Problem 1. Run the code as is to find ranges of **CAPACITY** and **NUMITEMS** that require around 10 seconds to run.

Capacity	Items	Memory inefficient (value & solution)	Memory efficient (value only)	Memory efficient (value & solution)
50,000	5,000	15.8426	5.81777	13.6626
25,000	10,000	12.6301	5.68191	13.2752
100,000	2,500	12.6167	5.97705	14.1457

Table 1: Some sets of capacity and number of items that run at around 10 seconds

Observation: Here, the products of capacity and item are all close to 250,000,000.

Conclusion: due to the runtime complexity of dynamic programming version of knapsack problem $\mathcal{O}(\log nW)$, we can see that the running time is positively proportional to the number of items and the capacity of the knapsack.

Problem 2. Read the code and see if there are any cache locality optimizations that could improve performance.

As Figure 1 and 2 show, in the memory inefficient part of the code, the way of storing the data doesn't match the way of accessing the data. To optimize this, either we change the way of indexing, or we change the way we access the table. Once they are in the same style (both in column major or both in row major), we achieve the spatial locality of cache.

```
// macro for linear indexing into 2D table (column-major)
inline int li(int w, int j, int W) {return w+j*(W+1);}
```

Figure 1: Index the data by column major

```
// loop over 2D table
for(w = 0; w <= W; w++) {
    for(j = 1; j < numItems; j++) {
        // if item doesn't fit or not including it is better
        if(w < wts[j] || K[li(w,j-1,W)] >= vals[j] + K[li(w-wts[j],j-1,W)]) {
            K[li(w,j,W)] = K[li(w,j-1,W)];
        } else { // else including it is better
            K[li(w,j,W)] = vals[j] + K[li(w-wts[j],j-1,W)];
        }
    }
}
```

Figure 2: Build up table by row major

Capacity	Items	Memory inefficient (value & solution)	Memory efficient (value only)	Memory efficient (value & solution)
1000	1000	0.0260408	0.0168568	0.384746
5000	1000	0.175559	0.109112	0.244531
5000	5000	1.28282	0.540613	1.20950
10000	1000	0.39598	0.226099	0.516556
10000	10000	5.555875	2.22395	5.07604
20000	2000	2.24177	0.921218	2.13556
50000	5000	15.8426	5.81777	13.6626
100000	10000	66.9352	23.3454	55.9303

Table 2: Running time of original sequential code

Table 2 and 3 are the records for the original sequential code before and after we achieve the spatial locality of cache for memory inefficient part (so only the results under Memory inefficient column change).

Capacity	Items	Memory inefficient (value & solution)	Memory efficient (value only)	Memory efficient (value & solution)
1000	1000	0.0212187	0.0168568	0.384746
5000	1000	0.133945	0.109112	0.244531
5000	5000	0.668835	0.540613	1.20950
10000	1000	0.273084	0.226099	0.516556
10000	10000	2.73563	2.22395	5.07604
20000	2000	1.12666	0.921218	2.13556
50000	5000	7.13521	5.81777	13.6626
100000	10000	29.0102	23.3454	55.9303

Table 3: Running time with cache locality

From the results, we can see that for small capacity and number of items, the improvement in time is about 20%; when the capacity and number become large, the improvement in time is increased above 50% which is decent.

Problem 3. Use **OpenMP** to parallelize each of the 3 algorithms in a new file called knapsack_parallel.cpp

Capacity	Items	Memory inefficient (value & solution)	Memory efficient (value only)	Memory efficient (value & solution)
1000	1000	0.030061	0.0168941	0.131404
5000	1000	0.0436337	0.02075	0.13755
5000	5000	0.181331	0.0728484	0.788706
10000	1000	0.0679167	0.0229069	0.151138
10000	10000	0.50116	0.20791	1.88364
20000	2000	0.18389	0.0587676	0.324667
50000	5000	1.0174	0.307317	1.32777
100000	10000	8.83435	1.10914	4.01752

Table 4: Parallelized code with spatial locality on 32 cores

- **Memory inefficient (value & solution)** This algorithm involves 2 methods: *build_table*() and *backtrack*().

  The *build_table*() method contains 2 *for* loops. The first *for* loop initializes first column, and we add the OpenMP parallel for directive before it. The second one is a nested *for* loop. We interchange the outter and inner *for* loop and then add OpenMP directive exactly before the inner loop.

  We do not change the *backtrack*() method. Even though it contains a *for* loop, there exists write-after-read dependency. The for loop cannot be parallelized.

- **Memory efficient (value only)** This algorithm is related with only 1 method: *compute_table*().

  The structure of that method is quite similar with the *build_table*() method, and it includes a simple *for* loop and a nested *for* loop. Similarly, we wrap the single *for* loop with OpenMP directive, and we add pragma to the inner *for* loop of the nested loop statements.

- **Memory efficient (value & solution)** This algorithm contains *backtrack_implicit*() and *convert_path_to_used*() methods.

The *backtrack_implicit*() is recursive and therefore we do not parallelize it directly. However, it depends on a subroutine called *findk*() method. We parallelize the first simple *for* loop and the inner *for* loop of the second nested *for* loop statements.

The *for* loop in *convert_path_to_used*() method needs to modify a variable, and we need to put the whole block of the *for* loop in a critical region if we want to parallelize the *for* loop. It is meaningless to create the paradox.

We run the *knapsack_parallel* on a node with 32 cores, and the running times for different configuration are recorded in Table 4. With comparing with the data in Table 3, the speedup seems trivial when the size of the problem is relative small, e.g. $capacity = 1000, items = 1000$. The running time for the first two algorithm is almost the same. The third algorithm is as three times fast as the original version. When we test these three algorithm with parameter $capacity = 10000, items = 10000$, the first algorithm is as 5 times fast as before, the second algorithm is as 11 times fast as before, and the third algorithm is as 3 times fast as the initial version. Figure 3 illustrates the trend of execution time on different configurations.

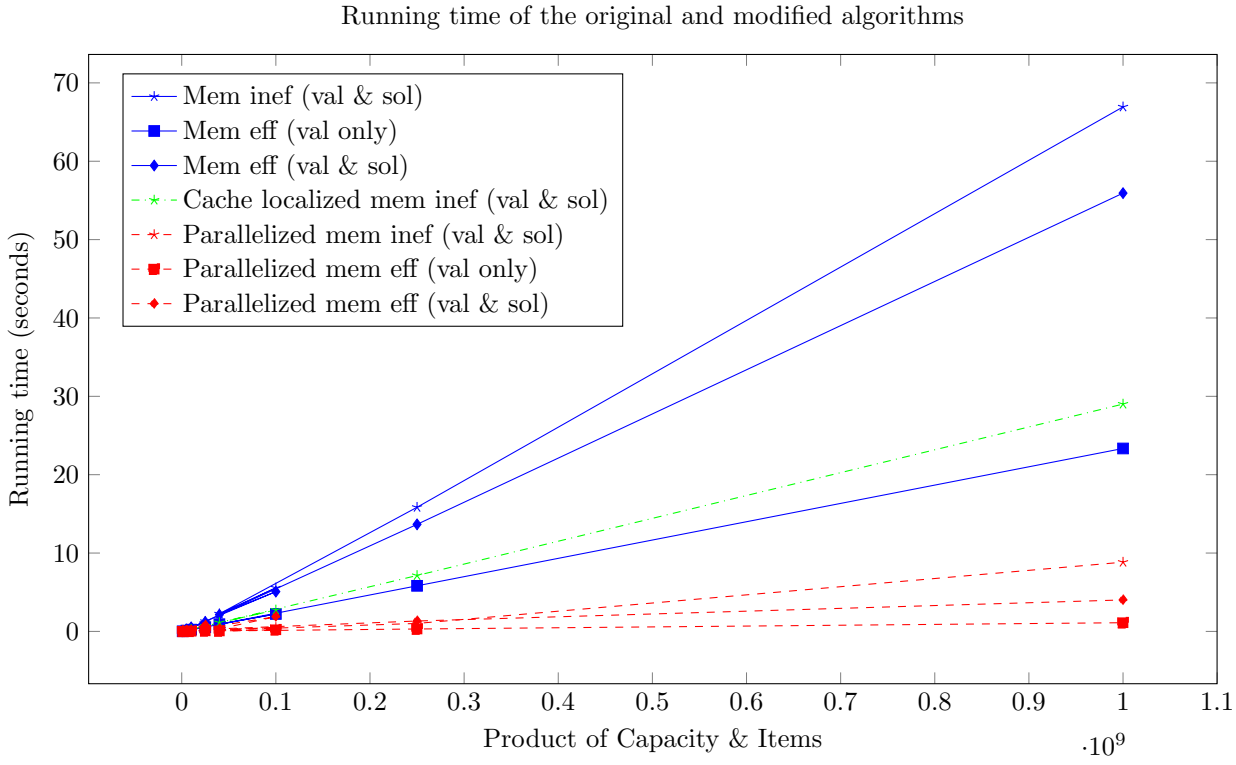

Figure 3: Execution time of the sequential and parallel algorithms, as well as the one with cache locality. The x axis represents the product of capacity value and item numbers. The y axis denote the running time in seconds. Blue solid lines represents sequential algorithms, and the red dashed lines indicate parallelized algorithms. The green dash dot line stands for running time of the memory inefficient algorithm with cache locality.