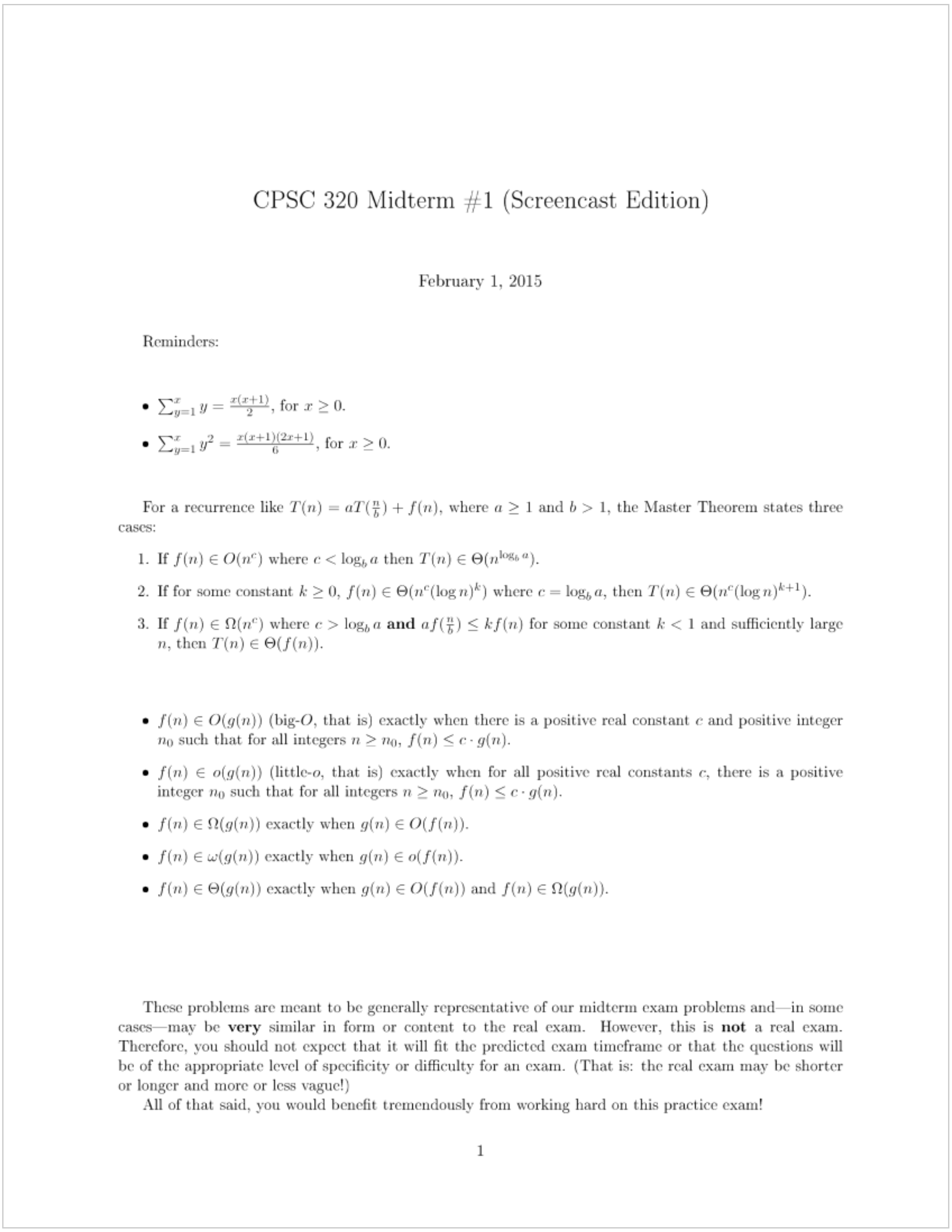
CPSC 320 2014W2 Midterm 1 Practice Problem Worked Sol'ns

February 1, 2015
11:32 AM



CPSC 320 Midterm #1 (Screencast Edition)

February 1, 2015

Reminders:

- $\sum_{y=1}^{x} y = \frac{x(x+1)}{2}$, for $x \geq 0$.
- $\sum_{y=1}^{x} y^2 = \frac{x(x+1)(2x+1)}{6}$, for $x \geq 0$.

For a recurrence like $T(n) = aT(\frac{n}{b}) + f(n)$, where $a \geq 1$ and $b > 1$, the Master Theorem states three cases:

1. If $f(n) \in O(n^c)$ where $c < \log_b a$ then $T(n) \in \Theta(n^{\log_b a})$.
2. If for some constant $k \geq 0$, $f(n) \in \Theta(n^c (\log n)^k)$ where $c = \log_b a$, then $T(n) \in \Theta(n^c (\log n)^{k+1})$.
3. If $f(n) \in \Omega(n^c)$ where $c > \log_b a$ **and** $af(\frac{n}{b}) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then $T(n) \in \Theta(f(n))$.

