Dynamic Programming

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Divide & Conquer

- Wikipedia: "an algorithm design paradigm based on multi-branched recursion. . . works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly."
- There are two main requirements for a divide & conquer algorithm:
  - The ability to split a problem up into several subproblems of the same form as the problem itself, or of a small number of different types. Crucially, given solutions to these subproblems, it must be possible to combine them to get the solution for the larger problem.
  - The guarantee that these problems will soon get small enough that they're trivial to solve (**base cases**).

Divide & Conquer: Notable Examples

- Top-down parsers (used in compilers for many programming languages, etc.)
  - Hard problem: parse an entire file
  - Easier problem (e.g.): parse an `if` statement
  - Of course, there's a large number of different kinds of things that are parsed, i.e. different types of subproblems, and the calling function chooses appropriately.
  - Most parsers complete in $O(\text{length of input})$ time.
- Merge sort
  - Hard problem: sort an array
  - Easier problem: sort a smaller array
  - Easiest problem: sort an array of one number
  - Combining two sorted arrays into a sorted array is pretty easy
  - 1 array of size n takes $O(n)$ time to merge, 2 arrays of size $\frac{n}{2}$ take $O(n)$ total time to merge, ..., n arrays of size 1 take $O(n)$ total time to merge, for $O(n \log n)$ total time.

Table of Contents

Reminder: Algorithmic Complexity

- 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 divide & conquer algorithms, this is a bit more complicated, as seen from the time complexity of merge sort. However, a lot of divide & conquer algorithms look very similar to merge sort, so you can infer the complexity from that.

Reminder: Algorithmic Complexity

- 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.
- You will be notified if a judge is significantly different to this guideline.
- The judge that was used for contest 1 will likely be used for the remaining contests and final exam. A rough guideline for it is 1.5 *billion* operations per second.
  - Problem C accepted an $O(n^2)$ solution for the subtask with bound $n \leq 50,000$.

Table of Contents

D Y N A M I C

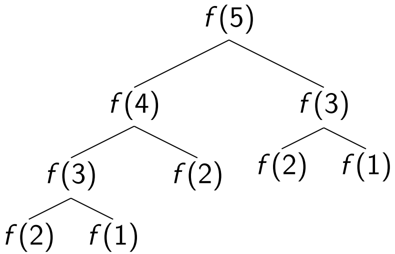
P R O G R A M M I N G

What is 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*

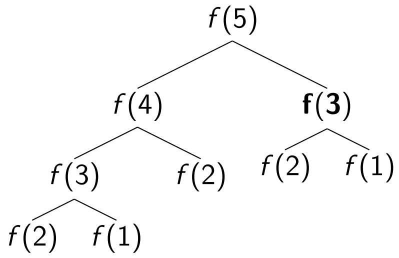
A problem we've seen before

- **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 $f(5)$:



A problem we've seen before

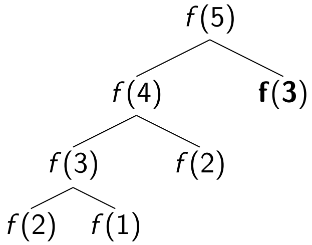
- Let's take a closer look at the call tree for $f(5)$:



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

A problem we've seen before

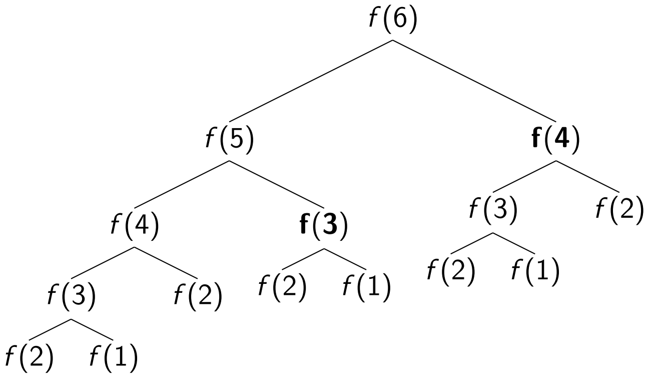
- If we don't duplicate work:



- The call tree gets a bit smaller.

