.g., \( \text{rep}_1(0, 1, 2, 3, 4) = (1, 0'1, 0'0'10, 0'0'11, 0'0'0'100) \). Note we use the prime mark (‘) for decoration. What is \( \text{rep}_1(99) \)? What is the length of \( \text{rep}_1(n) \) as a function of \( n \)?

(b) Now consider \( \text{rep}_2 : \mathbb{N} \to \{0, 1\}^* \) where \( \text{rep}_2(n) = \text{rep}_1(\text{len}(n))\text{bin}(n) \) for \( n \geq 0 \). E.g., \( \text{rep}_2(0, 1, 2, 3, 4) = (1, 0'1, 0'0'10, 0'0'11, 0'0'11) \). What is \( \text{rep}_2(99) \)? What is the length of \( \text{rep}_2(n) \) as a function of \( n \)?

(c) What is wrong with the suggestion to define \( \text{rep}(n) \) recursively as follows:

\[
\text{rep}(n) = \text{rep}(\text{len}(n))\text{bin}(n)
\]

for all \( n \geq n_0 \) (for some \( n_0 \))? How can you fix this issue? How small can \( n_0 \) be, and what can you do for \( n < n_0 \)? What is the length of your representation as a function of \( n \)? What is your representation of 99? For what values of \( n \) will your representation be shorter than \( \text{rep}_2(n) \)?
SOLUTION: (a) \( \text{bin}(99) = 1100011 \) so \( \text{len}(99) = 7 \). So \( \text{rep}_1(99) = 0000000'1100011 \).

The length of \( \text{rep}_1(n) = 2 \lg n + O(1) \).

(b) \( \text{bin}(7) = 111 \), so \( \text{rep}_1(7) = 000111 \). So \( \text{rep}_2(99) = 000111'1100011 \) (saves one bit over scheme \( \text{rep}_1(99) \)).

(c) The problem is that \( \text{rep}(n) = \text{rep}(\text{len}(n)) \text{bin}(n) \) is no longer prefix-free. So it is not self-limiting (you do not know when you have reached the end of the representation).

Suppose \( D(n) \) is the “depth” or number of times that \( \text{rep}(n) \) calls itself recursively. Then the solution is to define

\[
\text{rep}_*(n) := \text{rep}_0(D(n))\text{rep}(n)
\]

where \( \text{rep}_0(n) \) the following self-limiting representation of \( n \),

\[
\text{rep}_0(n) := 0^{n-1}_1.
\]

Of course, we could have used \( \text{rep}_1 \) or \( \text{rep}_2 \) instead of \( \text{rep}_0 \). Note that \( \text{D}(n) \) is roughly \( \lg^*(n) \) where \( \lg^*(n) \) is the number of logs (base 2) that must be taken to reduce \( n \) to be at most 1. E.g., \( \lg^* 2 = 1, \lg^* 3 = 2, \lg^* 5 = 3, \lg^* 19 = 4, \lg^*(1,000,000) = 5 \). Hence \( \text{rep}_*(n) \) has length at most

\[
\lambda(n) := 2 \lg^*(n) + \lg n + \lg \lg n + \lg \lg \lg n + \cdots + \lg^{(\lg^*(n))}(n).
\]

We also need to figure out \( n_0 \). For instance, if \( n = 2 \), then \( \text{rep}(2) = \text{rep}(\text{len}(2)) \text{bin}(2) = \text{rep}(2) \text{bin}(2) \). Oops, \( \text{rep}(2) \) is defined in terms of itself. We note that \( \text{rep}(3) = \text{rep}(\text{len}(3)) \text{bin}(3) = \text{rep}(2) \text{bin}(3) \) does not have this issue. Hence the smallest choice of \( n_0 \) is \( n_0 = 3 \). For \( n < n_0 \), let us define \( \text{rep}_0(n) \). To summarize,

\[
\text{rep}(n) = \begin{cases} 
\text{rep}_0(n) & \text{if } n \leq 2 \\
\text{rep}(\text{len}(n)) \text{bin}(n) & \text{else}.
\end{cases}
\]

\[
D(n) = \begin{cases} 
1 & \text{if } n \leq 2 \\
1 + D(\text{len}(n)) & \text{else}.
\end{cases}
\]

Let us first try to compute \( \text{rep}(99) \). We have

\[
\text{rep}(99) = \text{rep}(7) \text{bin}(99) = \text{rep}(3) \text{bin}(7) \text{bin}(99) = \text{rep}(2) \text{bin}(3) \text{bin}(7) \text{bin}(99) = 001'111, 111, 1100011.
\]

Also, \( D(99) = 4 \). Therefore

\[
\text{rep}_*(99) = 0001'\text{rep}(99) = 0001'001'11, 111, 1100011.
\]

This is longer than \( \text{rep}_2(99) \) or \( \text{rep}_1(99) \).

Comments: How large does \( n \) have to be if \( \text{rep}_*(n) \) is shorter than \( \text{rep}_2(n) \)?

Q2. (5 Points) Adam Gashlin in our class suggests an improvement upon the compressed bit representation of Huffman code trees in our text. Say a Huffman code tree \( T \) is canonical if at every level, every leaf must appear before any internal node in the left-right listing of the level.

(a) Show that every Huffman code tree can be re-oriented to become canonical.

(b) Modify our “compressed bit representation” for canonical Huffman code tree. Compare these two encoding schemes.

(c) (Open) Can you construct a representation of canonical Huffman trees using \( n + o(n) \) bits? One suggestion is to use the self-limiting encoding of the previous exercise: suppose the leaves of the canonical tree are \( L_1, L_2, \ldots, L_n \), sorted by non-decreasing depth. Let \( d_i \) be the depth of \( L_i \). The encoding consists of the sequence of bits: (1) A self-limiting representation of \( d(L_1) \). (2) For \( i = 2, \ldots, n - 1 \), some self-limiting encoding of \( d(L_{i+1}) - d(L_i) \).

NOTE: alternatively, is it true that if I tell you the number of leaves at each successive level, but you are not told the actual depth of each level, then you can uniquely reconstruct the depths of each level?
SOLUTION:  (a) Let us modify the tree in a top-down fashion (it seems slightly easier than a bottom up fashion). Begin at the level of the root (level 0). If every node in the current level is an internal node, there is nothing to do. Otherwise, move all the leaves at that level to the left of all the internal nodes (this can be done by pairwise exchange of nodes). Note that when we move a node $u$, we are really moving the entire subtree $T_u$. Go to the next level, or stop if there is no next level.

(b) We can now represent the shape of a canonical Huffman tree as follows: Suppose the leaves of the canonical $T$ are $L_1, L_2, \ldots, L_n$, sorted by non-decreasing depth. Let $d_i$ be the depth of $L_i$. This canonical representation of the canonical Huffman tree is given by the string

$$0^{d_1}10^{d_2-1}10^{d_3-2}1 \ldots 0^{d_n-d_{n-1}}1.$$ 

It is clear that we can reconstruct the shape of $T$ from this sequence, and that it is self-limiting. There will always be $n$ ones in the canonical representation.

It is clear that the canonical representation is never worst than our compressed bit representation. We show a case where the canonical representation and the compressed bit representation are identical: consider the full binary tree where $d_i = i$ ($i = 1, \ldots, n-1$) and $d_n = d_{n-1}$. Its canonical representation and also compressed bit representation is:

$$010101 \cdots 01011$$

of length $2n-1$ (with $n$ ones and $n-1$ zeros). On the other hand, the canonical representation can use as few as $n + \log n$ bits: if $n = 2^k$ and the $T$ is the complete binary tree, then its canonical representation is

$$0^k1^n.$$ 

(c) We suspect that there is an encoding of canonical Huffman trees using $n + o(n)$ bits. The suggested scheme still seems to use $2n$ bits in the worst case.

Comments: It seems that some form of this encoding of canonical Huffman trees is used in Audio-Video compression.

Q3. (3 Points) In class today, Therese Avitabile suggests the following algorithm for computing a random permutation of $\{1, \ldots, n\}$:

\begin{verbatim}
RandomPermutation
Output: A random permutation of $S_n$ stored in $A[1..n]$.
1. for $i = 1$ to $n$ do
2. \hspace{1em} $A[i] = 0$. \hspace{1em} \text{\triangleright} \text{Initially, each entry is "empty"}
3. for $i = n$ downto 1 do \hspace{1em} \text{\triangleright} \text{Main Loop}
4. \hspace{1em} $j \leftarrow 1 + \lfloor i \cdot \text{random}() \rfloor$.
5. \hspace{1em} “Put $i$ into the $j$-th empty slot of array $A$.”
\end{verbatim}

Line 5 has the obvious interpretation:

\begin{verbatim}
PUT($i, j$):
\hspace{1em} \text{\triangleright} \text{To put $i$ into the $j$-th empty slot}
\hspace{1em} for $k = 1$ to $n$
\hspace{2em} if ($A[k] = 0$)
\hspace{3em} $j \leftarrow j$
\hspace{2em} if ($j = 0$)
\hspace{3em} $A[k] \leftarrow i$
\hspace{3em} Return.
\end{verbatim}
Prove that this algorithm computes a uniformly randomly permutation on \{1, \ldots, n\}.

**SOLUTION:** Define the event \( A_{i,j} \) to mean that the entry \( A[j] \) contains \( i \). Given a permutation \( \pi \in S_n \), we have a sequence of events \( A_{i,j(i)} \) that holds for \( i = 1, \ldots, n \). Rename \( A_{i,j(i)} \) as \( A_i \). It suffices to prove that

\[
\Pr(A_i | A_n A_{n-1} \cdots A_{i+1}) = 1/i
\]

But this is immediately. The result follows by our formula

\[
\Pr(A_1, \ldots, A_n) = \prod_i \Pr(A_i | A_n A_{n-1} \cdots A_{i+1}).
\]

**Comments:**

Q4. (2+3+2+3 Points) Consider the example in the text VII.6, \( X_n = (01a)^n \) and \( Y_n = (10a)^n \). Let \( \lambda_n := \lambda(X_n, Y_n) \). We want to improve the bound \( \lambda_n \geq 2^n \) in the text.

(a) Compute \( L(X_2, Y_2) \) by filling in the usual matrix. Further determine \( \lambda(X_2, Y_2) = |LCS(X_2, Y_2)| \).

(b) Prove that \( L(X_n, Y_n) = 2n \). HINT: Use induction, but it can be quite tricky to find the right set of induction hypotheses!

(c) Using your result in part (a), show that \( \lambda_n = \Omega(\sqrt{n}) \).

(d) Provide an exact closed formula for \( \lambda_n \), and show that this is \( \Theta(4^n/\sqrt{n}) \). HINT: There is the usual digraph constructed from the entries of the matrix \( L[0..3n, 0..3n] \), with unique source \((3n, 3n)\) and sink \((0,0)\). Count the number of paths from \((3n, 3n)\) to \((0,0)\).
\textbf{SOLUTION:}  
(a) $L(X_2, Y_2) = 4$ and $\lambda(X_2, Y_2) = 6$.  
(b) The proof is tricky, as you need to get a reasonably small set of inductive hypotheses.  
I will give 3 inductive hypotheses: let $m, n \geq 0$.  
\begin{align*}  
L(X_n, Y_m) &\leq n + m \quad (- \text{hypothesis } A(n, m)) \\
L(X_n0, Y_m) &\leq n + m \quad (- \text{hypothesis } B(n, m)) \\
L(X_n, Y_m1) &\leq n + m \quad (- \text{hypothesis } C(n, m))  
\end{align*}  
(8)  

Note that our original goal is to show is $L(X_n, Y_n) = 2n$.  This easily follows from $A(n, n)$,  
and the fact that $L(X_n, Y_n) \geq 2n$.  
We will use induction on the pairs $(n, m)$, based on the the partial ordering where  
$(n, m) \geq (n', m')$ iff $m \geq m'$ and $n \geq n'$.  

Induction Basis: If $m = 0$ or $n = 0$, then $A(n, m)$ is true since the RHS equal 0.  
If $n = 0$, then $B(n, m)$ is true since the RHS equal $\delta(m > 0)$ (using Kronecker delta  
function, which is 1 iff the predicate $m > 0$ is true, and 0 otherwise).  
If $m = 0$, then  
$C(n, m)$ is true since the RHS equal $\delta(n > 0)$.  
We now prove $A(n, m)$, $B(n, m)$ and $C(n, m)$ (in this order), assuming $X(n', m')$ holds  
for all $X = A, B, C$ and all $(n', m') < (n, m)$.  You will see that it is important to prove  
$A(n, m)$ before $B(n, m)$ or $C(n, m)$.  

- $A(n, m)$: By way of contradiction, let $L(X_n, Y_m) \geq n + m + 1$.  
\begin{align*}  
L(X_n, Y_m) &= 1 + L(X_{n-1}01, Y_{m-1}10) \quad \text{(equality case of LCS)} \\
&= 1 + \max \{L(X_{n-1}0, Y_{m-1}10), L(X_{n-1}01, Y_{m-1})\} \quad \text{(inequality case of LCS)} \\
&= 2 + \max \{L(X_{n-1}, Y_{m-1}1), L(X_{n-1}0, Y_{m-1})\} \quad \text{(equality case of LCS)} \\
&\leq 2 + (n + m - 2) \quad \text{(by $C(n - 1, m - 1) < B(n - 1, m - 1)$)} \\
&= n + m \quad \text{(contradiction)}  
\end{align*}  

- $B(n, m)$: By way of contradiction, let $L(X_n0, Y_m) \geq n + m + 1$.  Note that we may now  
also assume $A(n, m)$!  
\begin{align*}  
L(X_n0, Y_m) &= \max \{L(X_n, Y_m), L(X_n0, Y_{m-1}10)\} \quad \text{(inequality case of LCS)} \\
&= L(X_n0, Y_{m-1}10) \quad \text{(by $A(n, m)$)} \\
&= 1 + L(X_n, Y_{m-1}1) \quad \text{(equality case of LCS)} \\
&\leq 1 + (n + m - 1) \quad \text{(by $C(n, m - 1)$)} \\
&= n + m \quad \text{(contradiction)}  
\end{align*}  

- $C(n, m)$: this is similar to the proof of $B(n, m)$.  