- $f(n) \in O(g(n))$ (big-O, that is) exactly when there is a positive real constant c and positive integer n_0 such that for all integers $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n) \in o(g(n))$ (little-o, that is) exactly when for all positive real constants c, there is a positive integer n_0 such that for all integers $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n) \in \Omega(g(n))$ exactly when $g(n) \in O(f(n))$.
- $f(n) \in \omega(g(n))$ exactly when $g(n) \in o(f(n))$.
- $f(n) \in \Theta(g(n))$ exactly when $g(n) \in O(f(n))$ and $f(n) \in \Omega(g(n))$.

These problems are meant to be generally representative of our midterm exam problems and—in some cases—may be **very** similar in form or content to the real exam. However, this is **not** a real exam. Therefore, you should not expect that it will fit the predicted exam timeframe or that the questions will be of the appropriate level of specificity or difficulty for an exam. (That is: the real exam may be shorter or longer and more or less vague!)

All of that said, you would benefit tremendously from working hard on this practice exam!

1

1 Vain-y Dividi Vici

Consider the following recursive algorithm called on an array of integers of length n. (Note: in this particular problem, it is not relevant, but generally if we refer to "fourths" of an array A with length n that is not divisible by 4, the "fourths" of A won't be exactly length $\frac{n}{4}$, but each will have length either $\lceil \frac{n}{4} \rceil$ or $\lfloor \frac{n}{4} \rfloor$. Typically, this has no effect on the asymptotic analysis.)

```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```

$O(1)$

1. Give a recurrence $T(n)$ describing the runtime of this algorithm. Be careful to clearly specify both any recursive case(s) and base case(s) and what conditions on the input identify them. To clarify, we've started a solution below, but you will need more than the case we have started.

T(n) = $2T(\frac{n}{4}) + 1$ (when n is div. by 4)

$T(n) = 1$ (when n is not div. by 4)

Repeated from problem intro:

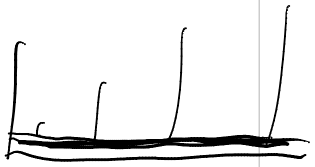
```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```

2. Would a good Ω-bound on the runtime of this algorithm in terms of n be **best** described as a best-case bound, a worst-case bound, or [✓ neither]? Choose **one** and briefly justify your answer.

Once we pick a problem size, the runtime is fixed.
No (distinct) best or worst-case exists.

Repeated from problem intro:

```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```



3. Give and briefly justify a **good** Ω-bound on the runtime of this algorithm in terms of n.

$\Omega(1)$

For any input of length $\underline{\text{not}}$ divisible by 4, ~~the~~ runtime is constant.

Repeated from problem intro:

```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```

$\frac{n}{4^k} = 1$

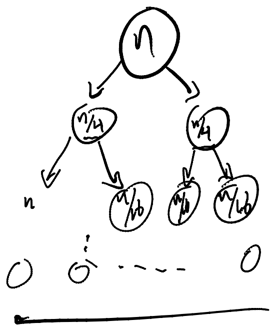
$n = 4^k$

$\lg n = \lg 4^k = k \lg 4$
$= 2k$

4. Draw a recursion tree for `CEDI(A)` labeled by the amount of time taken by each recursive call to `CEDI` and the total time for each "level" of calls, both in terms of n for an arbitrary value of n that is **a power of** 4 **greater than 1** (i.e., $n = 4^k$ for $k \geq 1$).

i	
0	n
1	$n/4$
2	$n/16$
k	$n/4^i$

$\frac{n/4}{2}$



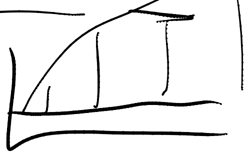
1	2^0
2	2^1
4	2^2
2^k	2

$T(n) = 2T(\frac{n}{4}) + 1$

$\sum_{i=0}^{2} 2^i = 2^{i+1} - 1$

$2^{\lg n/2} = (2^{\lg n})^{1/2} = n^{1/2} = \sqrt{n}$

$2\sqrt{n} - 1$ is $\Theta(\sqrt{n})$?



