Dynamic Programming

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

- The running time of your solution is important!
- If you don't think about the time complexity of your algorithm before coding it up, sooner or later you'll end up wasting a lot of time on something something that's too slow.
  - This is especially tragic in exam environments.
- For simple code, analysing complexity can be as simple as multiplying together the bounds of nested `for` loops.
- For recursive solutions, a rough bound is $O(\text{time spent in recursive function} \times \text{number of recursion branches}^{\text{recursion depth}})$
- For DP, it usually comes down to carefully determining the state space and cost of recursion.

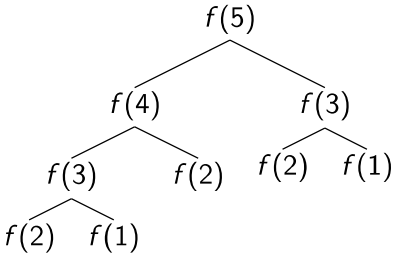
- On most online judges (this applies to the problem sets), a rough guideline is 50 million operations per second.
  - Constant factors occasionally matter, e.g. if you have no recursion, or only tail-recursion, you might manage more operations than this.
  - If you do float arithmetic, everything will be slow
- This means that for $n \leq 1,000,000$, an $O(n \log n)$ algorithm will probably run in time, but an $O(n^2)$ algorithm will definitely time out.
- Best way to get a gauge of an online judge's speed is to submit a simple for loop and compare the number of iterations it can do in 1 second to your local environment.

- We like greedy algorithms because they cut down the state space
- If a locally suboptimal choice can never contribute to the globally optimal solution, we don't have to expand nearly as many states
- But what if this doesn't work? Hill climbing etc
- We would like some way to explore many options at each stage, but efficiently - avoid repeating work

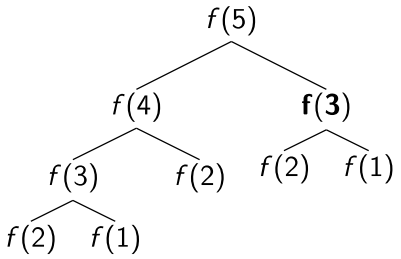
DYNAMIC PROGRAMMING

- Wikipedia: "a method for solving complex problems by breaking them down into simpler subproblems"
- If we can then keep recursively breaking down those simpler subproblems into even simpler problems until we reach a subproblem which is trivial to solve, we are done.
- This sounds a lot like Divide & Conquer...
- The key aspect of Dynamic Programming is **subproblem reuse**: If we have a divide & conquer algorithm that regularly reuses the same subproblem when breaking apart different larger problems, it'd be an obvious improvement to save the answer to that subproblem instead of recalculating it.
- In a way, dynamic programming is *smart recursion*

- **Problem statement** Compute the nth Fibonacci number ($1 \leq n \leq 1,000,000$)
- **Naïve solution** Recall that $f(1) = f(2) = 1$, and $f(n) = f(n-1) + f(n-2)$. Write a recursive function and evaluate.
- **Time Complexity** We recurse twice from each call to f, and the recursion depth is up to n. This gives a complexity of $O(2^n)$.
- Example call tree for $f(5)$:

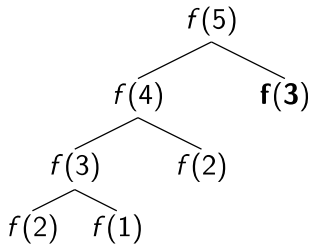


- Let's take a closer look at the call tree for $f(5)$:



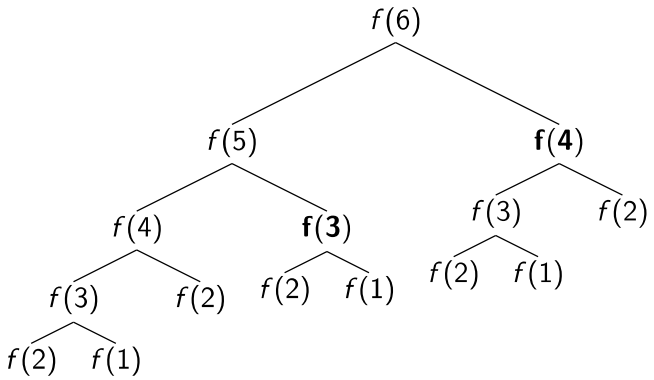
- What is $\mathbf{f(3)}$? *A problem we've seen before*.

- If we don't duplicate work:

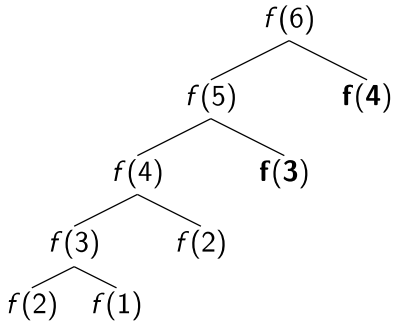


- The call tree gets a bit smaller.

- This'll make a much bigger difference on a bigger case:



- This makes a much bigger difference on a bigger case:



- In fact, we reduce the number of calls from $O(2^n)$ to $O(n)$.

- In general, a dynamic programming (DP) algorithm comes in three parts:
  - An exact definition of the subproblems. It is convenient to define these subproblems as entities in a *state space* and refer to individual subproblems as **states**.
    - In our example, each $f(i)$ is a state, and the state space includes all these states for i from 1 to n.
  - A **recurrence relation**, which facilitates the breaking down of subproblems. These define the *transitions* between the states.
    - In our example, the recurrence relation is $f(n) = f(n-1) + f(n-2)$.
  - **Base cases**, which are the trivial subproblems.
    - In our example, the base cases are $f(1) = 1$ and $f(2) = 1$.

