This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

You are playing a video game, where you control a character in a grid with m rows and n columns. The character starts at the square in the top left corner $(1, 1)$, and must walk to the square in the bottom right corner (m, n). The character can only move one square at a time *downwards or rightwards*. Every square (i, j), other than the starting square and the ending square, contains a known number of coins $a_{i,j}$.

1.1 [10 marks] Design an algorithm which runs in $O(mn)$ time and determines the maximum number of coins that your character can accumulate by walking from $(1, 1)$ to (m, n) using a combination of downwards and rightwards moves.

Subproblems: For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $P(i, j)$ be the problem of determining the value $\text{opt}(i, j)$, which we define as the maximum number of coins that your character can accumulate by walking from $(1, 1)$ to (i, j).

Recurrence: For $(i, j) \neq (1, 1)$, we have

$$\text{opt}(i, j) = a_{i,j} + \max \begin{cases} \text{opt}(i, j-1), & \text{if } j \neq 1 \\ \text{opt}(i-1, j), & \text{if } i \neq 1 \end{cases}$$

Here we define $a_{m,n}$ to be 0 since we can treat the bottom right square as having 0 coins. In the first case, we consider arriving at square (i, j) from the left, and in the second case, we consider arriving from above.

Base cases: There are no coins on the path from $(1, 1)$ to $(1, 1)$ since $(1, 1)$ contains no coins, so $\text{opt}(1, 1) = 0$.

Overall answer: $\text{opt}(m, n)$.

Order of computation: Each subproblem $P(i, j)$ depends on subproblems $P(i-1, j)$ and $P(i, j-1)$, so we can solve subproblems in lexicographic order.

Time complexity: There are $O(mn)$ subproblems, each solved in $O(1)$ time. Therefore the overall time complexity is $O(mn)$.

1.2 [10 marks] After playing this game many times, you have broken the controller, and you can no longer control your character. They now walk randomly as follows:

- if there is only one possible square to move to, they move to it;
- otherwise, they move right with probability p and down with probability $1 - p$.

Note that this guarantees that the character arrives at (m, n).

Design an algorithm which runs in $O(mn)$ time and determines the *expected* number of coins that your character will accumulate by walking from $(1, 1)$ to (m, n) according to the random process above.

Recall that for a discrete random variable X which attains values $x_1, \ldots, x_n$ with probabilities $p_1, \ldots, p_n$, the *expected value* of X is defined as

$$\mathbb{E}(X) = \sum_{i=1}^{n} p_i x_i.$$

First, we will compute the probability of passing square (i, j) on a random path.

Subproblems: For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $Q(i, j)$ be the problem of determining the value $f(i, j)$, which we define as the probability that your character will pass square (i, j) on a random path.

Recurrence: For $i, j > 1$, we have

$$f(i,j) = \begin{cases} p \times f(i,j-1) + (1-p) \times f(i-1,j) & \text{if } i < m, j < n \\ f(i,j-1) + (1-p) \times f(i-1,j) & \text{if } i < m, j = n \\ p \times f(i,j-1) + f(i-1,j) & \text{if } i = m, j < n \\ f(i,j-1) + f(i-1,j) & \text{if } i = m, j = n. \end{cases}$$

Suppose the square $(i, j-1)$ to the left of (i, j) is reached with probability x. Then such a path has probability p of continuing right to (i, j), *except* when $i = m$, where this is a forced move with probability 1. Thus the contribution to $f(i, j)$ from paths through $(i, j-1)$ is px if $i < m$ and x otherwise. Paths through $(i-1, j)$ are handled similarly, with cases $j < n$ and $j = n$.

Base cases:

- $f(1, 1) = 1$, since every path goes through the starting square.
- $f(1, j) = p^{j-1}$, since the jth square of the first row can only be reached by a sequence of $j - 1$ rightwards moves.
- $f(i, 1) = (1-p)^{i-1}$, since the ith square of the first column can only be reached by a sequence of $i - 1$ downwards moves.

Overall answer: all $f(i, j)$ (to be used in the expected value calculation below).

Order of computation: Each subproblem $Q(i, j)$ depends on subproblems $Q(i-1, j)$ and $Q(i, j-1)$, so we can solve subproblems in lexicographic order.

Time complexity: There are $O(mn)$ subproblems, each solved in $O(1)$ time. Therefore the overall time complexity is $O(mn)$.

