CS 170 Lecture Notes

Mokhalad Aljuboori

Fall 2023 – Prof Nika and John

08/24/23

Lecture 1

Introduction/Arithmetic

1.1 Introduction

Whenever we have an algorithm, there are three questions we always ask about it:

1. is it correct?
2. How much time does it take, as a function of n?
3. And can we do better?

Whenever wew ask these questions, we can answer them in two different frameworks: **Detail-oriented**, which includes precise definitions, rigorous proofs, covers corner cases and is very detailed in general. THe other framework is **Bigger Picture**, which is useful for getting an intuitive sense of how the algorithm works. Try to answer these questions from both perspectives, flip flopping.

1.2 Asymptotics

1.2.1 Big $\mathcal{O}$ Notation

Big O notation provides a way to generalize and express the runtime of an algorithms without worrying about architecture-specific details since the actual runtime varies from computer to computer as each basic computer step depends crucially on the particular processor, caching strategy, and other complex details.

Definition 1.1: Big $\mathcal{O}$

let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows **no faster than** g") if for all n, there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$

The constant c allows us to disregard what happens for small values of n and focus on large dependence on n. For example:

Example 1.2

Take $f_1(n) = n^2$ and $f_2(n) = 2n + 20$. Which is a faster? This depends on n. For $n \leq 5$, n^2 is smaller (faster), otherwise $2n + 20$ is smaller (faster/superior).

This superiority is captured by the Big-O notation:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22 \qquad n \text{ is all positive integers}$$
$$2n + 20 \leq 22 \cdot n^2 \qquad \text{Definition of Big } \mathcal{O}$$

Therefore we say $f_2 = \mathcal{O}(f_1)$. If we try $\frac{f_1}{f_2}$, we will see that it will get arbitrarily large as n increases, and so no constant c will make the definition work.

Example 1.3: Egyptian Multiplication

1. Repeat: Halve 1st number (floor) and double the second, until we get a 0 in the first number.
2. Remove all the rows where the
3. ...

TODO: Prove this algorithm is correct and find it's Big O

1.2.2 Divide and Conquer for Multiplication

Break up the multiplication of two integers with n digits into multiplication of integers with $\frac{n}{2}$ and recurse.

1.2.3 Big Ω Notation

$$f = \Omega(g) \text{ means } g = \mathcal{O}(f)$$

1.2.4 Big Θ Notation

$$f = \Theta(g) \text{ means } f = \mathcal{O}(g) \text{ and } f = \Omega(g)$$

Questions

1. How does the constant in Big O notation allow us to disregard what happens for small values of n?

(a) Sometimes a function is smaller than another for small values of n but larger for larger values of n (see example 0.2). The constant allows us to increase that function by a factor so that it will be greater than the other function for **all** values of n.

2. Cover in depth what big omega and big theta is.

06/15/23

Chapter 2

Algorithms with numbers

Questions

1.

08/15/23

Chapter 3

Divide and Conquer Algorithms

The *divide-and-conquer* strategy solves a problem by:

- Breaking it into it *subproblems* that are themselves smaller instances of the same type of problem.
- Recursively solving these subproblems.
- Appropriately combining their answers.

Recurrence Relations

Divide and conquer algorithms often follow a generic pattern: they tackle a problem of size n by recursively solving, say a subproblems of size n/b and then combining these answers in $\mathcal{O}(n^d)$ time, for some $a, b, d > 0$. For example, Karatsuba multiplication algorithm had a = 3, b = 2, and d = 1.

09/05/23

Lecture 4

Fast Fourier Transforms

4.1 Polynomial Multiplication

We are given two polynomials of degree d as inputs:

$$A(x) = a_0 + a_1 x^2 \cdots + a_d x^d$$

$$B(x) = b_0 + b_1 x^2 \cdots + b_d x^d$$

The goal is to find the coefficients of

$C(x) = (A \cdot B)(x)$

$C(x) = c_0 + c_1 x + \cdots + c_N x^N \quad N = 2d$, since the deg of the product of two deg d polynomials is a 2d deg poly.

Note $c_k = \sum_{i=0}^{k} a_i \cdot b_{k-i}$ $C(x)$ has $2d+1$ coefficients, and the kth coeff. is found in $\Theta(k)$ time. Therefore finding all $2d+1$ coeff. seems to require $\Theta(d^2)$ time. Can we do faster?

4.1.1 Algorithm 1: Karatsuba

We'll do the same trick as integer multiplication. Split the polynomials in half, multiply the 4 halves, and use a trick to decrease the multiplication from 4 to 3, giving us $\mathcal{O}(n^{\log_2 3})$ flops.

4.1.2 Polynomial Interpolation, FFT

To arrive at a fast algorithm for polynomial multiplication we take inspiration from an important property of polynomials.

Lemma 4.1: Alternate representation for polynomials

A degree d-polynomial is uniquely characterized by its values at any $d+1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ by either one of the following:

1. its coefficients $a_0, a_1, \ldots, a_d$
2. d + 1 values: $A(x_0), A(x_1), \ldots, A(x_d)$

Of the two representations, the second is the more attractive for polynomial multiplication. The product $C(x)$ has degree $2d$, and is completely determined by its values at any $2d+1$ points. It's value at and given point z is easy enough to figure out., just $A(z)$ times $B(z)$. Thus polynomial multiplication takes **linear time** in the ***value representation***. (assuming we compute the values of $A(z)$, $B(z)$ for free.).

The problem is that since we are given the input polynomials in the coefficient representation, we also expect the output to be specified by coefficients. So we'll need to first translate from coefficients to values, multiply the values, and translate back to coefficients (a process called *interpolation*).