- **Problem statement** Given an integer n ($0 \leq n \leq 1,000,000$), in how many ways can n be written as a sum of the integers 1, 3 and 4?
- **Example** If $n = 5$, there are 6 different ways:

$$\begin{aligned} 5 &= 1+1+1+1+1 \\ &= 1+1+3 \\ &= 1+3+1 \\ &= 3+1+1 \\ &= 1+4 \\ &= 4+1. \end{aligned}$$

- **Subproblems** Let $f(n)$ be the number of ways in which n can be represented using the numbers 1, 3 and 4. Each state is represented by a single integer, n.
- **Recurrence** For $n \geq 4$, if we already know the answers for $f(n-1)$, $f(n-3)$ and $f(n-4)$, then the answer for $f(n)$ is given by

$$f(n) = f(n-1) + f(n-3) + f(n-4)$$

- **Base cases** By inspection, we can see that $f(0) = 1$, $f(1) = 1$, $f(2) = 1$ and $f(3) = 2$.

- **Complexity** Since we have $O(n)$ values of f to calculate, each taking $O(1)$ time to calculate, assuming that the subproblems it depends on have already been calculated, the algorithm has overall time complexity $O(n)$.
- **Implementation**

```
f[0] = 1, f[1] = 1, f[2] = 1, f[3] = 2;
for (int i = 4; i <= n; i++)
  f[i] = f[i-1] + f[i-3] + f[i-4];
```

- A neat trick allows us to optimise this implementation to use $O(1)$ memory without changing the time complexity.

```
f[0] = 1, f[1] = 1, f[2] = 1, f[3] = 2;
for (int i = 4; i <= n; i++)
  f[i%4] = f[(i-1)%4] + f[(i-3)%4] + f[i%4];
```

- There are two main ways of implementing (and thinking about) DP solutions.
- These are most commonly referred to as **top-down** *(memoised)* and **bottom-up**

- Top-down dynamic programming takes the mathematical recurrence, and translates it directly into code.
- Answers to subproblems are cached to avoid solving them more than once. Caching function return values is widely known as *memoisation*.
- Top-down implementations are usually the easiest, because this is how most people think about DP solutions.

```
int f(int n) {
  // base cases
  if (n == 1 || n == 2) return 1;
  // return the answer from the cache if we already have one
  if (cache[n]) return cache[n];
  // calculate the answer and store it in the cache
  return cache[n] = f(n-1) + f(n-2);
}
```

- **Warning:** if 0 is a valid answer to a subproblem, initialise your cache array to something that isn't a valid answer.

- Bottom-up dynamic programming starts at the base cases and builds up all the answers one-by-one.

```
f[1] = 1, f[2] = 1;
for (int i = 3; i <= n; i++) f[i] = f[i-1] + f[i-2];
```

- **Warning:** when answering a subproblem, we must make sure that all subproblems it will look at are already answered.
  - In this example, the order in which states depend on each other is straightforward; this is not always the case.
  - In general, the dependency between DP states forms a **directed acyclic graph** (DAG).
  - If the state dependency graph has a cycle, it's not a valid DP
- Some algorithms are easier to think about this way. For example, the Floyd-Warshall algorithm is a DP, most easily implemented bottom-up.

- Top-down generally admits a more direct implementation after finding a recurrence.
  - It is more convenient on recursive structures like trees.
  - It only ever touches states that are necessary to compute, which can make it significantly faster for some problems.
- Often, there are characteristics of the state space that allow for space optimisations only possible going bottom-up.
- More generally, in bottom-up you tend to have more control which is useful for more advanced techniques. Especially if you want to speed up recurrence with a data structure.
- **Summary:** If you have a choice, pick your preference. However, for trees, we'll generally only do top down. And for many more advanced techniques, we'll probably only do bottom up.

- Cool. Now we know what DP is (hopefully).
- The above helps you recognize and (hopefully) code a DP someone tells you.
- But, personally, I don't actually find any of this useful for finding the right states or the right recurrence.
- I'll talk a bit about a useful strategy for me and do some examples to demonstrate.
- But really, the only method I know that for sure works is practice.

- To get a gauge of whether a problem is DP and what the DP might look like, I prefer to start with the recurrence rather than the state.
- I see DP as a way to compute the answer by taking 1 step at a time.
- Your state should contain the minimum amount of information needed so that we can figure out what steps are valid.
- I also think there's a lot of trial and error, which I haven't done a good job of emphasizing so far.

1. Choose some order to do the problem in. So essentially, how are you going to build up your solution?
   - Often implicit in the state but I find it useful to split off since it strongly suggests what your state might be.
2. Pick a tentative state. At minimum it should contain the parameters necessary to determine the end result.
   - E.g: If you need to output the best answer assuming you take X items, then number of items should probably be in your state.
3. Determine if you stored enough in your state to know what moves you can make. Try to get a recurrence for your state.
   - If your recurrence seems to need more than you are storing then try adding this data to your state and repeating.
4. Repeat this until it stabilizes or you realise you should try something else.

- Now, you should have at least some DP.
- Be happy about this, you've probably just changed something from exponential to polynomial.
- Is it good enough though? **Calculate** your complexity.
- What if it isn't? A few directions to go from here. I'll just list 2 common questions to ask yourself.

