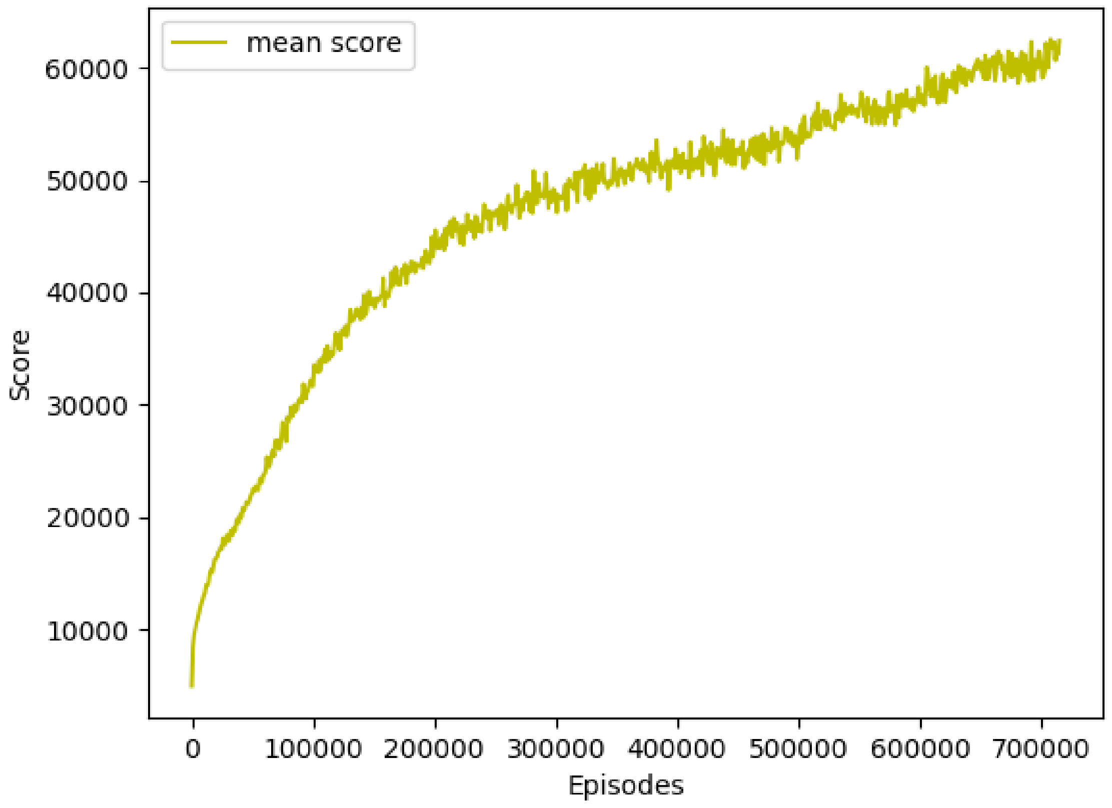
Lab2 : Temporal Difference Learning

資訊工程學系
學號:0816095
王軒

Table of Contents

1. Plot

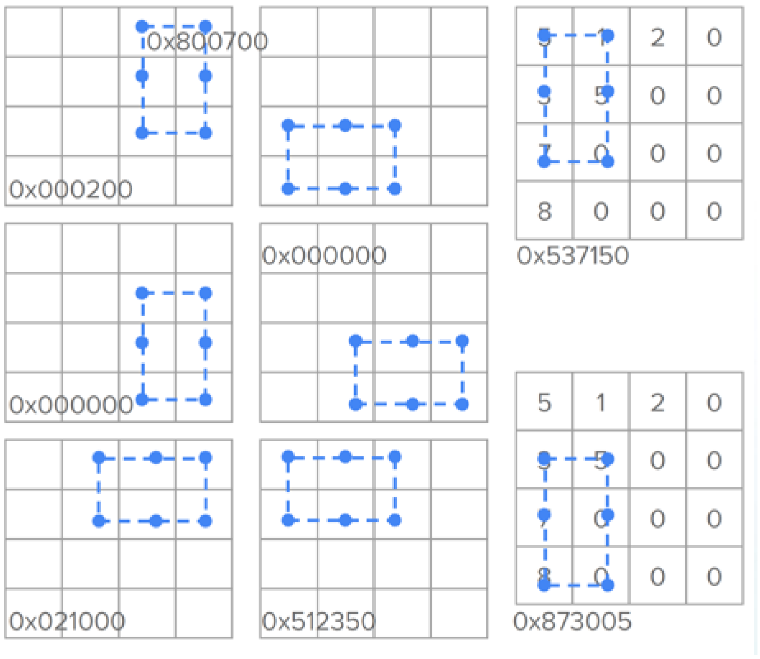
A plot that shows the mean score of 700000 episodes. We can see that it is still learning new things and still improving, so I kept on training. This graph is just an illustration of the training process.