\textbf{COMMENT:}  I briefly looked at some of your proofs here, and I feel that most of them  
are lacking.  If you have a slicker proof than this, please let me know!  This new proof is  
motivated by Adam Gashlin finding a bug in my previous solution; he suggested using  
the hypothesis $L(X_n, Y_n) < 2n$ and this made me realize that all my previous hypotheses  
where equalities and they were hard to generalize for arbitrary $m$ and $n$.  

13
(c) First, assume that $n$ is even. Break $X_n, Y_n$ into $n/2$ groups each. In each pair of
groups, there are 6 independent LCS (from part (a)). This gives $6^{n/2} = \sqrt[6]{6^n}$ versions of
LCS, so $\lambda_n = \sqrt[6]{6^n}$. If $n$ is odd, this yields $\lambda_n = \sqrt[6]{6^{n-1}} = K \sqrt[6]{6^n}$ (where $K = 6^{-1/2}$).
(d) The digraph in the HINT is just the straightforward graph with vertex set
$\{0, 1, \ldots, 3n\}^2$. If you compress this digraph as described in the text, you get the graph
$G_1(X_n, Y_n)$. Let $G_1(X_n, Y_n)$ be denoted $H_n$, whose vertices may be labeled by $(i, j)$
where $i + j$ denotes their level (source is level $2n$, sink is level $0$.) The nodes in level $j$
are denoted $(0, j), (1, j - 1), (2, j - 2), \ldots, (j, 0)$. Edges go from level $j$ to level $j - 1$.
There are two kinds of edges from any node $(i, j)$: horizontal $(i, j) - (i - 1, j)$ ($j \geq 1)$
and vertical $(i, j) - (i - 1, j)$ ($i \geq 1$). Moreover, $\lambda_n$ equals the number of paths from
$(n, n)$ to $(0, 0)$ in $H_n$.
It is important to prove that the structure of $H_n$ has the grid structure just described.
This is clear from explicit construction for $n = 1, 2, 3$. To prove the general case, you can
just look at the recursive equation for $L(X_n, Y_n)$.
Now we are ready to count (maximal) paths in $H_n$ which has a 1-1 bijection (why?)
with the elements on LCS$(X_n, Y_n)$. The number of such paths is $\binom{2n}{n}$ since each path
can be identified with an $n$ element subset $X$ of $\{1, \ldots, 2n\}$. The path corresponding
to $X$ starts from $(n, n)$ and ends at $(0, 0)$, comprising a sequence of $2n$ edges. The
ith edge is horizontal if $i \in X$. For $n = 1, 2, 3, 4$, this number is 2, 6, 20, 70. Using
Stirling’s approximation, we see that $\binom{2n}{n} = \Theta(4^n/\sqrt{n})$. (i) Extend the above proof
to show that $L(X_n, Y_{n+k}) = 2n + k$ for all $0 \leq k \leq n$. It is easy to see that if $k > n$,
$L(X_n, Y_{n+k}) = 3n$.
(ii) The fact that maximal paths in $H_n$ is in bijection with LCS$(X_n, Y_n)$ is a quirk of the
strings $X_n, Y_n$. Here is an example where this fails: $X_n := (01aa)^n$ and $Y_n := (10aa)^n$.
In this case, LCS$(X_n', Y_n')$ is different when viewed as a multiset than a set.

Q5. (2+3+3 Points) We consider the alignment problem for strings. Assume use the simplified cost model,
$$
\delta_m = -2, \quad \delta_{\neq} = 1, \quad \delta_* = 3.
$$

(a) Compute the matrix $A(\text{length, elongate})$.
(b) Describe in detail an $O(mn)$ space algorithm to compute any optimal alignment $(X_*, Y_*)$ for an
input pair $X, Y$ of strings. Be sure to justify the correctness of your algorithm.
(c) Derive a “small space” version of the problem in (b). Determine its space and time complexity.

SOLUTION: (c) In this context, small space means linear space. There is a simple
small space solution: just write a recursive algorithm to compute $A(X, Y)$.
$$
A(X, Y) = \min \{ A(X', Y') + \delta(x_m, y_n), A(X', Y) + \delta_*, A(X, Y') + \delta_* \}
$$
We can turn this into a method for computing the alignment by a bit more book keeping.
Clearly, the space is determined by the recursion depth, and this is linear. The time is
exponential however.
But clearly, that is not what we have in mind. Instead, you can emulate the small space
solution we gave for LCS.

Comments:

Q6. (2 Points) Consider the problem of Optimal Binary Search Trees ($\S$VII.7). Suppose $n = 3$ and all the
$q$’s are zero. We have five possible BST’s as shown in Figure 1: $T_a, T_b, \ldots, T_e$. Characterize the domain
$D_i \subseteq \mathbb{R}^3$ of $(p_1, p_2, p_3)$ for which the tree $T_i$ is the optimal binary search tree ($i = a, b, \ldots, e$). Are these
domains convex? Connected?
SOLUTION: The answer is quite simple: for each tree $T_i$, we have a linear function $L_i(p_1, p_2, p_3)$ that gives the cost of $T_i$:

$$L_a = p_1 + 2p_2 + 3p_3, \quad L_b = p_1 + 3p_2 + 2p_3, \quad L_c = 2p_1 + p_2 + 2p_3, \quad L_d = 2p_1 + 3p_2 + p_3, \quad L_e = 3p_1 + 2p_2 + p_3.$$  

Each $D_i$ is characterized by $L_i \leq L_j$ for all $j \neq i$. The inequality $L_i \leq L_j$ specifies a half space, and so $D_i$ is an intersection of half-spaces. Therefore each $D_i$ is a convex polytope. Clearly, it is connected.

Comments: Clearly the characterization of the domains $D_i$ where $i$ range over all binary trees with $n$ nodes is completely general. We can even allow the $q_i$’s to be non-zero.

4 Homework 5: Due on Thu Nov 1

Q1. (6 Points) A vertex cover for a bigraph $G = (V, E)$ is a subset $C \subseteq V$ such that for each edge $e \in E$, at least one of its two vertices is contained in $C$. A minimum vertex cover is one of minimum size. Here is a greedy algorithm to find a vertex cover $C$:

1. Initialize $C$ to the empty set.
2. Choose from the graph a vertex $v$ with the largest degree. Add vertex $v$ to the set $C$, and remove vertex $v$ and all edges that are incident on it from the graph.
3. Repeat step 2 until the edge set is empty.
4. The final set $C$ is a vertex cover of the original graph.

(a) Show a graph $G$, for which this greedy algorithm fails to give a minimum vertex cover. HINT: An example with 7 vertices exists.
(b) Let $x = (x_1, \ldots, x_n)$ where each $x_i$ is associated with vertex $i \in V = \{1, \ldots, n\}$. Consider the following set of inequalities:

- For each $i \in V$, introduce the inequality $0 \leq x_i \leq 1$.
- For each edge $(i, j) \in E$, introduce the inequality $x_i + x_j \geq 1$.

If $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ satisfies these inequalities, we call $a$ a feasible solution. If each $a_i$ is either 0 or 1, we call $a$ a 0–1 feasible solution. Show a bijective correspondence between the set of vertex covers and the set of 0–1 feasible solutions. If $C$ is a vertex cover, let $a^C$ denote the corresponding 0–1 feasible solution.
(c) Suppose $x^* = (x^*_1, \ldots, x^*_n) \in \mathbb{R}^n$ is a feasible solution that minimizes the function $f(x) = x_1 + x_2 + \cdots + x_n$, i.e., for all feasible $x$,

$$f(x^*) \leq f(x).$$
Call \( x^* \) an **optimum vector**. Note that \( x^* \) is not necessarily a 0–1 vector. Construct a graph \( G = (V, E) \) where \( x^* \) is not a 0–1 feasible solution. **HINT:** you do not need many vertices \( (n \leq 4 \) suffices).

(d) Given an optimum vector \( x^* \), define set \( \overline{C} \subseteq V \) as follows: \( i \in \overline{C} \) iff \( x_i \geq 0.5 \). Show that \( \overline{C} \) is a vertex cover.

(e) Suppose \( C^* \) is a minimum vertex cover. Show that \( |\overline{C}| \leq 2|C^*| \). **HINT:** what is the relation between \( |\overline{C}| \) and \( f(x^*) \)? Between \( f(x^*) \) and \( |C^*| \)? **REMARKS:** using Linear Programming, we can find an optimum vector \( x^* \) quite efficiently. The technique of converting an optimum vector into an integer vector is a powerful approximation technique.

**SOLUTION:**

(a) Consider the graph with vertex set \( \{a, a', b, b', c, c', d\} \) and edge set \( \{a' \!-\! a, b' \!-\! b, c' \!-\! c, a \!-\! d, b \!-\! d, c \!-\! d\} \). Our algorithm will first choose the vertex \( d \) of degree 3. The final vertex cover has size 4, but the optimum has size 3.

(b) For any 0–1 feasible solution, \( a = (a_1, \ldots, a_n) \) we have a corresponding set \( C = \{i \in V : a_i = 1\} \). This is easily seen to be a vertex cover. Conversely, for any vertex cover \( C \), we have a corresponding \( a = (a_1, \ldots, a_n) \) which is a 0–1 feasible solution.

(c) Consider the graph with \( V = \{a, b, c\} \) and \( E = \{a \!-\! b, b \!-\! c, c \!-\! a\} \). Then we have the inequalities \( x_a + x_b \geq 1 \), \( x_b + x_c \geq 1 \) and \( x_c + x_a \geq 1 \). Then \( f(x_a, x_b, x_c) = x_a + x_b + x_c \). If \( x = (x_a, x_b, x_c) \) is a 0–1 vector that arise from a vertex cover, then \( f(x) \geq 2 \). On the other hand, if \( x^* = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}) \) then \( f(x^*) = 1.5 \). It is not hard to see that \( x^* \) is optimum: if \( x_a < \frac{1}{2} \), then \( x_b > \frac{1}{2} \) and \( x_c > \frac{1}{2} \), and \( f(x^*) > 1.5 \).

(d) Note that for each edge \( (i, j) \), either \( i \in C^* \) or \( j \in C^* \). Hence \( C^* \) is a vertex cover.

(e) Let \( C^* \) be a minimum vertex cover. Then we know from (b) that \( |C^*| \geq f(x^*) \). Moreover, we have \( 2x_i^* \geq x_i \). Hence \( 2|C^*| \geq 2f(x^*) = \sum_i 2x_i^* \geq \sum_i x_i = |\overline{C}| \).

Comments:

Q2. (10 Points) We have the 2-car loading problem, but now imagine the 2 cars move along two independent tracks, say the left track and right track. Either car could be sent off before the other. We still make decision for each rider in an online manner, but our ith decision \( x_i \) now comes from the set \( \{L, R, L^+, R^+\} \). The choice \( x_i = L \) or \( x_i = R \) means we load the ith rider into the left or right car (resp.), but \( x_i = L^+ \) means that we send off the left car, and put the i-th rider into a new car in its place. Similarly for \( x_i = R^+ \). Consider the following heuristic: let \( C_0 > 0 \) and \( C_1 > 0 \) be the residual capacities of the two open cars. Try to put \( w_i \) into the car with the smaller residual capacity. This is the **best fit** strategy. If neither fits, we send off the car with the smaller residual capacity (and put \( w_i \) into its replacement car). This is open ended: prove or disprove that this strategy will never use more cars than the \( G_1 \) algorithm, or the First Fit Algorithm \( G_2 \).
SOLUTION: (a) Let \( G_2''(w) \) denote the minimum number of cars used by this best-fit two-car and two-track algorithms. We want to compare this to \( G_1(w) \) (the greedy 1-car algorithm). CLAIM: \( G_2''(w) \leq G_1(w) \). We use induction on the length of \( w \). This is clear if the length is 1 or \( G_1(w) = 1 \). Suppose \( w = (w_1, \ldots, w_n) \) of length \( n > 1 \) and \( G_1(w) > 1 \). Inductively, assume that we have shown \( G_2''(w') \geq G_1(w') \) for all \( w' \) of length \( < n \). We may write \( w = w' ; (w_m, w_{m+1}, \ldots, w_n) \) for some \( m < n \) such that \( G_1(w') = G_1(w) - 1 \geq 1 \). Consider the state of \( G_2''(w') \) after it has finished processing the input \( w' \). There are now two cars \( C \) and \( C' \) on the two tracks, both partially filled. (If \( G_2''(w') = 1 \), then one of these two cars will be empty.) We now continue to process the remaining weights \( (w_m, w_{m+1}, \ldots, w_n) \). Clearly, since these weights fit into a single car, \( G_2'' \) will never use more than one additional car. This shows that \( G_2''(w) \leq 1 + G_2''(w') \leq 1 + G_1(w') = G_1(w) \).

(b) We give a counter example to the claim that \( G_2''(w) \leq G_2(w) \).