1. Look at your state. Is it bad? If so, can you fix this?
   - Is everything in our state necessary? Can we determine the valid moves from a subset of the state? More difficult, can we move anything around to improve the state?
   - Maybe our order to start with was incorrect.
2. Look at your recurrence. Is there some nice structure to it? If so, it is likely a suitable data structure will speed it up.
   - In particular, any recurrences that are ranges should make you think of *range* tree

- All the DP problems we've seen so far have a simple, one-dimensional state.
- However, it is easy to extend DP to states of higher dimensions.
- The hardest part of finding a DP solution is usually identifying a state that makes sense for the problem, and more dimensions just add more possibilities.

- **Problem Statement** To further your academic career, you have decided to steal some textbooks from the library. Unfortunately the bag you have brought is far too small, and won't fit all of the books.
  There are N books, the ith has a given size s_i and a value v_i (representing how valuable it is to you). Your bag has a given maximum capacity S: the sizes of all the books you take with you must total less or equal to this.
  Security is coming, and you want to maximise the total value of the books you're taking. What is the maximum value you can fit in your bag?
- **Constraints** $1 \leq N \leq 1,000$, $1 \leq S \leq 1,000$.
  $1 \leq s_i, v_i \leq 1,000,000$. All numbers are integers.

- We can't try every possible selection of books, because that would be $O(2^N)$ possibilities.
- We can start optimising by first observing that the order we put the books in the bag doesn't matter.
- In order to place a book in our bag, what information do we need to know?
  1. The amount of space remaining in our bag
  2. A guarantee that we haven't already put this book in our bag
- If we order the books by their given numbers, we have an ordering for free: if we are up to book i, then we've already considered books 1 through $i-1$, and not books i through N.

- This suggests the state:

$$(i, r)$$

  where i is the book we are currently considering such that we have already considered all the books before i, and r is the amount of space remaining in the bag.
- We ask the question $f(i, r)$: how much value can I fit into r units of space, using only books i through N?
- Then $f(1, S)$ will give the answer to the problem.
- Can we find a recurrence that answers this question in terms of smaller ones?

- $f(i, r)$: how much value can I fit into r units of space, using only books i through N?
- If we consider only book i, we have two choices:
  - If we put book i in our bag, we will lose s_i space and gain v_i value. Then the best value we could get would be $f(i+1, r-s_i) + v_i$.
  - If we *don't* put book i in our bag, we will not lose any space or gain any value. Then the best value we could get would be $f(i+1, r)$. We increment i because we have already decided not to use book i, changing our mind later won't help.
- Thus we obtain the recurrence:

$$f(i, r) = \max(f(i+1, r-s_i) + v_i, f(i+1, r))$$

- What if $r - s_i < 0$? To simplify the recurrence, we can simply include in our base cases:
  $f(i, r) = -\infty$ for all $r < 0$, for all i.
  Then no solution that tries to use such an answer will ever be the best one.
- What about base cases that can actually result in successful answers?
- $f(i, 0) = 0$ for all i
- Also, $f(N + 1, r) = 0$ for all $r \geq 0$ (we've run out of books to look at).

- **Complexity** Our state includes two parameters, one with N possibilities and the other with S possibilities, so there are a total of NS states.
  Each state checks a constant number (at most 2) other states to obtain an answer, so each state takes $O(1)$ time to calculate.
  Thus, the total time complexity of this algorithm is $O(NS)$.

- **Top-Down Implementation**

```
// 2D cache, should be initialised to -1 because 0 is a valid answer
int cache[N+1][S+1];

int f(int i, int r) {
  // base cases
  if (r < 0) return -2e9;
  if (i > n || r == 0) return 0;
  // check cache
  if (cache[i][r] != -1) return cache[i][r];
  // calculate answer
  return cache[i][r] = max(f(i + 1, r - s[i]) + v[i], f(i + 1, r));
}
```

- This implementation reduces the need to bounds-check for the large number of base cases.

- To find a bottom-up implementation, we need to be very careful about the order of the loops.
- Note that each state depends on i which are *greater*, and r which are *less or equal*.
- Also, we need to bounds-check carefully now, to make sure we don't read outside our array.
- **Bottom-Up Implementation**

```
int dp[N+2][S+1];

for (int i = N; i >= 1; --i) {
  // everything from larger i will be available here
  for (int r = 0; r <= S; ++r) {
    // we have declared the array larger, so if i == N, dp[i+1][...]
         will be zero.
    int m = dp[i+1][r];
    // bounds check so we don't go to a negative array index
    if (r - s[i] >= 0) m = max(m, dp[i+1][r-s[i]] + v[i]);
    dp[i][r] = m;
  }
}
```

- Sometimes your state needs to be much bigger than the DPs you are used to.
- A common trick is to make your state a set.
- So your state space is $2^n \cdot (\text{extra metadata})$.
- Seems bad but still a lot better than $n!$ which is usually the alternative.
- Especially useful with NP-hard problems involving finding a permutation. TSP (Travelling Sales Person) is the most well-known such example.
- Practically, often these can be detected by having small bounds, like $N \leq 20$.

- To represent our state, we don't store an actual set. This is unwieldly and probably slow.
- Instead we use a *bitset*.
- A bitset is an integer which represents a set. The ith least significant bit is 1 if the ith element is in the set, and 0 otherwise. For example the bitset 01101101 represents the set $\{0, 2, 3, 5, 6\}$.
- In this way, we can use an integer to index any subset of a set, amongst other things.
- This is much faster, especially if you use built-in bit operations to manipulate the set.