5. Give and briefly justify—based on your tree—a good O-bound on the runtime of this algorithm in terms of n.

$O(\sqrt{n})$ The alg runs no slower on n not 4^i

Repeated from problem intro:

```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```

6. Briefly explain why your bound from the previous part is **not** a Θ-bound.

B/c the alg is $\underline{\text{not}}$ $\Omega(\sqrt{n})$ for n not a multiple of 4.

7. Briefly explain why we cannot use the Master Theorem to give a Θ-bound on the runtime of this algorithm.

That and the alg has an infinite # of base cases.

Repeated from problem intro:

```
CEDI(A):
  If the length of A is odd OR half of the length of A is odd:
    Return the first element of A
  Else:
    Note: If we reach here, the length of A is divisible by 4
    Let A1 be the 1st fourth of A,
        A2 be the 2nd fourth of A,
        A3 be the 3rd fourth of A, and
        A4 be the 4th fourth of A.
    Return CEDI(A2) + CEDI(A4)
```

$f(n) = n^{0}$ (c)

$$\begin{cases} T(n) = \overset{a}{\underline{2}}T(\frac{n}{\underset{b}{4}}) + \underline{1} & \text{for } n \text{ div. by } 4 \\ = 1 & \text{otherwise} \end{cases}$$

8. If we consider only values of n that are powers of 4, we **can** apply the Master Theorem. Indicate the key parameters of the Master Theorem in this case and use it to re-justify your O-bound.

We are in case 1 b/c $f(n) \in O(n^0)$ and $0 < \log_4 2 = \frac{1}{2}$

Thus we have $O(n^{\log_b a}) = O(n^{1/2}) = O(\sqrt{n})$

2 Easy as Θne, Two, Thre∈ (or not)

For each of the following code snippets, give and briefly justify good Θ-bounds on their runtime in terms of n.

Notes: 2^n below means 2^n.

```
Let count = 0                        — O(1)
For i = 1 -> n:                      — n iters
  For j = n -> i*i:                  — 0 or i^2 - n + 1 iters
    Increment count                  — O(1)
Output "Whee! Going down.."          — O(1)
While count > 0:                     ] count iterations  Θ(count) time
  Decrement count                    — O(1)
```

$$\sum_{i=1}^{n} i^2 = \boxed{\frac{n(n+1)(2n+1)}{6}}$$

$\Theta(n^3)$

$\Theta(n^3)$

LOOP OVERCOUNTS

$j = 1 \to n-1$

$$\boxed{\sum_{i=1}^{n} n-1 = n^2 - n}$$

overcount is at most $\Theta(n^2)$ (and so doesn't matter)

Overall: $\Theta(n^3)$

Repeated from the problem intro: For each of the following code snippets, give and briefly justify good Θ-bounds on their runtime in terms of n.

```
Let count = 0
For i = 1 -> n:
  If i*i < n:
    For j = 1 to i*i:
      Increment count
Output "Whee! Going down.."
While count*count > 0:
  Decrement count
```

$O(1)$

$\Theta(n^{3/2})$

$O(1)$

$O(1)$

FIRST PART

$$\sum_{i=1}^{\sqrt{n}} \underbrace{\sum_{j=1}^{i^2} 1}_{} = \sum_{i=1}^{\sqrt{n}} i^2 = \frac{\sqrt{n}(\sqrt{n}+1)(2\sqrt{n}+1)}{6}$$

$\Theta((\sqrt{n})^3)$

$= \Theta(n^{3/2})$

SECOND PART

$$\sum_{i=\sqrt{n}}^{n} 1 = n - \sqrt{n} + 1$$

$\Theta(n)$

$\Theta(\text{count}) = \Theta(n^{3/2})$

OVERALL: $\boxed{\Theta(n^{3/2})}$

9

Repeated from the problem intro: For each of the following code snippets, give and briefly justify good Θ-bounds on their runtime in terms of n.

Notes: `2^n` below means 2^n.