How efficient is this? We'll the $d+1$ selection of evaulation points and multiplication is just linear time. But how about when we evaluate the polynomials at the $d+1$ points? Evaluating a polynomial of degree $d \leq n$ at a single point is $\mathcal{O}(n)$ steps. Therefore the baseline for n points is $\Theta(n^2)$.

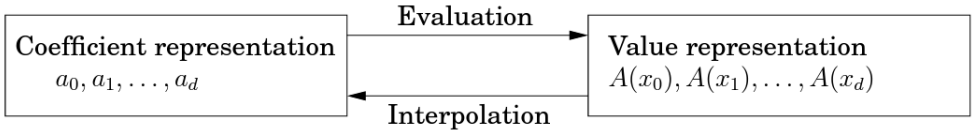


Figure 4.1: The resulting algorithm

Lemma 4.2: Runtime of Evaluating a Polynomial at a Single Point

Suppose we are given a polynomial of degree $n-1$: $p(x) = p_o + p_1 x + \cdots + p_{n-1}x^{n-1}$

$p(\alpha) = p_0$	0 mults
$+ p_1\alpha$	1 mults
$+ p_2\alpha \cdot \alpha$	2 mults
$+ p_3\alpha \cdot \alpha \cdot \alpha$	3 mults
$\ldots$	
$+ p_{n-1}\alpha \cdot \alpha \ldots \alpha$	n - 1 mults
$p(a)$	$\mathcal{O}(n^2)$mults

However notice that we are doing redundant work in each step. We already compute α^2 in the second step, so we don't need to compute it again in the next step.

With this, we can instead do each step in 1 multiplication by initializing an array where $A[i] = \alpha \cdot A[i-1]$. So evaluating a polynomial takes $\mathcal{O}(n)$

Fast Fourier Transform is able to do the evaluation step in just $\mathcal{O}(n \log n)$, for a particular clever choice of $x_0, \ldots, x_{n-1}$, using **complex numbers.**

4.1.3 Complex Numbers

Definition 4.3: complex number

a number that's in the form:

$$a + b \cdot i$$

Where a is called the real part, and b is called the imaginary part and $i = \sqrt{-1}$

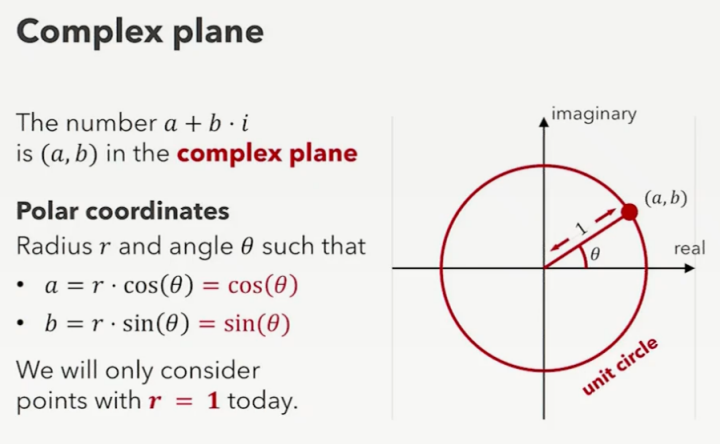


Figure 4.2: Complex Plane

Roots of Unity

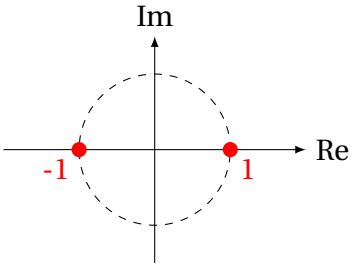
- **Unity** is just a fancy word for the number 1
- n^{th} roots of unity = {solutions to $x^n = 1$}
- it's a point x where you raise it to the nth power and you get 1.

Example 4.4: Roots of Unity

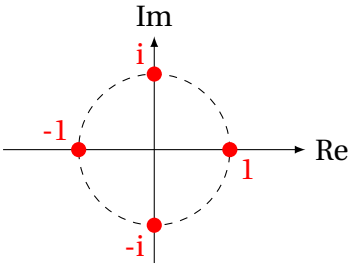
- 2^{nd} roots of unity = $\sqrt{1} = \{\pm 1\}$
- 4^{th} roots of unity = $\sqrt[4]{1} = \{\pm 1, \pm i\}$
- 8^{th} roots of unity = $\sqrt[8]{1} = \{\pm 1, \pm i, \pm\left(\frac{1+i}{\sqrt{2}}\right), \pm\left(\frac{1-i}{\sqrt{2}}\right)\}$

In general, the nth root of unity is n distinct numbers.

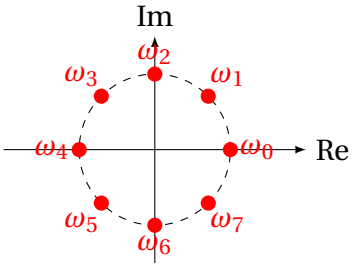
Visually, the nth roots of unity is the n *equally* spaced points on the unit circle



2nd Roots of Unity



4th Roots of Unity



8th Roots of Unity

Recall that when we multiply two complex numbers, we just add their angles, so we can come up with this formula.