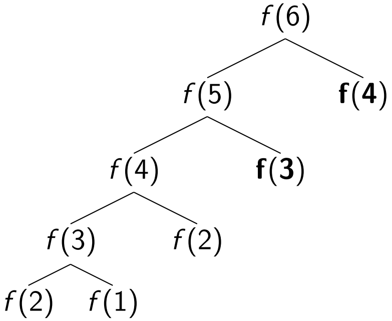
A problem we've seen before

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



A problem we've seen before

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



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

The essence of dynamic programming

- 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 $f(2) = 1$.

Table of Contents

Implementing & Understanding DP

- 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

- 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

- 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 or bottom-up?

- Top-down generally admits a more direct implementation after finding a recurrence.
    - It is also more convenient when doing DP over a structure like a tree.
    - It only ever touches states that are necessary to compute, which can make it significantly faster for some problems.
- Bottom-up can also be faster, since it doesn't have the recursive overhead inherent to the top-down approach.
- Often, there are characteristics of the state space that allow for space optimisations only possible going bottom-up. (Example in the next problem)
- Unless the problem only allows for one of these, then use the one that is most natural and is easiest to implement.

Table of Contents

Example problem: Sum Representation

- **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:

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

Example problem: Sum Representation

- **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 $f(3) = 2$.

Example problem: Sum Representation

- **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];
```

- **Exercise** optimise this implementation to use $O(1)$ memory without changing the time complexity.

Example problem: Fireworks

- **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?

Example problem: Fireworks

- 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).

Example problem: Fireworks

- Since we're restricted in the order that we can launch the fireworks, we have a natural definition of our subproblems based on the ordering.
- We ask, "What is the highest score I can obtain, ending at the *i*th firework that can be launched?"
- 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.

Example problem: Fireworks

- **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]);
}
```

Example problem: Fireworks

- 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?

Example problem: Fireworks

- **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 $l_i \leq x_j \leq r_i$.
- This is exactly the problem solved by our range tree data structure!
- So we can just directly plug in the data structure we already have, 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.

Example problem: Fireworks

- **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?

Table of Contents

Dynamic Programming in More Dimensions

- 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.

Example problem: 0-1 Knapsack

- **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.

Example problem: 0-1 Knapsack

- 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.

Example problem: 0-1 Knapsack

- 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?

Example problem: 0-1 Knapsack

- $f(i, r)$: 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:

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

Example problem: 0-1 Knapsack

- 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).

Example problem: 0-1 Knapsack

- **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 $O(NS)$.

Example problem: 0-1 Knapsack

- **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.

Example problem: 0-1 Knapsack

- 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]);
  }
}
```

Table of Contents

Example problem: Roof Tiling

- **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 ($1 \leq n \leq 60$, $1 \leq k \leq m \leq 15$), how many valid tilings are there?
- **Example** If $n = 2$, $m = 2$ and $k = 1$, there are 3 different tilings: BB, BW, or WB.

Example problem: Roof Tiling

- In a DP formulation of this problem, it seems natural that we should consider the position of the current tile in our subproblem. But this isn't enough.
- What else do we need? We need to be able to enforce the constraint that there must be at least k black tiles for every m consecutive tiles. We handle this in the simplest possible way, by including which of the last m tiles are black in our subproblem state.

Bitsets

- To represent which subset of the last m tiles are black, 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.

Bitsets

- We can manipulate bitsets using built-in bit manipulation operations:
  - Singleton set: `1<<i`
  - Set complement: `~x`
  - Set intersection: `x & y`
  - Set union: `x | y`
  - Symmetric difference: `x ^ y`
  - Membership test: `x & (1<<i)`
  - 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
  }
}
```

Example problem: Roof Tiling

- **Subproblems** 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.
- **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 $|S| \geq k$ and $f(i, S) = 0$ iff $i < m$ for any set S.

Example problem: Roof Tiling

- **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%2][set>>1] + dp[i%2][(set>>1)|(1<<(m-1))];
}

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