- Useful operations for manipulating bitsets:
  - Singleton set: `1<<i`
  - Set complement: `~x`
  - Set intersection: `x & y`
  - Set union: `x | y`
  - Symmetric difference: `x ^ y`
  - Membership test: `x & (1<<i)`
  - Size of set (with GCC): `__builtin_popcount(x)`
  - Least significant bit (or an arbitrary bit): `x & (-x)`
  - Iterate over all sets and subsets:

```
// for all sets
for (int set = 0; set < (1<<n); set++) {
  // for all subsets of that set
  for (int subset = set; subset; subset = (subset-1) & set) {
    // do something with the subset
  }
}
```

- **Problem statement** You want to tile your roof with n tiles in a straight line, each of which is either black or white. Due to regulations, for every m consecutive tiles on your roof, at least k of them must be black. Given n, m and k ($1 \leq n \leq 60$, $1 \leq k \leq m \leq 15$, $m \leq n$), how many valid tilings are there?
- **Example** If $n = 2$, $m = 2$ and $k = 1$, there are 3 different tilings: BB, BW, or WB.

- A counting problem, a bit different from what we might be used to. Our DP will associate with a state how many configurations correspond to that state. The rest (for now) will mostly be the same.
- Let's start with an obvious state. How about the number of valid tilings with n tiles. Then hopefully we can step to $n+1$ by laying one more tile.
- Then our step is to lay either a black tile or white tile.
- But uhhh... how do we know if we can lay a white tile?
- We need to know if there are only $k-1$ black tiles in the last $m-1$ tiles. If so, the next tile must be black.
- Whoops, we should add that to our state.

- Okay, our state is n, the number of tiles laid and b, the number of black tiles that are in the last $m-1$ tiles.
- Now we know whether the $n+1$-th tile must be black.
- So we compute

```
dp[n][b] = #tilings of n tiles
where b of the last m-1 tiles are black
```

- Then:

$$dp[n+1] = dp[n][k-1] + 2\sum_{j=k}^{m-1} dp[n][j]$$

- Oh oops, we should compute `dp[n][b]` not `dp[n]`.
- Well...ah crap.

- This is impossible.
- And it makes sense, we only store how many tiles in the last $m-1$ are black.
- But say we add a black tile to the right. Then how do we know if `(num black tiles)` increases or stays the same?
- You need to know what the $(m-1)$th tile was.
- And because of this, in one step's time you need to know what the $(m-2)$th tile was. And so on. So we really need to know what all these tiles are.
- So we should amend our state to be which of the last $m-1$ tiles are black (actually last m tiles for simplicity).

- Let $f(i, S)$ be the number of ways to have tiled the first i tiles, such that out of the last m tiles, the ones that are black are exactly the ones in the set S.
- For the recurrence, we can either set the new tile to be black or white. Reflecting this in our state is just applying the right bit operations from earlier.
- **Recurrence**

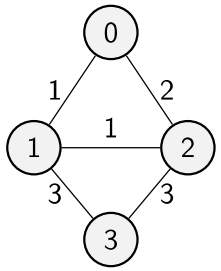
  $$f(i, S) = f(i-1, S >> 1) + f(i-1, (S >> 1) | (1 << (m-1))),$$

  where $|S| \geq k$, or 0 otherwise; in the first term, tile $(i - m)$ is white, and in the second it is black.
- **Base case** We have $f(m, S) = 1$ iff $|S| \geq k$. We don't need to consider $f(i, S)$ for any $i < m$.

- **Complexity** There are $O(n2^m)$ total states to calculate, and each state takes $O(1)$ to compute, so this algorithm runs in $O(n2^m)$ time. We also exploit the fact that the answer for $f(n, S)$ only ever relies on the answers for $f(n-1, T)$, allowing us to use only $O(2^m)$ memory.
- **Implementation**

```
// base case
for (int set = 0; set < (1<<m); set++) {
  dp[m%2][set] = (bitcount(set) >= k);
}

for (int i = m+1; i <= n; i++) {
  memset(dp[i%2], 0, sizeof(dp[i%2]));
  for (int set = 0; set < (1<<m); set++)
    if (bitcount(set) >= k)
      dp[i%2][set] = dp[(i+1)%2][set>>1] + dp[(i+1)%2][(set>>1)|(1<<(m
          -1))];
}

// answer is sum over all sets of dp[n%2][set]
```

- **Problem Statement:** Given a weighted, bidirectional graph with N nodes, find the shortest path starting from node 0 that visits every node exactly once.



- **Sample Input:**
- **Sample Output:** 5, path is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

- Let us try the problem in a natural order, in increasing number of nodes on a path.
- So our state is `(#nodes in path)` and we store the shortest path with that many nodes.
- Then our recursion is...whoops, we need to know where we are as well.
- So let us amend the state, our state is `(#nodes in path, last node in path)`.
- Then our recursion is to try every next node.
- But how do we know if we have visited a node twice?
- We have to keep this in our state...

- So our state is `(S, e)`. We store the shortest path that uses the nodes in S exactly once and ends at e. We will denote this $f(S, e)$.
- Now we know what next moves we can make. So we have a hope of forming a recurrence.
- Our recurrence is try all cities we could have came from.

$$f(S, e) = \min_{p \in S \setminus \{e\}} f(S \setminus \{e\}, p) + d[p][e]$$

- Alternatively, I find it more natural to push this forward and think of the next cities we can go to.
- Exercise: Translate the above into bit operations.