Lemma 4.5: Formula for nth roots of unity

Formula: n^{th} roots of unity

$$= \text{angle } 0\cdot\theta, 1\cdot\theta, 2\cdot\theta, \ldots \qquad \text{where } \theta = \frac{2\pi}{n}$$

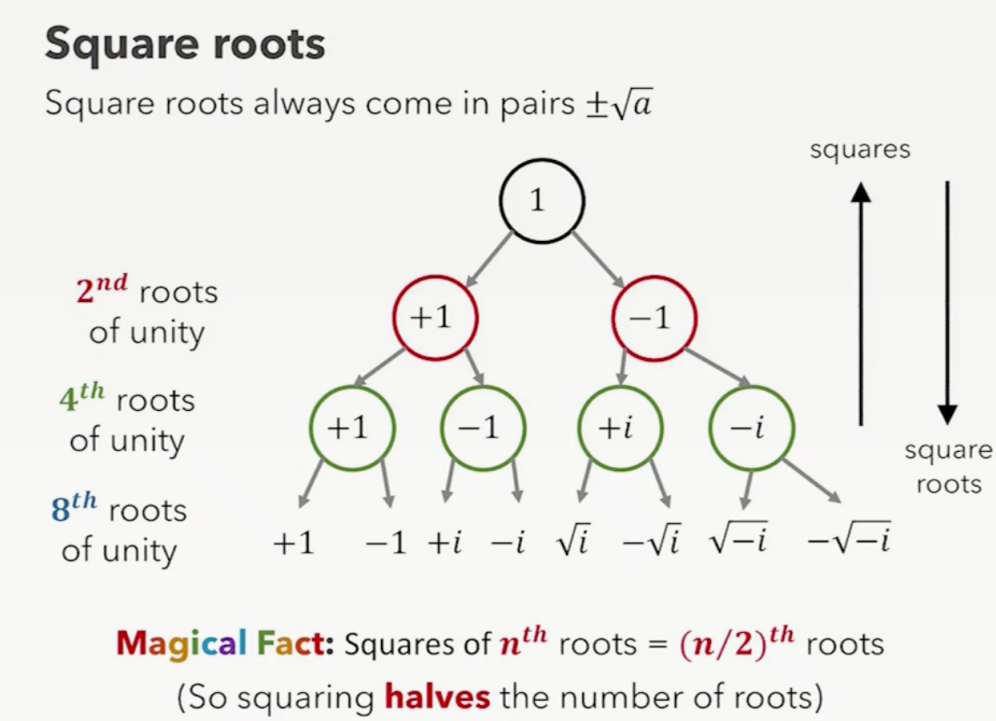


Figure 4.3

Complex number Takeaways

- Generator fact: for all $0 \le i \le m-1, \omega_i = \omega_1^i$
- The magical fact above

09/10/23

Lecture 5

Polynomail Multiplication

Recall that we are given two polynomials in their coefficient representation, and we want to compute their product. We saw last lecture that we can multiply two polynomials in their coefficient representation in $O(n^2)$

And also multiplying two polynomials in the value presentation takes $\mathcal{O}(n)$ time so it's optimal.

The bottle neck in the polynomial multiplication is the evaluation of the polynomials at $\mathcal{O}(n)$ points, which with the naive algorithm takes $\mathcal{O}(n^2)$ time. Can we possibly do this faster with a more clever choice of evaluation points? Yes

Lets design the algorithm:

Recall that if the degree of the polynomials we are multiply is of degree $n-1$, then their product $C(x)$, has degree $2(n-1) = 2n-2$. Therefore in the value representation, $C(x)$ is defined by $2n-1$ points. This means that we'll need to evaluate p and q on at least $2n-1$ points, call it m, since we will be multiplying the values the polynomials to get $C(x)$

Let m be the first power of 2 such that $m \geq 2n-1$. We will evaluate p and q on m^{th} roots of unit

$$\omega_0, \omega_1, \ldots, \omega_{m-1}$$

in time $\mathcal{O}(m\log(m)) = \mathcal{O}(nlog(n))$. This is the **Fast Fourier Transform.**

Fast Fourier Transform

The fast fourier transform is an algorithm for evaluating a polynomial of degree $n-1$ on m points in $\mathcal{O}(n\log n)$ **inputs:**

1. m, a power of two
2. a polynomial: $p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{m-1}x^{m-1}$

output : $[p(\omega_0), p(\omega_1), \ldots, p(\omega_{m-1})]$

Divide an Conquer

Let's write out $p(x)$ and split it into two parts, the evan part and odd part.

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{m-1}x^{m-1}$$

$$\begin{aligned}
\textbf{even part: } & p_0 + p_2 x^2 + p_4 x^4 + \cdots + p_{m-2}x^{m-2} \\
& p_0 + p_2 x^2 + p_4 (x^2)^2 + p_6 (x^2)^3 + \ldots \\
& \textbf{Even}(x^2) \\
\text{where } \textbf{Even}(z) &= p_0 + p_2 z + p_4 z^2 + p_6 z^3 + \ldots \\
& . \\
\textbf{odd part: } & p_1 x + p_3 x^3 + p_5 x^5 + p_7 x^7 + \cdots + p_{m-1}x^{m-1} \\
& = x \cdot (p_1 + p_3 x^2 + p_5 x^4 + x^6 + \cdots + p_{m-1}x^{m-1}) \\
& = x \cdot \textbf{Odd}(x^2) \\
\text{where } \textbf{Odd}(z) &= p_1 + p_3 z + p_5 z^2 + \ldots \\
& .
\end{aligned}$$

$$\therefore p(x) = \text{Even}(x^2) + x \cdot \text{Odd}(x^2)$$

where deg(**Even**) $= \dfrac{m-2}{2} = \dfrac{m}{2} - 1$, same for deg(**Odd**)

Recall what the goal is: we need to to evaluate the polynomial at m points, which will be the m^{th} roots of unity. We choose the m^{th} roots of unity because it has m solutions, $\omega_0, \ldots \omega_{m-1}$

$$[p(\omega_0), p(\omega_1), \ldots, p(\omega_{m-1})]$$

$$p(\omega_i) = p_e(\omega_i^2) + \omega_i p_o(\omega_i^2)$$

The recurrence relation we therefore get is as follows:

$$T(m) = 2T(m/2) + \mathcal{O}(m)$$

Which is $\mathcal{O}(n \log n)$ by the masters theorem. Next up is interpolation. How can we do it in $\mathcal{O}(n \log n)$?