```
Given: An array A of length n of integers

Let minDiff = infinity                        O(1)
For i = 0 -> (2^n - 1):                                               2^n - 1 iters
  Let inSum = 0                                O(1)
  Let outSum = 0                               O(1)
  For j = 0 -> (n - 1):                                    n-1 iters
    If the j'th bit of i is 1:
      Increase inSum by A[j]                      O(1)
    Else:
      Increase outSum by A[j]
  Let thisDiff = |inSum - outSum|                 O(1)
  If thisDiff < minDiff:                       } O(1)
    minDiff = thisDiff
Return minDiff                                 O(1)
```

$$\frac{n\cancel{2^n}}{\cancel{2^n}}$$

$$\boxed{\Theta(n2^n)}$$

$$(2^n-1)(n-1)(1)$$

$$\underline{n2^n} - \cancel{n} - \cancel{2^n} \cancel{+1}$$

3 Marriage Counselling

In this problem, we consider the Gale-Shapley algorithm with men proposing. For each statement, circle **one** answer to indicate whether the statement is **always** true, **never** true, or **sometimes** true (i.e., true for some instances but not for others).

Note: in some cases we restrict attention to just certain types of instances, in which case we're asking whether the statement is always, never, or sometimes true for instances **of that type**.

1. Two men both propose to n women.

✓

always true [✓ **never** true] **sometimes** true

$m_1 : w_1\ w_2\ w_3\ \ldots\ w_{n-1}\ w_n$

accepts

2. For any instance in which two men m_1 and m_2 both most prefer one woman w, the ordering of m_1's and m_2's proposals determines whether m_1 or m_2 marries w.

✓

always true [✓ **never** true] **sometimes** true

$m_1 : w$ ----

$m_2 : w$ ----

$m_1 : w_1$ --

$m_2 : w_2$ --

$w_1 : m_2$ --

$w_2 : m_1$ --

$m_1 : w_1$ --- $\Longleftrightarrow$ $w_1 : m_1$ ---

$m_1 : w_1$ $w_1 : m_1$

$m_2 : w_2$ --- $w_2 : m_2$ ----

$m_3 : w_3$ -- $w_3 : m_3$ ---

3. Every woman marries her most preferred man.

✓

always true **never** true [✓ **sometimes** true]

4. Some man marries his most preferred woman.

✓

always true **never** true [✓ **sometimes** true]

$m_1 : w_1$ $w_1 : m_1$

$m_1 : w_1\ w_2\ w_3$

$m_2 : w_1\ w_2\ w_3$

$m_3 : w_2\ w_1\ w_3$

$w_1 : m_3\ m_2\ m_1$

$w_2 : m_1\ m_3\ m_2$

$w_3 : m_1\ m_2\ m_3$

5. For any instance in which two women w_1 and w_2 both most prefer one man m, one of w_1 and w_2 marries m.

✓

always true **never** true [✓ **sometimes** true]

$m_1 :$ — $w_1 : m_1$

$m_2 :$ — $w_2 : m_1$

$m_1 : w_1$ -- $w_1 : m_3$ ...

$m_2 : w_2$ ... $w_2 : m_3$ ...

$m_3 : w_3$ -- $w_3 : m_1$ --

4 Demi-Glace

The minimum spanning tree problem becomes somewhat strange in the presence of negative edge weights. Imagine, for example, that you are a telecommunications company creating a communications network by connecting particular cities with fiber-optic cable. You want to ensure that all cities are connected by some path (i.e., that you've created a spanning tree). There is a cost to laying the cable, but some pairs of cities are also willing to pay you to do the job; so, the **net** cost of a particular connection may be positive, zero, or even negative.

It will be handy for this problem to define a "spanning subgraph" rather than a "spanning tree".

For a graph $G = (V, E)$, a spanning subgraph is a graph $G' = (V', E')$, where $V' = V$, $E' \subseteq E$, and G' is connected (the "spanning" part).

A "minimum spanning subgraph" would then be the spanning subgraph of a graph whose total edge weight is smallest.

1. Prove that for non-negative edge weights, the minimum spanning tree of a graph is a minimum spanning subgraph.

① An MST is a Spanning Subgraph b/c its vertices are all verts of G, its edges are a subset of G's, & it is connected.

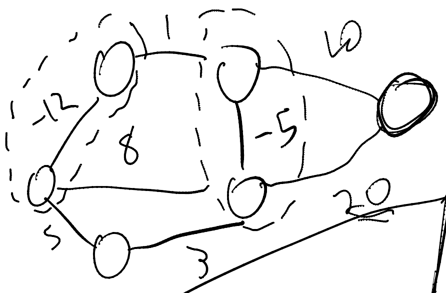
② It is min among SS. b/c any spanning subgraph will contain a spanning tree w/ cost ≥ the MST's. & b/c edges have non-neg weight, that is ≤ cost of SS.

