Dynamic Programming

Class 27

Divide and Conquer

- we have looked at divide and conquer
- mergesort and quicksort are classic examples
  1. partition a single large problem into smaller, non-overlapping problems that are easier to solve
  2. iterate over or recurse on the subproblems
  3. combine subproblem results into large problem result
- typically used for finding a solution, not for optimization

Binomial Coefficients

- consider the problem of finding the binomial coefficient $\binom{n}{k}$
- indexed by n and k

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- problem: computationally infeasible because factorials either overflow or require a (dog slow) large integer library
- one solution: use an alternate definition

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- this is a recurrence
- base cases $\binom{n}{0} = \binom{n}{n} = 1$

Algorithm

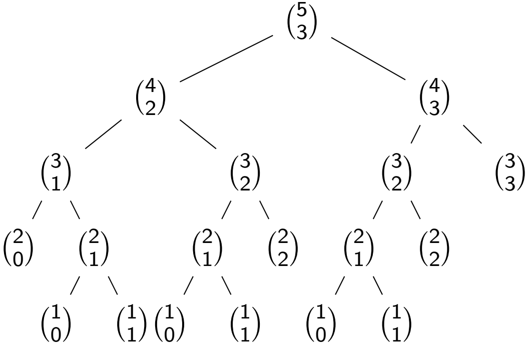
- implementing the recursive definition is straightforward
- it's simple, direct, works perfectly
- for small n, no problem
- performance with large n is unacceptable: $T(n, k) \in O(n!)$
- this kind of code gives recursion its bad name

```
uint64_t binomial(unsigned n, unsigned k)
{
  if (k == 0 || k == n)
  {
    return 1;
  }
  return binomial(n - 1, k - 1) + binomial(n - 1, k);
}
```

run code with $n = 25, 30, 35$

Recursion

- the problem with the recursive algorithm is not recursion per se
- the problem is that the recursive algorithm repeatedly computes the same results multiple times
- this problem looks like divide and conquer
- but it's not because the subproblems overlap



Massive Redundancy

- in the simple example of $\binom{5}{3}$ we have $6/19 = 32\%$ of recursive calls are repeats
- as n increases, the percentage of repeat calls approaches 100%

- we appear to be on the horns of a dilemma
  1. factorial-sized numbers requiring slow library calls
  2. spectacularly redundant recursion giving factorial-time performance

Dynamic Programming

- a solution is dynamic programming
- a misnomer: not dynamic, and has nothing to do with programming
- invented in the early 1950's by Richard Bellman
- linear programming was an optimization technique popular in the 1940s
- Bellman coined the phrase "dynamic programming" to sound cool and hide the fact that his new optimization technique used mathematics
- chairman of the congressional committee funding the work was a mathphobe

Dynamic Programming

- dp consists of two parts
    1. break a problem down into smaller subproblems (that may overlap) just like divide and conquer
    2. remember the solution to each subproblem, in case you encounter it again
- dp is used for
    1. finding a solution with overlapping subproblems
    2. finding an optimal solution when a brute force algorithm takes too long

Binomial Coefficient via Dynamic Programming

- the problem with recursively computing a binomial coefficient is repeatedly solving the same subproblems over and over
- the binomial function has two parameters, so we need to store a result for each n, k input
- each time we encounter n and k as parameters, check to see if we've already seen that pair of parameters
- if so, we already know the answer
- if not, we compute the answer and remember it

```
unit64_t binom(unsigned n, unsigned k, Matrix<unit64_t>& memo)
{
  if (memo.at(n, k) == 0)
  {
    if (k == 0 || n == k)
    {
      memo.at(n, k) = 1;
    }
    else
    {
      memo.at(n, k) =
        binom(n - 1, k - 1, memo) + binom(n - 1, k, memo);
    }
  }
  return memo.at(n, k);
}
```

DP Program Run

run program

- we overflow way before we exhaust computational limits
- storing the precomputed values in a table is called memoization

Dynamic Programming General Strategy

1. characterize the structure of a solution
2. give a recursive definition of the value of a solution
3. implement the definition of step 2 with memoization added, and compute the answer to your problem
4. construct the structure of the answer from the results of the computation in step 3

- sometimes step 4 is optional, depending on the original question

Amounts with Coins

- make an amount of money with an optimal number of coins (an optimization problem) using a greedy algorithm

Amounts with Coins

- make an amount of money with an optimal number of coins (an optimization problem) using a greedy algorithm

- a greedy algorithm does not always work
- try making 15¢ using denominations 1¢, 6¢, 10¢

Amounts with Coins

- make an amount of money with an optimal number of coins (an optimization problem) using a greedy algorithm

- a greedy algorithm does not always work
- try making 15¢ using denominations 1¢, 6¢, 10¢
- a brute force algorithm always works
- what is a brute force algorithm for making an amount of money with the optimal number of coins?

DP Amount with Coins

- a greedy algorithm doesn't always work
- brute force takes too long

- however, an amount with any set of denominations can be optimally found using dynamic programming

Step 1

- given an amount a
- a vector of m coin denominations `denom` where `denom.at(0)` is always 1¢ and `denom.at(m - 1)` is the largest coin denomination

denom	1¢	5¢	10¢	25¢

Step 1

- characterize the structure of an optimal solution

- $a_i, b_j, \ldots$ where $i, j, \ldots$ are in $0..m-1$ and $a_i + b_j + \ldots$ is minimal and $a_i * \text{denom.at(i)} + b_j * \text{denom.at(j)} + \ldots = a$
- the quantities involved are
  1. the amount of money to be made
  2. which coin denominations are used
  3. how many coins are used

Step 2

- give a recursive definition of the value of an optimal solution:
- consider the current amount and a current coin denomination
- we either do use a coin of the current denomination
- or we do not use a coin of this denomination
- let a be the amount of money
- let i be an index for the denom array
- let $\mathrm{opt}(i, a)$ be the function that gives us the optimum (minimum) number of coins — what does this function look like?