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 20;
const int INF = 1e9;
int N, adj[MAXN][MAXN]; // assume this is given.
int dp[1<<MAXN][MAXN];

int ans = INF;
for (int state = 0; state < (1<<N); state++) {
    for (int city = 0; city < N; city++) dp[state][city] = INF;
}
dp[1][0] = 0;

for (int state = 1; state < (1<<N); state++) {
    for (int city = 0; city < N; city++) {
        int cdp = dp[state][city];
        if (state == (1<<N) - 1) ans = min(ans, cdp);
        for (int nxt = 0; nxt < N; nxt++) {
            // Already visited nxt.
            if (state & (1<<nxt)) continue;
            dp[state|(1<<nxt)][nxt] =
                min(dp[state|(1<<nxt)][nxt], cdp + adj[city][nxt]);
        }
    }
}
```

- Sometimes you have the right state space but the cost of recursion is too high.
- In such cases, take a look at the recursion and see if it has a nice structure.
- Often the recursion step can be sped up with the right choice of data structure.
- One common example is your recursion naturally involves a *range* query.

- You have already seen one example of this in week 2, LIS.
- The recurrence we end up with by going left to right is:

$$\begin{aligned} &bestEndingAt[i] \\ &= \max\left(bestWithEnd[1], bestWithEnd[a[i]]\right) + 1 \end{aligned}$$

- Note in LIS, it was not clear from the outset that a range tree would play a role. You had to write down the recurrence to see it.

- **Problem statement** You have a number ($1 \leq N \leq 100,000$) of fireworks available to you, each situated at some point on the horizon, where each can only be launched at some particular time. You can use any number of the fireworks, but you cannot change the order in which they are launched. Furthermore, for each firework you launch, the previous firework launched must be located in the *combo range* of that firework. Given the ordering that you must launch the fireworks in, their positions x_i ($1 \leq x_i \leq 500,000$) on the horizon, their scores s_i and their combo ranges (l_i, r_i), what is the maximum sum of scores you can obtain?

- For example, suppose we had three fireworks located at positions 1, 2 and 3 in that order, and that their combo ranges were (1,5), (3,5) and (0,1) respectively.
- We can launch any of them singly, because there would be no previous firework restricting the combo range.
- However, we could never launch the second firework with any of the others, because the only firework before it (the first one) lies outside its combo range, and it does not lie in the combo range of the only firework after it (the third one).

- Due to the problem, we have a natural order to start with.
- Then the obvious state to start with is just the last firework we've launched. For each firework, we store the highest obtainable score with a chain ending at that firework.
- If for each previous firework, we have the best chain of zero or more fireworks leading up to it, we just want to pick the best eligible previous firework.
- In this case, we just need to consider if the firework we're coming from is inside our combo range.

- **Top-Down Implementation**

```
int f(int i) {
  if (i == 0) return 0;
  if (cache[i]) return cache[i];
  int m = s[i];
  for (int j = 1; j < i; j++) {
    if (x[j] >= l[i] && x[j] <= r[i]) {
      m = max(m, f(j) + s[i]);
    }
  }
  return cache[i] = m;
}
```

- **Bottom-Up Implementation**

```
ll res = 0;

for (int i = 1; i <= n; i++) {
  dp[i] = s[i];
  for (int j = 1; j < i; j++) {
    // try the jth as the penultimate firework
    if (x[j] >= l[i] && x[j] <= r[i]) {
      dp[i] = max(dp[i], dp[j] + s[i]);
    }
  }
  // update answer
  res = max(res, dp[i]);
}
```

- We have n states to calculate, and each one takes $O(n)$ time.
- This is an $O(n^2)$ algorithm, and with N up to 100,000, this is not going to run in time.
- It seems unlikely that we can change the state space to be anything other than linear, but the recurrence looks simple enough.
- What is the actual recurrence?

- **Recurrence** Let $f(i)$ be the best score possible ending at the ith firework (in the given order). Then

$$f(i) = s_i + \max(f(j)),$$

  where the maximum is taken over $j < i$ where $l_i \leq x_j \leq r_i$, and is zero if no such j exists.
- This is a *range* constraint.
- So we should be able to use a range tree, and obtain a better solution!
- There are still n states, but now each one takes only $O(\log n)$ time to calculate, so we obtain a solution in $O(n \log n)$ time.

- **Bottom-Up Implementation**

```
int query(int a, int b); // max query range tree of size 500,000
int update(int a, int v); // update index a to value v

ll res = 0;
for (int i = 1; i <= n; i++) {
  // calculate best score ending in i-th firework using the range tree
  dp[i] = query(l[i], r[i] + 1) + s[i];
  // add i-th firework to RMQ
  update(x[i], dp[i]);
  // update final answer if necessary
  res = max(res, dp[i]);
}
```

- A top-down implementation is not as easy to come up with in this case. Why?

- **Problem Statement:** Consider the $N+1$ points on the real line $0, 1, \ldots N$. You are given M segments, each has a range $[s_i, e_i]$ and a cost c_i.
  Output the minimum cost necessary to obtain a subset of the segments which covers all $N+1$ points.
- **Input Format:** First line, N, M. $1 \leq N, M \leq 100,000$.
  The following M lines describe a segment as a triple (s_i, e_i, c_i).

- **Sample Input:**

