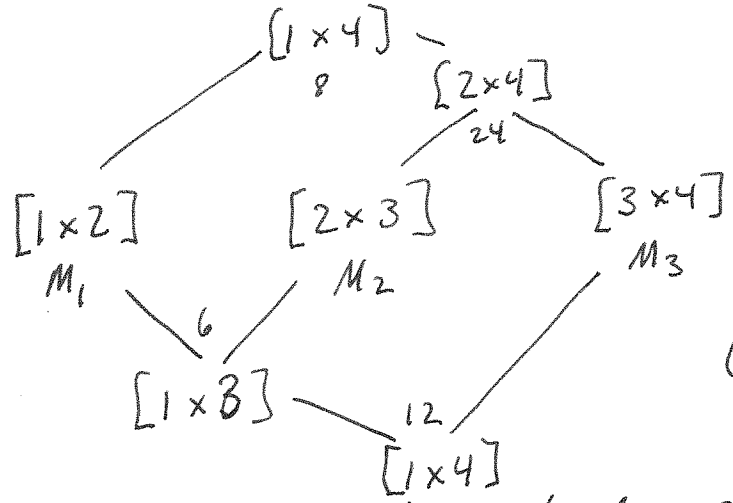
2A. Most divide-and-conquer algorithms split a problem into disjoint subproblems. Memoization is like dynamic programming – it exploits repeated subproblems. If divide-and-conquer subproblems are distinct, there will be no reuse of saved answers. In the case of Freq, if one call is a descendent of another call in the call tree, the former will be on a strict subset of the latter, hence the calls are distinct. If the calls are on different paths, their sets are disjoint and hence distinct.

2B. A number of you showed that the top-down approach failed to get the optimal answer. There are examples with as few as 3 matrices, for example:




Top-down
$M_1 (M_2 M_3)$
32 mults

Optimal
$(M_1 M_2) M_3$
18 mults

2C. This problem can be solved using the LCS algorithm. If S is the original input, let T = sorted(S) and U = $T T^{R1}$, where $T^{R1}$ is the reversal of T, with the largest value removed.

Claim: LCS(S, U) will give you the longest bitonic subsequence of S. Note that U is bitonic, so that any subsequence of it is also bitonic.* Further, any bitonic subsequence of S will be a subsequence of U, including the longest bitonic subsequence of S.

Example: S = 3 4 6 2 7 1 15 5

T = 1 2 3 4 5 6 7 15

U = 1 2 3 4 5 6 7 15 7 6 5 4 3 2 1

LCS(S, U) = 3 4 6 7 15 5

[*Note: U has subsequences such as 1 2 2 1 that are not quite bitonic. However, since S has no duplicates, LCS will never find one of these.]

Time complexity: Sorting S is $O(n \log n)$. Creating U is $O(n)$. LCS(S, U) is $O(len(S) \cdot len(U)) = O(n \cdot (2n-1)) = O(n^2)$. Thus, the overall complexity is $O(n^2)$.

2D — see ~~next~~ last page

2E. A greedy algorithm works here, where activities are scheduled by earliest start time, and we only add a room when all other rooms are already scheduled.

In order to get good time complexity, we can use a priority queue to organize both the activities and the finish of an activity in a room. So the queue will have two kinds of events:

SCHED(a, s, e): Activity a that runs from time s to e needs to be scheduled.

AVAIL(r, e): Room r will be available next at time e.

[C]

In the queue, items are ordered smallest-first by SCHED.s and AVAIL.e, with an AVAIL item coming before any SCHED item with the same sort key.

We also will maintain a list OPEN of currently open rooms.

The schedule will be in the form of items of the form ASSIGN(a, r), meaning activity a is assigned to room r.

Let $A = A_1, A_2, \ldots, A_n$ be the initial list of activities, and let PQ be the priority queue.

```
minRooms(A)
    Add A to PQ; OPEN ← nil
    While PQ not empty
        m ← EXTRACT-MIN(PQ)
        case m = SCHED(a, s, e)
            if OPEN = nil then add(OPEN, newRoom())
            r ← remove(OPEN)
            output ASSIGN(a, r)
            INSERT(AVAIL(r, e), PQ)
        case m = AVAIL(r, e)
            add(OPEN, r)
```

Time complexity: PQ initially has $n$ events, and never gets larger, since any INSERT follows an ~~extract~~ EXTRACT-MIN. If we implement PQ as a heap, then each operation will be $O(\lg n)$, plus one $O(n)$ for initialization. OPEN can be a stack, for $O(1)$ cost per operation. Each activity is in PQ twice, as SCHED and AVAIL. So the

while-loop executes $2n$ times, each time doing $O(\lg n)$ work. So overall, we have $O(n + 2n \cdot \lg n) = O(n \cdot \lg n)$.

Example: $A = [a_1, 1, 3], [a_2, 6, 8], [a_3, 5, 10], [a_4, 7, 11], [a_5, 4, 7], [a_6, 2, 5], [a_7, 4, 8], [a_8, 1, 2]$.

