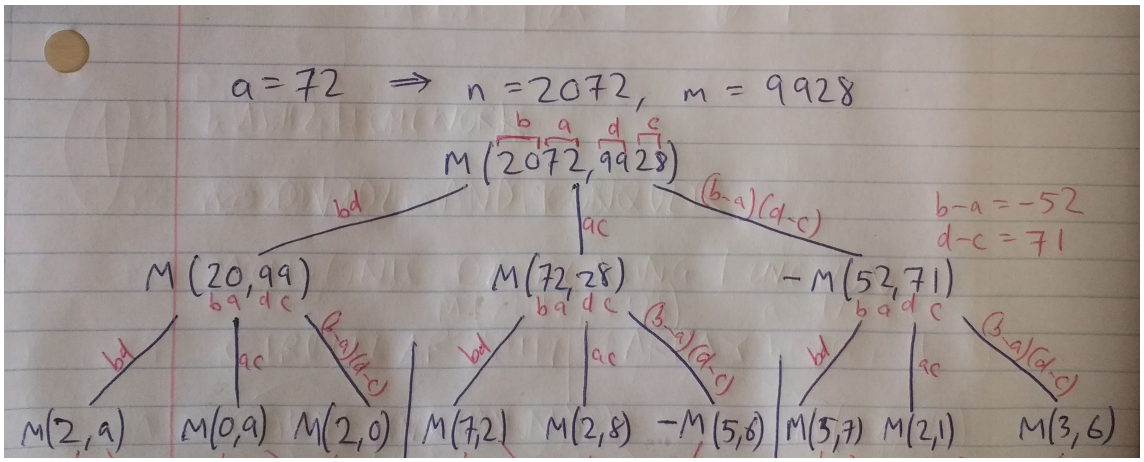
Homework 7: solutions

Isaac Henrion

Here is an example trace with $a = 72$.



Problem 2

We use the recursion tree method to solve the recurrence relation. Looking at the equation, we can read off the work done at node, branch factor and decay: they are $5n^2$, 3 and 2 respectively.

Level 0 of the tree (the root) has 1 node of size n which does $5n^2$ work. Level 1 has 3 nodes of size $n/2$, each doing $5n/4$ work for a total of $3 \times \frac{5n^2}{4}$. Level 2 has 9 nodes of size $n/4$, each doing $5(n/4)^2$ work for a total of $9 \times \frac{5n^2}{4^2}$. $[\dots]$ Level k has 3^k nodes of size $\frac{n}{2^k}$, each doing $\frac{5n^2}{4^k}$ work for a total of $3^k \times \frac{5n^2}{4^k}$. At the leaves, we have $3^{\log_2 n}$ nodes each doing 3 work for a total of $3 \times 3^{\log_2 n}$.

Putting this all together, we can write

$$T(n) = 5n^2 + 5n^2\left(\frac{3}{4}\right) + 5n^2\left(\frac{3}{4}\right)^2 + \cdots + 5n^2\left(\frac{3}{4}\right)^k \cdots + 5n^2\left(\frac{3}{4}\right)^{\log_2 n - 1} + 3 \times 3^{\log_2 n}$$

and perform the following gory details derivation. I have provided a line-by-line explanation of what's going on, but you might skip a few of these steps if you know exactly what you're doing.