Now, we will use these probabilities to retrieve the expected value. For a path s from $(1, 1)$ to (m, n), let $\mathbb{P}(s)$ be the probability of taking this particular path.

$$\begin{aligned} &\text{Expected number of coins} \\ &= \sum_{\text{path } s:(1,1)\to(m,n)} (\mathbb{P}(s) \times \text{Number of coins on path } s) \\ &= \sum_{\text{path } s:(1,1)\to(m,n)} \left(\mathbb{P}(s) \times \left[\sum_{(i,j)\in s} a_{i,j} \right] \right). \\ &= \sum_{\text{path } s:(1,1)\to(m,n)} \left[\sum_{(i,j)\in s} (\mathbb{P}(s) \times a_{i,j}) \right]. \end{aligned}$$

Now, we can rearrange this sum by grouping all the terms relating to a particular cell (i, j) together. Factorising $a_{i,j}$, which is common to all such terms, leaves

$$\sum_{\text{paths } s \ni (i,j)} \mathbb{P}(s),$$

which is exactly $f(i,j)$. Therefore the expected number of coins is simply

$$\sum_{i=1}^{m}\sum_{j=1}^{n}\left[f(i,j)\times a_{i,j}\right],$$

which we can calculate in $O(n^2)$ from our previous computed $f(i,j)$ values.

A simpler alternative solution is to consider paths from arbitrary (i,j) to the destination (m,n).

Subproblems: For $1 \le i \le m$ and $1 \le j \le n$, let $R(i,j)$ be the problem of determining the value $h(i,j)$, which we define as the expected number of coins that your character will accumulate by walking from (i,j) to (m,n) according to the random process.

Recurrence: We have

$$h(i,j) = a_{i,j} + \begin{cases} 0, & \text{if } i=m, j=n; \\ h(i,j+1), & \text{if } i=m, j\neq n; \\ h(i+1,j), & \text{if } j=n, i\neq m; \\ p\times h(i,j+1)+(1-p)\times h(i+1,j) & \text{otherwise.} \end{cases}$$

Here we define $a_{1,1}$ and $a_{m,n}$ to be 0 since we can consider treat the starting and ending square both as having 0 coins. The first case is the base case, where the only path from (m,n) to (m,n) contains 0 coins, so the expected number of coins on the path is 0. The second and third cases reflect the fact that the only way to continue from a square in the last row or column is to walk right or down respectively, making the expected value simply the sum of $a_{i,j}$ and the expected value from the next square.

The final case reflects the random process, since

$$\begin{aligned} h(i,j) &= \text{expected number of total coins starting at } (i,j) \\ &= \sum_{\text{paths from } (i,j)} \text{number of coins on path} \times \text{probability of taking path} \\ &= \sum_{\text{paths from } (i,j)} \text{coins on path} \times \text{probability of going right then from } (i,j+1), \text{ or down then from } (i+1,j) \\ &= a_{i,j} + \left[p \times \sum_{\text{paths from } (i,j+1)} \text{coins} \times \text{probability}\right] + \left[(1-p)\times \sum_{\text{paths from } (i+1,j)} \text{coins} \times \text{probability}\right] \\ &= a_{i,j} + p\times h(i,j+1) + (1-p)\times h(i+1,j). \end{aligned}$$

Overall answer: $h(1,1)$, the expected value from the starting cell.

Order of computation: Each subproblem $R(i,j)$ depends on subproblems $R(i+1,j)$ and $R(i,j+1)$, so we can solve subproblems in reverse lexicographic order.

Time complexity: There are $O(n^2)$ subproblems, each solved in $O(1)$ time. Therefore the overall time complexity is $O(n^2)$.

Question 2

You are managing a garage with two mechanics, named Alice and Bob. Each mechanic can serve at most one customer at a time.

There will be n customers who come in during the day. The ith customer wants to be served for the entire period of time starting at time s_i and ending at time e_i. You may assume that the customers are indexed by their order of arrival, i.e. $s_1 < s_2 < \ldots < s_n$.

For each customer i, the business will:

- make a_i dollars if customer i is served by Alice;
- make b_i dollars if customer i is served by Bob;
- lose c_i dollars if customer i is not served.

Your task is to maximise the net earnings of the garage, which is calculated as the total amount made minus the total amount lost.

2.1 **[8 marks]** Consider the following greedy algorithm.

Process each customer i in order of arrival as follows.

- If both Alice and Bob are available at time s_i:
  - if $a_i \geq b_i$, assign customer i to Alice;
  - otherwise, assign the customer to Bob.
- If only one mechanic is available at time s_i, assign customer i to that mechanic.
- If neither mechanic is available at time s_i, do not serve customer i.

Design an instance of the problem which is not correctly solved by this algorithm. You must:

- specify a number of customers n,
- for each customer, provide values for s_i, e_i, a_i, b_i and c_i,
- apply the greedy algorithm to this instance and calculate the net earnings achieved, and
- show that a higher net earnings figure can be achieved.

This greedy approach clearly fails if you occupy both mechanics early on with low profit, long duration tasks which causes them to miss high value customers that come afterwards. For example, suppose $n = 3$ and the customers are as tabulated below.

i	s_i	e_i	a_i	b_i	c_i
1	1	8	1	0	0
2	2	9	0	1	0
3	5	6	100	100	0

The greedy algorithm sends customer 1 to Alice and customer 2 to Bob for net earnings of 2. However, the maximum net earnings is 101, achieved by ignoring one of the first two customers and assigning customer 3 to the corresponding mechanic instead.

Another approach is to occupy a mechanic with a low value customer, only to be followed by a high value customer for that same mechanic. For example, suppose $n = 2$ and the

customers are as tabulated below.

i	s_i	e_i	a_i	b_i	c_i
1	1	8	2	0	0
2	2	9	3	0	0

The greedy algorithm sends customer 1 to Alice and customer 2 to Bob for net earnings of 2. However, the maximum net earnings is 3, achieved by sending customer 1 to Bob and customer 2 to Alice.

2.2 [12 marks] Design an algorithm which runs in $O(n^3)$ time and determines the maximum net earnings of the garage.

The following solution requires some work to set up, but gives the most natural extension to a quadratic time solution.

Let $e_0 = -\infty$.

Preliminaries:

- Relabel the customers in increasing order of end time in $O(n \log n)$ using merge sort.
- Let
$$p(i) = \max\{j : 0 \leq j < i, e_j \leq s_i\}.$$
This is the last customer to end no later than customer i starts[a].
  - The significance of this is that a mechanic can serve any customer up to $p(i)$ before serving customer i.
  - We can compute all $p(i)$ by brute force from the definition in $O(n^2)$.
  - This can be improved to $O(n \log n)$ using binary search or even $O(n)$, but other terms will dominate it in the final time complexity so these optimisations have no effect.
- Also, let
$$C(i, j) = \sum_{k=i+1}^{j-1} c_k.$$
  - This is the cost of skipping all customers between i and j exclusive.
  - We will discuss the computation of the $C(i, j)$ in the 'Time Complexity section below.

Subproblems: Suppose $0 \leq i_A, i_B \leq n$. Then let $P(i_A.i_B)$ be the problem of determining $\text{opt}(i_A, i_B)$, the maximum earnings considering only the first $m = \max(i_A, i_B)$ customers, where Alice hasn't served any customers after customer i_A and Bob hasn't served any after i_B. We interpret $i_A = 0$ to mean that Alice hasn't served any customers, and $i_B = 0$ similarly.

Recurrence: Let $m = \max(i_A, i_B)$ as above, and suppose $m > 0$.

If $i_A = m$ and $i_B < m$, we have

$$\text{opt}(m, i_B) = \max\left([\text{opt}(p(m), i_B) + a_m - C(\max(p(m), i_B), m)], [\text{opt}(m-1, i_B) - c_m]\right).$$

- The first term of the maximum corresponds to assigning customer m to Alice.
  - Before serving customer m, Alice's previous customer must have index at most $p(m)$.

- This gains customer m's price for Alice a_m.
- However, we cannot serve the customers between m and whichever is larger of $p(m)$ and i_B, so we incur the cost $C(\max(p(m), i_B), m)$.

- The second term corresponds to skipping customer m.
  - Alice can now serve customers up to $m-1$.
  - We incur cost c_m for skipping customer m.

Similarly, if $i_B = m$ and $i_A < m$, we have

$$\text{opt}(i_A, m) = \max\left([\text{opt}(i_A, p(m)) + b_m - C(\max(p(m), i_A, m)], [\text{opt}(i_A, m-1) - c_m]\right).$$

The first term corresponds to assigning customer m to Bob and the second term to skipping customer m.

Finally, if $i_A = i_B = m$, then we have

$$\text{opt}(m, m) = \max\left([\text{opt}(p(m), m-1) + a_m], [\text{opt}(m-1, p(m)) + b_m], [\text{opt}(m-1, m-1) - c_m]\right),$$

corresponding to assigning customer m to Alice, Bob or neither respectively.

Base case: $\text{opt}(0, 0) = 0$.

Final answer: The answer is simply $\text{opt}(n, n)$, when all customers are considered.

Order of computation: $P(i_A, i_B)$ depends on subproblems $P(i'_A, i'_B)$ where either $i'_A < i_A$ or we have $i'_A = i_A$ and $i'_B < i_B$. Therefore we can solve the subproblems in lexicographic order.

Time complexity: There are $O(n^2)$ subproblems, and each recurrence equation involves a constant number of operations *except* for the computation of the $C(i, j)$. Each $C(i, j)$ term is the sum of potentially $O(n)$ terms $c_{i+1}, \ldots, c_{j-1}$, so it appears that the recurrence takes $O(n)$ time.

However, we can improve this using some precomputation. For each $0 \leq i \leq n$, let $z_i = c_1 + \ldots + c_i$. These values can be computed in a total of $O(n)$ using the recurrence $z_i = z_{i-1} + c_i$. Then, we can query any $C(i, j)$ value in constant time since

$$\begin{aligned} C(i, j) &= c_{i+1} + \ldots + c_{j-1} \\ &= (c_1 + \ldots + c_i + c_{i+1} + \ldots + c_{j-1}) - (c_1 + \ldots + c_i) \\ &= z_{j-1} - z_i. \end{aligned}$$

With this optimisation, our algorithm now runs in $O(n^2)$.

[a] We allowed any interpretation regarding endpoints, i.e. customers with times (t_1, t_2) and (t_2, t_3) could be interpreted as conflicting or non-conflicting. This solution uses the latter assumption

This alternative solution allows a more direct processing of the customers in the original order, but cannot be optimised to $O(n^2)$.

Let $e_0 = -\infty$ and $s_{n+1} = \infty$.

Subproblems: Suppose $0 \leq i, j, k \leq n$ such that $i, j \leq k$ and $i \neq j$ unless both are zero. Then define subproblems as follows.

- Let $P(i, j, k)$ be the problem of determining the maximum earnings $f(i, j, k)$ from customers $1..k$, where i is the index of the last customer served by Alice, and j is the index

of the last customer served by Bob.

- When $i = 0$ or $j = 0$, this denotes that the corresponding mechanic has served no customers.

- Let $Q(j, k)$ be the problem of determining $g(j, k)$, the maximum earnings from customers $1..k$, where j is the index of the last customer served by Bob, and Alice is free at time s_{k+1}.
- Let $R(i, k)$ be the problem of determining $h(i, k)$, the maximum earnings from customers $1..k$, where i is the index of the last customer served by Alice, and Bob is free at time s_{k+1}.

Recurrence: For $i, j, k > 0$, we have

$$f(i,j,k) = \begin{cases} f(i,j,k-1) - c_k & \text{if } i, j < k \\ g(j,k-1) + a_k & \text{if } i = k \\ h(i,k-1) + b_k & \text{if } j = k, \end{cases}$$

where customer k is skipped in the first case, assigned to Alice in the second and assigned to Bob in the third. We also have

$$g(j,k) = \max\{f(p,j,k) : e_p \leq s_{k+1}, p \neq j\}$$

and

$$h(i,k) = \max\{f(i,q,k) : e_q \leq s_{k+1}, q \neq i\}$$

from the definitions of these quantities.

Base cases: $f(0,0,0) = g(0,0) = h(0,0) = 0$.

Final answer: The answer is obtained by $\max_{1 \leq i,j \leq n} f(i,j,n)$, where all customers have been considered and any customer could be the last served by either mechanic.

Order of computation:

- $P(i,j,k)$ depends on $P(\cdot,\cdot,k-1)$, $Q(j,k)$ and $R(i,k)$
- $Q(j,k)$ depends on $P(\cdot,j,k)$
- $R(i,k)$ depends on $P(i,\cdot,k)$.

Therefore, we iterate through the subproblems in increasing order of k, first solving all the $P(\cdot,\cdot,k)$, then all the $Q(\cdot,k)$ and $R(\cdot,k)$.

Time complexity: There are $O(n^3)$ subproblems $P(i,j,k)$ each solved in constant time, $O(n^2)$ subproblems $Q(j,k)$ each solved in $O(n)$ time and $O(n^2)$ subproblems $R(i,k)$ each solved in $O(n)$ time. The final answer is found in $O(n^2)$ time. Therefore the overall time complexity is $O(n^3)$.

Question 3

You are given a simple directed weighted graph with n vertices and m edges. The edge weights *may* be negative, but there are no cycles whose sum of edge weights is negative.

3.1 **[10 marks]** An edge e is said to be *useful* if there is some pair of vertices u and v such that e belongs to **at least one** shortest path from u to v.

Design an algorithm which runs in $O(n^3)$ and determines the set of useful edges.

Run the Floyd-Warshall algorithm on the graph, which runs in $O(n^3)$. After this, we may obtain $\text{dist}(u, v)$, the length of a shortest path from u to v, in constant time, for any pair of vertices u and v.

Now for each edge e (of which there are at most $O(n^2)$ in a simple graph), from a vertex u to a vertex v with weight w, we claim it is useful if and only if $\text{dist}(u, v) = w$.

First observe that $\text{dist}(u, v)$ cannot exceed w, since edge e is itself a candidate for this distance.

Now, if $\text{dist}(u, v) = w$, then the path from u to v using only the edge e costs w, and so is a shortest path. This satisfies the definition of usefulness of e.

On the other hand, suppose the shortest $u \to v$ path in the graph is of weight less than w. Let p be one such path. Now, if e is useful, then there is a shortest $u' \to v'$ path which includes e, but replacing e with p yields an even shorter path. Therefore e is not useful.

The steps take $O(n^3)$ and $O(n^2)$, so the final time complexity is $O(n^3)$.

3.2 **[10 marks]** An edge is said to be *very useful* if there is some pair of vertices u and v such that e belongs to **every** shortest path from u to v.

Design an algorithm which runs in $O(n^3)$ and determines the set of very useful edges.

Let e be an edge from vertex u to vertex v of weight w.

Claim: Edge e is very useful if and only if it belongs to every shortest $u \to v$ path.

Proof: The 'if' direction holds by the definition of "very useful". For the "only if" direction, suppose that there is a shortest $u \to v$ path p which does not include e, but that e belongs to every shortest $x \to y$ path. This is clearly impossible, since replacing all instances of e with p on any shortest $x \to y$ path gives a new path without e of no greater weight, contradicting the assumption. It follows that an edge which does not belong to every shortest path between its endpoints cannot be very useful.

Is the following statement (which we will designate as $(\star)$) equivalent to the above claim?

> Edge e is very useful if and only if every $u \to v$ path consisting of two or more edges has weight strictly greater than w.

The 'if' direction is certainly true, but unfortunately, the 'only if' direction is not. It is possible that edge e is very useful even if there is a $u \to v$ path consisting of two or more edges with weight equal to w, so long as e belongs to this path also. Such a path must be of one of the following forms:

$$\underbrace{u \to \ldots \to u}_{C_u} \underbrace{\to}_{e} v$$

$$u \underbrace{\to}_{e} \underbrace{v \to \ldots \to v}_{C_v}$$

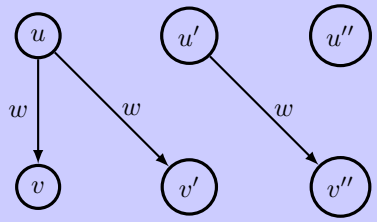
$$\underbrace{u \to \ldots \to u}_{C_u} \underbrace{\to}_{e} \underbrace{v \to \ldots \to v}_{C_v}$$

where C_u and C_v are cycles. Since there are no negative weight cycles, any such cycle must be of zero weight.

If there are no cycles of zero weight, then the statement $(\star)$ above is equivalent to our earlier claim, and we will see that the problem can be solved in $O(n^3)$. Unfortunately, to the best of our knowledge there is no $O(n^3)$ algorithm which solves the problem without this added restriction[a]. Therefore, in the rest of this solution, we assume that the graph has no cycles of zero weight.

Let the original graph be $G = (V, E)$. Construct a new graph G' initially with the same vertices and edges as G, then:

- for each vertex $v \in V$, add vertices v' and v'' to G'.
- for each edge $(u, v) \in E$ of weight w, add edges (u, v') and (u', v'') of weight w to G'.



Now, run the Floyd-Warshall algorithm on G'. A path from any vertex u to a vertex v'' corresponds to a path in G with two or more edges. It follows that the shortest $u \to v''$ path in G' has the same path weight as the shortest $u \to v$ path in G with two or more edges. Therefore, by statement $(\star)$ above, we can test whether an edge is very useful by comparing its weight to this path length.

There are $3n$ vertices, so the Floyd-Warshall algorithm runs in $O((3n)^3) = O(n^3)$ time, and then each of $m < n^2$ edges can be checked in constant time, so the overall time complexity is $O(n^3)$.

[a]The best algorithm that we are aware of is as follows: for each useful edge (u, v), delete it and run the Bellman-Ford algorithm from u on the remaining graph, then compare the distance found to v against the weight of the deleted edge. This has a total time complexity of $O(nm^2)$ in the worst case.

Question 4

There are $2n$ players who have signed up to a chess tournament. For all $1 \leq i \leq 2n$, the ith player has a known skill level of s_i, which is a non-negative integer. Let $S = \sum_{i=1}^{2n} s_i$, the total skill level of all players.

In the tournament, there will be n matches. Each match is between two players, and each player will play in exactly one match. The *imbalance* of a match is the absolute difference between the skill levels of the two players. That is, if a match is played between the ith player and the jth player, its imbalance is $|s_i - s_j|$. The *total imbalance* of the tournament is the sum of imbalances of each match.

The organisers have provided you with a value m which they consider to be the ideal total imbalance of the tournament.

Design an algorithm which runs in $O(n^2 S)$ time and determines whether or not it is possible to arrange the matches in order to achieve a total imbalance of m, assuming:

4.1 [**4 marks**] all s_i are either 0 or 1;

Call players 'good' if they have skill level 1, and 'bad' otherwise. Suppose there are g good players and b bad players.

Each match has imbalance of either 0 or 1. For total imbalance m, we require m matches between a good player and a bad player, and $n - m$ matches between two good players or two bad players. Therefore, we require that:

- $g, b \geq m$
- $g - m$ and $b - m$ are even.

These two conditions are not only necessary but also sufficient; after making matches between m good-bad pairs, we can make pairs arbitrarily among the remaining $g - m$ good players and among the remaining $b - m$ bad players.

Therefore, it suffices to count the number of good players and bad players in $O(n)$ time, and confirm whether these two criteria are met.

4.2 [**16 marks**] the s_i are distinct non-negative integers.

Begin by sorting the players in descending order of skill, and relabelling accordingly. Using mergesort, this takes $O(n \log n)$ time.

We will refer to the player with higher skill level in a match as the 'winner' and the other player as the 'loser'. The winners must have total skill m greater than the losers, and the total of all skill levels is S, so it follows that the winners must have total skill $\frac{S+m}{2}$. If $S + m$ is not even, the answer is trivially F.

Subproblems: For $0 \leq i \leq 2n$, $0 \leq j \leq n$ and $0 \leq k \leq S$, let $P(i, j, k)$ be the problem of determining the boolean $f(i, j, k)$, which records whether it is possible to select j winners with total skill k from players $1..i$. Note that if $i > 2j$, the answer is immediately F, since we can never have more losers than winners encountered so far.

Recurrence: For $i > 0$, we have

$$f(i, j, k) = f(i - 1, j, k) \vee f(i - 1, j - 1, k - s_i).$$

In the first case, we designate player i as a loser, so the j winners with total skill k must be

selected from the first $i-1$ players. In the second case, we designate player i as a winner, leaving $j-1$ winners of total skill $k-s_i$ to be selected from the first $i-1$ players.

Base cases: When no players have been considered, there are no winners to select, so $f(0,0,0) = T$ and $f(0,j,k) = 0$ if $j \neq 0$ or $k \neq 0$. As above, we also have $f(i,j,k) = F$ if $i > 2j$.

Overall answer: The matches can be arranged if and only if n winners can be selected from all $2n$ players with total skill $\frac{S+m}{2}$, i.e.

$$f\left(2n, n, \frac{S+m}{2}\right) = T.$$

Order of computation: Each subproblem $P(i,j,k)$ depends on subproblems $P(i-1,\cdot,\cdot)$, so we must solve subproblems in increasing order of i, e.g. lexicographic order.

Time complexity: Sorting the players by skill level takes $O(n \log n)$. Then, there are $O(n^2 S)$ subproblems, each solved in constant time. Therefore the overall time complexity is $O(n^2 S)$.