Faster Interpolation

The algorithm input is a polynomial at m values, and output the coefficient of that polynomials.

Interpolation (Inverse FFT)

Here is an idea for how to pick the m points at which to evaluate a polynomial $A(x)$ of degree $\leq n-1$. If we choose them to be positive-negative pairs, that is:

FT formula:

$$p(\omega_l) = \sum_{j=0}^{m-1} p_j \cdot (\omega_l)^j$$

Inverse FT Formula:

$$\begin{aligned} p_l &= \frac{1}{m} \cdot \sum_{j=0}^{m-1} p(\omega_j) \cdot (\omega_{m-l})^j \\ &= \frac{1}{m} \cdot q(\omega_{m-l}) \end{aligned}$$

The matrix viewpoint

Recall we have a polynomial and we compute it's values at m points, which are the roots of unity.

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{m-1} x^{m-1}$$

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

$$\begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ p(\omega_2) \\ \vdots \\ p(\omega_{m-1}) \end{bmatrix} = \begin{bmatrix} p_0 & +p_1\omega_0 & +p_2\omega_0^2 & +\cdots+p_{m-1}\omega_0^{m-1} \\ p_0 & +p_1\omega_1 & +p_2\omega_1^2 & +\cdots+p_{m-1}\omega_1^{m-1} \\ p_0 & +p_1\omega_2 & +p_2\omega_2^2 & +\cdots+p_{m-1}\omega_2^{m-1} \\ & \vdots & & \\ p_0 & +p_1\omega_{m-1} & +p_2\omega_{m-1}^2 & +\cdots+p_{m-1}\omega_{m-1}^{m-1} \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \dots & \omega_0^{m-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{m-1} \\ 1 & \omega_2 & \omega_2^2 & \dots & \omega_2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{m-1} & \omega_{m-1}^2 & \dots & \omega_{m-1}^{m-1} \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{m-1} \end{bmatrix}}_{[p]}$$

This is called a **Fourier Transform Matrix**. Naively, this matrix vector product computes in $O(m^2)$, but FFT solves this problem in $\mathcal{O}(m \log m)$ time.

Note: it solves this **without ever writing down** M

Inverse Fourier Transform: We start off with the values of a polynomial at m points and want the coefficients of a polynomial.

$$M^{-1} \cdot \begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ p(\omega_2) \\ \vdots \\ p(\omega_{m-1}) \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{m-1} \end{bmatrix}$$

Then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even powers of x_i coincide with those of $-x_i$.

Lecture 6

Lecture 7

Lecture 8

Lecture 9

09/12/23

Lecture 10

Paths in Graphs

Lecture 11

10/5/23

Lecture 12

Dynamic Programming Pt 1

12.1 Elements of Dynamic Programming

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them on by one, smallest first, using the answers to small subproblems to help figure out larger one, until the whole lot of them is solved.

Large problems break up into smaller sub problems, i.e., the solution of some big problem can be expressed in terms of the solutions to the smaller sub-problems.

eg. fib(n) = fib(n - 1) + fib(n - 2)

But what's different about dynamic programming from divide and conquer? After all in divide and conquer we also solved smaller subproblems. The problems that are solved with dynamic programming have a lot of overlapping subproblems. This means that we can save resources by solving a subproblem once and storing its value, and then use that subproblem many times over.

eg. fib(i + 1), fib(i + 2), ... All use fib(i) indirectly, so we can store the value of fib(i) instead of having to repeatedly compute it.

12.1.1 Here is a general template to solve DP problems

- **step 1:** Identify subproblems. What makes a good subproblem?

- Not too many of them (the more subproblems the slower the DP algorithm is).
- Must have enough information in it to compute subproblems recursively. (needed for step 2.)

- **step 2:** Find a recursive formulation for the subproblems
- **step 3:** Design the dynamic programming algorithm and memoize computation starting from the smallest subproblems and build up

12.2 Two ways to do DP

- **Top Down**: start from the biggest problem and recurse to smaller problems. Looks just like recursion, with one exception: **memoization**; keeping track of what smaller problems we have solved already.
- **Bottom up:** start from the smallest problems first and then bigger problems. Still memoize, but usually doesn't have a recursive call.

12.2.1 Fibonacci Numbers

Recall the naive recursive way to compute fibonacci:

```
1 def fib(n):
2     if n <= 1:
3         return n
4     else:
5         return fib(n - 1) + fib(n - 2)
```

The runtime of this algorithm is exponential, extremely slow. The first observation we can make about why this algorithm is so slow is that the recursion tree repeats a lot of sub-problems. For every node, in the recursive tree, it recomputes the problem from scratch.