$$\begin{aligned}
T(n) &= \left[\sum_{k=0}^{\log_2 n - 1} 5n^2\left(\frac{3}{4}\right)^k\right] + 3 \times 3^{\log_2 n} && \text{pulling all the non-leaf terms into a sum} \\
&= \left[5n^2 \sum_{k=0}^{\log_2 n - 1}\left(\frac{3}{4}\right)^k\right] + 3 \times 3^{\log_2 n} && \text{taking constant factor } 5n^2 \text{ out of the sum} \\
&= 5n^2\left(\frac{1 - \left(\frac{3}{4}\right)^{\log_2 n}}{1 - \frac{3}{4}}\right) + 3 \times 3^{\log_2 n} && \text{using formula for geometric series with ratio} = 3/4 \\
&= 5n^2\left(\frac{1 - \left(\frac{3}{4}\right)^{\log_2 n}}{\frac{1}{4}}\right) + 3 \times 3^{\log_2 n} && \text{evaluating the arithmetic in the denominator} \\
&= 20n^2\left(1 - \left(\frac{3}{4}\right)^{\log_2 n}\right) + 3 \times 3^{\log_2 n} && \text{flipping term of } 1/4 \text{ to the top} \\
&= 20n^2 - 20n^2\left(\frac{3}{4}\right)^{\log_2 n} + 3 \times 3^{\log_2 n} && \text{expanding the brackets} \\
&= 20n^2 - 20n^2 \cdot n^{\log_2 \frac{3}{4}} + 3 \times 3^{\log_2 n} && \text{using the identity } x^{\log_a y} = y^{\log_a x} \text{ with } x = \frac{3}{4}, y = n, a = 2 \\
&= 20n^2 - 20n^2 \cdot n^{\log_2 \frac{3}{4}} + 3 \times n^{\log_2 3} && \text{using the identity } x^{\log_a y} = y^{\log_a x} \text{ with } x = 3, y = n, a = 2 \\
&= 20n^2 - 20n^2 \cdot n^{\log_2 3 - \log_2 4} + 3 \times n^{\log_2 3} && \text{since } \log \frac{a}{b} = \log a - \log b \\
&= 20n^2 - 20n^2 \cdot \frac{n^{\log_2 3}}{n^{\log_2 4}} + 3 \times n^{\log_2 3} && \text{since } x^{a-b} = \frac{x^a}{x^b} \\
&= 20n^2 - 20n^2 \cdot \frac{n^{\log_2 3}}{n^2} + 3 \times n^{\log_2 3} && \log_2 4 = 2 \\
&= 20n^2 - 20n^{\log_2 3} + 3 \times n^{\log_2 3} && \text{canceling terms of } n^2 \\
&= 20n^2 - 17n^{\log_2 3} && \text{collecting terms of } n^{\log_2 3}
\end{aligned}$$

and we observe that because $2 > \log_2 3$, the dominant term is the one in n^2.

Therefore $T(n) = \Theta(n^2)$.

Problem 3

This problem demands a kind of binary search. If the numbers were decreasing from positive to negative, how would you find the first negative value? You would look at the element halfway through the array A, and if it's negative then you know that you have to look to the left, if positive then look to the right. This problem is solved in exactly the same way. In our array, each element falls into one of two categories: either it is equal to its index (with 1-based indexing) or it is one more than its index. In the first case, the missing element must lie to the right of that element. In the second case the missing element is to the left. In either case, we have narrowed the problem down to a half-sized one. We then recursively search the half-sized array.

What about the base case? If the sub-array being considered has length 1, then we perform a simple check. If the value of the singleton element is equal to its index in the original array, then the missing element must be that index plus one. Otherwise the missing element is the index. (Think about the problem for $n = 2$, where the possible arrays are [1] and [2]. In the first case, the missing element is 2 (i.e. the index + 1). In the second case, the missing element is 1 (i.e. the index)).

For the sake of argument, let's say n is a power of 2, plus one. That way A will have length 2^k for some k, and we don't have to worry about fractional indices, rounding down and the like. We can always pad A to have length a power of 2, in any case. Then we have the following algorithm.

Require: array A of length $n-1$, consisting of the numbers 1 to n in increasing order, but one is missing.

```
function MissingWrapper(A)
    function Missing(i, j; ; A)
        i is the start index and j is the end index, so we are considering sub-array A[i, j].
        if j − i = 1 then
            if A[i] = i then
                return i + 1
            else
                return i
            end if
        end if
        mid ← (j − i)/2
        if A[i + mid] = i + mid then
            return Missing(i + mid, j, A)
        else
            return Missing(i, i + mid, A)
        end if
    end function
    return Missing(1, n, A)
end function
```

What is the complexity of this algorithm? We need to look at a recurrence relation. Let $T(m)$ be the complexity of a call to `Missing` of size m (i.e. one that considers a sub-array of size m). This call does some constant work to determine whether it is a base case and if so, what to do. If it is not a base case, then it will launch a single call to `Missing` but this time of size $m/2$. Hence the recurrence relation is

$$T(m) = \begin{cases} T(m/2) + C & \text{if } m > 1 \\ C & \text{if } m = 1 \end{cases}$$

where C is a constant summarizing the work done for the base case.