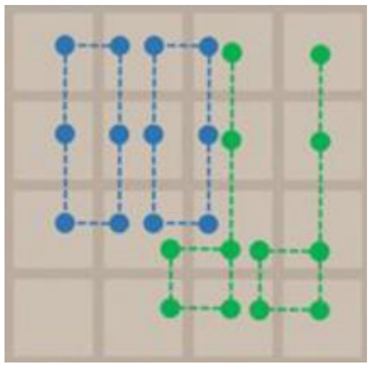
2. Implementation and the usage of n-tuple network

The reason for n-tuple network is because for some problem, there are just too many situations to consider, such that it is impossible for us to map a state to each of them (In the 2048 case there are a total of 16 blocks in the board, and each block has 17 different possible outcomes, that brings us to a total of 17^{16} states). Thus, we use tuples to overcome this problem by sacrificing some performance.

Instead of mapping a state to a situation one-to-one, we define tuples that contain part of the situation (part of the board), and use isomorphisms to cover every corner of the board and mirrored pattern.

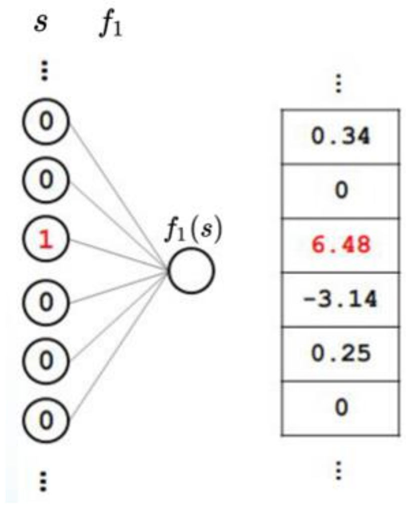


In the 2048 case, we can define several tuples of different shapes to try to identify useful patterns as much as possible.



In this lab I used 4 6-tuple to represent the board. Each tuple has 6 blocks and each block has 17 possible outcomes as previously mentioned. This gives us 17^6 possible state for each tuple, and we have 4 tuples, so that is $4 * 17^6$ parameters. This is a much smaller number compared to 17^{16} , and is easier to work with.

For this configuration, we have $4 * 17^6$ parameters to train, each representing the estimated value of their respected pattern. For each board of the 2048 game, we can apply the 4 tuples and all their isomorphisms to the board and get the estimated values of 4 * 8 different patterns. Sum up all the values to get the overall estimation of the board.



This graph illustrates the evaluation process of a single tuple

$$V(s) = f_1(s) + f_2(s) + f_3(s) + f_4(s)$$

V(s) represents the estimated value of a board, and each f(s) represents the sum of all 8 isomorphisms of a single pattern

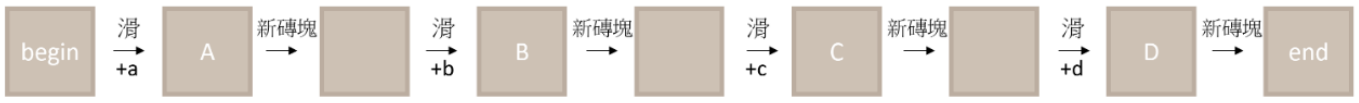
3. Mechanism of TD(0)

Temporal Difference Learning is a type of reinforcement learning, different from Monte Carlo learning, it doesn’t need to play through an entire episode to do an update, the difference between states are adequate to make an update.