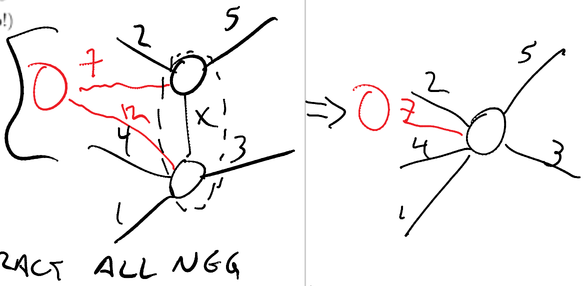
Repeated from the problem intro: For a graph $G = (V, E)$, a spanning subgraph is a graph $G' = (V', E')$, where $V' = V$, $E' \subseteq E$, and G' is connected (the "spanning" part).

A "minimum spanning subgraph" would then be the spanning subgraph of a graph whose total edge weight is smallest.

2. Give an efficient, correct reduction from the problem of finding a minimum spanning subgraph in a weighted undirected graph $G = (V, E)$ with real-valued (and possibly negative) edge weights to the minimum spanning tree problem on a graph with non-negative real edge weights.

Hint: think about "edge contractions". (Never heard of them? Look them up!)



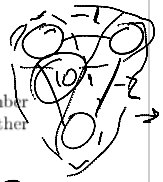


A_1: CONTRACT ALL NEG EDGES IN G, DELETING LARGER EDGES WHEN THE RESULT HAS > 1 EDGE BTW ONE PAIR OF NODES.

A_2: UNION THE SOL'N W/THE SET OF ALL NEG WEIGHTED EDGES.

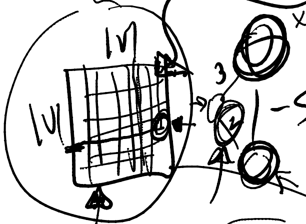
$\Theta(|V|+|E|)$

$g_2 \leftrightarrow g_3$



$V_1 \to g_1$	
$V_2 \to g_1$	
$V_3 \to g_2$	
$V_3 \to g_3$	
$V_5 \to g_2$	

3. Give and briefly justify a good Θ-bound on your reduction's worst-case runtime in terms of the number of nodes $|V|$ and edges $|E|$. Assume the input is in the form of an adjacency list. Describe any other data structures details necessary to justify the bound.



DISJ SET UNION-FIND DS TO MERGE/CONTRACT ALL NEG WEIGHT EDGES

$|V|+|E|$ FOUND

AVG COST ~~$\Theta(\alpha(|V|))$~~

TOTAL $\boxed{\Theta(|E|\alpha(|V|))}$

CREATE NEW ADJ LIST OVER GROUPS OF VERTS. $\Theta(|V|^2)$

a. CREATE ADJ MATRIX

b. FILL IN WITH EDGES. SKIP EDGES W/IN A GROUP & OVERWRITE MORE EXPENSIVE EDGES.

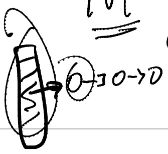
KEEP TRACK OF ORIG VERT'S.

c. CREATE ADJ LIST FROM UNDERLYING PROBLEM

$|V|^2$

$\Theta(|V|+|E|)$

$|V|^2 + \cancel{|V|} + \cancel{|E|}$



OVERALL: $\boxed{\Theta(|V|^2 + |E|\alpha(|V|))}$

4. Prove that your reduction—paired with an optimal solution to the MST problem—is optimal.

A1: CONTRACT ALL NEG WEIGHT EDGES; DELETE HIGHER WEIGHT EDGES WHERE > 1 EDGE NOW COINCIDES

A2: ADD ALL NEG WEIGHT EDGES TO SOL'N TO UNDERLYING PROBLEM

① NO MSS CAN LEAVE OUT ANY NEG WEIGHTED EDGE.
ASSUME FOR CONTRA THAT ~~IT~~ DOES.
THEN WE CAN ADD IT BACK IN.
THE RESULT IS STILL A SS & IT'S CHEAPER

ASSUME FOR CONTRA THAT IT DOES.
THEN WE CAN ADD IT BACK IN.
THE RESULT IS STILL A SS & IT'S CHEAPER.
THAT'S A CONTRADICTION.

② TO CONNECT THE CONN. COMPONENTS AFTER INCLUDING THESE EDGES, THE LEAST EXPENSIVE SOLUTION IS EXACTLY A MIN COST SPANNING TREE OVER THE CONN COMPONENTS, IGNORING MORE EXPENSIVE EDGES.

5 A Capital Idea

1. Prove that if $f(n) \in o(g(n))$, then $f(n) \in O(g(n))$.

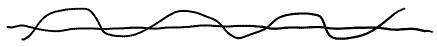
Assume $f(n) \in o(g(n))$.

So for all constants c, $\exists n_0$, $f(n) \leq cg(n)$ for $n \geq n_0$.

MUST SHOW: $f(n) \in O(g(n))$.

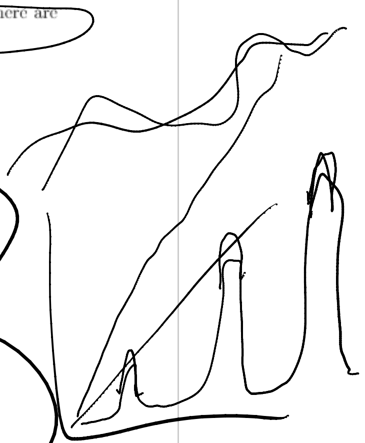
Let $c_0 = 1$. By assumption n_0 exists s.t. $f(n) \leq c_0 g(n)$ for all $n \geq n_0$.

2. In each row below, circle the correct statement if we know that for all positive integers n, there are two larger integers n_1 and n_2 such that $f(n_1) < g(n_1)$ and $f(n_2) > g(n_2)$.



$f(n) = 2$

$g(n) = \boxed{2 + \sin n}$



~~$f(n) \in O(g(n))$~~ ~~$f(n) \notin O(g(n))$~~ [✓ $f(n)$ may or may not be in $O(g(n))$]

~~$f(n) \in \Omega(g(n))$~~ ~~$f(n) \notin \Omega(g(n))$~~ [✓ $f(n)$ may or may not be in $\Omega(g(n))$]

$n \qquad n+1$

~~$f(n) \in \Theta(g(n))$~~ ~~$f(n) \notin \Theta(g(n))$~~ [✓ $f(n)$ may or may not be in $\Theta(g(n))$]

$f(n) \in o(g(n))$ [✓ $f(n) \notin o(g(n))$] $f(n)$ may or may not be in $o(g(n))$

$$\boxed{\begin{array}{l} f(n_1) < g(n_1) \\ f(n_2) > g(n_2) \end{array}}$$

$f(n) \overset{?}{=} g(n)$

$f(n) \in \omega(g(n))$ [✓ $f(n) \notin \omega(g(n))$] $f(n)$ may or may not be in $\omega(g(n))$

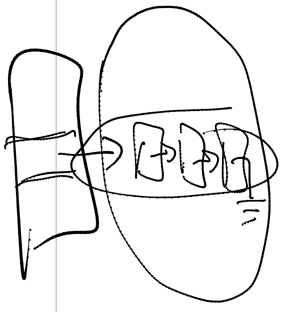
3. Consider the following pseudocode:

```
For each edge (u, v) in E:
  For each edge (u, v') in E incident on the node u:
    UnknownComputation(G, v, v')
  For each edge (u', v) in E incident on the node v:
    UnknownComputation(G, u, u')
```

$\# \text{iters} = |E|$

$\Omega(1)$

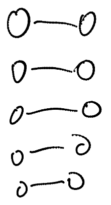
The directed graph `G = (V, E)` given as input uses an adjacency list representation as does the algorithm itself. You're given no further information about `UnknownComputation`, however. Give a good asymptotic lower-bound on the runtime of the algorithm in terms of the number of nodes $|V|$ and edges $|E|$. Briefly justify your bound by annotating the code above. (Note: the same bound is correct for both best- and worst-case.)



$\sum_{\text{lists}} |\text{list}|^2$

WRONG

$\Omega(|E|)$



FOR COMPLETE: $\Omega(|E| \cdot |V|) = \Omega(|V|^3)$

4. If $h_1(n) \in O(h_2(n))$, is $h_1(n)! \in O(h_2(n)!)$? Prove or disprove your answer.

FALSE. COUNTEREXAMPLE IS

$\exists c, n_0, \forall n \geq n_0, h_1(n) \leq c\, h_2(n).$

$\exists ?\ d, n_1, \forall n \geq n_1, (h_1(n))! \leq d (h_2(n))!$

$n \quad 2n$

$n \quad n+1$

$$\frac{(n+1)!}{n!} = \frac{(n+1)\cancel{n!}}{\cancel{n!}} = \underline{\underline{n+1}}$$

$h_1(n) = n+1 \qquad h_2(n) = n$

$(h_1(n))! = (n+1)! \qquad (h_2(n))! = n!$

$h_2(n) \in o(h_1(n)) \qquad h_1(n) \notin O(h_2(n)).$

6 Pairs of Apples and Oranges

For each of the following, indicate the most restrictive true answer of $f(n) \in o(g(n))$, $f(n) \in O(g(n))$, $f(n) \in \Theta(g(n))$, $f(n) \in \Omega(g(n))$, and $f(n) \in \omega(g(n))$.

$\frac{n}{\sqrt{n}\lg n} = \frac{\sqrt{n}}{\lg n} = \frac{n^{1/2}}{\lg n}$ $\frac{1}{2}\frac{1}{\sqrt{n}}$ $\frac{\frac{1}{2\sqrt{n}}}{\frac{c}{n}} = \frac{n}{2c\sqrt{n}} = \frac{\sqrt{n}}{2c} \to \infty$

n

n^{ϵ} $\epsilon > 0$

$\lg n \in o(n^{\epsilon})$

$\lg(n^{\sqrt{n}}) \in$ o (~~100n − lg n~~)

$\sqrt{n}\lg n$ little

$\lg n = c\ln n$

$\frac{c}{n}$ ϵ

$2^{n/2} \in$ o (3^n)

$2^n \in o(3^n)$ little

$2^{n/2}$ 2^n

$2^{n\cdot\frac{1}{2}}$

$2^{\frac{1}{2}\cdot n} = (2^{1/2})^n = \sqrt{2}^n$

$\lg(4^n) \in$ Θ ($\lg(3^n)$)

$n\,\cancel{\lg 4}$ $n\,\cancel{\lg 3}$

$\sqrt{n} \cdot \sqrt{n}$

$(\ln n)(\ln(n+1)) \in$ o (n)

little

$\sqrt{n^n} \in$ Θ ($(\sqrt{n})^n$)

$n^{n\cdot\frac{1}{2}} = (n^n)^{1/2}$ $(\sqrt{n})^n = (n^{1/2})^n = n^{\frac{1}{2}\cdot n}$

7 Greedy Straw-Man Pessimality

You're solving the optimal caching problem except **maximizing** the number of cache misses rather than minimizing it.

UNNECESSARY FLAVOR TEXT: A systems research group (somewhere besides UBC) is trying to show how great their new caching algorithm is. They decide to test against the **worst** algorithm they can create. So, given the number of pieces of data n, the cache size $k < n$, the sequence of data items $d_1, d_2, \ldots, d_m$, and the initial contents of the cache $\{c_1, c_2, \ldots, c_k\}$—which for this version of the problem are "dummy" data items may never appear in the sequence of data items, i.e., the cache is effectively empty—they want an algorithm that gives an eviction schedule $e_1, e_2, \ldots, e_j$ that **maximizes** the number of cache misses, but (1) **never** evicting an element unless the cache is full and a cache miss occurs and (2) **always** replacing the evicted item with that caused the cache miss. (I.e., it's a plausible strategy, even if terrible.)