Step 2

- give a recursive definition of the value of an optimal solution:
- consider the current amount and a current coin denomination
- we either do use a coin of the current denomination
- or we do not use a coin of this denomination
- let a be the amount of money
- let i be an index for the denom array
- let $\text{opt}(i, a)$ be the function that gives us the optimum (minimum) number of coins — what does this function look like?

$$\text{opt}(i, a) = \begin{cases} \text{opt}(i, a - \text{denom.at(i)}) + 1 & \text{if we use this coin} \\ \text{opt}(i - 1, a) & \text{if we do not use this coin} \end{cases}$$

Step 2

- but how do we decide whether to use this coin denomination or not?

Step 2

- but how do we decide whether to use this coin denomination or not?
- we choose the one that gives us the minimum
- thus our function becomes

$$\text{opt}(i, a) = \min \begin{cases} \text{opt}(i, a - \text{denom.at(i)}) + 1 & \text{if we use this coin} \\ \text{opt}(i - 1, a) & \text{if we do not use this coin} \end{cases}$$

- what is the base case?

Step 2

- but how do we decide whether to use this coin denomination or not?
- we choose the one that gives us the minimum
- thus our function becomes

$$\mathrm{opt}(i, a) = \min \begin{cases} \mathrm{opt}(i, a - \mathrm{denom.at(i)}) + 1 & \text{if we use this coin} \\ \mathrm{opt}(i - 1, a) & \text{if we do not use this coin} \end{cases}$$

- what is the base case?
- zero amount requires no coins: $\mathrm{opt}(i, 0) = 0$

Step 3a

- implement the definition using recursion (no memo yet)

Step 3a

- implement the definition using recursion (no memo yet)

```
unsigned opt(size_t i, unsigned a)
{
  if (a == 0) // amount is 0
  {
    return 0;
  }
  return min(opt(i, a - denom.at(i)) + 1, opt(i - 1, a));
}
```

- unfortunately, we can't quite use this code, because `a - denom.at(i)` might underflow (if the current coin is bigger than a)
- and `i - 1` also might underflow (if i is already 0)

Step 3a

- implement the definition using recursion, no memo, and protect from underflow

```
unsigned opt(size_t i, unsigned a)
{
  if (a == 0) // amount is 0
  {
    return 0;
  }
  if (i == 0) // protect from underflowing i
  {
    return opt(i, a - 1) + 1;
  }
  if (a < denom.at(i)) // protect from underflowing a
  {
    return opt(i - 1, a);
  }
  return min(opt(i, a - denom.at(i)) + 1, opt(i - 1, a));
}
```

Step 3b

- memoize the code
- base cases can be:
    - built into the memo table initialization (for iterative implementation; see below)
    - built into the code (for recursive implementation)

Step 3b

```
size_t opt(size_t i, size_t a, Matrix<size_t>& memo,
           const vector<size_t>& denom)
{
  if (memo.at(i, a) == SIZE_MAX)
  {
    if (a == 0) // amount is 0
    {
      memo.at(i, a) = 0;
    }
    else if (i == 0) // only pennies; don't overflow i
    {
      memo.at(i, a) = opt(i, a - 1, memo, denom) + 1;
    }
    else if (a < denom.at(i)) // don't overflow a
    {
      memo.at(i, a) = opt(i - 1, a, memo, denom);
    }
    else
    {
      memo.at(i, a) = min(opt(i, a - denom.at(i), memo, denom) + 1,
                          opt(i - 1, a, memo, denom));
    }
  }
  return memo.at(i, a);
}
```

Step 3b

- what does the memo table look like?
- i (denomination index) is row, a (amount) is column
- 11¢ using 1¢, 6¢, 10¢, and 15¢ coin denominations

		a											
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	1	2	3	4	5	6	7	8	9	10	11
	1	0	1	2	3	4	5	1	2	3	4	5	6
i	2	0	1	2	3	4	5	1	2	3	4	1	2
	3	0	1	2	3	4	5	1	2	3	4	1	2

(this memo table was produced by an iterative version)

The Memo Table

- two things are crucial to understanding dynamic programming
  1. what one entry in the memo table represents: the optimal number of coins for this entry's amount, using any number of coins from pennies up to this entry's denomination inclusive
  2. where an entry in the memo table came from: look at the recurrence

Step 4

- now we know how many coins total are required to make the amount
- but we don't know which coins to use
- we could
  1. keep a list as we go
  2. backtrace through the memo table
- keeping a list is ok, but requires extra storage
- backtracing requires no extra storage
- backtrace starts at the spot representing the "final answer"
- stops when it hits a base case
- where this is depends on the algorithm
- in coin amounts, stops at the left column ($a = 0$)
- backtrace can be iterative or recursive
- uses memo table plus any helper info, e.g., denom

Step 4

- backtrace through the completed memo table
- meaning of left arrow and up arrow
- 11¢ using 1¢, 6¢, 10¢, and 15¢

		a											
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	1	2	3	4	5	6	7	8	9	10	11
	1		1				5						6
i	2		1										2
	3												2

- this is the recursive version of the table