We can consider a recursion tree with work done C, branch factor 1, and decay rate 2. At each level k of the tree, there is one node performing work C, for $k = 0, \ldots, \log_2 n$. Hence the total work is $C(1 + \log_2 n) = O(\log n)$. Alternatively, we can just observe that there are $O(\log_2 n)$ calls of constant computation, so it's $O(\log n)$.

Problem 4

This problem is quite similar to the original round-robin problem, but we have one fewer team. The twist is that team i cannot play on day i. How can we represent this constraint. The key idea here is to represent it as *another team*, taking the total back to a power of 2. This "ghost team" is labeled $n+1$. We just schedule a round-robin tournament using the algorithm from the textbook. (Recall: this involves splitting the teams in half, scheduling the matches between one half and the other, and then recursively making a round-robin for each half.)

When team i meets team $n+1$ in this tournament, they will have a rest day. So we have a way of organizing a round-robin with a rest day for each team. That's good. But we need team i to rest on day i. At the moment we have no idea when team i will rest.

Well, actually, we do. The list of team $n+1$'s matches gives the resting teams on each day. Some notation: let $a_1, \ldots, a_n$ be the teams assigned to team $n+1$, so team $n+1$ plays team a_i on day i. The sequence $a_1, \ldots, a_n$ is just a different ordering of $1, \ldots, n$. We want team i to play team $n+1$ on day i, not a_i. So we can just go through the schedule, replacing the matches on day a_i with the matches on day i. That would take time $O(n^2)$ because there are $O(n^2)$ matches, and we can do an array lookup in constant time. Creating the lookup table takes time $O(n)$, because we have to initialize an array of length n, and fill it with i at index a_i for $i = 1, \ldots, n$.

It is clear that after reordering teams according to the lookup table, team i would have a break on day i. But what if we ruined the round-robin structure in some way by doing this? It's easy to see that we haven't. Suppose for contradiction that the structure had been broken. That would mean that some pair of teams (j, k) don't play each other. But (j, k) appeared as (a_j, a_k) in the original schedule, which used the original round-robin algorithm. So a_j did play a_k in the original, hence j plays k in the permuted schedule. Therefore all teams play each other.

So the algorithm will work correctly. Moreover, it terminates in time $O(n^2)$ because the original round-robin algorithm works in time $O(n^2)$, and we do an additional $O(n^2)$ computations (relabeling the teams).

Require: $n = 2^k - 1$
  function OriginalRoundRobin(m)
    Given m a power of 2, returns a matrix M scheduling matches for teams $1, \ldots, m$.
    M is $m \times (m-1)$, and $M[i, j]$ stores the team that plays team i on day j.
    Implementation is in textbook!
    return M
  end function
  $M \leftarrow$ OriginalRoundRobin($n+1$)
  $a \leftarrow M[n+1]$ ▷ The list of matches for the "ghost team" $n+1$
  let $b \leftarrow$ array of length n initialized to `null`
  for $i = 1, \ldots, n$ **do**
    $b[a[i]] \leftarrow i$
  end for
  $M_{\text{new}} \leftarrow$ copy of M
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, n$ **do**
      $M_{\text{new}}[i, j] \leftarrow M[i, b[j]]$
    end for
  end for
  delete the final row of M_{new}
  return M_{new}

Problem 5

This problem is a lot like the original knapsack problem. How can we represent the constraint that the number of items must be even? Remember – when you are given a constraint in a dynamic programming problem, it is almost always good to "extend the dimensionality" of the problem. What does this mean? It means that you make the constraint into an extra dimension of your problem. Just like how we made the weight constraint a dimension of the original knapsack problem, we will keep track of the *parity* (even/oddness) of our solution.

In the original knapsack problem, when adding an item would cause us to violate the capacity condition, we didn't take it. Similarly, in the even knapsack problem, if adding an item would cause us to violate the parity condition, we won't take it. But we don't know whether we have an even number of items until we have considered all the items. So if we get to the end, and we have an odd number of items, we will assign $-\infty$ to that solution, meaning it will be rejected by its parent call.