Consider the input

\[
w = \left( \frac{1}{2} + \epsilon, \quad \frac{1}{2} + 2 \epsilon, \quad 2 \epsilon, \quad \frac{1}{2} - 3 \epsilon, \quad \frac{1}{2} - 2 \epsilon \right).
\]

It is easy to see that \( G_2(w) \) will fill two cars with the weights

\[
C_1 = \left( \frac{1}{2} + \epsilon, \frac{1}{2} - 3 \epsilon \right) \\
C_2 = \left( \frac{1}{2} + 2 \epsilon, \frac{1}{2} - 2 \epsilon \right)
\]

and thus \( G_2(w) = 2 \). However, \( G_2''(w) \) will fill three cars as follows:

\[
C_1 = \left( \frac{1}{2} + \epsilon, \frac{1}{2} - 3 \epsilon \right) \\
C_2 = \left( \frac{1}{2} + 2 \epsilon, 2 \epsilon \right) \\
C_3 = \left( \frac{1}{2} - 2 \epsilon \right)
\]

Both strategies put the first two weights into two cars as follows:

\[
C_1 = \left( \frac{1}{2} + \epsilon \right) \\
C_2 = \left( \frac{1}{2} + 2 \epsilon \right)
\]

The arrival of \( 2 \epsilon \) now makes \( G_2'' \) and \( G_2 \) diverge: the best-fit algorithm \( G_2'' \) puts \( 2 \epsilon \) into the second car while the first-fit \( G_2 \) puts \( 2 \epsilon \) into the first car. The last two weights will now exploit this difference.

Comments:

Q3. (2+2+3+5 Points) Huffman code is based on transmitting bits. Suppose we transmit in ‘trits’ (a base-3 digit). Then the corresponding 3-ary Huffman code \( C : \Sigma \rightarrow \{0, 1, 2\}^* \) is represented by a 3-ary code tree \( T \) where each leaf is associated with a unique letter in \( \Sigma \) and each internal node has degree at most 3. If \( f : \Sigma \rightarrow \mathbb{N} \) is a frequency function, this assigns a weight to each node of \( T \): the leaf associated with \( x \in \Sigma \) has weight \( f(x) \), and each internal node has a weight equal to the sum of the weights of its children. The cost of \( T \) is defined as usual, as the sum of the weights of the internal nodes of \( T \). We are interested in optimal trees \( T \), i.e., whose cost is minimum.

(a) Show that in an optimal 3-ary code tree, there are no nodes of degree 1 and at most one node of
degree 2. Furthermore, if a node has degree 2, it must have leaves as both of its children.
(b) Let $T$ be a tree whose internal nodes have degrees 2 or 3. If there are $d_i$ nodes of degree $i$ ($i = 0, 2, 3$) in $T$ show that $d_0 = 1 + d_2 + 2d_3$.
(c) Show that there are optimal 3-ary code trees with this property: if $|\Sigma|$ is odd, there are no degree 2 nodes, and if $|\Sigma|$ is even, there is one degree 2 node. Moreover, if the unique node $u$ of degree 2 we may assume its children have minimum frequencies among all the leaves.
(d) Give an algorithm for constructing an optimal 3-ary code tree and prove its correctness.

SOLUTION: (a) Like the binary case, it is clear that an optimal code tree cannot have any node with degree 1. If two nodes $u$ and $v$ have degree 2, and assuming depth of $u$ is less than or equal to the depth of $v$, then we can move one child of $v$ to become a third child of $u$ without increasing the cost of the code tree. But now this new code tree is non-optimal (since $v$ has degree 1), and this contradicts the claim that the original tree was optimal. This proves that any optimal code tree has at most one node of degree 2. Next, suppose $u$ has degree 2, and one of its children is also a leaf. Then we can take any leaf below that child and make it a child of $u$. This will reduce the cost of the code tree, again a contradiction.
(b) Use induction on the number of leaves $d_0$. The result is clearly true if $d_0 = 1$. If $d_0 > 1$, then there is at least one internal node $u$ with leaves as children. We remove the children of $u$ to get a new tree $T'$ in which $u$ is now leaf. Also assume $T'$ has $d'_i$ nodes of degree $i$ ($i = 0, 2, 3$). Since $T'$ has fewer leaves than the original tree, our induction hypothesis tells us that

$$d_0 = 1 + d'_2 + 2d'_3 = 1 + d_2 - 1 + d_3. \quad (9)$$

There are two cases: CASE (i) where $u$ has 2 children. Thus $d'_0 = d_0 - 1$, $d'_2 = d_2 - 1$ and $d'_3 = d_3$. Plugging this into (9), we obtain $d_0 = 1 + d_2 + 2d_3$. CASE (ii) where $u$ has 3 children. Thus $d'_0 = d_0 - 2$, $d'_2 = d_2$ and $d'_3 = d_3 - 1$. Plugging this into (9), we again obtain $d_0 = 1 + d_2 + 2d_3$.
(c) From parts (a) and (b), we know that the optimal tree has degrees $|\Sigma| = d_0 = 1 + d_2 + 2d_3$, where $d_2$ is either 0 or 1. If $|\Sigma|$ is even (odd), this equation implies that $d_2$ must be 1 (0).
(d) The algorithm is as follows: we start with $|\Sigma|$ trees, where each tree consists of just one node (the root) labeled by a letter of $\Sigma$ and with the weight equal to the frequency of that letter. If $|\Sigma|$ is even, we merge any two trees of least weight to form a new tree. In general, suppose there is an odd number of trees, we just merge any 3 trees of least weight to form a new tree. The number of trees remain odd. At the end we are left with exactly one tree, which is our optimal 3-ary Huffman tree. The correctness is proved exactly as in the binary case, using induction and part (c).

Comments:

Q4. (2+2+2+2+5+1 Points) We consider the 4-ary version of the previous question. Let $T$ be an optimum 4-ary code tree for some frequency function $f : \Sigma \rightarrow \mathbb{N}$.
(a) Give a short inductive proof of the following fact: Suppose $T$ is any 4-ary tree on $n \geq 1$ leaves, and let $N_d$ be the number of nodes with $d$ children ($d = 0, 1, 2, 3, 4$). Thus, $n = N_0$. Give a short inductive proof for the following formula: $n = 1 + N_2 + 2N_3 + 3N_4$.
(b) Show that if $T$ is an optimal code tree, then $N_1 = 0$ and $3N_2 + 2N_3 \leq 4$, and every non-full internal node has only leaves as children and the depth of these leaves must equal the height of $T$.
(c) Moreover, we can always transform $T$ from part (b) into $T'$ such that the corresponding degrees satisfy $N'_1 = 0$ and $N'_2 + N'_3 \leq 1$. Also, for any non-full internal node of $T'$, its children have weights no larger than any other leaves.
(d) Suppose $r = (n - 1) \mod 3$. So $r \in \{0, 1, 2\}$. Show how $N'_2, N'_3$ in part (b) is determined by $r$.
(e) Describe an algorithm to construct an optimal code tree from a frequency function $f$.
(f) Show the optimal 4-ary Huffman tree for the input string hello world!. Please state the cost of this optimal tree.
Q5. GOOD!!!

Suppose we want to transmit a sequence involving only three characters, a,b,c. Using our splay tree protocol what is the worst case sequence of length n?

Q6. Compute \(A(X_n, Y_n)\) and \(L(X_n, Y_n)\) where \(X_n = (01ab)^n\) and \(Y_n = (10ba)^n\).

REMARK: \(L(X_n, Y_n)\) seems to be 2n.

Q7. Adam Gashlin wants to use the predicate \(C_2(I) : 0 \notin \square f''(x)\) to help do root isolation. Design an algorithm with

Q8. Order 4 Fibonacci search?

Q9. (12 Points) Exercise VI.3.3 (p.21). Let us define the potential of node \(u\) to be \(\Phi(u) = \text{lg}(\text{SIZE}(u))\), instead of \(\Phi(u) = \lfloor \text{lg}(\text{SIZE}(u)) \rfloor \).

(a) How does this modification affect the validity of our Key Lemma about how to charge \texttt{SplayStep}?
In our original proof, we had 2 cases: either $\Phi'(u) - \Phi(u)$ is 0 or positive. But now, $\Phi'(u) - \Phi(u)$ is always positive. Thus it appears that we have eliminated one case in the original proof. What is wrong with this suggestion?

(b) Consider Case I in the proof of the Key Lemma. Show that if $\Phi'(u) - \Phi(u) \leq \lg(6/5)$ then $\Delta \Phi = \Phi'(w,v) - \Phi(u,v) \leq -\lg(6/5)$. HINT: the hypothesis implies $a + b \geq 9 + 5c + 5d$.

(c) Do the same for Case II.

**SOLUTION:**

(a) It is not true that we have gotten rid of one case. There remain two cases. Just because $\Phi'(u) - \Phi(u)$ is positive is not enough. We must show that it is bounded away from 0, say

$$\Phi'(u) - \Phi(u) \geq K \tag{10}$$

for some (any) fixed positive constant $K$. This is the constant that will pay for the $O(1)$ work that is performed in the splay step, in case the $\Delta \Phi > 0$.

There is also an important converse: in case (10) fails, we must show that

$$\Delta \Phi \leq -K. \tag{11}$$

This release in potential will pay for the work done in SplayStep.

(b) We will choose the constant $K$ to be $\lg(6/5)$. Suppose $\Phi'(u) - \Phi(u) = \lg((3 + a + b + c + d)/(1 + a + b)) \leq \lg(6/5)$. Then $5(3 + a + b + c + d) \leq (1 + a + b)$, i.e.,

$$9 + 5(c + d) \leq a + b. \tag{12}$$

We need to show that (12) implies $\Phi'(v,w) - \Phi(u,v) \leq \lg(5/6)$. I.e., $\lg((2 + b + c + d)(1 + c + d)) - \lg((1 + a + b)(2 + a + b + c)) \leq \lg(5/6)$, i.e.,

$$6(2 + b + c + d)(1 + c + d) \leq 5(1 + a + b)(2 + a + b + c). \tag{13}$$

This follows since

$$6(2 + b + c + d)(1 + c + d) = [3(1 + c + d)][2(2 + b + c + d)] \leq [1 + a + b][2(2 + b + c + d)] \quad \text{(by (12), } a + b \geq 3(1 + c + d))$$

$$\leq [1 + a + b][2(2 + a + 2b + c)] \quad \text{(by (12), } a + b \geq d)$$

$$\leq [1 + a + b][5(2 + a + b + c)]$$

which proves (13).
Q10. (2+2+2 Points) Consider the transmission of the string “hello world!” using the Dynamic Huffman tree algorithm and the splay tree algorithm.
(a) First describe the bit string that is transmitted using the Dynamic Huffman tree algorithm. Do not just give the bit string, but summarize the processes that goes into the production of each substring.
(b) Do the same, but using the Splay tree approach.
(c) Draw some conclusion about the relative merits of the two methods.

Q11. (2+2 Points) Here is an ASCII string $X = b^m a^n c^p$ where $m, n, p$ are positive integers. So $|X| = m + n + p$. The ASCII ordering on the characters is $a < b < c$ (of course). Recall our protocol for transmitting strings using External Splay trees.
(a) If $m = 10, n = 2, p = 20$, what is the length of $E(X)$? The answer is a single number, but explain how you get this number.
(b) TRUE or FALSE: there exist constants $\alpha, \beta, \gamma, \delta$ such that the length of $E(X)$ is equal to

$$|E(X)| = \alpha m + \beta n + \gamma p + \delta$$
SOLUTION: (b) We first answer part (b): the answer is FALSE. Let us see why: Let $n' = \lfloor n/2 \rfloor$.

$$E(X) = \text{ASC}(a)1^m0\text{ASC}(b)(0110)^{n'}0\text{ASC}(c)1^p$$

if $m$ is even. Thus when $n$ is even, we have $|E(X)| = 8 + m + 1 + 8 + 2n + 1 + 8 + p = 26 + m + 2n + p$. But when $m$ is odd, $E(X) = \text{ASC}(a)1^m0\text{ASC}(b)(0110)^{n'}1000\text{ASC}(c)1^p$.

But when $n$ is odd, $|E(X)| = 27 + m + 2n + p$.

(a) The answer 60. This comes from the formula in part (b) when $n$ is even: $|E(X)| = 26 + m + 2n' + p = 26 + 10 + 4 + 20 = 60$.

Comments: We seek a formula of the form:

$$|E(X)| = \alpha m + \beta n + \gamma p + \delta$$

From the preceding development, we see that the constants $\alpha = 1$, $\beta = 2$ and $\gamma = 1$ are well-defined. However, the constant $\delta$ can be 26 (if $n$ is even) or 27 (if $n$ is odd).
5 Homework 4: Due on Wed Oct 10

This is a short homework, in view of the midterm on Thu Oct 11. It is due just before midnite on Wed Oct 10th. We will post the solution immediately after midnite. Sorry we cannot accept late homework for this reason.

Q1. (2+2+3+3+3 Points) Let $u$ be a node in a binary tree. Define $\text{esize}(u)$ to be 1 more than the size of the subtree $T_u$ rooted at $u$. (Alternatively, $\text{esize}(u)$ is the number of nil nodes in the extended binary tree corresponding to $T_u$. See Lect.III ¶30.) The ratio of node $u$ in a binary tree is given by $\text{ratio}(u) := \text{esize}(u\text{.left})/\text{esize}(u\text{.right})$. E.g., If $T_u$ has size 1, then $\text{ratio}(u) = 1$. If $T_u$ has size 2, then $\text{ratio}(u) = 1/2$ if $u\text{.left} = \text{nil}$, and $\text{ratio}(u) = 2$ if $u\text{.right} = \text{nil}$. If $T_u$ has size 3, then $\text{ratio}(u) \in \{1/3, 1, 3\}$.