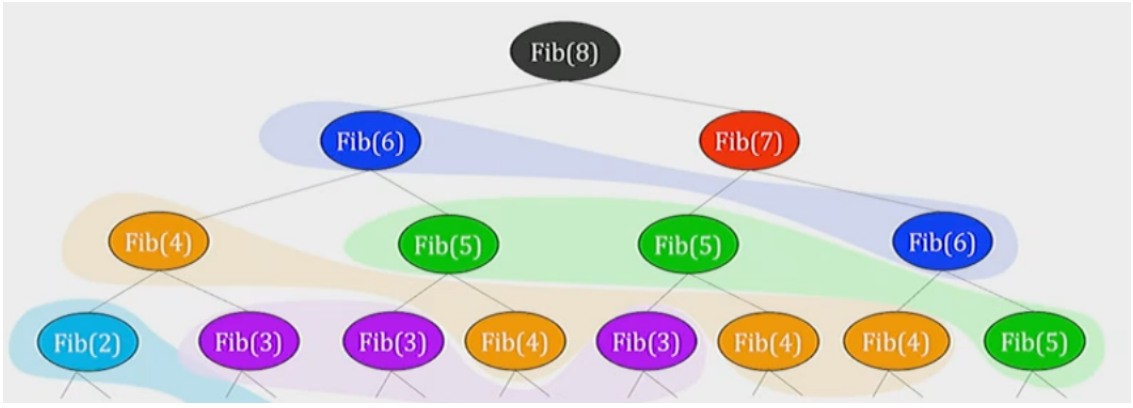


Figure 12.1: Repetition in fib recursion tree

So how can we fix this? We can keep an array and remember the computation we did so that we don't have to recompute this in another subproblem (**memo-ization**).

```
memo = [0, 1, None, None, ..., None] # Global array

def fastFibTopDown(n):
    if memo[n] != None:
        return memo[n]
    else:
        mem[n] = fastFibTopDown(n - 1) + fastFibTopDown(n - 2)
        return mem[n]
```

We can also do a bottom up.

```
def fastFibBottomUp(n):
    mem = [None] * (n + 1)
    mem[0] = 0
    mem[1] = 1
    for i in range(2, n + 1):
        mem[i] = mem[i - 1] + mem[i - 2]
    return mem[n]
```

An additional optimization we can make with this approach to the space complexity. Notice that in fib, we only need to keep track of the two previous elements (f(n - 1), f(n - 2)), so we instead of keeping track of all fib values from 0...n, lets only keep track of the two previous values so that we can compute the next value.

```
def fibBottomUpSpaceSaving(n):
    mem = [0, 1]
    for i in range(2, n + 1):
        x = mem[0] + mem[1]
        mem[0] = mem[1]
        mem[1] = x
    return mem[1]
```

12.3 DP order of Computation and Dags

There is an implicit DAG in dynamic programming problems! The nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem B, we need the answer to subproblem A, then there is a conceptual edge from A to B.

To see this for fib, lets draw a directed edge (i, j) if the computation of i directly depends on/uses the solution to subproblem j.

BottomUp solves problems in the order (or reverse order, depending on how you drew edges, in this draw it's reverse order) of this topological sort.

In Top Down: we starting recursing at the top, but but the memo the memo-ization table starts to get filled according to the reverse topological sort graph.

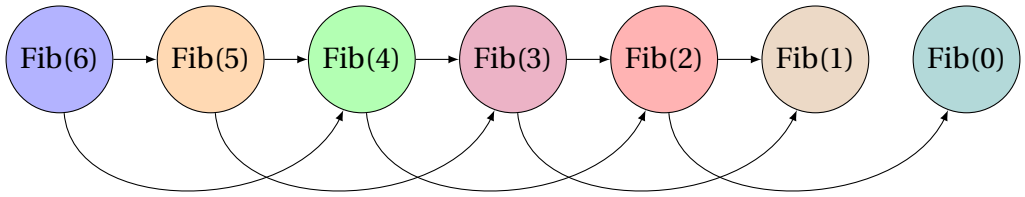


Figure 12.2: Fib(6) drawn as a DAG

12.4 SSSP on DAG's

Recall the shortest paths problem on dags. We're given a weighted graph G (positive or negative weight) and a starting vertex s, and want to compute the shortest path to every vertex u from s. How do we do that?

$$f(u) = \text{length of shortest path from s to u}$$

$$f(u) = \begin{cases} 0, & \text{if } u = s \\ \infty, & \text{elif indeg}(u) = 0 \\ \min\limits_{(v,u)\in E} f(v) + w(v,u), & \text{otherwise} \end{cases}$$

The recursive case takes the minimum path from every edge coming to u + the weight of the edge. But how do we code this?

In code we are given an adjacency list, a list where the linked list contains all the outgoing edges of the current vertex, we want all the incoming edges though, so we reverse the graph.

12.5 Bellman Ford

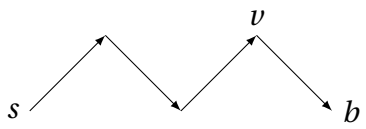
```
1 def Bellman(G, s):
2     T[1...n] = all infinity
3     T[s] = 0
4     for k = 1 to |V| - 1:
5         for e = (u, v) in E:
6         ½
7             T[v] = min{T[v], T[u] + w(u, n)}
8     return T;
```

The first step in figuring out the solution to a DP problem is to find a function that has a nice recurrence relation that we can then memoize. So what is the function that we are gonna use that will eventually lead us to solve bellman ford?

- $f(b,k)$ = length of shortest path from $s \to b$ using $\leq$ edges
- since we can assume that the shortest path without a negative cycle connecting that shortest path cannot use more than n - 1 edges, otherwise if it did then it would not be a tree, i.e., there is a cycle. So we want $f(t, n-1)$
- $$f(b,k) = \begin{cases} 0, & b = s, k = 0 \\ \infty, & b \neq s, k = 0 \\ \min(f(b,k-1), \min\limits_{(v,b)\in E}(f(v,k-1) + w(v,b))), & \text{otherwise} \end{cases}$$

To understand this recurrence relation, lets explain the three cases.