1. Here is a greedy strategy that does **not** always cause the largest number of cache misses: Each time a data item is not in the cache (a miss occurs), evict the item that was brought into the cache most recently. (The initial "dummy" data items are evicted in an arbitrary order.)

   Now, give a small example that shows that this strategy can fail.

$n = 3$ $\quad$ $\cancel{c_1}$ 1 $\qquad\qquad$ $\cancel{c_1}$ $\cancel{1}$ $\cancel{3}$ $\cancel{1}$ 3

$k = 2$ $\quad$ $\cancel{c_2}$ $\cancel{2}$ $\cancel{3}$ $\cancel{2}$ $\qquad\qquad$ $\cancel{c_2}$ $\cancel{2}$ $\cancel{1}$ $\cancel{2}$ 1

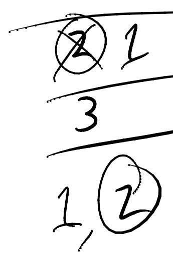
data: 1, 2, 3, 1, 2, 1, 3, 1

Repeated from the problem intro: You're solving the optimal caching problem except **maximizing** the number of cache misses rather than minimizing it.

UNNECESSARY FLAVOR TEXT: A systems research group (somewhere besides UBC) is trying to show how great their new caching algorithm is. They decide to test against the **worst** algorithm they can create. So, given the number of pieces of data n, the cache size $k < n$, the sequence of data items $d_1, d_2, \ldots, d_m$, and the initial contents of the cache $\{c_1, c_2, \ldots, c_k\}$—which for this version of the problem are "dummy" data items may never appear in the sequence of data items, i.e., the cache is effectively empty—they want an algorithm that gives an eviction schedule $e_1, e_2, \ldots, e_j$ that **maximizes** the number of cache misses, but (1) **never** evicting an element unless the cache is full and a cache miss occurs and (2) **always** replacing the evicted item with that caused the cache miss. (I.e., it's a plausible strategy, even if terrible.)

2. Give a **new** greedy algorithm (either in English like the one above or in pseudocode) that **correctly** solves this problem.

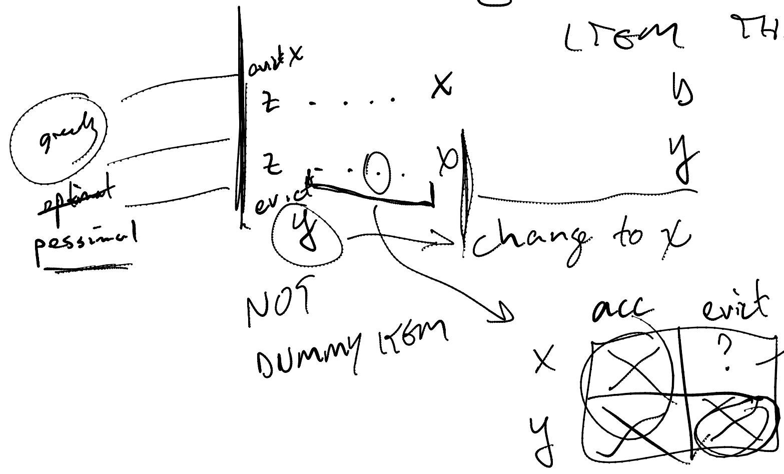
When we have a choice of non-dummy items to evict, evict the item that will be used soonest.



Repeated from the problem intro: You're solving the optimal caching problem except **maximizing** the number of cache misses rather than minimizing it.

3. Prove that your strategy is correct.

[EVICT THE NEXT CACHED ITEM THAT WILL BE USED.]



in this case, evict y

IF NONE OF THESE HAPPENS, EVICT Y AT ACCESS TO X.

CONSIDER THE FIRST DATA ITEM ON WHICH GREEDY & A PESSIMAL SOL'N DIFFER IN EVICTING A NON-DUMMY ITEM.
WLOG, GREEDY EVICTS X, PESS EVICTS Y.
X & Y MUST BOTH BE IN THE CACHE.
X MUST BE THE NEXT CACHED ITEM IN THE DATA STREAM, BTW THIS EVICTION & THE ACCESS TO X, ① THERE CAN BE NO ACCESSES TO X OR Y (ELSE GREEDY WOULD CHOOSE BASED ON THOSE).
② PESS CANNOT EVICT Y (B/C IT CAN ONLY BE BROUGHT BACK IN ON AN ACCESS TO Y.
WE WILL CHANGE PESS TO EVICT X LIKE GREEDY.
IF BTW THAT EVICTION & THE ACCESS TO X, PESS EVICTED Y THEN CHANGE THAT EVICTION TO X, & PESS HAS THE SAME STATE @ THE ACCESS TO X AS IT HAD & SAME # OF EVICTIONS.
ELSE, EVICT Y AT THE ACCESS TO X & IT HAS SAME STATE BUT MORE EVICTIONS (CONTRADICTION).
WE'VE MADE PESSIMAL ONE STEP MORE SIMILAR TO GREEDY @ THE SAME # OF EVICTIONS. A FINITE # OF SUCH CHANGES WILL MAKE IT = GREEDY. QED.

8 Declaration of (a Degree of) Independence

Let's see if we can find a bound on the minimum size of an independent set in an undirected graph given the maximum degree $d_{\max}$ of any node in the graph. (Recall that the degree of a node in an undirected graph is the number of edges incident on that node.)

Here's a naïve algorithm to try to find an independent set in a graph:

```
Initialize the solution to the empty set {}          — list ← O(1)
While there are remaining nodes in the graph          — max iters |V|
  Pick a node and add it to the solution               — O(1)
  Remove it and all nodes adjacent to it from the graph
```

arbitrarily

TOTAL: O(|V|)

O(1)

$O(|V|^2)$ ≡ IMPL: [for i = 0; i < |V|; i++ if !removed[i]] pick & process

exactly |V| iters

in adj list: easy to traverse these. O(1) per node

1. Give and justify (i.e., by annotating the code and explaining any complex annotations) a good, worst-case big-O-bound on the runtime of this algorithm in terms of the number of vertices in the graph n and the maximum degree of any vertex $d_{\max}$.

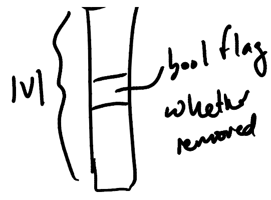
|V|

TOTAL: O(|V|) O(|V|dmax) ← largest # of inc. edges

|V| { bool flag

dmax ≤ |E| ≤ |V|dmax

|V| + |V|(dmax)



i.e. edges

$$d_{max} \le |E| \le |V| d_{max}$$

$$O(\cancel{1} + |V| + |V| + |V| + |V| + |V| d_{max})$$

$$= \boxed{O(|V| + |V| d_{max})}$$

23

Repeated from the problem intro:

```
Initialize the solution to the empty set {}
While there are remaining nodes in the graph
  Pick a node and add it to the solution
  Remove it and all nodes adjacent to it from the graph
```

n = # of verts. $|V|$

d_{max} = max degree

2. Give and justify a good, **non-asymptotic lower-bound** on the number of iterations of the loop performed on any graph. (A precise formula, **not** a O, o, Θ, Ω, or ω bound!)

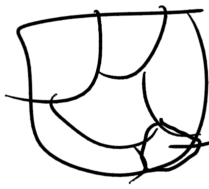
$1 + d_{max}$

$$\boxed{\left\lceil \frac{n}{1+d_{max}} \right\rceil}$$

Ex. $d_{max} = 4$, $n = 6$

rem 5

(maybe ceiling?)



Even if leftovers are too small, still need 1 iter.

(loop)

3. **Briefly** explain why a lower-bound on the number of iterations of the algorithm above also gives a lower-bound on the size of the independent set in the input graph.

maximum?

The alg gives an ind set.

Its size is equal to the # of iters.

So, the # of iters is a lower-bound on the size of the max ind set.