  ```
  5 4
  0 5 10
  0 3 4
  2 5 4
  0 4 5
  ```

- **Sample Output:**

  ```
  8
  ```

- **Explanation:** Take the second and third segments.

- What order should we attempt the problem in?
- Let's try just processing the segments one by one.
- What is the state?
- To answer the question, we need to store what indices we have covered.
- Okay, that's a bit excessive.
- **Note:** There was nothing special about the order of the segments, we essentially processed the segments in an arbitrary order.
- It is generally better to at least have some meaningful order.

- Alternatively, we can focus on the array.
- Then a natural order is left to right.
- But what would our state be then? This is less obvious.
- A first guess would be i, where we are up to, and S which indices to the left of i are covered.
- But this just gets us back to where we started.
- The **key** here is to make the state just i. Then we denote $dp[i]$ to be the min cost of segments to cover exactly $[0, i]$.

- A very reasonable question is, is this enough to form a recursion?
- Let's just try!
- To go from i to $i+1$ what do we need to do?
- We need to cover index $i+1$. What does this mean for our set of segments?
- We need to pick a segment ending at $i+1$! (Note, we are defining $dp[i]$ to be the min cost to cover *exactly* $[0, i]$. So we ignore segments ending past $i+1$)
- So we now have our choices. How do we form the recurrence?

- Let's say we pick segment $[s, i+1]$ with cost c. What do the rest of our segments have to satisfy?
- **Answer:** They must cover a range $[0, e]$ where $e \geq s-1$.
- So assuming we pick this segment, we have

$$dp[i+1] = c + \min_{j \in [s-1, i]} dp[j]$$

- Complexity? $O(N \cdot M)$

- A bit too slow but at least not exponential.
- But we have a hope of speeding this up, the state space is good (optimal even). It is the recurrence blowing us out.
- Let's look at it again:

$$dp[i+1] = c + \min_{j \in [s-1, i]} dp[j]$$

- Looks structured...
- *Ranges!* omg...
- So we would hope we can support this with a range tree!

Example: Segment Cover


```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int MAXN = 100000;
const ll INF = (1ll << 61);
// (start, cost)
vector<pair<int, ll>> segments[MAXN+5];
int N, M; ll dp[MAXN+5];
// (Point Update, Range Min) range tree
void update(int p, ll v);
ll query(int s, int e); // [s, e)

int main() {
    cin >> N >> M;
    for (int i = 0; i < M; i++) {
        int s, e, c;
        cin >> s >> e >> c;
        segments[e].emplace_back(s, c);
    }
    for (int i = 0; i <= N; i++) {
        dp[i] = INF;
        for (auto seg : segments[i]) {
            ll prevcost = seg.first == 0 ? 0 : query(seg.first-1, i);
            dp[i] = min(dp[i], prevcost + seg.second);
        }
        update(i, dp[i]);
    }
    cout << dp[N] << '\n';
}
```

- Complexity? $O(N + M \log N)$.
- There are many problems along these lines, of doing DP on a line with choices given by intervals. It shouldn't be a surprise many involve range trees.
- A key was to be clear what the state represented. *Exercise:* An alternative is to make $dp[i]$ the min cost for covering **at least** $[0, i]$. Work out the details for this approach.
- *Exercise:* What changes if we require the intervals to cover the entire interval $[0, N]$ not just the integer parts?

- **Problem Statement:** You have a $H \times W$ block of marble, divided into 1×1 cells. Each cell has a value. You want to pick a subset of cells to make a building. The restrictions are:
  - Each cell of the building must lie on either the ground or another cell of the building.
  - The cells chosen for the building for each of the H rows must be contiguous.

  What is the maximum possible sum of values of a valid building?
- **Input:** First line 2 integers, H, W. $1 \leq H, W, \leq 1000$. Next H lines each have W integers, the value of the cells. These values can be negative.
- **Source:** Australian Informatics Invitational Olympiad 2014.

- **Sample Input:**