In the original knapsack problem, we represented the problem with capacity C and having considered all items with index greater than i as $K(C, i)$. Now we will maintain a bit that records the parity of the current collection of items, so we write $K(C, i, p)$ let's say. Here, $p = 1$ if an odd number of the items $i + 1$ to n have been chosen, and 0 if even.

Consider item i. If its weight w_i is greater than C, then we can't take it. In this case, we have to move on to item $i - 1$, and the parity remains the same. So if $w_i > C$, then $K(C, i, p) = K(C, i - 1, p)$.

On the other hand, if its weight is at most C, then we can take it. In this case, we have to decide between taking it and not taking it. If we take the item i, the parity of the collection switches, the capacity goes down by w_i, and the value goes up by v_i. If we don't take it, then everything remains the same. Therefore, if $w_i \leq C$, then $K(C, i, p) = \max\left(K(C - w_i, i - 1, 1 - p) + v_i, K(C, i - 1, p)\right)$.

What about base cases? It's evident that $K(0, i, p) = 0$ for any i or p, since we can't take anything if we have 0 capacity. The more interesting base case is $K(C, 0, p)$. To clarify, this is the case where all items have been considered, and we just need to determine whether the solution is valid. If $p = 0$ (even number of items collected so far) then the solution is valid, and we get 0 (from picking up no items). So $K(C, 0, 0) = 0$. On the other hand, if $p = 1$, then the whole collection built so far is invalid, as it has odd size. In this case, we assign the value $-\infty$ (any suitably large negative number would do, e.g. the negative sum of all the values). So $K(C, 0, 1) = -\infty$. That means that the solution will be discarded when a maximum is taken. If no maximums are taken, that means that no items have been picked up, in which case the parity is 0 (because 0 is even). So we'll never get a ∞ solution.

Putting this all together, we have the recurrence relation:

$$K(C, i, p) = \begin{cases} \max\begin{cases} K(C - w_i, i - 1, 1 - p) + v_i \\ K(C, i - 1, p) \end{cases} & \text{if } w_i \leq C,\ i > 1 \text{ and } C > 0 \\ K(C, i - 1, p) & \text{if } w_i > C,\ i > 1 \text{ and } C > 0 \\ 0 & \text{if } C = 0,\ i > 0 \\ 0 & \text{if } i = 0, p = 0 \\ -\infty & \text{if } i = 0, p = 1 \end{cases}$$

We can write pseudocode with lookup tables to make it more efficient.

```
function EvenKnapsack(C, i, p; ; K)
    C is the capacity
    i is the index of the item we are currently considering
    p is the parity of the current collection of chosen items
    K is the lookup table that stores previous results (called by reference!)
    if K[C, i, p] is null then
        if i = 0 then
            if p = 0 then
                K[C, 0, 0] ← 0
            else
                K[C, 0, 1] ← −∞
            end if
        else if C = 0 then
            K[0, i, p] ← 0
        else if C < w[i] then
            K[C, i, p] ← EvenKnapsack(C, i − 1, p, K)
        else
            K[C, i, p] ← max { EvenKnapsack(C − w[i], i − 1, 1 − p, K) + v[i]
                             { EvenKnapsack(C, i − 1, p, K)
        end if
    end if
    return K[C, i, p]
end function
```

```
function EvenKnapsackWithTrace(C, v, w)
    n ← len(v)
    initialize (C + 1) × n × 2 array K to null
    MaxValue ←EvenKnapsack(C, n, 0, K)
    trace ← empty list
    p ← 0
    for i = n, . . . , 1 do
        if K[C, i, p] ≠ K[C, i − 1, p] then
            trace.append(i)
            C ← C − w[i]
            p ← 1 − p
        end if
    end for
    print trace
end function
```

The complexity of this algorithm is the number of subproblems multiplied by the complexity of each subproblem. In this case, there are $(C + 1) \times n \times 2 = O(nC)$ subproblems. Each one does a constant number of operations – a constant number of assignments, additions, comparisons and accesses. So the overall complexity is $O(nC)$.