Let $\rho \in (0, 1)$. We say $u$ is $\rho$-ratio balanced if

$$\rho < \text{ratio}(u) < 1/\rho.$$  (18)

Notice that we have strict inequalities in this definition. A BST $T$ is $\rho$-ratio balanced (or, it is $\text{RB}[\rho]$) if every node in $T$ is $\rho$-ratio balanced.

![Figure 2: Ratio Rebalancing: (i) Single Rotate (ii) Double Rotate](image)

Consider the BST $T_u$ in Figure 2: it is rooted at $u$ with $v = u\text{.left}$ and $w = u\text{.right}$. Assume $\text{ratio}(u) = \rho$, $\text{ratio}(v) = \rho'$ and $\text{ratio}(w) = \rho''$.

(a) After we perform $\text{rotate}(v)$, the ratios of $v$ and $u$ are $\sigma$ and $\sigma'$ as shown Figure 2(i). Express $\sigma$ and $\sigma'$ in terms of $\rho$ and $\rho'$.

(b) After we perform $\text{rotate}^2(v)$, the ratios of $w, v, u$ are $\sigma, \sigma'$ and $\sigma''$, as shown Figure 2(ii). Express $\sigma, \sigma', \sigma''$ in terms of $\rho, \rho', \rho''$. HINT: it is useful to use the notation $\tilde{\rho} := 1 + \rho$.

(c) Suppose all nodes in $T_u$ are 1/3-ratio balanced with the exception that $\text{ratio}(u) = 3$. (The other case $\text{ratio}(u) = 1/3$ is similar). Prove that if $\text{ratio}(v) = \rho' \geq 3/5$, then after the single rotation $\text{rotate}(v)$ as in Figure 2(i), the resulting tree remains 1/3-ratio balanced. HINT: The expression for $\sigma$ is monotonic increasing or monotonic decreasing as a function of each of the variables $\rho, \rho'$. Using
this observation, you determine the minimum $\sigma_{\min}$ and $\sigma_{\max}$ as $\rho, \rho'$ varies in the range $[1/3, 3]$. Similarly, determine $\sigma'_{\min}, \sigma'_{\max}$.

(d) Suppose all nodes in $T_u$ are 1/3-ratio balanced with the exception that $\text{ratio}(u) = 3$. (The other case $\text{ratio}(u) = 1/3$ is similar). Prove that if $\text{ratio}(v) = \rho' \leq 3/5$, then after the double rotation $\text{rotate}^2(w)$ as in Figure 2(iii), the resulting tree remains 1/3-ratio balanced.

(e) Conclude from this that there is an algorithm to maintain 1/3-ratio balanced trees under insertion and deletion. HINT: suppose $T$ is not exactly 1/3-ratio balanced, but $1/3 + \delta$ for some small enough $\delta > 0$. Does the above argument work?

**SOLUTION:**

(a) We have

$$\rho = \frac{a + b}{c}, \quad \rho' = \frac{a}{b}. \quad (19)$$

Thus,

$$a = b \rho', \quad (20)$$

$$a + b = c \rho, \quad (21)$$

$$b(\rho' + 1) = c \rho, \quad (22)$$

$$b = c \frac{\rho}{\rho'}, \quad (23)$$

$$a = c \frac{\rho \rho'}{\rho}, \quad (24)$$

$$b + c = c \left( \frac{\rho}{\rho'} + 1 \right), \quad (25)$$

$$= c \frac{\rho + \rho'}{\rho}. \quad (26)$$

Finally,

$$\sigma' = \frac{b}{c} = \frac{\rho'}{\rho}, \quad \sigma = \frac{a}{b + c} = \frac{\rho \rho'}{\rho' + \rho} = \frac{\rho \rho'}{\rho' + \rho}. \quad (27)$$

(b) We have

$$\rho = \frac{a + b + c}{d}, \quad \rho' = \frac{a}{b + c}, \quad \rho'' = \frac{b}{c}. \quad (27)$$

Thus,

$$b = c \rho'' \quad (28)$$

$$a = \rho'(b + c) = c \rho' (\rho'' + 1) = c \rho' \tilde{\rho}' \quad (29)$$

$$d \rho = a + b + c = c(\rho' \tilde{\rho}' + \rho'' + 1) \quad (30)$$

Finally,

$$\sigma' = \frac{a}{b} = \frac{\rho' \tilde{\rho}'}{\rho''}, \quad \sigma'' = \frac{a}{c} = \frac{\rho}{\rho' \rho'' + \rho'' + 1}, \quad \sigma = \frac{a + b}{c + d} = \frac{\rho' \tilde{\rho}' + \rho''}{1 + \rho' \tilde{\rho}' + \rho''} = \frac{\rho(\rho' \tilde{\rho}' + \rho'' \tilde{\rho}'')}{1 + \rho' \tilde{\rho}' + \rho'' \tilde{\rho}'}. \quad (31)$$
(c) and (d):
Using the hint, we see that the \( \sigma \)'s depend on the \( \rho \)'s as indicated by this table:
under a variable \( \rho \), a + indicates increasing, a – indicates decreasing, a 0 indicates independence.

There are some tedious calculations in what follows (sorry).
For instance, in row R1, we see that \( \sigma \) is increasing with \( \rho \) and \( \rho' \) but independent of \( \rho'' \).
Both rows R1 and R2 are easy to check.
For row D1, we easily see that \( \frac{\partial \sigma}{\partial \rho} > 0 \).
What is more interesting is how \( \sigma \) vary with \( \rho' \):
\[
\frac{\partial \sigma}{\partial \rho'} = \frac{\rho \tilde{\rho}' (1 + \rho + \rho'' + \rho' \tilde{\rho}'') - \tilde{\rho}''}{D}
\]
where the denominator \( D \) is positive. The numerator is \( \tilde{\rho}'' (\rho (1 + \rho + \rho'' + \rho' \tilde{\rho}'') - 1) \geq \tilde{\rho}'' [3(1 + 3 + \cdots) - 1] > 0 \).
Here, we use the fact that \( \rho = 3 \). This proves that \( \sigma \) also increases with \( \rho' \). Similarly, the numerator of \( \frac{\partial \sigma}{\partial \rho''} \) is positive.
The monotonicity properties in D2 and D3 are similarly checked.
With these monotonicity properties, it is now easy to plug in the extreme values of \( \rho, \rho', \rho'' \) that will maximize or minimize \( S \in \{ \sigma, \sigma', \sigma'' \} \).
Let \( S_{\min} \) and \( S_{\max} \) denote these maxima. These entries are shown in the table.

<table>
<thead>
<tr>
<th>New Ratio</th>
<th>( \rho )</th>
<th>( \rho' )</th>
<th>( \rho'' )</th>
<th>( S_{\min} )</th>
<th>( S_{\max} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 ( \sigma = \frac{\rho \tilde{\rho}'}{\rho + \rho'} )</td>
<td>+ + 0</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{1}{3} )</td>
<td>( \frac{9}{23} )</td>
<td>( \frac{9}{7} )</td>
</tr>
<tr>
<td>R2 ( \sigma' = \frac{\tilde{\rho}'}{\rho} )</td>
<td>+ – 0</td>
<td>( \frac{1}{4} )</td>
<td>( \frac{3}{15} )</td>
<td>( \frac{7}{15} )</td>
<td>( \frac{15}{8} )</td>
</tr>
<tr>
<td>D1 ( \sigma = \frac{\rho (\rho'' + \rho' \tilde{\rho}'')}{1 + \rho + \rho'' + \rho' \tilde{\rho}''} )</td>
<td>+ + +</td>
<td>( \frac{2}{1} )</td>
<td>( \frac{1}{3} )</td>
<td>( \frac{8}{3} )</td>
<td>( \frac{3}{2} )</td>
</tr>
<tr>
<td>D2 ( \sigma' = \frac{\rho' \tilde{\rho}''}{\rho} )</td>
<td>0 + –</td>
<td>( \frac{4}{9} )</td>
<td>( \frac{1}{3} )</td>
<td>( \frac{12}{5} )</td>
<td>( \frac{1}{1} )</td>
</tr>
<tr>
<td>D3 ( \sigma'' = \frac{\rho \tilde{\rho}''}{\rho' \rho'' + \rho' \tilde{\rho}'' + 1} )</td>
<td>+ – –</td>
<td>( \frac{15}{32} )</td>
<td>( \frac{3}{4} )</td>
<td>( \frac{27}{16} )</td>
<td></td>
</tr>
</tbody>
</table>

The entries in the last two columns show that
\[
1/3 < S_{\min} \leq \text{ratio}(x) \leq S_{\max} < 3
\]
holds for every \( x \in \{ u, v, w \} \).
(e) Are the above derivations enough to give us algorithms for insertion and deletion for 1/3-ratio balanced trees? Well, what do these algorithm look like? As in AVL trees, we first perform the standard BST insert or delete. Then we go back along the path to rebalance any node \( u \) whose ratio is \( \leq 1/3 \) or \( \geq 3 \) (wlog, assume the case \( \text{ratio}(u) \geq 3 \)). The problem in the above analysis is that we assume that \( \text{ratio}(u) = 3 \). What if \( \text{ratio}(u) > 3 \)? Well, you can see from the numbers that there is a bit of ”play” left. Clearly, by continuity argument, there is an \( \epsilon_0 > 0 \) such that if \( \text{ratio}(u) = 3 + \epsilon \), then the above analysis still holds. I.e., we can restore the tree’s 1/3-balance condition.

For any \( \epsilon_0 \), it is clear that when \( \text{esize}(u) \) is larger than some \( n_0 \), then \( \text{ratio}(u) \leq 3 + \epsilon_0 \). What do we do when \( \text{esize}(u) < n_0 \)? For the sake of argument, we simply say: we rebuild a perfectly balanced tree at \( u \). This work is \( O(1) \), and so we will have an \( O(\log n) \) time algorithm for insertion/deletion. Of course, this crude argument is not practical. A careful analysis would show that if \( \epsilon_0 = 1 \), the above analysis goes through! Furthermore, the ONLY case where \( \text{ratio}(u) > 3 + \epsilon_0 = 4 \) can occur is when you delete from a tree of size 6. It turns out that even for this case, our rebalancing method gives the correct answer.

Q2. (10 Points) The text described a DFS algorithm for detecting if a bigraph is biconnected. Extend this algorithm to compute a representation of the reduced graph \( G^c \) of a bigraph \( G \). For simplicity, assume \( G \) is connected. Naturally, we want you to do ”Shell Programming”, but it is important that you explain your ideas in words (do not just write ”code”). Be sure to justify correctness of your algorithm.

**REPRESENTATION:** How should you represent \( G^c \)? We suggest encoding this information in three arrays:

\[
cut[v \in V], \rep[v \in V], \text{component}[v \in V].
\]

Initially, \( \cut[v] \leftarrow \text{false} \). Eventually, \( \cut[v] = \text{true} \) iff \( v \) is a cut-vertex. If there \( m \) biconnected components, let \( C_i \) refer to the \( i \)-th biconnected component (for \( i = 1, \ldots, m \)). For each \( C_i \), we will pick an arbitrary vertex \( r_i \in C_i \) as its **representative**. Actually \( r_i \) is not arbitrary, but is the second (!) vertex of \( C_i \) that is seen by DFS. We indicate this information by setting \( \rep[r_i] \leftarrow i \). If a vertex \( v \) is not the representative of any biconnected component, we set \( \rep[v] \leftarrow 0 \). As usual, we maintain the DFS tree by using a parent array \( p[u \in V] \) where \( p[u] \) is the parent of \( u \) in the DFS tree (\( p[u] = u \) iff \( u \) is root). Note that if \( r_i \) is the rep of \( C_i \), then \( p[r_i] \) is the first vertex of \( C_i \) that is seen by DFS. Finally, each DFS tree edge \( (p[u], u) \) is contained in a unique biconnected component. We set \( \text{component}[u] \leftarrow i \) iff the edge \( (p[u], u) \) belongs to the \( i \)th biconnected component.
**SOLUTION:** Observe that when the DFS tree is restricted to the edges in one biconnected component, then it is a tree that is rooted at its representative node! Our solution will be a 2 pass algorithm, and for the first pass, we only want to count the number of biconnected components and identify all the cut vertices. The algorithm is now specified by the following shell macros:

```plaintext
INIT(G(V, E; s₀)):
  clock ← 0
  count ← 0
  p[s₀] = s₀  # mark s₀ as root
  for each (u ∈ V)
    cut[u] ← false
    rep[u] ← 0
```

Previsiting v from u amounts to processing the edge u−v. The action depends on whether we have seen v before or not:

```plaintext
PREVISIT(v, u):
  If (v is unseen),
    firstTime[v] ← clock++
    p[v] ← u  # (u−v) is “tree-edge”
    mft[v] ← ∞
    If (p[u] = u)  # if u is root
      If (++numChildren > 1)
        Return(“u is cut-vertex”)  # cut vertex detected!
      Elseif (firstTime[u] > firstTime[v])  # u−v is “back edge”
        mft[u] ← min{mft[u], firstTime[v]}
```

Postvisiting u is the place where we detect cut-vertices and hence biconnected component representatives:

```plaintext
POSTVISIT(u):
  If (mft[u] ≥ firstTime[p[u]])  # cut vertex detected!
    cut[p[u]] ← true  # may be redundant
    rep[u] ← count +
```

At the end of this DFS search, the value of count is the number of biconnected components.

So far, we have not updated the array component[u ∈ V]: We want component[u] = i iff the tree edge p[u]−u belongs to the i-th component. This can be accomplished by another call to DFS. We leave this as an Exercise.

**Comments:**

6 Homework 3: Due on Thu Oct 4

NOTE: some students noted that they spent too much time preparing solutions in LaTeX. Feel free to scan your hand written solutions. Or, you can mix-and-match.