- The first case is when $b = s$ and we use no edges, therefore we've already reached b.
- The second case is when $b \neq s$ and we use no edges, therefore it's impossible to get to b since we cannot use any more edges.
- The third case is best explained by picture. The first term in the argument of min is $f(b,k-1)$ since certainly if we are allowed to use at most k edges, then are allowed to try and get to b with $k-1$ edges. The second term is another min: $\min\limits_{(v,b)\in E}(f(v,k-1) + w(v,b))$ which is best explained by the picture below:



Take the previous edge of to b with the minimum shortest distance + the weight of that edge.

12.6 All Pair Shortest Paths Algorithm

This algorithm output should be a 2d-array ($n \times n$) $T[1\ldots n][1\ldots n]$, s.t. $T[i][j]$ = length of shortest path from node i to node j.

The naive solution would be to run bellman ford from every vertex. What's the runtime? Each run of bellman ford takes $\mathcal{O}(nm)$, and you run it n times, so in total this algorithm has runtime: $\mathcal{O}(n^2 m)$ which is potentially $\mathcal{O}(n^4)$ if the graph is dense.

We'll discuss a faster algorithm (*Floyd Warshall*) that has runtime $\mathcal{O}(n^3)$

```
def FloydWarshall(G):
    init T[1...n][1...n] = all ∞
    for each vertex i to n:
        T[i][i] = 0

    for each edge (u, v) ∈ E:
        T[u][v] = w(u, v)

```

```
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                a = T[i][j]
                b = T[i][k] + T[k][j]
                T[i][j] = min(a, b)
    return T
```

let $f(i,j,k)$ = length of shortest path from i to j using intermediate vertices that must be in the set $\{1,...,k\}$

$$f(i,j,k) = \begin{cases} 0, & \text{if } i = j, k = 0 \\ w(i,j), & \text{if } i \neq j, k = 0 \\ \min \left\{ \begin{array}{c} f(i,j,k-1), \\ f(i,k,k-1) + f(k,j,i-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

The recurrence relation is best explained by the picture in DPV. There are two cases, the shortest path from i to j is better than the shortest path i to k + 1 and the k + 1 to j.

10/10/23

Lecture 13

Dynamic Programming Pt 2

Agenda for this lecture:

- Longest Increasing Subsequence
- Edit Distance
- Knapsack

13.1 Longest Increasing Subsequences

- **input**: an array of n integers $a = [a_1, \ldots, a_n]$
- **output**: The length of the longest increasing subsequence, not necessarily contiguous, of the input.

Subproblem: let $L[j]$ = length of LIS in the array $[a_1, \ldots, a_j]$ that **ends in** a_j for $j = 1, ..., n$.

Why must it end in a_j? Because how else would we know that we can add a new element to the subsequence? The next element must be greater than a_j, the largest element so far in the current subsequence.

This gives us a recursive formula:

$$L[j] = 1 + \max_{i<j}(L[i] \mid a_i < a_j)$$

The pseudo code will be as follows:

```
def LIS(a):
    L[1...n] = all 0
    for j = 1 to n:
        best = 0
        for i = 1 to j:
            if a_i < a_j:
                best = max(best, L[i])
        L[j] = 1 + best
return max(L)
```

It's easy to see that the runtime of this is $\mathcal{O}(n^2)$.

13.2 Edit Distance

- **input**: two strings $S[1 \dots m]$ and $T[1 \dots n]$
- **output**: compute the smallest number of edits to turn S into T.

The edits that are allows are:

1. insert a character into S
2. delete a character from S
3. change on character to another character

Applications of edit distance:

- auto correct
- word suggestions in search engines
- dna analysis of similarities

Defining the subproblem

For all $0 \le i \le m$ and $0 \le j \le n$, let:

$$E(i, j) = \text{EditDist}(S[1 \dots i], T[1 \dots j])$$

Be the cost of optimal alignment between $S[1 \dots i], T[1 \dots j]$. How many subproblems do we have? We have a subproblem for every i, j pairs, each having $m+1$ and $n+1$ different possible numbers respectively, therefore we have $\mathcal{O}(m \cdot n)$ subproblems.

13.2.1 Finding the Recurrence relations

. What are the three ways of aligning $S[1\dots i]$ and $T[1...j]$ using the smaller subproblems. There are three ways to do this.

1. Case 1: $S[i]$ is deleted, (aligned with an empty space in T)
2. Case 2: $T[j]$ is added.
3. Case 3: Change $S[i]$ to $T[j]$

$$E(i,j) = \min \left\{ \begin{array}{c} 1 + E(i-1,j), \\ 1 + E(i,j-1), \\ 1 + E(i-1,j-1) \end{array} \right\}$$

The bases cases are when we align the a string with an empty string, which has an alignment cost of the non empty string since we'd have to delete that many characters to align them. Therefore: $E(0,j) = j$ and $E(i,0) = i$.

What's the runtime? Each subproblem relies on only 3 smaller subproblems, which could be computed first to and memoized. There are $\mathcal{O}(mn)$ subproblems, and each does an $\mathcal{O}(1)$ operation, therefore the runtime is $\mathcal{O}(mn)$

13.3 Knapsack With Repetition

input: a weight capacity W, and n items with (weights, values): $(w_1, v_1), \dots, (w_n, v_n)$.

output: Most valuable combination of items, whose total weight is at most W.

Two Variants:

1. With repetition (aka unbounded supply, aka with replacement), i.e., for each item i, we can take as many copies of it as we want.
2. Without repetition (0-1 knapsack, aka without replacement), i.e., for each item i, we either take 1 copy or 0 copy of it.