```
3 4
-9 1 -9 1
1 1 -9 1
1 1 1 1
```

- **Sample Output:**

```
7
```

- **Explanation:** Pick all 4 cells in the bottom row, the left 2 in the second row and just the 2nd in the third row.

- This is less obviously DP. Let's just pick something to look at and start from there.
- One natural starting point is to consider the problem cell by cell.
- What order? Let's say bottom to top, and left to right within each row.
- So our state is cell we are considering.
- And our choice needs to be whether to put the next cell into the building.
- But how do we know if we can include the next cell?

- Okay, our state is too small, what do we need to add?
  - What cells on the previous row are in our building.
  - Something about what cells in our current row are in our building. **Why?**
  - Actually we need to store exactly what cells in our current row are in our building.
- What is the size of our state space now?
- Depends a bit on your implementation. Maybe something like $O(WH \cdot W^2 \cdot W^2)$.
- This is a problem. We can optimize recurrence but if our state space is too large we are just dead in the water.

- Let's try a different state.
- We can note that each row is very structured, perhaps we can do a row at a time.
- Instead of building cell by cell, let's build row by row.
- So instead of having $O(1)$ moves, our moves are "pick a contiguous selection of cells on this row and put it in our building".
- What is our state? How do we know if a move is valid?

- We need to know if the segment we have chosen lies within the segment of the building on the previous row.
- So our state should be what row we are up to and what cells in the previous row are in the building.
- How big is our state?
- $O(H \cdot W^2)$.
- Still a bit too large but this is progress. It is possible to speed up the recurrence to $O(1)$ so we are nearly there.

- If you were solving the problem yourself, you would probably keep trying this direction for a while.
- But ultimately, it seems this is the limit of how small we can make the state space.
- Each of the parameters is necessary.
- We probably need to pick a different direction for our DP.
- Let's go back to the drawing board.
- **Draw things!**

- Let's try going left to right. So we are now making our building column by column.
- What are our moves? Same as before, let's try picking the cells for each column.
- What is our state? How do we know a move is valid? Boils down to, what are the constraints for a valid building in terms of its columns?
- **Key Observation:** It is ternary! The building increases height up to a point then decreases in height.

- **Key Observation:** It is ternary! The building increases height up to a point then decreases in height.
- Okay, let us start with the obvious state, which column we are up to.
- Is this enough to know which moves we can make?
- No. We don't even know if the building increased in height from the previous column.
- What do we need to store to know this?
- Okay, our state is now `(current column, height of previous column)`.

- **Key Observation:** It is ternary! The building increases height up to a point then decreases in height.
- State:
  `(current column, height of previous column)`.
- Is this enough?
- Note: It is always okay to decrease in height. But if our move increases the height, how do we know if it is a valid move?
- No. Not enough. We need to know if at least once in the past our building has decreased in height.
- New state:

  `(current column, height of previous column,`
  `has the height of the building ever decreased?)`

- **Key Observation:** It is ternary! The building increases height up to a point then decreases in height.
- State:

  `(current column, height of previous column, has the height of the building ever decreased?)`
- Is this enough?
- It is enough to tell us what moves are valid in the current column.
- What is the state space?
- $O(WH)$.
- So we should be optimistic. So we should try to define a recurrence.

- State:

  `(current column, height of previous column, has the height of the building ever decreased?)`
- Let $f(w, h, b)$ be the best building up to and including column w, with h cells picked in column w and $b = 1 \iff$ the height of the building has decreased at some stage.
- Recurrence? Try $b = 0$ and $b = 1$ separately.

- For $b = 0$. We can't decrease in height from our previous column and we can't ever have decreased in height in the past.

$$f(w+1, h, 0) = \max_{h' \leq h} f(w, h', 0) + \sum_{i=0}^{h} b[w+1][i]$$

  where we say $f(w, -1, 0) = 0$ for convenience.
- For $b = 1$. Then due to the ternary condition we can't increase from our previous column.

$$f(w+1, h, 1) = \max_{h' \geq h, b' \in \{0,1\}} f(w, h', b') + \sum_{i=0}^{h} b[w+1][i]$$

- Complexity?
- $O(WH)$ state space, $O(H)$ recurrence. Overall $O(WH^2)$.

- But state space is good, we can hope to speed up the recurrence.
- Let's take a look at it. Let's try $b = 0$.

$$f(w+1, h, 0) = \max_{h' \le h} f(w, h', 0) + \sum_{i=0}^{h} b[w+1][i]$$

- This looks relatively structured. **Why?**
- Everything is a range yay.
- The first part is a range max. We know how to do these.
- The second part is a range sum of input. We can precompute this. **How?**
- So we should be able to do $O(WH \log H)$.

- A bit overkill it turns out. Here's a useful trick.
- Let's look at it again:

$$f(w+1,h,0) = \max_{h' \le h} f(w,h',0) + \sum_{i=0}^{h} b[w+1][i]$$

- For any specific $f(w+1,h,0)$, it takes $O(H)$ to compute the first max.
- However, we need this for all $h \le H$. And we can do all of them at once in $O(H)$. **Why?**
- Cause your algo for finding $\max_{h' \le h} f(w,h',0)$ finds $\max_{h' \le j} f(w,j,0)$ for all $j \le h$ too.
- So for each column we can compute all these values in $O(H)$ at once.
- This reduces complexity to $O(WH)$.

- Recap. Our state: $f(w, h, b)$.
- Recurrences:

$$f(w+1, h, 0) = \max_{h' \le h} f(w, h', 0) + \sum_{i=0}^{h} b[w+1][i]$$

$$f(w+1, h, 1) = \max_{h' \ge h, b' \in \{0,1\}} f(w, h', b') + \sum_{i=0}^{h} b[w+1][i]$$

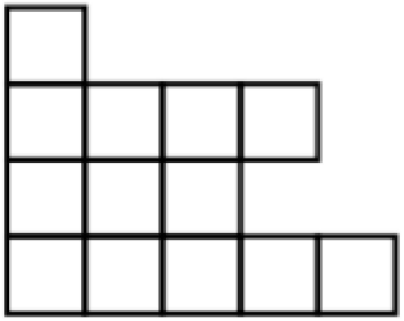
- We speed up the sums by precomputing a cumulative sum for each column. We speed up the maxes by doing them all at once for each column.
- $O(WH)$ states with recursion cost $O(H)$ per column, hence overall recursion cost $O(WH)$.

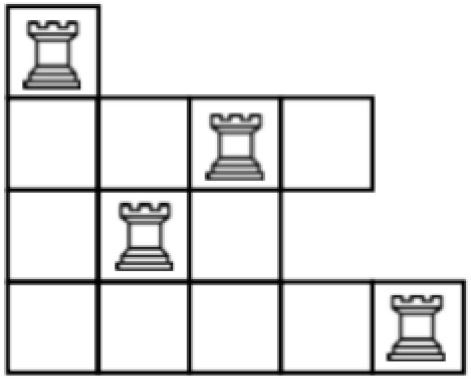
```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_DIM = 1000;
const long long INF = (1ll << 60);
// input and precomp.
int W, H;
long long b[MAX_DIM][MAX_DIM]; //(w,h)
long long columnsum[MAX_DIM][MAX_DIM];
long long dp[MAX_DIM][MAX_DIM][2]; // (w,h,b)

void precomp() {
    // TODO: read input here.
    for (int w = 0; w < W; w++) {
        for (int h = 0; h < H; h++) {
            dp[w][h][0] = dp[w][h][1] = -INF;
            columnsum[w][h] = b[w][h];
            if (h > 0) columnsum[w][h] += columnsum[w][h-1];
        }
    }
}
```