Q1. (2+2+2+2 Points) In the previous homework, we considered algorithms for locating the maxima \( x^* \in [0,1] \) of a strictly unimodal function \( f(x) \). Our algorithms know the number \( k \) of probes it can use, and we obtained optimal bounds \( A(k) = 1/F_{k+1} \) for worst case width \( b−a \) of the final interval \([a,b] \subseteq [0,1]\). We now consider a variant of this search where our algorithm is not given \( k \) in advance. We develop...
this algorithm, denoted $AL^*$, as a **stateless algorithm** because its actions are uniform, independent of the number of probes used. In this sense, binary search is also stateless.

Recall (hw-hints.pdf) the concept of “states” and the idea of a 2-player game between an algorithm $AL$ and an adversary $AD$. $AL$ begins by choosing a 3-state, and $AD$ reduces it to a 2-state. Then $AL$ extends the 2-state into a 3-state, which $AD$ again reduces to a 2-state. Each **round** of the game consists of an extension by $AL$, followed by a reduction by $AD$. After $k − 1$ rounds, the game ends, and the **value** $Val(AL, AD)$ of the game is the width of the final 2-state. Of course, $Val(AL) := \min_{AD} Val(AL, AD)$.

If $s = (a_1, \ldots, a_k)$ is a $k$-state, let its **reverse state** be denoted $s^R := (a_k, a_{k−1}, \ldots, a_1)$. Two states $s, s'$ are **proportional** if $s' = s \cdot \alpha := (\alpha a_1, \ldots, \alpha a_k)$ for some $\alpha > 0$. E.g., $(\frac{1}{3}, \frac{2}{3})$ is proportional to $(1, 2)$.

Two states $s$ and $s'$ are said to be **similar** if $s$ is proportional to either $s'$ or the reverse of $s'$.

(a) Let $s^- := \langle 2 − \phi, \phi − 1 \rangle$ where $\phi$ is the golden ratio. Show that $s^+ := \langle 2 − \phi, 2\phi − 3, 2 − \phi \rangle$ is an extension of $s^-$, and any reduction of $s^+$ is similar to $s^-$. Determine the $\alpha$ such that $s^- = \alpha s^+$ or $s^- = \alpha (s^+)^R$.

**SOLUTION:** For this question, it is useful to remember the nice relations: $\phi^2$ is just $\phi + 1$ and $1/\phi$ is just $\phi − 1$. Note that $2\phi − 3, 2 − \phi$ are positive and their sum is equal to $\phi − 1$. Thus $s^+$ extends $s^-$. We check that $\alpha s^- = s^+$ when $\alpha = \phi − 1 = 1/\phi$.

**Comments:**

(b) Use part (a) to design the stateless algorithm $AL^*$.

**SOLUTION:** The initial state of $AL^*$ will be $s^+$, and it is clear from part (a) that any reduction of $s^+$ is similar to $s^-$. Moreover, part (a) shows that we can extend any state similar to $s^-$ to another state that is similar to $s^+$. In more detail: after before $i$th step, the reduced state is equal to $\phi^{i−1} (2 − \phi, \phi − 1) = \phi^{−i} s^−$ (or the reverse of this). The algorithm will just extend this state to $\phi^{1−i} (2 − \phi, 2\phi − 3, 2 − \phi) = \phi^{1−i} s^+$. Now, the adversary can only reduce this to $\phi^{1−i} (2 − \phi, 2\phi − 3) = \phi^{1−i} s^−$ (or its reverse).

**Comments:**

(c) Determine $Val(AL^*, k)$ (the maximum width of the final state after $k \geq 1$ probes). How many probes do you need to ensure a width of at most 0.001? (Answer is an explicit number like 123)

**SOLUTION:** Part (a) established that for $k = 2$, $Val(AL^*, 2)$ is the width of the reduced state $\phi^{−1} s^−$, i.e., $Val(AL^*, 2) = \phi^{−1}$. Clearly, $Val(AL^*, 1) = 1$ and the reduced state after 1 probe is $s^-$. Inductively, suppose $Val(AL^*, 1 + k) = \phi^{−k}$ with reduced state $\phi^{−k} s^−$. When $k = 0$, this is true. Inductively, with each additional probe, we convert the state $\phi^{−k} s^−$ to $\phi^{−k−1} s^−$. This proves our claim.

Answer is 16 probes. To get $\phi^{−k} \leq 0.001$, we need $−k \leq \log(0.001)/\log_\phi$ or $k \geq \log(1000)/\log_\phi = 14.35 \cdots$. So $k = 15$, and the number of probes is $1 + k = 16$.

**Comments:**

(d) In some sense, this algorithm is “optimal”. How can we quantify this optimality? HINT: think of $A(k)$.
SOLUTION: One sense is “asymptotic optimality up to a fixed relative error”, even against an “omniscient algorithm” that knows in advance how many probes will be used. Let us use the fact (Lect.II, Exercise) that $F_k \sim \phi^k / \sqrt{5}$ as $k \to \infty$. The omniscient algorithm $A_k$ (that knows $k$) has value $A(k) = 1/F_{k+1} \sim 5/\phi^{k+1}$. On the other hand, we know $Val(AL^*, k) = 1/\phi^{-k}$. Therefore the ratio $Val(AL^*, k)/A(k)$ approaches $\sim \sqrt{5}/\phi^2 = \sqrt{5}/(1 + \phi) \sim 2.236/2.618 \sim 0.85$. This is 15% less than to the omniscient algorithm.

You might want to check how large $k$ has to be for this analysis to be correct. Further, if you give $A^*$ one extra probe, it would already exceed the omniscient algorithm!

Here is another attempt to show a “weak optimality” against any deterministic strategy that does not know the total number of probes in advance. For any such strategy, the instant it deviates from $A^*$, algorithm, then we can stop the game is at most two more steps, and $A^*$ will produce a state with superior (strictly smaller) width.

More precisely: wlog, say you deviated at the first step. So you choose start at some state $s'$ that is not $s^-$ or its reverse. Alternatively, there is a first time when you extend $\phi^{1-i}s^+$ to some state that is different than $\phi^{1-i}s^+$. After reduction, we might as well assume you have $\phi^{-i}s'$. We let you take the next step, extending $s'$ to $s'' = \langle a, b, c \rangle$. We know that either $a \neq 2 - \phi$ or $a + b \neq 2\phi - 3$.

CASE A: suppose $a < 2 - \phi$. In this case we reduce the state to $\langle b, c \rangle$, and we stop the game immediately. Thus our stateless algorithm has a strictly better performance.

CASE B: suppose $1 - c = a + b > 2 - \phi$. We can reduce the state to $\langle a, b \rangle$ and stop the game. Again $A^*$ is better.

CASE C: Suppose $a > 2 - \phi$ and $1 - c < 2 - \phi$. In this case, we can reduce it to $\langle a, b \rangle$ and continue one more step. Regardless of the next probe of the other algorithm, an adversary can reduce it to a state with width at least $a$. But $a > 1/\phi^2$, and so this is inferior to $A^*$’s performance which attains $1/\phi^2$.

Comments:

Q2. (0+3+1 Points) AVL Trees Rebalancing Acts.

(a) For your own practice, do some exercises in Lect.III §6 that require hand simulation of insertion and deletion into AVL trees.

(b) Given $k \geq 1$, what is the smallest $n = n(k)$ such that there is an AVL tree of size $n$ with the property that a deletion will cause the tree to have $k$ rebalancing acts? NOTE: a “rebalancing act” is either a single-rotation or a double-rotation under one of the cases D.a, D.b or D.c.

(c) How do you control the number of single- or double-rotations in the $k$ rebalancing acts of the previous part?

Q3. (6 Points) Let $T$ be an AVL tree with $n$ nodes. We consider the possible heights for $T$.

(a) What are the possible heights of $T$ if $n = 15$?

(b) What if $T$ has $n = 16$ or $n = 20$ nodes?

(c) Are there arbitrarily large $n$ such that all AVL trees with $n$ nodes have unique height?

SOLUTION: (a) Use the $\mu(h)$ function that tells you the minimum number of nodes in an AVL tree of height $h$. We have $\mu(3, 4, 5) = 7, 12, 20$. Thus an AVL tree with $n = 15$ nodes has height at most 4 (since $15 < \mu(5)$). But can it have height 3? Yes: a binary tree of height 3 have at most 15 nodes.

(b) What about $n = 16$ or $n = 20$? The same analysis shows that the height must be exactly 4.

(c) Answer is NO. To prove this, we use the exact formula for $\mu(h)$ in a previous exercise. Moreover, if $M(h)$ is the maximum number of nodes in binary tree with height $h$, we have the well-known formula that $M(h) = 2^{h+1} - 1$. If the answer is yes, then such $n$ must lie in the range $(M(h - 1), \mu(h + 1))$ for some $h$. But it is easy to see that for $h \geq 6$, this range is empty.

Comments:
Q4. (8 Points) We focus on the details of implementing the AVL tree algorithm. Suppose at each node \( u \) of the AVL tree, we only store its balance state, \( u.BAL \in \{-1, 0, +1\} \), not its height.
(a) Show how to maintain these fields during an insertion. In particular, write the while-loop for the REBALANCE PHASE of our algorithm.
(b) Do the same for deletion.

**SOLUTION:** Omitted.

**Comments:**

Q5. (1 Point) Suppose we have a \((5, 6)\)-search tree. What is the smallest possible value of \( c \) so that we have a valid \((5, 6, c)\)-search tree?

**SOLUTION:** Answer is \( c = 4 \). From the text, we know that \( c = 4 \) will do. But we can check that \( c = 3 \) will fail.

**Comments:** This implies that \( c = 2 \) will also fail.

Q6. (16 Points) Consider tradeoffs among the following 4 schemes to organize the nodes of an \((a, b)\)-search tree:

(i) an array, (ii) a singly-linked list, (iii) a doubly-linked list, (iv) a balanced binary search tree.

Use a specific numerical example: block size is \( 4096 = 2^{12} \) bytes, and each block pointer is 4 bytes, and each key 6 bytes. A local pointer within the block uses 12 bits, but for simplicity treat this as two bytes.

(a) What is the maximum value of the parameter \( b \) under each of the schemes (i)-(iv)? Be sure to show your calculations. The text has some discussion of these issues.

(b) What is the worst case time to search for a key in an \((a, b)\)-search tree with two million items? Your answer must be an explicit number (like 123, not an expression like \((2^{13} + 7)/3\)) for each schemes. The time unit is CPU cycles (or “CPUC”).

(c) Draw some general conclusions.

**PLEASE MAKE THESE ASSUMPTIONS**

**Be sure to state any other assumptions you need.**

Assume \((a', b') = (a, b)\), i.e., number of items in leaves is between \( a' \) and \( b' \). Each disk I/O takes 1000 CPU cycles. If searching for a key takes \( O(\lg n) \) or \( O(n) \) CPU time, always assume that “4” is the constant in big-Oh notation. E.g., searching for a key in a balanced BST with \( n = 100 \) keys takes \( 4 \lg n = 4 \times 7 = 28 \) CPUC. Searching for in a list of length \( n = 100 \) takes \( 4n = 400 \) CPUC. The root of the search tree is always in main memory, so you never need to read or write the root. Assume \( a = \lfloor (b + 1)/2 \rfloor \). You can use calculators, but I encourage you to make calculator-free (simplified) estimates whenever possible (e.g., log base 2 of one million is 20). We do not mind if your answer is off by some small constants.
SOLUTION  (a) Regardless of which method (i)-(iv) is used, we need to keep track of
the degree of the node (2 bytes), and parent pointer (4 bytes). Note that knowing
the degree is useful to check if a node is overfull or underfull. So the available memory in
each block is 4096 − 6 = 4090.

(i) Array representation. With \( b \) children, we need \( 10b \) bytes. So \( 10b \leq 4090 \). So we can
choose \( b = 409 \).

Free List Management. Before providing the solution for (ii)-(iv), we note a common
feature of these three solutions: you want to divide the space in a block into a set of
nodes of some fixed size. Each node is either used or free. When you insert into your
data structure (linked list, doubly-linked list or BST), you need to get a free node. When
you delete, you need to create a free node. Thus, you must manage the set of free nodes,
and the usual way is to put them together into a linked list called a FREELIST. In
this FREELIST, you can use one of the local pointers in the nodes to get to the next
free node. Therefore, the management of FREELIST has no impact on the size of your
nodes. But you need to allocate 2 bytes in the block to point to the beginning of the
FREELIST; so each block now has 4088 bytes.

(ii) Singly-linked list. Each node of the linked list needs 12 bytes: 4 bytes for a block
pointer, 6 bytes for the key, and and 2 bytes for the next node local pointer. Thus the
number of 12\( b \) \leq 4088. So the largest possible choice is \( b = 340 \).

(iii) Doubly-linked list. Each node needs 14 bytes, because we need an extra local pointer
compared to (ii). Thus the maximum value of \( b \) is \lfloor 4088/14 \rfloor = 292.