13.4 With Repetition

Defining a subproblem: for all $c \leq W$, let $K(c)$ = best value achievable for knapsack of capacity c.

Recurrence relation: How can we write $K(c)$ interms of smaller subproblems? Well if $K(c)$ includes item i, then removing it would give us an optimal solution to $K(c - w_i)$. Therefore $K(c) = K(c - w_i) + v_i$ for some item i. We don't know which item so we need to try all possibilities:

$$K(c) = \max_{i:w_i \leq c} (v_i + K(c - w_i))$$

```
def KnapSackWRepetition(W, (w_1, v_1),...,(w_n, v_n))
    init K to array of size W + 1
    K(0) = 0 # bag with zero capacity has no value.
    for c = 1 to W:
        K(c) = max_{i:w_i ≤ c} (v_i + K(c − w_i))
    return K(W)
```

13.4.1 Analying Runtime:

How many subproblems do we have? $O(W)$, we start from 0 and go all the way to W

How much work do we do per subproblem? We take a max of at most n items, therefore $\mathcal{O}(n)$ time per subproblem.

Total runtime = $\mathcal{O}(Wn)$

13.5 Knapsack Without Repetition

input: same input as above

output: most valuable subet of items, whose total weight is $\leq W$.

Without repetition, our previous subproblem now becomes completely useless. Knowing the value of $K(c-w_i)$ is the maximum value doesn't help us, since we don't know whether or not item n already got used up in this partial solution. We must refine our subproblem to carry additional information about the items being used.

13.5.1 Finding a subproblem

Idea: Solve knapsack for smaller capacities and **also** smaller set of items. We add a second parameter $0 \leq j \leq n$.

$$K(c, j) = \text{max value achievable using a knapsack of capacity w and items } 1, \ldots, j.$$

The answer we seek is $K(W, n)$.

13.5.2 Find Recurrence Relation

How can we write $K(c, j)$ in terms of smaller subproblems? There are two cases.

- Case 1: Opt sln using items 1,..., j doesn't actually use item j.
- Case 2: Opt sln using items $1, \ldots, j$ uses item j.

Therefore we take the best (max) of these two cases

$$K(c, j) = \max \left\{ \begin{array}{c} K(c, j-1), \\ K(c-w_j, j-1) + v_j. \end{array} \right\}$$

13.5.3 Designing the Algorithm

How do we memo-ize the subproblems in this recurrence relation?

The bases cases are as follows:

- $K(c,0) = 0$, since we aren't allowed to use any items.
- $K(0,j) = 0$, since we no don't have capacity for more items.

What's the memoization order? We should be thinking about the subproblems as a 2d array. If the columns of this table is the weights, and the rows is the items, each subproblem relies on the previous row, $K(c, j-1)$, which is directly above it, and $K(c-w_j, j-1)$, which is also above it, but somewhere to the left.

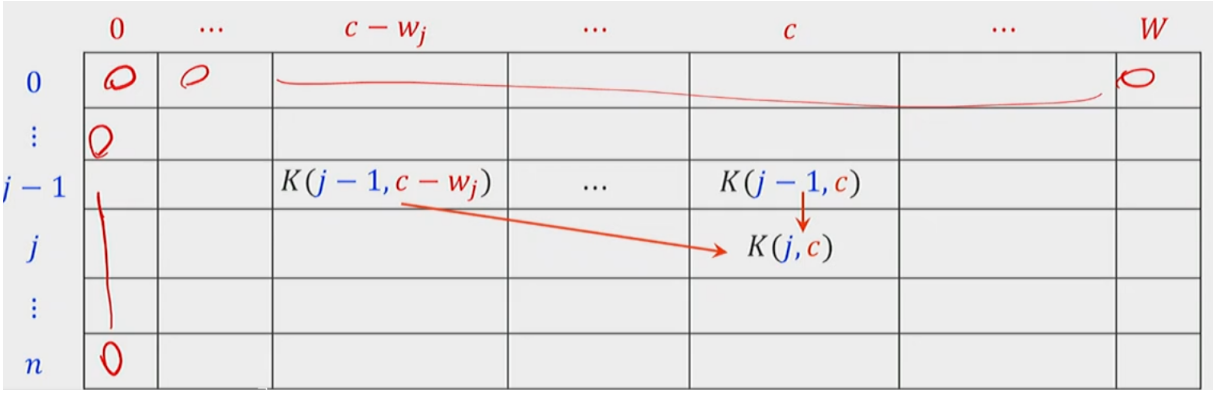


Figure 13.1: Memoization order

13.5.4 Runtime of this algorithm

There is $\mathcal{O}(W \cdot n)$ subproblems. Each subproblem consists of taking a max over 2 values, therefore $\mathcal{O}(1)$ time per subproblem.

Therefore, the total runtime is: $\mathcal{O}(W \cdot n)$ runtime. (*.pseudo polynomial time alg*)

Space complexity: we can naively store a 2d array, but notice we only need the previous row to compute the current subproblem. Therefore we only need to store $\mathcal{O}(W)$ amount of space; the current row, and the previous row.

10/12/23

Lecture 14

Dynamic Programming Pt.3

14.1 Traveling Salesman Problem

Input: cities $1 \ldots n$ and pairwise distances d_{ij} between cities i and j. (a fully connected graph)

Output: Find a "tour" of minimum total distance.

Definition 14.1: Tour

A tour is a path through the cities, that

1. starts from city 1
2. visits every city, exactly once
3. returns to city 1.

14.1.1 Finding a Subproblem

Suppose we started at city 1 as required, have visited a few cities, and now in city j. What information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them.