“Suppose you wish to predict the weather for Saturday, and you have some model that predicts Saturday’s weather, given the weather of each day in the week. In the standard case, you would wait until Saturday and then adjust all your models. However, when it is, for example, Friday, you should have a pretty good idea of what the weather would be on Saturday – and thus be able to change, say, Saturday’s model before Saturday arrives.”

This is the core idea of temporal difference learning, and depending on how many steps you want to take before updating, we have TD(λ), where λ is the parameter indicating how many steps you want the model to consider.

Consider the following graph:

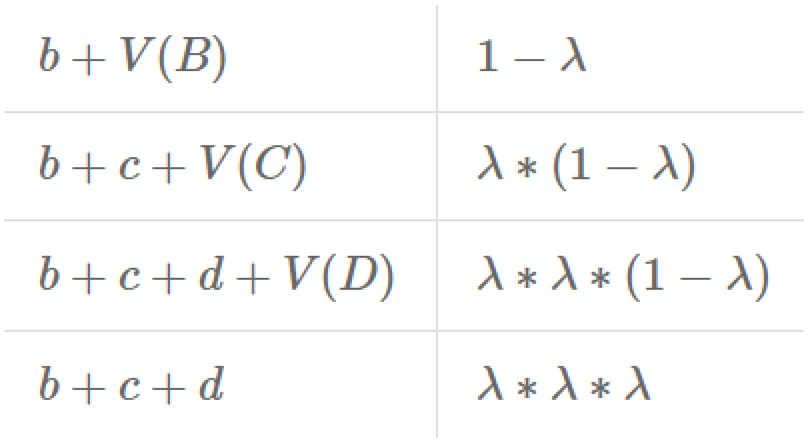


This is an episode of the game 2048, and suppose we want to update state A’s estimation value.

In Monte Carlo learning, V(A)’s learning target should be b + c + d, but that requires us to finish the episode before we can update.

In TD(0) we only consider the next state, thus V(A)’s learning target becomes b + V(B).

TD(λ) is just a combination of all different targets with different step number, and by using the parameter λ we can determine if we want to put emphasis on shorter step updates or longer step updates.



$b + V(B)$	$1 - \lambda$
$b + c + V(C)$	$\lambda * (1 - \lambda)$
$b + c + d + V(D)$	$\lambda * \lambda * (1 - \lambda)$
$b + c + d$	$\lambda * \lambda * \lambda$

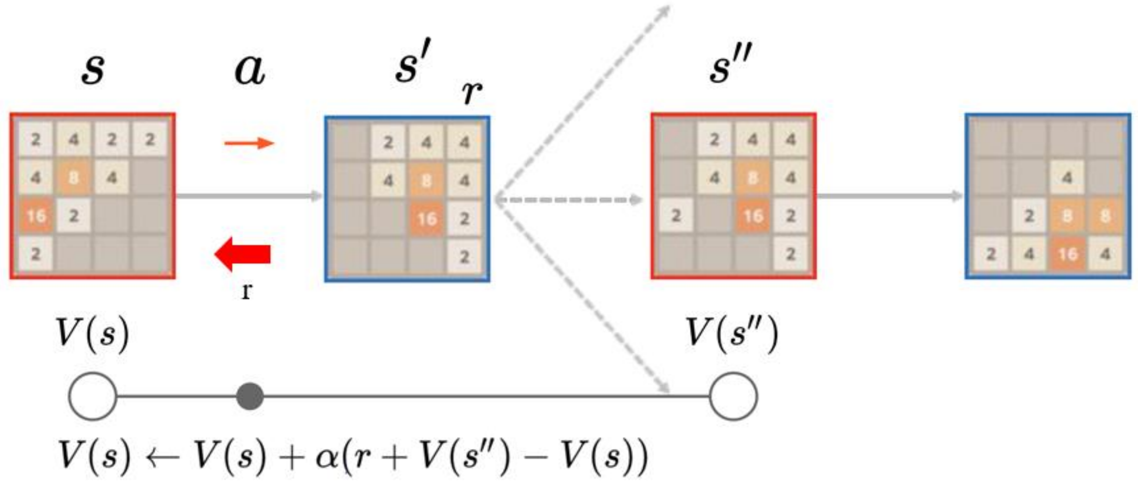
The left represents the targets of different steps, and the right represents their proportion in the update.