(iv) Balanced binary search tree. For our purposes, assume it is an AVL tree. Each node
needs 16 bytes: this is two bytes more than (iii) because each node needs a left-, right-
and parent pointer. So the number of bytes is 16, and a simplistic answer for this part
is that \( b = \lfloor 4088/16 \rfloor = 255 \). But this solution is too optimistic. Let us see why. Each
AVL node needs to maintain two bits of balance information (we might as well allocate
one byte to this. A more important issue is this: when you lookup a key \( K \), you get
back the closest key \( K_i \) in the BST. We may have \( K < K_i \) or \( K \geq K_i \). Thus, you need
to get to either \( P_i \) or \( P_{i-1} \) where \( P_j \)'s are the pointers to children nodes of the \((a, b)\)-
search tree. Suppose you store \( P_i \) with \( K_i \). But how to you get to \( P_{i-1} \)? The simplest
solution is to assume that your AVL nodes also has a predecessor pointer (these are
local pointers). But in order to maintain predecessor pointers, we also need to maintain
successor pointers. That means that each AVL node needs, not 16 bytes, but 21 bytes
(1 byte for balance info, four bytes for succ/pred pointers). So \( b = \lfloor 4088/21 \rfloor = 194 \) is a
more realistic figure.
Q7. (10 Points) We consider the worst case behavior of insertion and deletion in \((a, b)\)-search trees. Assume the standard insertion and deletion algorithms (not the enhanced ones). We are interested in counting the number of blocks that must be written into memory. For instance, in a LookUp, we must read \(h\) blocks of memory if the height of the tree is \(h\). Finally, assume \(a\) to be stored in one block in the disk. The root is always in main memory, so does not need to be written.

(a) Suppose \(b = 2a - 1\). For each \(h\) and \(n\), construct a \(B\)-tree of height \(h\) and a sequence of \(2n\) requests

\[(D_1, I_1, D_2, I_2, \ldots, D_n, I_n)\]

such that these requests results in \(3nh + O(1)\) block writes. Here \(D_i\) is a deletion and \(I_i\) is an insertion \((i = 1, \ldots, n)\).

(b) Suppose \(b = 2a\). The potential \(\phi(u)\) of node \(u\) in the \((a, b)\)-search tree is defined to be

\[
\phi(u) = \begin{cases} 
4 & \text{if } \text{degree}(u) = a - 1 \\
1 & \text{if } \text{degree}(u) = a \\
0 & \text{if } a < \text{degree}(u) < b \\
3 & \text{if } \text{degree}(u) = b \\
6 & \text{if } \text{degree}(u) = b + 1 
\end{cases}
\]

Note that during insertion (resp., deletion), we temporarily allow nodes to have degree \(b + 1\) (resp., \(a - 1\)). Moreover, the potential of the \((a, b)\)-tree is just the sum of the potential of all the nodes in the tree.

(b.1) Show that each split, borrow, or merge operation results in the reduction of (at least) 2 units of potential in the tree.

(b.2) Conclude that if we are given any sequence of \(n\) requests

\[(R_1, R_2, R_3, \ldots, R_n)\]

where each \(R_i\) is either an insert or a delete, the total number of block writes is only \(O(n)\).
SOLUTION:  (a) This is easy: just arrange for a single path in the tree to have nodes of degree $b$. Delete any key in this path, and we will cause each node on this path to split into two. Each of these blocks must be written, and hence it costs $2h$ writes. Next insert the same deleted key. If we take care, we can cause every node along this path to be merged, causing $h$ block writes.

(b.1) We check the maximum $\Delta \phi$ (the increase in potential) is for each operation. Note that $\Delta \phi$ is negative if there is decrease in potential. The maximum increase in potential is therefore the minimum decrease in potential.

- **SPLIT**: $b + 1 \to (a, a + 1)$, and the degree $p$ of parent increases by 1.
  
  In the worst case, the $p = b$ and
  
  $\Delta \phi \leq (-6 + 1 + 0) + (-3 + 6) = -2$

- **BORROW**: $(a - 1, d) \to (a, d - 1)$, and in the worst case $d = a + 1$.
  
  $\Delta \phi \leq (-4 - 0 + 1 + 1) = -2$.

- **MERGE**: $(a - 1, a) \to b - 1$, and the degree $p$ of parent decreases by 1.
  
  In the worst case, the $p = a$ and
  
  $\Delta \phi \leq (-4 - 1 + 0) + (-1 + 4) = -2$.

(b.2) Each insert and delete can be reduced to these split/borrow/merge operations.

INSERTION: At the beginning of an insert, we may need to change a node of degree $b$ to degree $b + 1$, or from degree $b - 1$ to degree $b$. In either case, we increase the potential by $-3 + 6 = 3$. THEREFORE WE MUST CHARGE THE INSERTION 3 units of potential for this initial insertion. Thereafter, all the operations can pay for themselves because of the release of potential. (Think of potential as a bank account, so decreasing the potential as drawing from the bank account.) But we did not discuss the root operation which is special. Since this is a singular event, we can directly pay for this case as well. Overall, we pay only a constant amount of potential.

DELETION: At the beginning of a deletion, we may change a degree from $a + 1 \to a$, or $a \to a - 1$. The increase in potential is 1 or 3, respectively. As before, we must pay for this increase in potential, and also for operations at the root. All other operations pay for themselves.

Comments:

7 Homework 2: Due on Thu Sep 20

Most problems are based on Lecture II on Recurrences, but a few on Lecture I. For some problems, especially geometric ones, feel free to draw figures to illustrate your arguments. You can scan hand-drawn figures to include them into your latex files. I use the free software called ”xfig” and would be happy to share with you my macros if you ask me.

Q1.  · (0 Points) Ex.II.1.3. Choice of DIC can affect the $\Theta$-order of solution. NOTE: please avoid using DIC that leads to the trivial solution.
SOLUTION: Use the “generalized exponential function”
\[
\exp(n, x) = \begin{cases} x & \text{if } n = 0 \\ \exp(n-1, 2^x) & \text{if } n > 1 \end{cases}
\]
Consider the recurrence
\[
T(n) = \exp(T(n-1), n)
\]
using two different initial conditions: \(T(n) = 1\) for \(n \leq 1\) and \(T(n) = 2\) for \(n \leq 1\). Let \(T_1, T_2\) be the respective solutions. Then we see that \(T_1(2) = 2^2, T_2(2) = 2^2 = 16\). To show that \(T_1\) and \(T_2\) do not have the same \(\Theta\)-order, it suffices to show that \(T_2(n) \geq 2^n T_1(n)\) for all \(n \geq 2\). This is true for \(n = 2\): \(T_2(2) = 16 = 2^2 T_1(2)\). For \(n \geq 2\),
\[
T_2(n + 1) = \exp(T_2(n), n + 1) \geq \exp(2^n T_1(n), n + 1) \geq 2^{n+1} \exp(T_1(n), n + 1) = 2^{n+1} T_1(n + 1).
\]

Comments:

- (1+1+2 Points) Ex.II.3.3. Use the EGVS Method to solve the following recurrences
  (a) \(T(n) = n + 8T(n/2)\).
  (b) \(T(n) = n + 16T(n/4)\).
  (c) Can you generalize your results in (a) and (b) to recurrences of the form \(T(n) = n + aT(n/b)\) when \(a, b\) are in some special relation?

SOLUTION: (a) By EGVS, \(T(n) = n \sum_{j=0}^{i-1} 4^j + 8^i T(n/2^i)\) after \(i\) expansion steps. By DIC, and choosing \(i = \lceil \log_2 n \rceil\), we get \(T(n) = n \sum_{j=0}^{\log_2 n} 4^j = \Theta(n 4^{\log_2 n}) = \Theta(n^3)\).

(b) By EGVS, \(T(n) = n \sum_{j=0}^{i-1} 4^j + 16^i T(n/4^i)\) after \(i\) expansion steps. By DIC, and choosing \(i = \lceil \log_4 n \rceil\), we get \(T(n) = n \sum_{j=0}^{\log_4 n} 4^j = \Theta(n 4^{\log_4 n}) = \Theta(n^2)\).

(c) EGVS gives \(T(n) = n \sum_{j=0}^{i-1} (a/b)^j + a^i T(n/b^i)\). Setting \(i = 1 + \lfloor \log_b n \rfloor\), by DIC, we get \(T(n) = n \sum_{j=0}^{\log_b n} (a/b)^j\). In (a) and (b), we use the fact that this is an exponential increasing sum. In order to use this, we need the relation
\[
a > b.
\]
In that case, we conclude that \(T(n) = n \Theta((a/b)^{\log_b n}) = n \Theta(n^{\log_b(a/b)}) = n \Theta(n^{\log_b(a)})\).

Comments: What if you want the exact constants in part (a)? Note that the largest term in this sum corresponds to \(j = \lfloor \log n \rfloor\), not \(j = \log n\). Let \(\epsilon = \log n - \lfloor \log n \rfloor\). \(T(n) = n \sum_{j=0}^{\lfloor \log n \rfloor} 4^j = n \left \lfloor 4^{\log n-1} \right \rfloor = n 4^{\log n-\epsilon} = n 4^{-\epsilon} / 3 - O(n)\). Thus, \(T(n) \sim C n^3\) where \(C = 4^{-\epsilon} / 3\).

What if \(a < b\) in part (b)? There are two possibilities: If \(a = b\), then we get \(T(n) = \Theta(n \log n)\). If \(a < b\), we get \(T(n) = \Theta(n)\).

- (0 Points) Ex.II.4.1. Proof of Theorem 1 on Real Induction.
- (2 Point) Ex.II.4.4. Solving by Real Induction. Sorry, but we want you to show the REAL BASIS (not just invoke our theorem about growth functions which would render the REAL BASIS automatic).
SOLUTION: The lower bound is easy. So let us give the upper bound. The naive induction hypothesis would be \( T(x) \leq K x^{\log_5} \) (ev.). But you quickly realize that we need to strengthen our hypothesis (we saw that in class, when we proved the hard case of the Multiterm Master Recurrence). We need an upper bound hypothesis of the form \( T(x) \leq K x^{\log_5} - K' x^2 \) for positive \( K, K' \). It is easy to figure out a lower bound on \( K' \) in order to maintain the hypothesis. You can choose \( K' \) to be this lower bound from the start if you like, but it is easier to derive it as needed. The choice of \( K \) will depend on \( K' \) (to get the induction basis). For now, let us focus on the inductive step:

\[
T(x) = 5T(x/2) + x^2 \leq 5[K(x/2)^{\log_5} - K'(x/2)^2] + x^2 = K x^{\log_5} - ((5K'/2^2) - 1)x^2
\]

provided \( K' \leq (5K'/2^2) - 1 \) or \( 1 \leq K'(5/4 - 1) \). We may choose \( K' = 4 \).

We now need to get the Real Basis for \( T(x) \leq K x^{\log_5} - 4x^2 \). I suggest picking \( K = 4 \);
so we want to show \( T(x) \leq 4(x^{\log_5} - x^2) \). By DIC, assume that \( T(x) = 0 \) for all \( x \leq 2 \).
Thus the Real Basis is verified if we choose the cut-off constant to be \( x_1 = 2 \).
Next, for \( x > x_0 = 2 \), we have \( x/2 \geq 1 \). So we can choose the gap constant to be \( \delta = 1 \).
With these constants, the Real Induction is verified by our preceding derivation. The lower bound by real induction is easy, omitted.

Comments:

- (0 Points) Ex.II.5.2. Cute problem (or proof by pictures).
- (2 Points) Ex.II.5.13. Applications of range and domain transformations.
- (3 Points) Ex.II.6.1. Growth types of functions.
- (1 Point) Ex.II.6.7. Use the method of grouping to show that \( S(n) = \sum_{i=1}^{n} \frac{\log i}{i} \) is \( \Omega(\log^2 n) \).

SOLUTION: A simple way to do this is to break up the sum into two groups, \( A(n) := \sum_{i=1}^{\sqrt{n}} \frac{\log i}{i} \) and \( B(n) := S(n) - A(n) \). Then

\[
S(n) \geq B(n) \geq \log \sqrt{n} \sum_{i \geq \sqrt{n}} \frac{1}{i} \geq \frac{1}{2} \log n(H_n - H_{\sqrt{n}}).
\]

The desired lower bound follows since

\[
H_n - H_{\sqrt{n}} \geq \ln n - \ln \sqrt{n} - 2 \geq \frac{1}{2} \ln n - O(1)
\]

Comments: The above method depends on knowing that \( H_n = \ln n + g(n) \) where \( 0 < g(n) < 1 \). A “thorough going” grouping solution would be to emulate the proof in Lect.II.¶19 on Harmonic series where we showed how to break \( H_n \) into \( N = \log n \) groups, each summing to a number between 1/2 and 1. But for the present summation, the \( i \)th group sum up to \( \Theta(i) \). Since there are \( N \) groups, the sum is \( \Theta(N^2) = \Theta(\log^2 n) \).

Q2. (10 Points) Let \( S \) be a set of \( n \) distinct points in the plane. All our lines \( \ell \) in the plane are assumed to be directed; so each line has an orientation (angle) \( \theta \in [0, 2\pi) \). Therefore, we may speak of the open left half-plane \( \ell^+ \) and the open right half-plane \( \ell^- \). HINT: feel free to draw diagrams to illustrate your proofs.

(a) A line \( \ell \) is a bisector of \( S \) if \( |\ell^+ \cap S| \leq n/2 \) and \( |\ell^- \cap S| \leq n/2 \). A bisector is simple if, in addition, \( |\ell \cap S| \leq 1 \), and \( |\ell^+ \cap S| = |\ell^- \cap S| \). Show that simple bisectors exist for any set \( S \).

SOLUTION: Easy and omitted. Indeed, if \( \ell \) is a bisector, then \( |\ell \cap S| = 1 \) iff \( n \) is odd.
Moreover, except for \( \binom{n}{2} \) directions, there is a simple bisector with any given direction.

Comments:
(b) Let $b$ be a bisector of $S$. Then $\ell$ is a (simple) **conjugate bisector** of $(S, b)$ if $\ell$ is simultaneously a (simple) bisector of $S \cap b^+$ and of $S \cap b^-$. It is also known as a “ham sandwich cut” of the two sets $S \cap b^+$ and $S \cap b^-$. **Show that conjugate bisectors of $(S, b)$ exist.** Moreover, if $S$ is in general position (i.e., no three points lie on a line) then simple conjugate bisectors exist for $(S, b)$.