Subproblems refer to partial solutions, and in this case, the most obvious partial solution is the initial portion of a tour. It starts from city 1, ends in city j, passing through all cities in a set S (which includes 1 and j)

Subproblems: for all $j \leq n$ and $S \subseteq \{1, \ldots, n\}$, s.t. S includes 1 and j, let

$T(S, j)$ = length of the shortest path visiting all cities in Sexactly onces starting from 1 and ending at j.

14.1.2 Finding the Recurrence Relation

How can we compute $T(S, j)$ using smaller subproblems? What should we pick as the second-to-last city before visiting j. It has to be some city in S, since we must pass it before visiting j. Let that city be i.

Therefore the overall length from 1 to j is $T(S - j, i) + d_{ij}$, we must pick the best such i:

$$T(S, j) = \min\{T[S - \{j\}, i] + d_{ij} \qquad | \; i \in S \wedge i \neq j\}$$

What about the bases cases?

- $T(\{1\}, 1) = 0$, since the distance starting at 1, and ending at 1, using only city 1, is 0, you don't move.
- For all other $|S| \geq 2$, $T(S, 1) = \infty$ since we don't wait a partial tour to start and end at the same city, our final solution will.

Final solution: Recall that we want to start and end at city i, therefore we want to find the best j, (the best city to end in) such that the cost is minimized, i.e.,

$$\min_{j \neq 1} T(\{1, \ldots, n\}, j) + d_{j1}$$

14.1.3 Designing the Algorithm

```
def TSP(D):
    init array T of size 2^n × n # 2^n different possible subsets S
    T[{1}, 1] = 0
    for set size s = 2 to n:
        for sets S of size s and containing 1:
            T[S, 1] = ∞
            for all j ∈ S ∧ j ≠ 1:
                T[S, j] = min_{i∈S∧i≠j} {T[S − j, i] + d_ij}
    return min_{j≠1} T[{1, ..., n}, j] + d_j1
```

14.1.4 Runtime anaylsis

There is $2^n \cdot n$ subproblems and each takes linear time to solve. Therefore the total runtime is $\mathcal{O}(2^n \cdot n^2)$

14.2 Independent Sets in Trees

We consider independent sets in trees and not general graphs because this problem can't be in done polynomial time.

Input: Undirected graph $G = (V, E)$, which is a tree.

Output: Largest "independent set" of G. An independent set of G, is $S \subseteq V$, if there are no edges between any $u, v \in S$

14.2.1 Finding the Subproblem:

We can start by rooting the tree at any node r, which gives us a natural ordering of children, and grand children, etc, of the rooted node. Now, each nodes defines a subtree–the one hanging from it.

This immediately suggests subproblems:

$$I(u) = \text{size of largest independent set of subtree hanging from } u.$$

Our goal is to find $I(r)$.

14.2.2 Finding a Recurrence Relation:

There are two cases:

- case 1: The independent set includes u.
- case 2: The independent set doesn't includes u.

Therefore the recurrence relation is as follows:

$$I(u) = \max \begin{pmatrix} 1 + \sum\limits_{\text{grandchildren w of u}} I(w), \\ \sum\limits_{\text{children w of u}} I(w) \end{pmatrix}$$

Lecture 15

Linear Programming

Example 15.1

Imagine there is a factory that produces two items, *foo* and *bar* with some ingredients to make. Foo sells for \$4/oz and bar sells for \$5/oz.

- foo takes 1 oz butter, 2 oz water, and 4 oz goo
- bar takes 6 oz butter, 3 oz goo

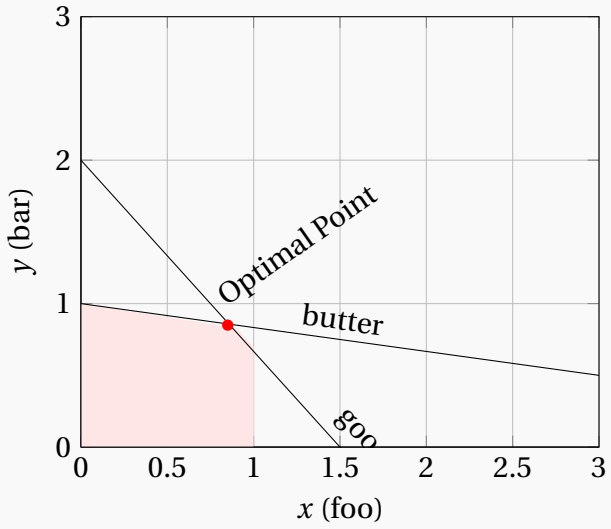
The factory has a limited number of ingredients:

- W = 10 oz of water
- B = 6 oz of butter
- G = 6 oz of goo

How many foos and bars should the factory make to maximize profit? This could be modeled as a linear program. Here are the equations:

$$\max 4x + 5y \qquad \text{Object: profit} \tag{1}$$
$$1x + 6y \leq 6 \qquad \text{butter constraint} \tag{2}$$
$$2x + 0y \leq 10 \qquad \text{water constraint} \tag{3}$$
$$4x + 3y \leq 6 \qquad \text{goo constraint} \tag{4}$$
$$x, y \geq 0 \qquad \text{non negative production of foo and bar constraint} \tag{5}$$

The plot with these equations is as follows.



If we rotate this plot where the object function points in the z axis, and have a marble roll down this hill, it will end up at the optimal point.

15.1 Simplex Algorithm

The simplex algorithm is an algorithm that solves linear programs. It's a greedy algorithm that starts at a vertex in the feasible region, and then moves to a better vertex solution, until it can't move anymore.

It's a greedy algorithm because it always moves to a better solution, and it's a local search algorithm because it only looks at the neighbors of the current solution.