Let S be SCHED and A be AVAIL in PQ

Start: PQ = $S(a_1), S(a_8), S(a_6), S(a_5), S(a_7), S(a_3), S(a_2), S(a_4)$

1. $m = S(a_1)$, OPEN = r1, ASSIGN$(a_1, r1)$, insert $A(r1, 3)$
   PQ = $S(a_8), S(a_6), A(r1, 3), S(a_5), S(a_7), S(a_3), S(a_2), S(a_4)$
2. $m = S(a_8)$, OPEN = r2, ASSIGN$(a_8, r_2)$, insert $A(r2, 2)$.
   PQ = $A(r2, 2), S(a_6), A(r1, 3), S(a_5), S(a_7), S(a_3), S(a_2), S(a_4)$
3. $m = A(r2, 2)$, OPEN = r2
   PQ = $S(a_6), A(r1, 3), S(a_5), S(a_7), S(a_3), S(a_2), S(a_4)$
4. $m = S(a_6)$, OPEN = r2, ASSIGN$[a_6, r_2]$, insert $A(r_2, 5)$
   PQ = $A(r1, 3), S(a_5), S(a_7), A(r2, 5), S(a_3), S(a_2), S(a_4)$
5. $m = A(r1, 3)$, OPEN = r1
   PQ = $S(a_5), S(a_7), A(r2, 5), S(a_3), S(a_2), S(a_4)$
6. $m = S(a_5)$, OPEN = r1, ASSIGN$[a_5, r1)$, insert $A(r1, 7)$
   PQ = $S(a_7), A(r2, 5), S(a_3), S(a_2), A(r1, 7), S(a_4)$
7. $m = S(a_7)$, OPEN = r3, ASSIGN$(a_7, r3)$, insert $A(^{r3}_{a_7}, 8)$
   PQ = $A(r2, 5), S(a_3), S(a_2), A(r1, 7), S(a_4), A(^{r3}_{a_7}, 8)$
8. $m = A(r2, 5)$, OPEN = r2
9. PQ = $S(a_3), S(a_2), A(r1, 7), S(a_4), A(^{r3}_{a_7}, 8)$

E

9. $m = S(a_3)$, OPEN = r2, ASSIGN($a_3$, r2), insert $A(a_3^{r2}, 10)$
   PQ = $S(a_2)$, $A(r1, 7)$, $S(a_4)$, $A(a_4^{r3}, 8)$, $A(a_3^{r2}, 10)$

10. $m = S(a_2)$, OPEN = r4, ASSIGN($a_2$, r4), insert $A(a_2^{r4}, 8)$
   * PQ = $A(r1, 7)$, $S(a_4)$, $A(a_4^{r4}, 8)$, $A(r3, 8)$, $A(r2, 10)$

11. $m = A(r1, 7)$, OPEN = r1
   PQ = $S(a_4)$, $A(r4, 8)$, $A(r3, 8)$, $A(r2, 10)$

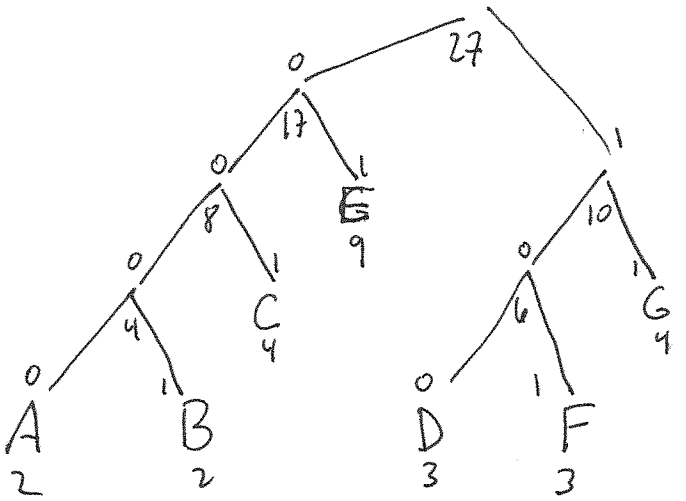
12. $m = S(a_4)$, OPEN = r1, ASSIGN($a_4$, r1), insert(r1, 11)
   PQ = $A(r4, 8)$, $A(r3, 8)$, $A(r2, 10)$, $A(r1, 11)$

13.–16. We remove all the A's, OPEN = r1, r2, r3, r4

Minimality: Note that if $A(r1, i)$ and $A(r2, j)$ are simultaneously in PQ, then they correspond to two activities $A_i$ and $A_j$ that overlap in time. Also note that we don't add room rk unless rooms r1 – r(k-1) are in PQ. So whenever we add room rk, we must have k mutually overlapping events, as at (*). So no fewer than k rooms will suffice.

Correctness: We never assign an activity to an occupied room. If we are processing $S(a_m)$, then all rooms with a next available time after the start time of $a_m$ are behind $S(a_m)$ in PQ, and are not candidates for assignment.

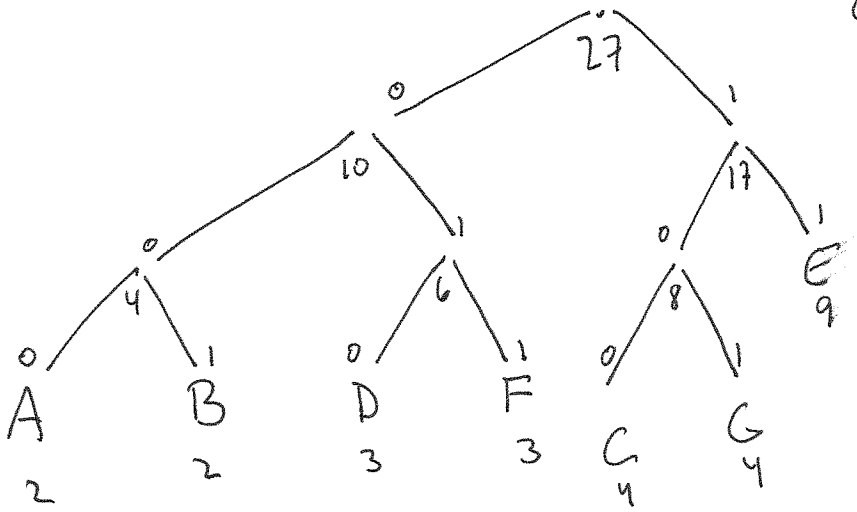
2F. The are choices in whether to merge DF with AB, C or G. (Plus there are left-right choices.)




code (ACEBAG) =

0000, 001, 01, 0001, 0000, 11

19 bits *




code (AGEBAG) =

~~0000, 100, 11, 0001, 0000, 101~~

~~20 bits *~~

000, 100, 11, 001, 000, 101

17 bits *

* Why does the difference in length ~~not~~ not represent a contradiction to the minimality of the second code?

2-D

## 410 Only

7. **Dynamic Programming** (25 points) Let A[1..n] be an array of floating-point numbers.
a. Describe an O(n) algorithm to find values i and j that maximize the sum of the values A[i] through A[j], that is A[i] + A[i + 1] + … + A[j]. If you give pseudocode, also provide an English explanation.

Let B[k] = the ~~be~~ largest sum you can get ending at k

S[k] = where that sum starts

Initialize
B[1] ← A[1]
S[1] ← 1

For k = 2 to n
if B[k-1] + A[k] < A[k]
then B[k] ← A[k]
S[k] ← k
else B[k] ← B[k-1] + A[k]
S[k] ← S[k-1]

Scan B for the maximum value

At any point, if adding A[k+1] to B[k] ~~goes negative, restart the chain, otherwise extend it.~~ is > A[k+1] alone, extend the chain. Otherwise restart at A[k+1]

Clearly O(n), because we make one pass through A, doing constant work per element, then one scan through B for the max.

b. Illustrate your method on array A below.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| A[i] | 3.1 | -4.2 | 6.7 | -3.2 | 7.5 | 2.0 | -5.4 | 4.7 |
| B[i] | 3.1 | -1.1 | 6.7 | 3.5 | 11.0 | 13.0 | -2.4 | 4.7 |
| S[i] | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 8 |

↑ restart (3)

↑ restart (8)

↑ max B value, so ~~3-6~~ A[3]-A[6] is the maximum sum.

2-D

[14]

## 584 Only

7. **Dynamic Programming** (25 points) We have n people with integer weights $w_1, w_2, \ldots, w_n$. There are two elevators, and we want to know if we can divide the people into two groups that will put equal weights on each elevator. (For example, with weights 3, 4, 6, 2, 11, 4 we can split them into Group 1 = 4, 11 and Group 2 = 3, 6, 2, 4, each with total weight 15).

a. Describe a method to solve the problem in $O(n^2W)$ time, where W is the maximum weight of any person. If you give pseudocode, also provide an English explanation.

Let $S = \sum w_i$. The goal is to see if some subset of the weights adds up to $S/2$. Then the remain weights must add up to $S/2$ as well. (If $S/2$ is not an integer, report "no solution")

Let B be a two dimensional array $B[0..n, 0..\frac{S}{2}]$

Entry $B[i,v] = 1$ if some subset of $w_1, \ldots, w_i$ can add up to $v$. Note that $B[i,v] = 1$ if either $B[i-1, v] = 1$ (leave out $w_i$) or $B[i-1, v-w_i] = 1$ (add $w_i$ to the subset of $w_1, \ldots w_{i-1}$ that sums to $v-w_i$). Initialize ~~$B[0,v] = B[i,0] = 0$~~.

$B[i,0] \leftarrow 1$, $B[0,v] \leftarrow 0$ for $v > 0$

b. Illustrate your method with the weights 2, 3, 2, 7, 2. Here $S = 16$ so $S/2 = 8$

| $w_i$ | i \ v | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 3 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 7 | 4 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 2 | 5 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

This value is 0, so even split not possible.

```
for i = 1 to n
    for v = 1 to S/2
        B[i,v] ← B[i-1,v] or B[i-1, v-w_i]
```

Test if $B[n, \frac{S}{2}] = 1$

since $S/2 < n \cdot W$, this is $O(n \cdot S/2) \subseteq O(n \cdot n \cdot W)$

There is also a formulation that just keeps a list of the 1's in each row.