**HINT:** a direction $\theta \in [0, 2\pi)$ is said to be **exceptional** for $S$ if there is a (directed) line with direction $\theta$ that passes through two or more points of $S$. For any direction $\theta$ that is non-exceptional for $S$, consider the bisector $\ell(\theta)$ of the set $S \cap b^+$. If $\ell(\theta)$ does not contain any points of $S \cap b^+$, then $\ell(\theta)$ is non-unique; we can make it unique by specifying that the distance of $\ell(\theta)$ from the set $\ell(\theta)^+ \cap S \cap b^+$ is equal to its distance from the set $\ell(\theta)^- \cap S \cap b^+$. Moreover, even when $\theta$ is exceptional, we can define $\ell(\theta)$ so that the parametrized line $\ell(\theta)$ changes continuously with $\theta$. Now, consider the fraction

$$f(\theta) = \frac{\ell(\theta)^- \cap S \cap b^-}{\ell(\theta)^+ \cap S \cap b^+}.$$ 

Wlog, assume $b$ has direction 0. How does $f(\theta)$ change as $\theta$ changes 0 to $\pi$?

**SOLUTION:** $f(0) = |S \cap b^-|/0 = \infty$. $f(\pi) = 0/|S \cap b^-| = 0$. Moreover, $f(\theta)$ is monotonic in $\theta$; specifically, $f(\theta)$ is non-increasing as $\theta$ increases. Intuitively, there must be some $\theta_0$ when $f(\theta_0) = 1$. The corresponding line $b(\theta_0)$ is our conjugate bisector. But there are some some subtleties.

Let us observe that the fraction $f(\theta) = a/b$ for some $a + b \leq n/2$. By general position of $S$, $a + b \geq (n/2) - 2$. Suppose $f(\theta)$ decreases from some $\frac{a}{b} > 1$ to some $\frac{a}{b} \leq 1$. If $\frac{a}{b} = 1$, we are done. If not, we note that $\frac{a}{b}$ can be one of

\[
\frac{a - 1}{b}, \frac{a - 2}{b}, \frac{a}{b + 1}, \frac{a}{b + 2}.
\]

We must show that in each case, we can find a conjugate bisector, perhaps by perturbing $\ell(\theta)$. There are two cases: (A) Suppose $a'/b' = (a - 1)/b$. From the fact that $a/b > 1 \text{ and } (a - 1)/b \leq 1$, we conclude $a > b$ and $(a - 1) \leq b$. Thus $a - 1 - b$. Thus $a' = (a - 1)/b = 1$. Similarly, if $a'/b' = a/(b + 1)$, we conclude that $a/(b + 1) = 1$.

(B) Suppose $a'/b' = (a - 2)/b$. If $a'/b' < 1$ this means that $(a - 1)/b = 1$. Let us suppose $f(\theta) = (a - 2)/b$. Thus $\ell(\theta)^- \cap S \cap b^- = a - 2$ and $\ell(\theta)^+ \cap S \cap b^- = b$ and $|\ell(\theta)^- \cap S \cap b^-| = 2$. Thus the $\theta$ is exceptional for $S$. Since $S$ is non-degenerate, $\ell(\theta)$ does not intersect any point of $S \cap b^+$. So, if $\ell'$ is a sufficiently small perturbation of $\ell(\theta)$, then $\ell'$ is still a bisector of $S \cap b^+$. We can choose this $\ell'$ so that $|\ell'^- \cap S \cap b^-| = a - 1$ and $|\ell'^+ \cap S \cap b^-| = b$ and $|\ell(\theta) \cap S \cap b^-| = 1$. This proves that $\ell'$ is a conjugate bisector of $(S, b)$.

**Comments:** Note that we need to assume that $S$ is in general position because of Huck’s example (say $S$ are four points that lie on a line). Then, even if $S$ is not in general position, a conjugate bisector of $(S, b)$ will always exist, but it is not necessarily simple.

(c) A **conjugate tree** for a set $S$ is a full binary tree $B$ such that

- Each node $u$ of $B$ is a pair of the form $u = (S', b')$ where $b'$ is a bisector of $S'$. We call $S'$ and $b'$ the **underlying set** and **bisector** at node $u$.
- The underlying set at the root is $S$, and the underlying set at any leaf is a singleton.
- If $u = (S', b')$ is a node and $v$ is the left (resp., right) child, then the bisector at $v$ is the conjugate bisector of $(S', b')$. and the underlying set at $v$ is $S' \cap b'^+$ (resp., $S' \cap b'^-$).
- If $u, v$ are siblings, then the bisectors at $u$ and $v$ are the same.
Let $B$ be a conjugate tree for $S$. The **half-space point retrieval** problem is to compute the set $S \cap \ell^+$ for any given “query” line $\ell$, using the data structure $B$. Make any reasonable assumptions needed in terms of data structures and $B$. Consider the recurrence

$$T(n) = 1 + T(n/2) + T(n/4).$$  \hfill (32)

Remark: our lecture notes show that $T(n) = \Theta(n^{0.69\ldots})$. Describe an algorithm which, for any “query” line $\ell$, takes $\Theta(T(n))$ time in order to output the set $S \cap \ell^+$.

**SOLUTION:** When searching the conjugation tree $B$ rooted at $(S, b)$, we can look at the bisector $b'$ at a child of $(S, b)$. One of the four quadrants defined by $(b, b')$ is fully contained in $\ell^+$ or fully contained in $\ell^-$. Let $v$ denote the grandchild of $(S, b)$ corresponding to this quadrant. The underlying set at $v$ is therefore contained in $S \cap \ell^+$ or disjoint from $S \cap \ell^-$. We mark the node $v$ provided the underlying set is contained in $S \cap \ell^+$. We then recursive search the $s$ sibling of $v$ and the uncle $u$ of $v$. Note that $u$ is a child of the root, and its underlying set has size at most $n/2$; similarly, the underlying set at $s$ has size at most $n/4$. At the end of this algorithm we have a list of all the marked nodes, and the list of marked nodes is an implicit representation of $S \cap \ell^+$. Note also that $T(n)$ counts the number of such marked nodes. Hence, if $T(n)$ is the time to search the node $(S, b)$, then it satisfies the recurrence (32).

**Comments:**

Q3. (3 Points) Consider the following recurrence originating in the Strassen-Schönhage work on integer multiplication:

$$T(n) = 2\sqrt{n}T(\sqrt{n}) + n \log n$$

(a) How many domain and range transformations does it take to bring this to “standard form”? Show these transformations.

(b) Solve for $T$ using the transformations in (a).

**SOLUTION:** (a) Answer is 4. We need a range and domain transform to bring it to the form $t(N) = 2t(N/2) + N$ where $t(N) = T(2^N)/2^N$. Note that $t(N)$ transforms both the range and the domain of $T(N)$. This is an instance of the Master Recurrence, and we know that with 2 more transformations, we get the standard form. (b) By the Master Theorem, we solve for $t(N)$: $t(N) = N \log N$. Plugging back, $T(n) = Nt(\log N) = N \log N \log \log N$.

**Comments:**

Q4. (8 Points) Recall the unbounded search problem discussed in class. Given an unknown $x^* \in \mathbb{R}$, you want to find an interval $[a, b]$ containing $x^*$ such that $b - a \leq 1$. When this happens, we say $x^*$ is located. You are allowed queries of the form “Is $x^* \leq x$?” and the answer is either YES or NO. Denote this query by “[x* : x]”. How many queries do you need? NOTE: in this problem, we are interested in complexity functions up to (what I will call) “+-dominance”. We define “$f \lesssim g” to mean that $2^f \leq 2^g$. This definition extends to “$f \lesssim g” in the natural way. For instance, $\log(2n) \lesssim \log(n)$ but $\log(n) \lesssim 2 \log(n)$. Intuitively, we distinguish functions up to an additive constant. The usual notion of dominance distinguish functions up to a multiplicative constant.

(a) Describe an algorithm $A_1$ achieving the bound $T_1(x^*) = 2\log |x^*| + O(1)$.

(b) Describe an algorithm $A_2$ achieving the bound $T_2(x^*) = \log |x^*| + 2\log \log |x^*| + O(1)$.

(c) What is the general scheme for these algorithms?
Q5. (10 Points) Recall the \textbf{k-th selection problem}: given a set \(S\) of \(n\) distinct numbers, we want to find the \(k\)-th largest, i.e., the element in \(S\) smaller than \(k-1\) other elements and larger than \(n-k\) other elements in \(S\). We described the general scheme for this problem in class (if you missed class on Thur Sep 13, please check with me on this). The scheme is simply parametrized by the size \(t\) of each ”group”
Q6. (10 Points) Suppose \( f : \mathbb{R} \rightarrow \mathbb{R} \) is a continuous and total function. You are told that \( f \) is strictly unimodal inside the interval \([0, 1]\) and it attains its maximum at a unique point \( x^* \in (0, 1) \). Strictly unimodal means \( f \) is strictly increasing before \( x^* \) and strictly decreasing after \( x^* \). E.g., \( f(x) = -(x - 1/2)^2 \) and \( x^* = 1/2 \). Our goal is to locate an interval \([a, b] \subseteq [0, 1]\) that contains \( x^* \), where \( b - a \) is as small as possible. Here, \( b - a \) is just the width of the interval \([a, b]\). You are allowed to “probe” the function \( f \) at any point \( x \) by evaluating \( f(x) \).

(a) Prove this key observation: suppose you know the values of \( f \) at four points, \( x_1 < x_2 < x_3 < x_4 \) and \( x^* \in [x_1, x_4] \). Then you can either conclude that \( x^* \) lies in \([x_1, x_3]\) or conclude that \( x^* \) lies in \([x_2, x_4]\).

(b) Let \( A(k) \) denote the infimum of the value of \( b - a \) that can be obtained with \( k \) probes. Please determine the values of \( A(4), A(5), \) and \( A(6) \). Of course, tell us the algorithm that achieve these bounds.

(c) Give the best upper bound on \( A(k) \) you can obtain for a general \( k \).
Let us show that also proved that as in AL3, yielding a value of 1

The width to less than 1 initial 3-state SOLUTIONS: the slope

A state after these reductions is s (ii) If /b, c / (5) ≥ |2. This means final width is at least 3. This means final width is at least |x|, by describing an adversary AD4: suppose the initial state is (a, b, c). There are two possibilities: (i) If a ≤ 2/5, then AD4 will choose the R-reduction (b, c). Since A(3) = 1/3, we know that no algorithm using 2 additional probes can reduce the width to less than 1/3 of b + c; since (b + c) ≥ 3/5, the final width is at least 1/5.

(ii) If a + b ≤ 3/5, then AD4 will for the next two rounds, choose R-reductions. If the state after these reductions is s, then |s| ≥ 2/5 (since a + b ≤ 3/5). Since A(2) = 1/2, we know that with one more probe, no algorithm can reduce |s| by a factor greater than 1/2. This means final width is at least |s|/2 ≥ 1/5. (iii) if a + b > 3/5 then c < 2/5, and the arguments of (i) implies the final width is at least 1/5. This completes our proof that A(4) ≥ 1/5.

We now A(5) = 1/8. First, we show A(5) ≤ 1/8: our algorithm AL5 will choose the initial 3-state (3/8, 2/8, 3/8). This will reduce to the 2-state (3/8, 2/8) (wlog). Now AL4 will extend this to the 3-state (1/5, 1/5, 1/5), and we know that this can be completed as in AL3, yielding a value of 1 + ε (for any ε > 0). Thus A(4) ≤ 1/5.

Let us show that A(5) ≥ 1/8, by describing an adversary AD5: suppose the initial state is (a, b, c). There are two possibilities: (i) If a ≤ 3/8, then AD5 will choose the R-reduction (b, c). Since A(4) = 1/5, we know that no algorithm using 3 additional probes can reduce the width to less than 1/5 of b + c; since (b + c) ≥ 5/8, the final width is at least 1/8.

(ii) If a + b ≤ 5/8, then AD5 will for the next two rounds, choose R-reductions. If the state after these reductions is s, then |s| ≥ 3/8 (since a + b ≤ 5/8). Since A(3) = 1/3, we know that with two more probes, no algorithm can reduce |s| by a factor greater than 1/3. This means final width is at least |s|/3 ≥ 1/8. (iii) if a + b > 5/8 then c < 3/8, and the arguments of (i) imply the final width is at least 1/8. This completes our proof that A(5) ≥ 1/8.

You should now be able to generalize these arguments to show A(6) = 1/13.
(c) Using the fact that $F_0 = 0$, $F_1 = 1$ and $F_{k+1} = F_k + F_{k-1}$, we see from the above derivation that $A(k) = 1/F_{k+1}$ for $k = 2, 3, 4, 5, 6$. It is clear that we can generalize this argument (do it!) for all $k$.

**Comments:** (I) Indeed, you can generalize the above algorithm to AL$k$ and adversary to AD$k$ for any $k$. Together, they prove that $A(k) = 1/F_{k+1}$ where $F_{k+1} = F_k + F_{k-1}$ for $k \geq 1$ and $F_i = i$ for $k = 0, 1$. Thus, this is called the Fibonacci search.

(II) We had viewed the bound $A(k)$ as a purely combinatorial game. It is not obvious that there exists unimodal functions $f_k(x)$ that enforces the lower bound $A(k) \geq 1/F_{k+1}$ required by our adversary AD$k$. This has been done by Kiefer (1953) who invented Fibonacci search.