```
int main() {
    precomp();
    vector<long long> maxdown(H, 0), maxup(H, 0);
    long long ans = 0;
    for (int w = 0; w < W; w++) {
        for (int h = 0; h < H; h++) {
            dp[w][h][0] = maxup[h] + columnsum[w][h];
            dp[w][h][1] = maxdown[h] + columnsum[w][h];
            ans = max(ans, max(dp[w][h][0], dp[w][h][1]));
        }
        // Remember: we could have chosen no blocks as well.
        maxup[0] = max(0ll, dp[w][0][0]);
        for (int h = 1; h < H; h++) {
            maxup[h] = max(maxup[h-1], dp[w][h][0]);
        }
        maxdown[H-1] = max(dp[w][H-1][0], dp[w][H-1][1]);
        for (int h = H-2; h >= 0; h--) {
            maxdown[h] = max(maxdown[h+1], max(dp[w][h][0], dp[w][h][1]));
        }
    }
    printf("%lld\n", ans);
}
```

- Let's summarize.
- The problem seems difficult and convoluted. But I think all of it is very natural except for looking left to right and characterizing the condition for going left to right.
- Even for that part, it is always worth making examples and trying different orders.
- Our choice of state space was, mostly, dictated just by the order we picked.
- Our recurrences followed through by translating our requirement.
- Speeding up the recurrence is natural from looking at the formula.
- **Moral (hopefully):** None of this is magic. Almost all of it is fairly logical.

- **Problem statement** You have a set of N ($1 \leq N \leq 30$) rows from a chessboard of N rows and M columns, with some of the squares cut out from the right. How many ways are there to place K rooks on this chessboard without any rook threatening any other rook?



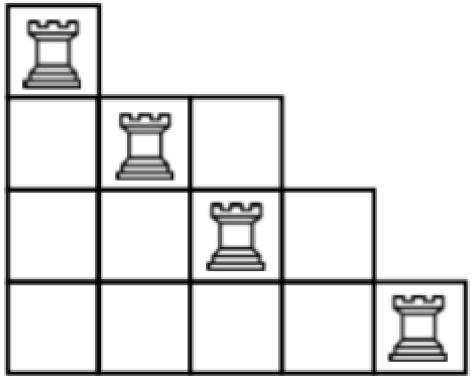


$N = 4, K = 4$
8 ways

- Let us pick an order.
- How about row by row.
- Then our state is which row we are up to, we will store the number of ways to put rooks up to this row.
- Our moves are try all places we can put a rook on the curent row.
- How do we know if a move is valid?

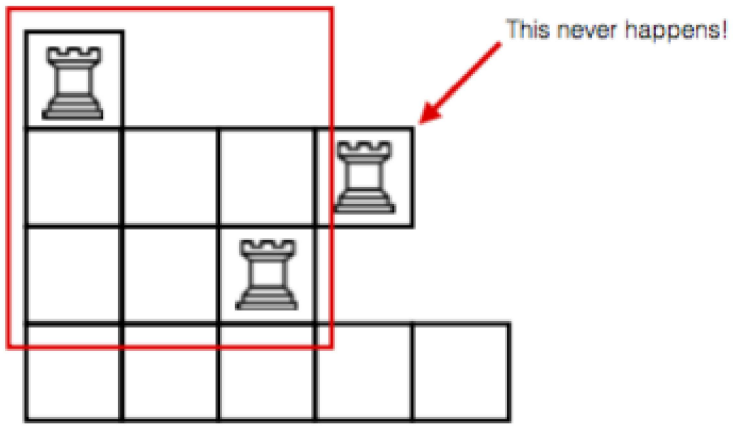
- In other words, how do we know which columns are free?
- Crap...okay we need to keep the set of free columns in our state.
- We can use a bit set like before.
- State space?
- About $O(2^M \cdot N)$, not good enough.

- But we are counting things. Can we instead just store the number of rooks we've placed. Then hopefully the number of ways we can place a rook on our new row is `(length of row) - (#rooks placed)`.
- Alas, no dice :(we don't know whether the rooks we've placed are in a column we can place to or not.
- Can we fix this?
- **Key Observation:** There was nothing special about the ordering of rows in this problem. Usually the order is suggested by the problem but in this case, there is no reason to have the rows in the order they are in.



$N = 4, K = 4$
8 ways

- Let's sort the sequence, so that the size of each row is non-decreasing
- Now, we know that if we place a rook on a row, we know that we can assume that every previously placed rook is in a cell that our row covers



- Now, we can say that any rooks already placed will be either to the left or directly above us
- We can then formulate a recurrence that only needs to know about the current row and the number of rooks
- If we're on a row i on length L, then for every configuration of the rows above with k rooks already placed, we can place a rook for this row in $(L - k)$ places

- **Subproblems** Let $f(i, j)$ be the number of ways to place j rooks on the first i rows, sorted by length.
- **Recurrence**

$$f(i, j) = f(i-1, j) + f(i-1, j-1) * (L_i - (j-1))$$

- **Base case** The number of ways to place 0 rooks on 0 rows is 1.

- **Implementation**

```
cache[0][0] = 1;
for (int i = 1; i <= n; i++) {
  for (int j = 0; j <= k; j++) {
    // can place no rooks in this row
    cache[i][j] = cache[i-1][j];
    // or place a rook in this row
    if (j > 0)
      cache[i][j] += cache[i-1][j-1] * (L[i-1] - (j-1))
  }
}
cout << cache[n][k] << endl;
```

- This time around, the magic was in reordering the rows.
- This **is** magic. But this general idea is very useful. If you aren't given an order, make an order. It can't be worse than having no order.
- You see a similar idea in 0-1 knapsack. To avoid storing a bitset of used items, we just pick any arbitrary order to process the items in.

- Another path to a solution is to do the problem in column order.
- But you have to do it from right to left.
- Then you get the same property as this solution.
- Note: column order isn't symmetrical as all rows are required to start from column 0.