We can see that TD(0) is just a special case of TD(λ), and TD(1) is just Monte Carlo learning.

4. Implementation detail

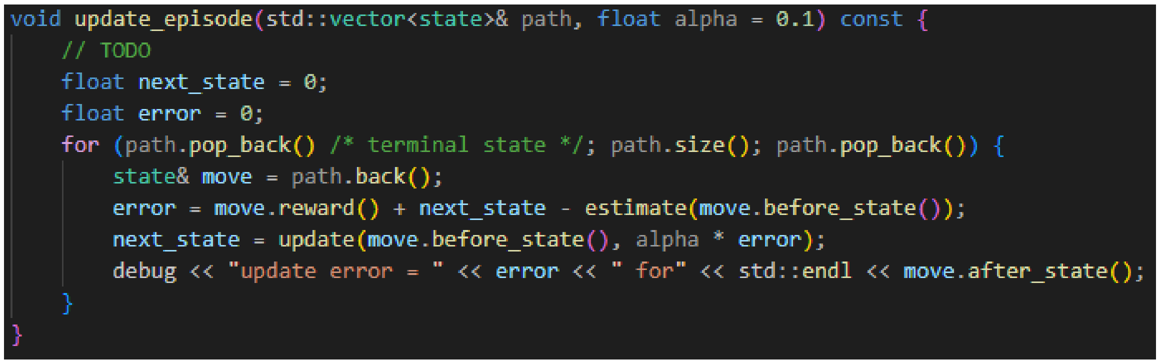
A. TD-Backup diagram

Because 2048 is a special game, after you take an action, and the environment changes, it will have a popup tile that is completely random. Thus after you take an action, it doesn’t immediately goes to next state, instead it becomes a temporary state after the current state, so called after_state (blue), and the state before is called before_state (red).



According to spec, we need to update before_state instead of after_state.

So, after every episode, we store the entire trajectory of that episode for later training. During training, we move from the end of the trajectory to the beginning, updating each before_state according to TD(0).



```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float next_state = 0;
    float error = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        error = move.reward() + next_state - estimate(move.before_state());
        next_state = update(move.before_state(), alpha * error);
        debug << "update error = " << error << " for" << std::endl << move.after_state();
    }
}
```

In the code, next_state represents next state’s before_state’s estimated value, because we start from the end of the trajectory, it is initially set to 0, and error represents TD error.

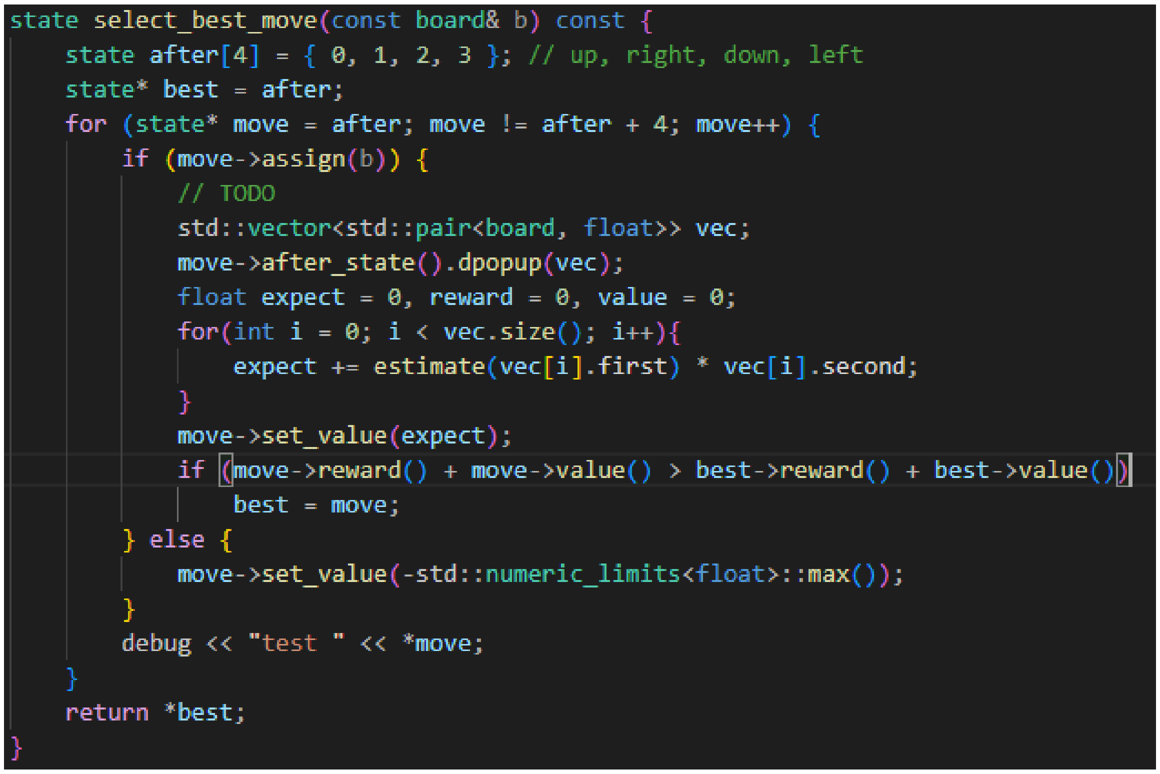
B. Action selection

Because we chose to update before_state, we have to choose our next action by considering V(before_state), not V(after_state), so our target should be reward + V(next before_state).

Reward is self-explanatory, but the next before_state contains probability, thus we have to calculate expectation, so the target becomes reward + E[V(next before_state)].

According to the rule of 2048, the next popup tile has a probability of 0.9 to be a 2 and 0.1 to be a 4, with completely random position within the empty tiles. So, to calculate the expected Value of the next before_state, we multiply the probability of all different possible next before_states with their respective estimations, and sum them up.

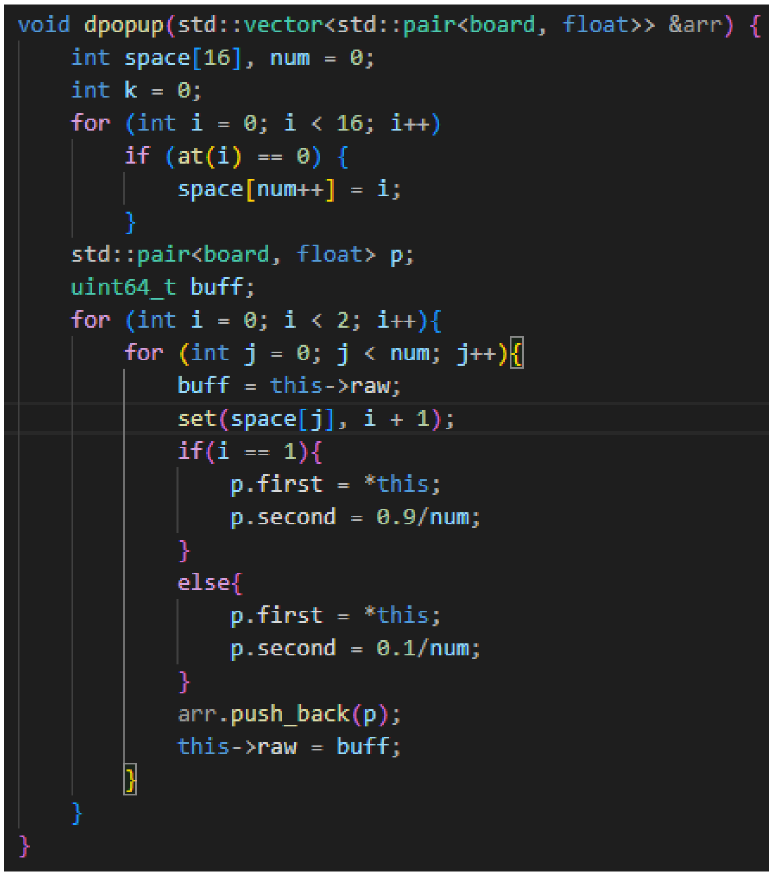
Finally we choose the action with the highest reward + E[V(next before_state)].



```
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
    state* best = after;
    for (state* move = after; move != after + 4; move++) {
        if (move->assign(b)) {
            // TODO
            std::vector<std::pair<board, float>> vec;
            move->after_state().dpopup(vec);
            float expect = 0, reward = 0, value = 0;
            for(int i = 0; i < vec.size(); i++){
                expect += estimate(vec[i].first) * vec[i].second;
            }
            move->set_value(expect);
            if (move->reward() + move->value() > best->reward() + best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

The first for loop loops through all different action. And to simulate the popups, I wrote a function called dpopup() that takes a vector<pair<board, float>> that's made up of pairs of board and float as input, and store all the possible next before_state and their respective probabilities into the vector. Then the second for loop loops through all the possible next before_state in the vector and calculate the expected value.

Finally we choose the highest as our best move.

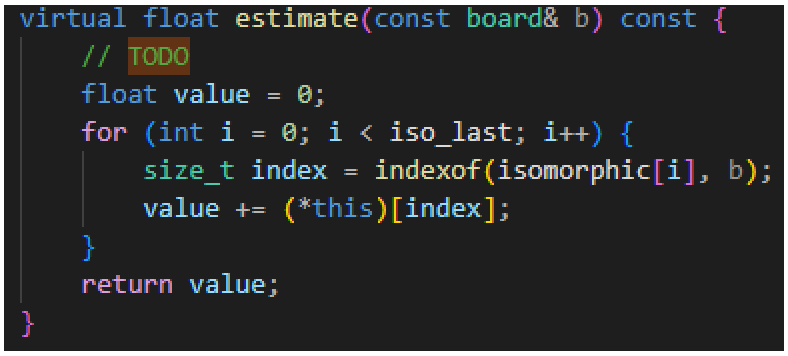


```cpp
void dpopup(std::vector<std::pair<board, float>> &arr) {
    int space[16], num = 0;
    int k = 0;
    for (int i = 0; i < 16; i++)
        if (at(i) == 0) {
            space[num++] = i;
        }
    std::pair<board, float> p;
    uint64_t buff;
    for (int i = 0; i < 2; i++){
        for (int j = 0; j < num; j++){
            buff = this->raw;
            set(space[j], i + 1);
            if(i == 1){
                p.first = *this;
                p.second = 0.9/num;
            }
            else{
                p.first = *this;
                p.second = 0.1/num;
            }
            arr.push_back(p);
            this->raw = buff;
        }
    }
}
```

Implementation of dpopup()

C. Pattern estimate

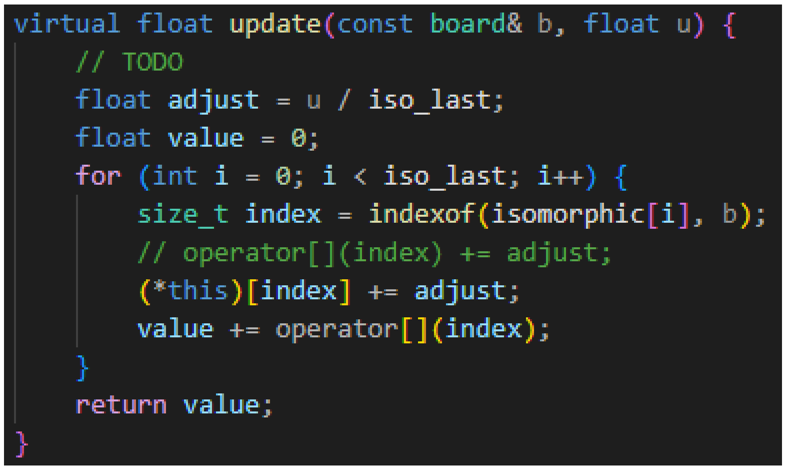
Loops through all 8 isomorphisms of a tuple and estimate each of value and return the sum, which is the estimated value of this board according to this particular pattern (feature).



```
virtual float estimate(const board& b) const {
    // TODO
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        value += (*this)[index];
    }
    return value;
}
```

D. Pattern update

Loops through all 8 isomorphisms and adjust the weight, the error is split evenly for all isomorphisms.



```
virtual float update(const board& b, float u) {
    // TODO
    float adjust = u / iso_last;
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        // operator[](index) += adjust;
        (*this)[index] += adjust;
        value += operator[](index);
    }
    return value;
}
```