(IV) Note that the above algorithm AL$k$ depends on the parameter $k$. If you do not know $k$ in advance, there is a simpler and continuous version of the algorithm where you begin with the initial state $(2 - \phi, 2\phi - 3, 2 - \phi)$ where $\phi$ is the golden ratio.
8 Homework 1: Due on Thu Sep 13

Recall that you must always justify your answers or give proofs. Do not hand in problems with 0 points. All problems are in Lecture I, Introduction to Algorithmics. We say write “Ex.I.3.2” (say) in referring to Exercise 3.2 in Lecture I.


- (0 Points) Ex.I.3.1. We interpreted the programs in Figure 3(a) and (b) as “algorithms for finding the maximum of \{x, y, z\}”. But the notion of an “algorithm” is a semantical concept. So the same programs can be given different interpretations. Please give a different interpretation to these two programs. I.e., view them as solving a different problem.

![Comparison Tree](attachment://comparison_tree.png)

![Comparator Circuit](attachment://comparator_circuit.png)

\textbf{SOLUTION:} Consider the problem \( P' \) of computing any \textit{non-maxima} of \( x, y, z \). We can interpret both programs as algorithms for this non-maxima problem \( P' \). We just have to change the output to return \textit{any} element different than the maximum. In Figure 3(b) for the circuit model, we had tacitly assume the output element is \( x' \). Now, we can assume the output element is \( z' \) (or \( y' \)). Actually, there is a problem with this interpretation when not all the input elements are distinct. For instance, if \( x = y = z \), which element do we output as “non-maxima”? You need to address this issue explicitly.

One simple solution is to declare that the elements must be all distinct. Another (better) solution is to declare that if \( x = y \) then \( y \) should be considered as “smaller” than \( x \). More generally, identify \((x, y, z)\) with \((x_1, x_2, x_3)\), and if \( x_i = x_j \) then we break-ties by declaring \( x_i < x_j \) iff \( i > j \). Now, we know how to specify the correct output even when the numerical values of \( x_1, x_2, x_3 \) are not distinct.

For the circuit model, when two input wires to a comparator have the same numerical value, we assume the element in the top input wire is carried out to the top output wire. With this convention, we can now specify that the output wire \( z' \) is a non-maxima.

\textbf{Comments:} Some students say that the tree program solves the problem of deciding “if \( z \) the largest among \( x, y, z \)?”. So we would put a YES or NO at each leaf, accordingly. That is a valid answer for part (a). But in the circuit model, we cannot such such a problem.

- (0 Points) Ex.I.3.2.

(a) Extend the comparison tree in Figure 3(a) so that it sorts three input elements \( \{x, y, z\} \).

(b) Extend the comparator circuit in Figure 3(a) so that it sorts three input elements \( \{x, y, z\} \).

\textbf{SOLUTION FIGURE:}

\textbf{SOLUTION:} See Figure 4(a) and (b). These are clearly extensions of the corresponding algorithms in Figure 3.

\textbf{Comments:}
• (0 Points) Ex.I.3.3. Design tree programs for four elements $a, b, c, d$:
  (a) To find the second largest element. The height of your tree should be 4 (the optimum).
  (b) To sort the four elements. The height of your tree should be 5 (the optimum).  

SOLUTION:

Figure 5: Tree program to find second largest of $a, b, c, d$

SOLUTION: (a) The solution is shown in Figure 5. We use symmetry to avoid drawing the entire tree. To understand the solution, we suggest you draw the partial order on $a, b, c, d$ at each node of the tree.

Comments:

• (5 Points) Ex.I.3.4. (a) Show that the median of 4 elements can be computed with 4 comparisons in the worst case.
  (b) Show that the median of 5 elements can be computed with 6 comparisons in the worst case.

HINT: you could use your solution in part(a) for this part.

NOTE: the median of a set $X$ of elements is the element of rank $\lceil |X|/2 \rceil$. An element has rank $k$ if it is smaller than or equal to $k - 1$ other elements and larger than or equal to $|X| - k$ other elements. Thus rank 1 is the largest, and rank $|X|$ is the smallest element. In case the elements are non-distinct, an element could have several ranks. For instance, if all the elements are identical, then each element could have ranks 1 to $|S|$. 

43
SOLUTION:  
(a) The median of four elements is the element of rank 2, i.e., the second largest. Let the 4 elements be \( X = \{a, b, c, d\} \). Suppose we begin with 2 comparisons: \( a : b \) and \( c : d \). Wlog, say \( a > b \) and \( c > d \). Then we compare the winners \( a : c \), and wlog, say \( a > c \). The resulting partial order has a Hasse diagram representation that we propose to call \( H_4 \). Now, \( a \) cannot be the median of \( X \); moreover, the median is either \( b \) or \( c \). Using one more comparison \( b : c \), suppose \( b > c \). Then \( b \) is the median. Check that we have used a total of four comparisons.

(b) Let the five elements be \( Y = \{a, b, c, d, e\} \). Using (a), we can use the \( H_4 \) structure on the set \( X = \{a, b, c, d\} \) using 3 comparisons, starting with the comparisons \( a : b \) and \( c : d \), followed by comparing the winners of these two comparisons. Assume \( a \) is the largest element in this structure, we know that it cannot be the median of \( Y \). Discarding it, our goal is now to compute the median of \( Y \); the median of \( Y \) is the 2nd largest (median) of the set \( Y' = \{b, c, d, e\} \). By part (a), we could do this with four comparisons, starting with the pair of comparisons \( c : d \) and \( b : e \). But we need not perform the comparison \( c : d \) since this has been done in computing the \( H_4 \) structure of \( X \). Therefore, three more comparisons suffice. This gives a total of 6 comparisons.

Comments:

- Ex.I.3.6. Design a tree program to merge two sorted lists \((x, y, z)\) and \((a, b, c, d)\). The height of your tree should be 6 (the optimum).

SOLUTION FIGURE:

![Tree Program](image)

Figure 6: Tree program to merge \( x < y < z \) with \( a < b < c < d \)

SOLUTION: The solution is shown in Figure 6, partially drawn. To design the tree program, you can just “unroll” the Merge Algorithm in the text. To see that 6 is the best possible, you can either use the information theoretic bound (ITB) or adversary argument, both in the next section. For ITB, note that any tree has at least \( \binom{7}{3} \) leaves because the sorted order on the 7 elements are completely determined once we determine which 3 of the sorted output comes from the set \( \{x, y, z\} \). The height is therefore at least \( \lceil \lg \binom{7}{3} \rceil = 6 \).

Comments:


- (0 Points) Ex.I.4.1.
- (0 Points) Ex.I.4.2.
- (5 Points) Consider this problem (it is a more open ended form of Ex.I.4.3): Let \( L(n) \) denote the minimum number of leaves in any comparison tree that sorts \( n \) elements. Clearly, \( L(n) \geq n! \) by the ITB argument. Give the best upper bound you can for \( L(n) \) when \( n = 3, 4, 5 \).
SOLUTION: Actually, we can show that \( L(n) = n! \) for all \( n \). Proof by induction: let \( T(n) \) denote a comparison tree that sorts \( X_n = \{x_1, x_2, \ldots, x_n\} \) using \( n! \) leaves. This is trivial for \( n = 1, 2 \). Then we can construct \( T(n+1) \) as follows: let \( B(n+1,Y) \) be the binary tree that inserts \( x_{n+1} \) into a sorted list \( Y \) on \( x_1, x_2, \ldots, x_n \). Clearly, \( B(n+1,Y) \) has exactly \( n+1 \) leaves. Then \( T(n+1) \) is obtained by installing \( B(n+1,Y) \) at each leaf of \( T(n) \) (note that \( Y \) depends on the leaf). The number of leaves in \( T(n+1) \) is \((n+1)\) times the number of leaves in \( T(n) \), and so inductively, this number is \((n+1)!\).

Comments: I should have formulated my original question as follows: what is the minimum number of leaves among all comparison trees that sort \( n \) elements in optimal height \( S(n) \)? Let \( L^*(n) \) denote this number. Clearly, \( L^*(n) \geq n! \) but it is not clear that \( L^*(n) \leq n! \). So let us see what is the best upper bound we can get. Consider the height of \( T(n) \) in the last solution. If we make \( B(n+1,Y) \) a binary search tree, then it has height \( \lceil \lg(n+1) \rceil \). So the height of \( T(n) \) is \( \sum_{i=2}^{n} \lceil \lg(i) \rceil \). Thus \( T(3) \) has height \( 1 + 2 = 3 = \lceil \lg(3!) \rceil \), \( T(4) \) has height \( 3 + 2 = 5 = \lceil \lg(4!) \rceil \), \( T(5) \) has height \( 5 + 3 = 8 > \lceil \lg(5!) \rceil \). So the construction from (a) shows \( L^*(3) = 3! \), \( L^*(4) = 4! \) and \( 7 \leq L^*(5) \leq 8 \). But I suspect \( L^*(5) = 7 \).


- (0 Points) Ex.I.7.5. Assume \( f(n) \geq 1 \) (ev.).
  (a) Show that \( f(n) = n^{O(1)} \) iff there exists \( k > 0 \) such that \( f(n) = O(n^k) \). This is mainly an exercise in unraveling our notations!
  (b) Show a counter example to (a) in case \( f(n) \geq 1 \) (ev.) is false.

SOLUTION: (a) Let \( f(n) \geq 1 \) (ev) and assume that \( f(n) = n^{O(1)} \). Therefore there exists a \( g \in O(1) \) such that \( g(n) \leq k \) (ev) and \( f(n) \leq n^{g(n)} \leq n^k \) (ev). This shows that \( f(n) \in O(n^k) \). In the other direction, suppose \( f = O(n^k) \). Then \( f \leq Cn^k \) (ev) for some \( C > 1 \). Thus \( f \leq n^{k+\epsilon} \) (ev) for any \( \epsilon > 0 \). This shows \( f \in n^{O(1)} \).
  (b) To find a counter example, we exploit the fact that if \( g = O(1) \) implies \( g \geq 0 \) (ev). Let us choose \( f(n) = 1/2 \). Clearly \( f = O(n) \). But \( f(n) \neq n^{O(1)} \) because if \( f(n) = n^{g(n)} \) for some function \( g(n) \), then clearly \( g(n) < 0 \) (ev). But this means \( g(n) \neq O(1) \).

Comments:

- (16 Points) Ex.I.7.10. Provide either a counter-example when false or a proof when true. The base \( b \) of logarithms is arbitrary but fixed, and \( b > 1 \). Assume the functions \( f, g \) are arbitrary (do not assume that \( f \) and \( g \) are \( \geq 0 \) eventually). The next question will make assume additional properties of \( f, g \).
  (a) \( f = O(g) \) implies \( g = \Theta(f) \).
  (b) \( \max\{f,g\} = \Theta(f+g) \).
  (c) If \( g > 1 \) and \( f = O(g) \) then \( \ln f = O(\ln g) \). HINT: careful!
  (d) \( f = O(g) \) implies \( f \circ \log = O(g \circ \log) \). Assume that \( g \circ \log \) and \( f \circ \log \) are complexity functions.
  (e) \( f = O(g) \) implies \( 2^{f} = O(2^{g}) \).
  (f) \( f = o(g) \) implies \( 2^{f} = O(2^{g}) \).
  (g) \( f = O(f^{2}) \).
  (h) \( f(n) = \Theta(f(n/2)) \).
SOLUTION: Only two statements are true:
(a) False. \( f = n \) and \( g = n^2 \).
(b) False. Take \( f \) any function that is eventually non-zero. Then take \( g = -f \).
(c) False. Take \( f = 1/2 \) and any function \( g \) such that \( g > 1 \). But \( \log(f) < 0 \) and so, by definition of the \( O \)-notation, \( f \notin O(\log g) \).
(d) True. We have \( f = O(g) \) implies there is some \( C > 0 \) and \( x_0 \) such that for all \( x > x_0 \), \( f(x) \leq C g(x) \). Thus, \( f(\log(x)) \leq C g(\log(x)) \) for all \( x \geq e^{x_0} \).

Comments:

SOLUTION:
(e) False. Let \( f = 2n \) and \( g = n \).
(f) True. \( f = o(g) \) implies that for all \( C > 0 \), \( 0 \leq f \leq C \cdot g \) (ev). Taking \( C = 1 \), we obtain that \( 0 \leq 2f \leq 2g \) or in other words \( 2f \in O(2g) \).
(g) False. Let \( f = 1/n \).
(h) False. \( f = 2^n \).

Comments: Note: \( f = 1/x \) is unbounded, but not unbounded eventually!

- (8 Points) Ex.I.7.14. You should read the star paragraph, ¶ 30. Provide four ways of stating upper bounds on complexity functions, in analogy to the four ways of stating lower bounds. Describe their logical relations.

SOLUTION: The four ways are
(a) \( f = O(g) \): \( (\exists C, x_0)(\forall x > x_0)[f(x) \leq C g(x)] \)
(b) \( f \neq \Omega(g) \): \( (\forall C, x_0)(\exists x > x_0)[f(x) \leq C g(x)] \)
(c) \( f = o(g) \): \( (\forall C)(\exists x_0)(\forall x > x_0)[f(x) \leq C g(x)] \)
(d) \( f \neq \omega(g) \): \( (\exists C)(\forall x_0)(\exists x > x_0)[f(x) \leq C g(x)] \)

We can see
\[
\begin{array}{ccc}
\uparrow & \downarrow & \uparrow \\
\text{f = O(g)} & \text{f = o(g)} & \text{f \neq \omega(g)} \\
\downarrow & \uparrow & \downarrow \\
\text{f \neq \Omega(g)} & \text{f = O(g)} & \text{f \neq \omega(g)}
\end{array}
\] (33)

Comments: