Algorithms

Muchang Bahng

Spring 2024

Contents

1 Complexity

A course on the study of algorithms.

Definition 1.1 (Algorithm)

An **algorithm** is a procedure for solving a mathematical problem in a *finite* number of steps. It should be

1. finite,
2. correct,
3. efficient

An algorithm, with respect to some inputs $\mathbf{n}$, will have a runtime that is some function f. We would like a formal way to analyze the asymptotic behavior between two functions.

Definition 1.2 (Complexity)

Given two positive functions f, g,

1. $f = O(g)$ if f/g is bounded.[a]
2. $f = \Omega(g)$ if g/f is bounded, i.e. $g = O(f)$.
3. $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

There are two notions of complexity here. We can compare f and g with respect to the *value* N of the input, or we can compare them with respect to the *number of bits* n in the input. While we mostly use the complexity w.r.t. the value, we should be aware for certain (especially low-level operations), the bit complexity is also important.

Let's do a quick comparison of various functions. Essentially, if we want to figure out the complexity of two positive functions f, g,[1] we can simply take the limit.

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = \begin{cases} 0 & \implies f = O(g) \\ 0 < x < +\infty & \implies f = \Theta(g) \\ +\infty & \implies f = \Omega(g) \end{cases} \tag{1}$$

Most of the time, we will have to use L'Hopital's rule to derive these actual limits, but the general trend is

1. $\log n$ is small
2. $\text{poly}(n)$ grows faster
3. $\exp(n)$ grows even faster
4. $n!$ even faster
5. n^n even faster

Theorem 1.1 (Properties)

Some basic properties, which shows very similar properties to a vector space.

1. Transitivity.

$$f = O(g), g = O(h) \implies f = O(h) \tag{2}$$
$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h) \tag{3}$$
$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h) \tag{4}$$
$$\tag{5}$$

[a] Note that it is more accurate to write $f \in O(g)$, since we consider $O(g)$ a class of functions for which the property holds.
[1] These will be positive since the runtime must be positive.

2. Linearity.

$$f = O(h), g = O(h) \implies f + g = O(h) \tag{6}$$
$$f = \Omega(h), g = \Omega(h) \implies f + g = \Omega(h) \tag{7}$$
$$f = \Theta(h), g = \Theta(h) \implies f + g = \Theta(h) \tag{8}$$
$$\tag{9}$$

Example 1.1 ()

Compare the following functions.

1. $f(n) = \log_{10}(n), g(n) = \log_2(n)$. Since they are different bases, we can write $f(n) = \log(n)/\log(10)$ and $g(n) = \log(n)/\log(2)$. They differ by a constant factor, so $f = \Theta(g)$.
2. $f(n) = (\log n)^{20}, g(n) = n$. We have

$$\lim_{n\to\infty} \frac{(\log n)^2 0}{n} = \lim_{n\to\infty} \frac{20 \cdot (\log n)^{19} \cdot \frac{1}{n}}{1} = \ldots = \lim_{n\to\infty} \frac{20!}{n} = 0 \implies f = O(g) \tag{10}$$

3. $f(n) = n^{100}, g(n) = 1.01^n$. We have

$$\lim_{n\to\infty} \frac{n^{100}}{1.01^n} = \lim_{n\to\infty} \frac{100n^{99}}{1.01^n \cdot \log(1.01)} = \ldots = \lim_{n\to\infty} \frac{100!}{1.01^n \cdot (\log 1.01)^1 00} = 0 \implies f = O(g) \tag{11}$$

Let's do a slightly more nontrivial example.

Example 1.2 ()

Given the following algorithm, what is the runtime?

```
for i in range(1, n+1):
  j = 1
  while j <= i:
    j = 2 * j
```

Now we can see that for each i, we will double up to $\log_2(i)$ times. Therefore summing this all over i is

$$\sum_{i=1}^{n} \log_2(i) = \log_2(n!) \leq \log_2(n^n) = n \log_2(n) \tag{12}$$

and so we can see that the runtime is $O(n \log n)$. Other ways to do this is to just replace the summation with an integral.[a]

$$\int_1^n \log_2(x)\, dx = x \log(x) - x\Big|_1^n = n \log(n) - n + 1 = O(n \log n) \tag{13}$$

2 Brute Force Algorithms

2.1 Basic Arithmetic

In here, we use basic deductions from elementary algebra to give us a starting point at which we analyze fundamental arithmetic algorithms.

[a] Need more justification on why this is the case. Mentioned in lecture.

Theorem 2.1 (Complexity of Addition)

The complexity of addition of two $O(N)$ values with n bits is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

By the same logic, the complexity of subtraction is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

Proof.

To see bit complexity, we are really taking each bit of each number and adding them together, plus a potential carry operation. Therefore, we are doing a bounded number of computations per bit, which is $O(1)$, but we must at least read through all of the bits, making this $O(\max\{n, m\})$.

Theorem 2.2 (Complexity of Multiplication)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.[a]
2. $O((\log n)^2)$ complexity.
3.

Theorem 2.3 (Complexity of Division)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 2.4 (Complexity of Modulus)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 2.5 (Complexity of Exponentiation)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 2.6 (Complexity of Square Root)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Definition 2.1 (Factorial)

[a] It turns out we can do better, which we will learn later.

2.2 Lists

Definition 2.2 (Max and Min of List)

Definition 2.3 (Bubble Sort)

Definition 2.4 (Binary Search)

2.3 Stack, Queues, Heaps

A heap is sort of in between a sorted array and an unsorted array.

2.4 Cryptography

Example 2.1 (GCD of Two Numbers)

Take a look at the following algorithm.

```
def gcd(a, b):
  if a == b:
    return a
  elif a > b:
    return gcd(a - b, b)
  else:
    return gcd(a, b - a)

print(gcd(63, 210))
```

Definition 2.5 (Primality Testing)

Definition 2.6 (Integer Factorization)

2.5 Matrix Operations

Definition 2.7 (Matrix Multiplication)

Definition 2.8 (Singular Value Decomposition)

Definition 2.9 (QR Decomposition)

Definition 2.10 (LU Decomposition)

Definition 2.11 (Matrix Inversion)

3 Divide and Conquer

Definition 3.1 (Divide and Conquer Algorithms)

The general idea is two steps:

1. Divide an input into smaller instances of the same problem. The simplest form is merge sort.
2. Conquer/solve these smaller instances, which takes less time.
3. Merge/Combine the smaller solutions into the original, bigger solution.

This is usually recursive, but does not need to be.

It has its applications in the most elementary operations, in sorting and multiplication. To prove correctness, we use induction by proving the correctness of the base case and then the inductive step to show that there is an invariant.

3.1 Recursive Algorithms and Recurrence Relation

I assume that the reader is familiar with recursive algorithms. Now to evaluate the runtime of a recursive algorithm, one must implicitly solve for the runtime of its recursive calls, and we can visualize it.

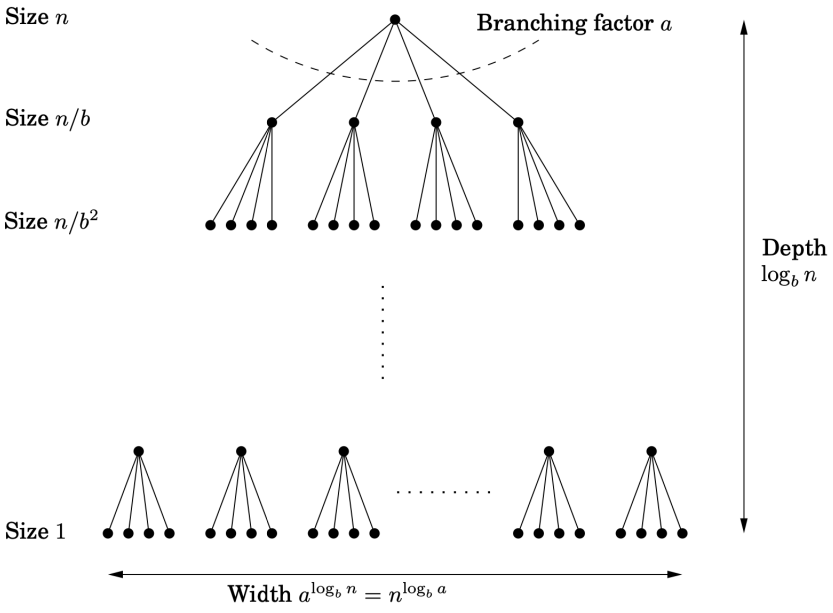


Figure 1

An important theorem for divide-and-conquer algorithms is

Theorem 3.1 (Master Theorem)

Given a recurrence relation of form

$$T(N) = aT(N/b) + O(N^c) \tag{14}$$

then the following holds

$$a > b^c \implies T(N) = O(N^{\log_b a}) \tag{15}$$
$$a = b^c \implies T(N) = O(N^c \log N) \tag{16}$$
$$a < b^c \implies T(N) = O(N^c) \tag{17}$$

Proof.

To intuit this, see that if $a > b^c$, then there arises a lot of subproblems, so our complexity is greater. If $a < b^c$, then we have few subproblems and can get a better runtime. If $a = b^c$, we get somewhere in between. To actually solve this, we can just unravel the recurrence to get the infinite series

$$T(N) = aT(N/b) + O(N^c) \tag{18}$$
$$= a^2 T(N/b^2) + N^c\left(\frac{a}{b^c} + 1\right) \tag{19}$$
$$= N^c\left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \dots\right) \tag{20}$$

So, if $a < b^c$, then even as $N \to \infty$, the sum is finite, so it is of order $O(N^c)$. If $a = b^c$, then the series is just $1 + \dots + 1$, which scales on the order of $O(\log_2 N)$. If > 1, then we have to calculate the last term, which contributes to our runtime and overpowers c.
Therefore, if a is large, our algorithm will have an exponential number of subproblems and will be bottlenecked by the.

3.2 Merge Sort and Counting Inversions

Algorithm 3.1 (Merge Sort)

Merge sort is the first instance.

Algorithm 1 Merge Sort

```
Require: Array nums
  function MergeSort(nums)
    n = len(nums)
    if n < 2 then                                                    ▷base case
      return nums
    end if
    mid = n // 2
    left_sorted = MergeSort(nums[:mid])                              ▷Divide into left half
    right_sorted = MergeSort(nums[mid:])                             ▷Divide into right half
    res = [0] * n
    i = j = k = 0
    while k < n do                                                   ▷Merge the sorted subarrays
      if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
        res[k] = left_sorted[i]          ▷We should add the next element from left array
        i += 1                           ▷if it is smaller or if right array is filled already
      else
        res[k] = right_sorted[j]
        j += 1
      end if
      k += 1
    end while
    return res
  end function
```

The recurrence relation for the runtime is as follows. Let $T(n)$ represent the worst-case runtime of `MergeSort` of size n. Then, we have

$$T(n) = 2 \cdot T(n/2) + O(n) \tag{21}$$

Consisting of two recursive calls with input size $n/2$ and then the merge step which is $O(n)$. But if we take a look at this, we have

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + O(n) && (22)\\ &= 2 \cdot \big(2 \cdot T(n/4) + O(n/2)\big) + O(n) && (23)\\ &= 4 \cdot T(n/4) + O(n) + O(n) && (24) \end{aligned}$$

and the number of times $O(n)$ is added up is $\log n$, meaning that this recurrence relation turns into $O(n \log n)$.

Definition 3.2 (Inversions)

Given two lists of ranked items, say

$$\begin{aligned} \text{Alice} &: a > b > c > d > e && (25)\\ \text{Bob} &: b > d > a > e > c && (26) \end{aligned}$$

We want to measure the dissimilarity between two rankings by counting the number of *inversions*, which are pairs of items for which open persons orders the opposite of the other (e.g. a, b for above).[a] So how many inversions are there? We can do this in $\Theta(n^2)$ by explicitly looking at all n pairs. Without loss of generality, we can assume that the first list is sorted simply by bijectively relabeling

these elements for both lists. Therefore, the set of inversions is defined to be

$$\{(i,j) \text{ s.t. } i < j \text{ and } \texttt{A[i] > A[j]}\} \tag{27}$$

Algorithm 3.2 (Counting Inversions)

The idea is very similar. By assuming that the first list is sorted, we can simply count the number of inversions in a single list A.

```
A = 5 6 1 3 4 8 2 7
```

1. In the divide step, we count all inversions in A_l, A_r, which are the left and right sides of A, *and* we sort A_l, A_r to add additional structure.

   ```
   1 3 5 6 | 2 4 7 8
   ```

2. In the conquer step, we merge them linearly but every time we add an element from A_r into our result, this reveals that there are k additional inversions added where k is the number of elements left in A_l to add.

Algorithm 2 Counting Inversions

```
Require: Array nums
  function INVERSIONS(nums)
    n = len(nums)
    if n < 2 then                                                    ▷base case
      return nums
    end if
    mid = n // 2
    left_sorted, left_invs = Inversions(nums[:mid])                  ▷Divide into left half
    right_sorted, right_invs = Inversions(nums[mid:])                ▷Divide into right half
    res = [0] * n
    i = j = k = 0                                                    ▷left, right, and combined index
    inv = 0                                                          ▷number of inversions
    while k < n do                                                   ▷Merge the sorted subarrays
      if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
        res[k] = left_sorted[i]          ▷We should add the next element from left array
        invs += len(left_sorted - i)     ▷Increment inversions by # of elems in left array
        i += 1                           ▷if it is smaller or if right array is filled already
      else
        res[k] = right_sorted[j]
        j += 1
      end if
      k += 1
    end while
    return res, inv + left_invs + right_invs
  end function
```

The recursion relation is still

$$T(n) = 2 \cdot T(n/2) + O(1) + O(n) \implies O(n \log n) \tag{28}$$

[a] Also known as Kendall-Tau distance in statistics.

3.3 Selection and Quick Sort

The next problem is a generalization of finding the median of an array, which we present can be done in $O(n)$ time *on average*. It turns out that that this idea of choosing a pivot and then dividing and conquering happens often in general selection and sort algorithms.

Algorithm 3.3 (Select kth Largest Element from Unsorted Array)

Algorithm 3 Find Kth Largest Element

Require: Array of numbers nums, integer k where 1 <= k <= length(nums)

```
function FindKthLargest(nums, k)
    if length(nums) = 1 then                          ▷Base case: if array has only one element
        return nums[0]
    end if
    pivot ← ⌊length(nums)/2⌋                          ▷Select middle element as pivot
    left ← [ ]                                        ▷Unsorted array for elements smaller than pivot
    right ← [ ]                                       ▷Unsorted array for elements larger than pivot
    n_pivots ← 0                                      ▷Count of elements equal to pivot
    for each n in nums do                             ▷Start filling in the arrays
        if n = nums[pivot] then
            n_pivots ← n_pivots + 1
        else if n ≤ nums[pivot] then
            left.append(n)
        else
            right.append(n)
        end if
    end for
    if length(right) ≤ k - 1 and length(left) ≤ length(nums) - k then
        return nums[pivot]          ▷Found the kth element which was in the middle in n_pivots
    else if length(right) ≥ k then
        return FindKthLargest(right, k)               ▷Largest is in the array of bigger numbers.
    else
        return FindKthLargest(left, k - length(right) - n_pivots)   ▷Largest is in the array of
smaller numbers.
    end if
end function
```

Algorithm 3.4 (Quick Sort)

Algorithm 4 Quicksort Algorithm

Require: Array A of comparable elements

```
function QUICKSORT(A, low, high)
    if low < high then
        p ← PARTITION(A, low, high)          ▷Get pivot position
        QUICKSORT(A, low, p - 1)             ▷Sort left subarray
        QUICKSORT(A, p + 1, high)            ▷Sort right subarray
    end if
end function

function PARTITION(A, low, high)
    pivot ← A[high]                          ▷Choose rightmost element as pivot
    i ← low - 1                              ▷Index of smaller element
    for j ← low to high - 1 do               ▷Scan through array
        if A[j] ≤ pivot then
            i ← i + 1                        ▷Increment index of smaller element
            swap A[i] and A[j]               ▷Swap current element with pivot
        end if
    end for
    swap A[i + 1] and A[high] ▷Put pivot in its correct position return i + 1 ▷Return pivot's final position
end function
```

3.4 Closest Pair of Points

The next problem simply takes a series of points and calculates the closest pair of points. This can be done trivially in $O(N^2)$ by taking all combinations, but with clever divide and conquer, we can reduce this down. The idea is that we want to divide them into a left and a right side, which we can sort in $O(N \log N)$ and find the median in $O(1)$. This now reduces to computing the closest pair of points in each $N/2$ half. If we can find the closest pair in each half, then we must compare it to all pairs of points across the halves, meaning that we must do $(N/2)^2$ comparisons again, leading to

$$T(N) = 2T(N/2) + O(N^2) \tag{29}$$

which isn't any better than $O(N^2)$. But imagine that we found that the smallest distance of the left and right were δ_1, δ_2, then for each point in the left side, we don't have to check all $N/2$ points on the right.

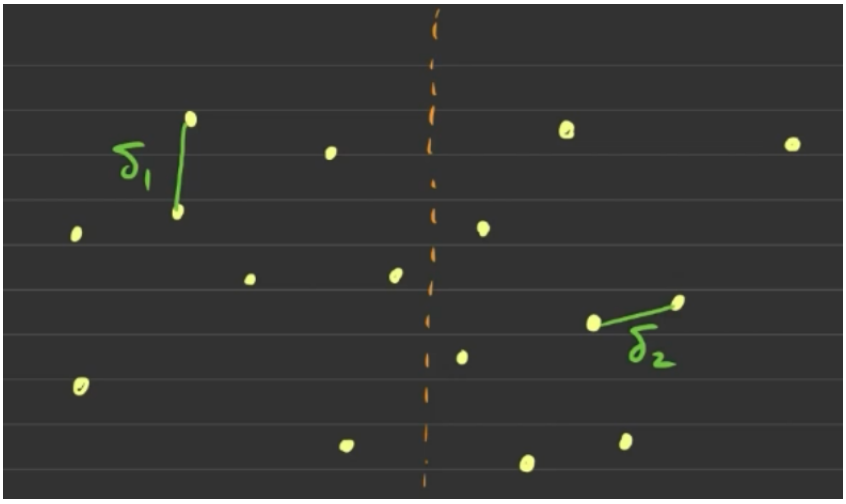


Figure 2

We just have to check those with distance at most $\delta = \min\{\delta_1, \delta_2\}$ from each point. Furthermore, we can discard all points that are too far away from the boundary. However, all N points could like in the relevant space, leading to $O(N^2)$ computations of distance.

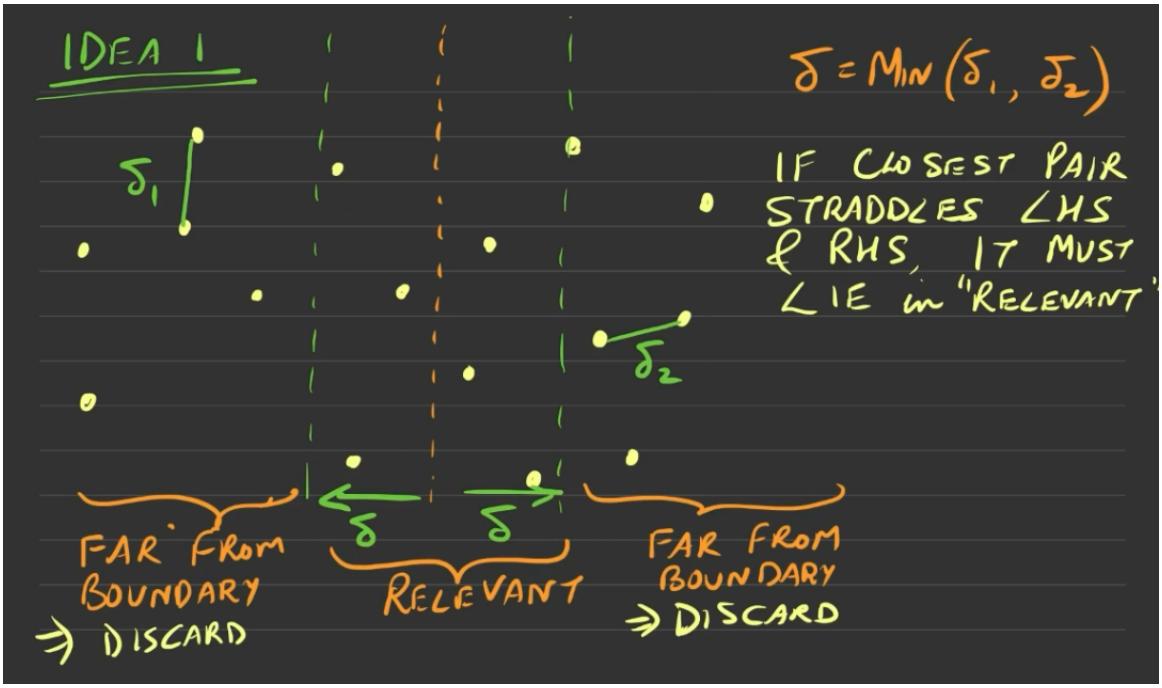


Figure 3

Therefore, we want to incorporate vertical distances and tile our relevant space into square of side length $\delta/2$.

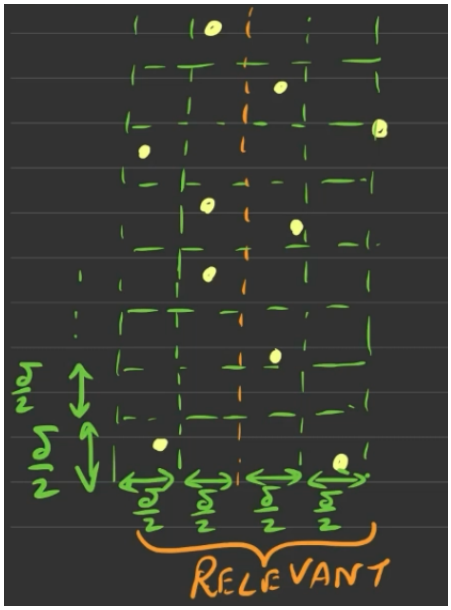


Figure 4

We claim that each square has at most 1 point, since if there were two, then their distance would be less than $\delta/\sqrt{2}$, which contradicts the distance between the two points being greater than δ. Therefore given a point, we only need to check a bounded constant.

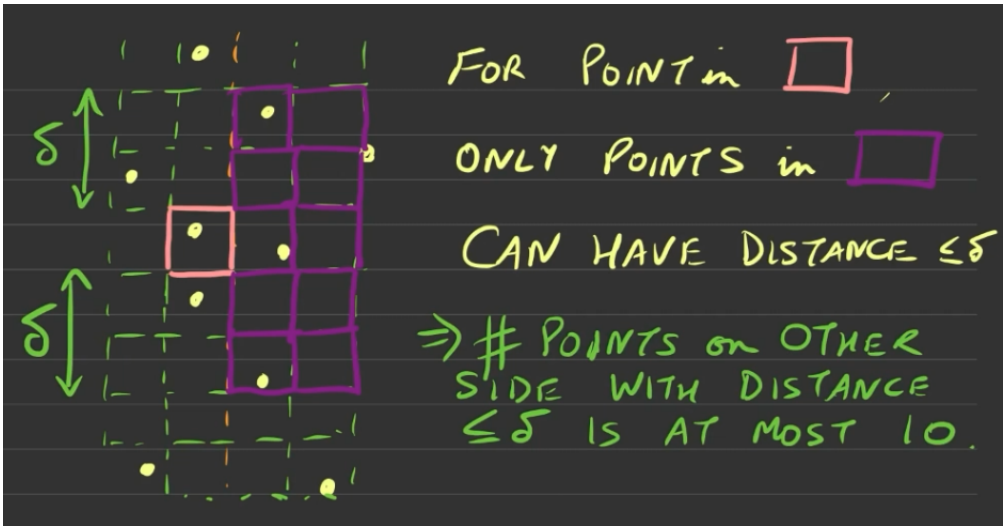


Figure 5: The number of points on the other side with distance $\leq \delta$ is at most 10. If we are more careful, we can reduce the number down to 7.

It turns out that we just need to compute $5N = (N/2) \cdot 10$ distances at most, and now our question reduces to how do we find these 5 points? Well we can first sort the points on the left and right by their y-coordinates, so we can just take a sliding window that encapsulates these points on the right for every on the left. The pointers of the sliding window should point to a point where the y-coordinate is at most δ away, and this can be done in constant time. Therefore, our recurrence relation is

$$T(N) = 2T(N/2) + O(N \log N) \implies T(N) = O(N \log^2 N) \tag{30}$$

This sorting along the y-coordinates is a bottleneck, but we can just shove this into the recursion step by telling the left and right to not just return the δ_1, δ_2, but also the points sorted in the y-coordinate. Then at the end of the merge, we also merge the lists L, R so that $S = L \cup R$ is sorted on y, which also takes $O(N)$ and doesn't add extra runtime on the merge step. This reduces the runtime to $O(N \log N)$.

Algorithm 3.5 (Closest Pair of Points)

The next problem simply takes a series of points and calculates the closest pair of points. This can be done trivially in $O(N^2)$ by taking all combinations, but with clever divide and conquer, we can reduce this down.

Algorithm 5 Closest Pair of Points

Require: N points $\{(x_i, y_i)\}$.

```
function ClosestPair(P)
    Sort points by x-coordinate.                          ▷Bottleneck of O(N log N)
    if len(P) = 2 then
        return d(p1, p2), sorted P by y-coord
    end if
    δ1, L ← ClosestPair(PL)
    δ2, R ← ClosestPair(PR)
    δ = min{δ1, δ2}
    min ← δ
    for l ∈ L s.t. distance to boundary ≤ δ do            ▷O(N) iterations
        Wl ← δ-window of points in R around l.            ▷Can be done in O(1) using sliding window.
        for r ∈ Wl do                                     ▷This is bounded by O(10)
            if d(p, l) < δ then
                min ← d(p, l)
            end if
        end for
    end for
    merge L and R into sorted S                           ▷O(N)
    return min, S
end function
```

3.5 Multiplication

3.5.1 Karatsuba Algorithm

3.5.2 Strassen Algorithm

We can solve matrix multiplication of two $N \times N$ matrices in a slightly more clever way than $O(N^3)$. Note that we can take the 2×2 block form of matrices A, B and multiply them to get $C = AB$, where

$$C_{11} = A_{11}B_{11} \cdot A_{12}B_{21} \tag{31}$$
$$C_{12} = A_{11}B_{12} \cdot A_{12}B_{22} \tag{32}$$
$$C_{21} = A_{21}B_{11} \cdot A_{22}B_{21} \tag{33}$$
$$C_{22} = A_{21}B_{12} \cdot A_{22}B_{22} \tag{34}$$

This requires us to compute a total of 8 $N/2 \times N/2$ multiplications and 4 additions, each of which is $O(N^2)$. Therefore, our recurrence relation is

$$T(N) = 8T(N/2) + O(N^2) \tag{35}$$

Using the master theorem, we find that $a = 8 > 2^2 = b^c$, so our runtime is $O(N^{\log_b a}) = O(N^3)$, which brings us right back to where we started. The problem with this is that $a = 8$, which is large. If we could get $a = 7$, then this would be an improvement. We want to reduce this number of multiplications, and we can do this using the Strassen algorithm, which uses the following values.

$$\begin{aligned}
P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_2 &= (a_{21} + a_{22})b_{11} \\
P_3 &= a_{11}(b_{12} - b_{22}) \\
P_4 &= a_{22}(b_{21} - b_{11} \\
P_5 &= (a_{11} + a_{12})b_{22} \\
P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
P_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

Then, we claim that we the entries of C are

$$\begin{aligned}
c_{11} &= P_1 + P_4 - P_5 + P_7 \\
c_{12} &= P_3 + P_5 \\
c_{21} &= P_2 + P_4 \\
c_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

So we have reduced 8, 4 mult/add to 7, 18 mult/add. Addition is cheap and the number of additions is bounded, so now we have decreased a to 7.[2] We can then solve the new recurrence relation

$$T(N) = 7T(N/2) + O(N^2) \implies O(N^{\log_2 7}) \approx O(N^{2.81}) \tag{36}$$

3.6 Polynomial Multiplication with Fast Fourier Transform

Given as inputs 2 degree N polynomials,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} \tag{37}$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1} \tag{38}$$

we want to multiply them to $C(x) = A(x)B(x)$ defined

$$C(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2} \tag{39}$$

Clearly, we must multiply every coefficient in A with B, which takes $O(N^2)$ time. It is also called the convolution operation.

Convolution of $(a_0, a_1, \ldots, a_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1})$

$$\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
c_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
&\vdots \\
c_{2n-2} &= a_{n-1} b_{n-1}
\end{aligned}$$

We can actually compute this convolution of two vectors in $O(N \log N)$ time using the FFT algorithm. Let's ease into this idea.

[2] We can reduce it even further, down to 2.37. Whether $O(N^2)$ is possible is an open problem.

Lemma 3.1 (Evauating a Polynomial at x)

If we are given an x and want to evaluate $A(x)$, we can just incrementally evaluate the terms up each degree in $O(N)$ time.

Algorithm 6 Evaluate polynomial $A(x)$ at $x = p$

```
S ← a_0
R ← x
for i = 1, 2, ..., n − 1 do
    S ← S + a_i · R
    R ← R · x
end for
```

Great, we can make some progress, but what does this have to do with finding the actual polynomial? Recall that from the fundamental theorem of algebra, a set of $n+1$ points will uniquely determine a nth degree polynomial. This at first glance doesn't help, since evaluating all $n+1$ points is $O(n^2)$, and even if we did, this doesn't really tell us how to reconstruct the polynomial in some fast time (e.g. matrix inversion won't work). But note that if we have a 1st degree polynomial, then evaluating it at ± 1 will retrieve the whole polynomial back

$$f(x) = a_0 + a_1 x \implies \begin{cases} f(+1) &= a_0 + a_1 \\ f(-1) &= a_0 - a_1 \end{cases} \implies \begin{cases} a_0 & = \frac{1}{2}\big(f(+1) + f(-1)\big) \\ a_1 = \frac{1}{2}\big(f(+1) - f(-1)\big) \end{cases} \tag{40}$$

We can think of this as sort of our base case. For Nth degree polynomials, we can divide it into a even and odd powers part.

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \dots && (41)\\ &= (a_0 + a_2 x^2 + a_4 x^4) + x(a_1 + a_3 x^2 + a_5 x^4 + \dots) && (42)\\ &= A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) && (43) \end{aligned}$$

where each of the splits have degree $N/2$. Then we want to evaluate the even and odd parts.

Let's jump ahead and focus on the problem of evaluating $A(x)$ of degree N at the Nth roots of unity.

Example 3.1 ()

For $N = 4$, we evaluate at $\pm 1, \pm i$.

$$A(x) = (a_0 + a_2 x^2) + x(a_1 + a_3 x^2) \tag{44}$$

which gives us

$$\begin{aligned} A(+1) &= A_e(+1) + A_o(+1) && (45)\\ A(-1) &= A_e(+1) - A_o(1) && (46)\\ A(+i) &= A_e(-1) + iA_o(-1) && (47)\\ A(-i) &= A_e(-1) - iA_o(-1) && (48) \end{aligned}$$

Note that even though we had $\pm i$ evaluated on A, they were all squared in each split so evaluating the 4th of unity, which we denote $U(4)$, has been reduced to finding $U(2)$ for each of the left and right polynomials.

Therefore, to evaluate A of degree N at $U(N)$, it suffices to evaluate A_e, A_o each at $U(N/2)$, followed by combing them using addition and multiplication, which turns out to be $O(N)$.

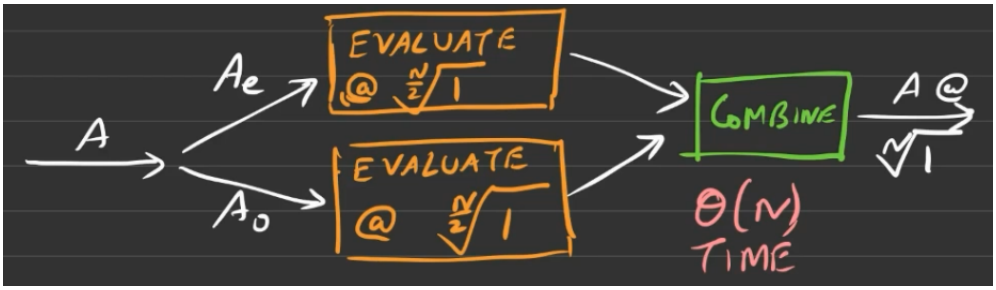


Figure 6

Algorithm 3.6 (Evaluate Nth Degree Polynomial at Nth Roots of Unity)

Require:

function FUNC(x)
end function

Therefore, we have divided the problem of evaluating over $U(N)$ to be

$$T(N) = 2T(N/2) + O(N) \implies T(N) = O(N \log N) \tag{49}$$

So we have shown that in general, evaluating N points of a polynomial takes $O(N^2)$ time, but if you're clever about what points to evaluate, you can get $O(N \log N)$.

Now going back to the original problem, we can evaluate $A(u), B(u)$ for $u \in U(N)$, and then multiply them to get $C(u)$. Great. Now to reconstruct the polynomial using the roots of unity, it turns out that there is a method in $O(N \log N)$ time as well.

4 Greedy Algorithms

Greedy algorithms are easy to implement and easy to conceptualize, but they are often not correct. So the only interest in them is proving if they are correct. We have done greedy algorithms like Dijkstra and finding MSTs. Every step, greedy algorithms make an irrevocable decision of what to do and cannot undo it. That is, we cannot backtrack. In fact, some algorithms like Kruskal's algorithm for MSTs is precisely a greedy algorithm.

Example 4.1 (Interval Scheduling)

Let there be 1 classroom and n courses, each with a start and end time (s_i, e_i). We want to find the largest subset of courses that can be scheduled in a day such that they don't overlap. There are two things we want to do. How do we code this up? How do we show that a greedy algorithm will give a correct answer? Two greedy approaches are as follows.

1. Sort them by the start times, and add them as long as they do not overlap with what you have. This will not work.
2. Sort them by the time interval lengths and keep adding them as long as they do not overlap with what you have. This will not work either.

It turns out that if we sort based on finish time, this will work. So why is this correct? Assume that the first k decisions of this greedy algorithm are correct. Then, to find the next interval that we will include, the optimal algorithm must choose one that does not overlap with what we have. This is good since we do the same. The second part is that the next added interval must have the

smallest end time t from all viable left intervals. Assume that it was not and that the optimal next interval had end time $s > t$. Then, the next interval must start after s, but this also starts after t, so we are sacrificing unnecessary extra space. An optimal solution with the endpoint at s can also be constructed with the same interval ending at t without anything to lose.

Algorithm 7 Find Max Classes to Fit into 1 Room

```
Require: classes C = {(s_i, e_i)}_i of type List[tuple(int, int)]
  function SCHEDULE(C)
      res ← []
      sort C by increasing end time
      for s, e ∈ C do
          if s < res[-1][-1] then                ▷If start overlaps with previous end time
              continue
          end if
          add (s, e) to res                      ▷Otherwise add class to final schedule
      end for
      return res
  end function
```

To get the run time, the sorting takes $O(n \log n)$ time and iterating is $O(n)$, so a total of $O(n \log n)$ time.

Example 4.2 (Classroom Scheduling)

A slightly more practical example is that you have n classes and you want to minimize the number of classrooms m. For this, you can really use any order. The general idea is

1. Consider intervals in increasing start time s_i.
2. If it can be scheduled in an existing classroom, then schedule it there.
3. If not, then open a new classroom.

Even then, we can conduct a line search through all the time periods, incrementing the current number of classes if an incoming time is a start time and decrementing if it is an end time. We report the max Δ over all times.

This algorithm is correct since by definition, Δ is the lower bound for the number of classrooms needed and the greedy algorithm attains it. The greedy algorithm reports Δ since if it reported $\Delta + 1$ classroom, then at some time t it opened the $\Delta + 1$th classroom. But this is impossible since this must mean that there are $\Delta + 1$ classrooms concurrently in use, contradicting our assumption that Δ is the optimal solution.

Algorithm 8 Minimizing Number of Rooms With n Classes (Leetcode 253, Meeting Rooms II)

```
Require: classes C = {(s_i, e_i)}_i of type List[tuple(int, int)]
  function MeetingRooms(C)
      start ← all start times sorted increasing
      end ← all end times sorted increasing
      res ← 0                                          ▷Our max classes count
      count ← 0                                        ▷Our curr. classes count
      s, e ← 0                                         ▷pointers for start, end
      while s < len(start) do                          ▷Don't need to check for end since
          if start[s] == end[e] then                   ▷start will always finish faster.
              s += 1, e += 1                           ▷One started and one ended, so don't change
          else if start[s] > end[e] then
              e += 1, count -= 1                       ▷One ended, so decrement count
          else if start[s] < end[e] then
              s += 1, count += 1                       ▷One started, so increment count
              res = max(res, count)                    ▷Update res if we got past it
          end if
      end while
      return res
  end function
```

The runtime is $O(n \log n)$, which is to sort. However, you don't even need to sort since you can go through all intervals in any order and place it in an existing classroom or open a new classroom.

The next one isn't as trivial, but requires us to devise a custom sorting method by comparing two sequences with a swap difference.

Example 4.3 (Quiz Problem)

Suppose you have n questions with question i having reward r_i, but the probability that you get it correct is p_i. You keep collecting the reward until you get one question wrong, and the game terminates. What order would you answer the questions in?
Suppose we have $q = (10, 5)$ and $r = (0.1, 0.2)$.

1. If you choose the first and then second, then the expected return is

$$10 \cdot 0.1 + 5 \cdot 0.1 \cdot 0.2 = 1.1 \tag{50}$$

2. If you choose the second and then first, then the expected return is

$$5 \cdot 0.2 + 10 \cdot 0.1 \cdot 0.2 = 1.2 \tag{51}$$

Clearly, finding all possibilities is too computationally expensive since it increases exponentially w.r.t. n. Intuitively, if I have a question with a high reward, I want to answer it first, but if I have a low probability of getting it correctly, then I can't answer future questions if I get it wrong. So I have to balance these two forces. So we want to sort the score of each question with some function $f(r_i, p_i)$, which is increasing in both r_i and p_i. It is not $r_i p_i$, but this is a good start.
Rather, we can take a different approach. Assume that the tuples were sorted in some order, but this was not the optimal order. Then this indicates that we can swap to adjacent elements and it will get a better order. This is really like bubble sort, and now we need to find out the conditions on which it can be improved.

1. If we answer $q_1 \to q_2 \to q_3 \ldots$, our expected reward is

$$\mathbb{E}[R] = r_1 p_1 + r_2 p_1 p_2 + r_3 p_1 p_2 p_3 \tag{52}$$

2. If we swap q_2 and q_3 and answer $q_1 \to q_3 \to q_2 \ldots$, then our expected reward is

$$\mathbb{E}[R] = r_1 p_1 + r_3 p_1 p_3 + r_2 p_1 p_2 p_3 \tag{53}$$

Note that the swap does not affect higher order terms. Doing some algebra, the swap is better iff

$$r_2 p_2 (1 - p_3) < r_3 p_3 (1 - p_2) \implies \frac{r_2 p_2}{1 - p_2} < \frac{r_3 p_3}{1 - p_3} \tag{54}$$

where the final swap saves computational time since we can compute for a single element rather than comparing pairwise elements. So, we sort it (in descending order!) according to these values to maximize this value. Note that the numerator measures your expected reward, while the denominator measures the probability of you screwing up.

Not all greedy problems admit to scoring functions however. This was just one example.

Example 4.4 (Minimizing Max Lateness)

Suppose there are n jobs that all come in at once at time 0, with job j having length p_j and deadline d_j. We want to schedule the order of jobs on one machine that can handle one job at a time, where you must minimize the maximum lateness, where the lateness of job i is $\max\{f_i - d_i, 0\} = [f_i - d_i]^+$. For example, given jobs $(p, d) = \{(10, 9), (8, 11)\}$, we can run it in two ways.

1. Running 1 then 2. The first job finishes at 10 and the second at 18. The lateness is $10 - 9 = 1$ and $18 - 11 = 7$ respectively, so the maximum lateness is 7.
2. Running 2 then 1. The first job finishes at 8 and the second at 18. The lateness is 0 and $18 - 9 = 9$, so the maximum lateness is 9.

Therefore, $1 \to 2$ beats $2 \to 1$. Clearly, brute forcing this over $N!$ jobs orderings is unfeasible, but there is a greedy approach to this. We can

1. schedule jobs in increasing order of deadlines.
2. try to exchange by taking a sequence, swapping it, and computing the score like we did for previous examples.

Both lead to the same score/principle that we should schedule jobs in increasing order of deadlines. Let's prove this by taking jobs i and j, where $d_j \leq d_i$. Then, if we schedule $i \to j$, then we have

$$l_i = [t + p_i - d_i]^+ \tag{55}$$
$$l_j = [t + p_i + p_j - d_j]^+ \tag{56}$$

with l_j being the greatest late time. If we did $j \to i$, then we have

$$\hat{l}_j = [t + p_j - d_j]^+ \tag{57}$$
$$\hat{l}_i = [t + p_i + p_j - d_i]^+ \tag{58}$$

with $\hat{l}_i$ being the greatest late time. But this is a good sacrifice since we can see that both

$$l_j > \hat{l}_j \text{ and } l_j > \hat{l}_i \tag{59}$$

meaning that if we swap, then we will decrease the lateness of j at the smaller cost of increasing that of i. Therefore, $j \to i$ is better. Note that even though the end times of the second job will always be the same between the two choices, starting with j will give a later deadline time. So this means that in the optimal solution, we have to place them in this order of deadlines since we can always improve them by swapping.

Example 4.5 (Gift Wrapping a Convex Hull)

Given n points in $\mathbb{R}^2$, we want to find the subset of points on the boundary of its convex hull. We can intuit this by taking the bottom most point and imagine "wrapping" a string around the entire region.[a] Here's a basic fact. If we have a point $p_1 = (x_1, y_1)$ and we're looking at $p_2 = (x_2, y_2)$, and then $p_3 = (x_3, y_3)$ shows up, then we can look at the cross product

$$P = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \tag{60}$$

and if $P > 0$, then p_3 is counterclockwise from p_2. The algorithm is as such.

1. You choose the bottommost point and label it p_1.
2. You construct a vector pointing to the right and start turning it counterclockwise until it hits the first point. This can be done by iterating through all of the n points and computing the angle.
$$\tan\theta = \frac{y_i - y_1}{x_i - x_1} \tag{61}$$
and the points with the smallest such tangent value will be the desired point, which will be $\Theta(n)$. We set this point to be p_2 and claim that p_1, p_2 is in the convex hull. This is true since all point must lie above the line p_1p_2. Now you do the same thing with p_2 starting with a vector facing to the right and rotating it left until it hits the first peg. Once you get to the topmost point, you can start the vector from the right and rotate it counterclockwise, but this is an implementation detail.
If your convex hull has L vertices, then this algorithm will have $O(nL)$, which will be $O(n^2)$ worst case but if we sample uniformly, then we have $L \approx \log n$ and on average it is $O(n \log n)$.

A lot of geometric algorithms is output sensitive.

Example 4.6 (Convex Hull with Graham Scan)

In fact, we can take a different approach and get $O(n \log n)$ time.

1. You still start with p_1 with the minimum y-coordinate and we sort in increasing $\theta_i = \arctan \frac{y_i - y_1}{x_i - x_1}$.
2. Then it will walk from p_i to p_{i+1} and check at p_{i+1} if it is on the convex hull or not.
   (a) If you turned counterclockwise to go to p_{i+1} then it's good and you're on the convex hull.
   (b) If you turned clockwise it is bad, and p_i is not on the convex hull, so remove p_i. However, p_{i-1} could also be clockwise as well, so you must backtrack and keep on removing points until you find a point p_j such that the previous turn is counterclockwise.

   Note that this is implemented as a stack.

This can be calculated in $O(1)$ by the previous cross product formula.

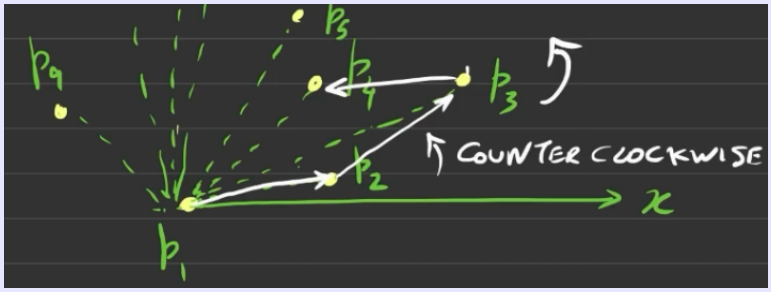


Figure 7: Overview of the steps mentioned above.

[a] This is called the gift wrapping algorithm.

Algorithm 9 Finding Convex Hull with Graham Scan

Require: $p_1, \dots, p_n \in \mathbb{R}^2$
  function GRAHAM(**p**)
    s $\leftarrow$ stack()
    push p_1, p_2 onto s
    for $i = 3, 4, \dots, n$ **do**
      push p_i onto stack
      while top 3 on stack are clockwise **do**
        pop second from top point
      end while
    end for
    return s
  end function

Note that once we have 3 points, the first three in the stack will always be counterclockwise, so we will never need to check if there are enough points in the stack. Even in the beginning, the next point (third) added is guaranteed to be counterclockwise because of ordering. Note that even though we might do a bunch of pushes and pops, the maximum number of pushes we can do is n and pops is n, so this is $O(n)$. In fact, the bottleneck is the sorting, which is $O(n \log n)$, so the total runtime is $O(n \log n)$.

5 Graphs

A huge portion of problems can be solved by representing as a *graph* data structure. In here, we will explore various problems that can be solved through *graph algorithms*.

5.1 Representations and Properties

All graphs consist of a set of vertices/nodes V and edges E. This tuple is what makes up a graph. We denote $|V| = n, |E| = m$.

Definition 5.1 (Undirected Graphs)

An **undirected graph** $G(V, E)$ is a tuple, where $V = \{v_1, \dots, v_n\}$ is the vertex set and $E = \{\{v_i, v_j\}\}$ is the edge set (note that it is a set of sets!).

1. The **degree** d_v of a vertex v is the number of edges incident to it.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in E. It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.
4. A graph is **connected** if for every pair of vertices $e_i, e_j \in E$, there is a path from e_i to e_j.
5. A **connected component** is a maximal subset of connected vertices.

Definition 5.2 (Directed Graph)

A **directed graph** $G(V, E)$ is a tuple, where $V = \{v_1, \dots, v_n\}$ is the vertex set and $E = \{(v_i, v_j)\}$ is the edge set (note that it is a set of tuples, so $(i, j) \neq (j, i)$).

1. The **in/out degree** $d_{v,i}, d_{v,o}$ of a vertex v is the number of edges going in to or out from v.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in E. It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.

4. A directed graph is **strongly connected** if for every pair of vertices $e_i, e_j \in E$, there is a path from e_i to e_j.[a]
5. A **strongly connected component** is a maximal subset of connected vertices.

In fact, from these definitions alone, we can solve an ancient puzzle called *the Bridges of Konigsberg*. Euler, in trying to solve this problem, had invented graph theory.

Example 5.1 (Bridges of Konigsberg)

Is there a way to walk that crosses each bridge *exactly* once?

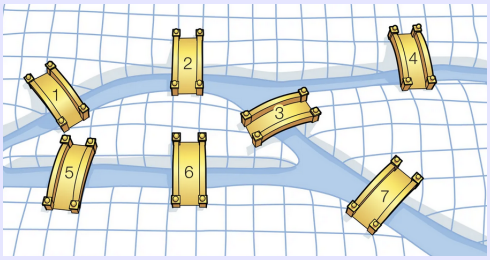


Figure 8: Bridges of Konigsberg

It can be decomposed into this undirected graph.

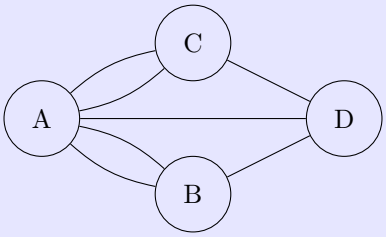


Figure 9: Graph representation.

Euler's observation is that except for start and end points, a talk leaves any vertex by different edge that the incoming edge. Therefore, the degree (number of edges incident on it) must have an even number, so all but 2 vertices must have an even degree. Since every vertex has an odd degree, there is no way of doing it.

In addition to the *adjacency list* representation, another way in which we represent a directed graph is through *adjacency matrices*.

Definition 5.3 (Adjacency Matrix)

In a finite directed graph (V, E), we can construct a bijection from V to the natural numbers and so we label each element in V with $i \in \mathbb{N}$. Then, we can construct a matrix A such that

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases} \tag{62}$$

While the adjacency matrix does have its advantages and has a cleaner form, usually in sparse graphs this is memory inefficient due to there being an overwhelming number of 0s.

[a] Obviously, a connected undirected graph is also strongly connected.

Definition 5.4 (Trees)

An undirected graph $G(V, E)$ is a **tree** if
1. G is connected.
2. G has no cycles.[a]

Removing the first requirement gives us the definition of a **forest**, which is a collection of trees. Conversely, if $G(V, E)$ is connected and $|E| = n - 1$, then G is a tree.

Theorem 5.1 (Properties of Trees)

If $G(V, E)$ is a tree, then
1. There exists a $v \in V$ s.t. $d_v = 1$, called a **leaf node**.
2. $|E| = |V| - 1 = n - 1$.

Proof.

The outlines are quite intuitive.
1. There must be some leaf node since if there wasn't, then we would have a cycle. We can use proof by contradiction.
2. We can use proof by induction. We start off with one vertex and to construct a tree, we must add one edge and one vertex at every step, keeping this invariant.

5.2 Exploration

Given two $v, s \in V$ either directed or undirected, how can we find the shortest path from v to s? We can do with either with DFS or BFS.

Definition 5.5 (DFS)

The recursive algorithm is

```
visited = set()
def dfs(start):
  if start not in visited:
    visited.add(start)
    # do something
    neighbors = ...
    for neighbor in neighbors:
      dfs(neighbor)
```

The iterative algorithm uses a stack, which mirrors the function call stack.

```
visited = set()

def dfs(start):
  toExplore = []
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.pop()
    # Do something
    neighbors = ...
```

[a] This makes sense, since to get back to a previous vertex you must backtrack.

```
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        toExplore.append(neighbor)
```

Theorem 5.2 (Runtime of DFS)

The runtime of DFS is $O(n+m)$.

Proof.

Definition 5.6 (BFS)

The iterative version is shown.[a]

```
visited = set()
def bfs(start):
  toExplore = collections.deque()
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.popleft()
    # Do something
    neighbors = ...
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        toExplore.append(neighbor)
```

Theorem 5.3 (Runtime of BFS)

The runtime of BFS is $O(n+m)$.

Proof.

To get the running time, we know that each vertex is popped only once from the queue, giving us $O(n)$. For each pop, we are exploring all the neighbors of V.

$$O\left(\sum_{v\in V} |\text{neighbors of } v| + 1\right) = O\left(\sum_{v\in V} d_v + 1\right) \tag{63}$$

$$= O(2|E| + |V|) = O(m+n) \tag{64}$$

which is linear in input size!

The more straightforward application is in reachability.

[a] The recursive version of BFS is very nontrivial.

Example 5.2 (Reachability)

Given a directed graph and a node v, find all nodes that are reachable from v.

Exercise 5.1 ()

Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ s.t. G remains connected after removing v.

Proof.

Let u be such a leaf node of T, and let G' be the subgraph of G resulting by removing u and its incident edges from G. For sake of contradiction,[a] suppose G' has more than one connected component. Let C be a connected component in G' that does not contain s, the root of the BFS tree T. Since G was connected before the removal of u, it must be that every path from s to any vertex v in C includes u (otherwise there would remain a path from s to v in G' and s would be in C). Then u is the only vertex not in S with edges to vertices in S, so all vertices in C must be "visited" during BFS only after visiting u. Furthermore, the vertices of S must be in the subtree of T rooted at u. But u is a leaf, which is a contradiction.

Exercise 5.2 ()

Two parts.

1. Give an example of a strongly connected directed graph $G = (V, E)$ s.t. that every $v \in V$, removing v from G gives a directed graph that is not strongly connected.
2. In an undirected graph with exactly two connected components, it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

Proof.

Listed.

1. A graph whose edges form a cycle, having at least three nodes.
2. Two strongly connected components with no edges between them.

5.3 Directed Acyclic Graphs and Topological Sorting

Definition 5.7 (Directed Acyclic Graph)

A DAG is a directed graph that has no cycles. Note that a DAG must have a node that has no in-edges.

To determine if a graph is a DAG, then we can brute force it by taking a node $s \in V$, running DFS/BFS, and if a neighbor is already in visited, return False. Then go through this for all starting nodes $s \in V$. This again has quadratic runtime. Can we do better? This introduces us to topological sorting.

It may be helpful to take a graph $G(V, E)$ and induce some partial order on the set of nodes V based off of E. It turns out that we can do this for a specific type of graph.

[a] We provide an alternative direct proof as follows: Since G is given to be connected, T contains all vertices of G. Let T' be the BFS tree minus u and its single incident edge connecting it to its parent in T. Since u is a leaf, T' remains a connected tree with all other vertices of G. The edges of T' exist in G', so G' is connected.

Definition 5.8 (Topological Sort)

Given a directed acyclic graph (DAG), a linear ordering of vertices such that for every directed edge $u - v$, vertex u comes before v in the ordering is called a **topological sort**. It satisfies the facts:

1. The first vertex must have an in-degree of 0.
2. A topological sort is not unique.

Example 5.3 (Simple Topological Sort)

The graph below

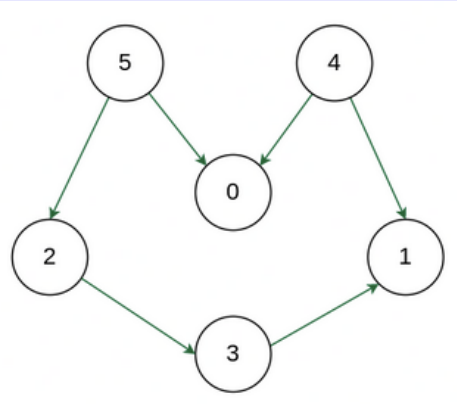


Figure 10

can have the two (not exhaustive) topological sortings.

1. $[5, 4, 2, 3, 1, 0]$
2. $[4, 5, 2, 3, 1, 0]$

To determine if a graph is a DAG, note the following theorem.

Theorem 5.4 (Topological Order and DAGs)

G has a topological order if and only if it is a DAG.

Proof.

To prove that a DAG has a topological order, we use induction. Pick a v such that its indegree is 0. Then, delete v, and therefore $G \setminus v$ is also a DAG with a topological order since we are only deleting edges. We keep going.

Therefore, if we can successfully topologically sort, we know it is a DAG. So we can kill two birds with one stone. Let's see how this is implemented. We can do it iteratively and recursively (the proof above should hint that this can be recursive).

Algorithm 5.1 (Iterative Topological Sort, Determine If Graph is DAG)

The general idea is that we first find the node that as 0 in-degree. From here, we can do DFS, and when we run out of neighbors to explore, then we push this into a queue. This is essentially a post-order traversal, where at the end are going to be the end nodes with no more neighbors, and the node we started from will be added last. Then we loop through and do this for all start nodes. We

first need a slightly modified form of DFS.

```
Require: Nodes V, adjacency list E
visited ← set()
res ← stack()
is_acyclic ← True
function DFS(v ∈ V)
    if v ≠ visited then
        add v to visited
        N_v ← neighbors of v
        for n ∈ N_v do
            if n ∈ visited then
                is_acyclic ← False
            end if
            DFS(n)
        end for
        push v onto res
    end if
end function

function TopologicalSort(V, E)
    for v ∈ V do
        DFS(v)
    end for
    if ! is_acyclic then
        return False
    end if
    return reversed of res
end function
```

Note that this runtime is $O(|V|+|E|)$ since we are just running DFS with a constant amount of work on top of each call.

Algorithm 5.2 (Recursive Topological Sort)

We want to see that while G is nonempty, we want to find the $v \in V$ such that it has indegree 0. Then place v next in order, and then delete v and all edges out of v. The problem is finding which vertex has indegree 0 (if we brute force it by looking through all remaining nodes and edges, you have quadratic runtime). To do this fast, the idea is

1. initially scan over all edges to store the indegrees of every node to a list `indeg`.
2. store all nodes with indegree 0 to a queue.
3. Run through the queue, and during each loop, when we remove a node, we look at all of its out-nodes s and decrement `indeg[s]`. If `indeg[s] = 0`, then add it to the queue.

```
Require: Nodes V, Edges E
q ← queue()
indeg ← list()
visited ← 0
function RECUR(x)
    initialize the indeg and q
    while q is nonempty do
        v ← pop(q)
        visited += 1
        for each w ∈ E[v] do
            indeg[w] -= 1
            if indeg[w] = 0 then
                push w into q
            end if
        end for
    end while
    if visited != |V| then
        return False
    end if
    return True
end function
```

Notice that the inner for loop is $O(d(v)+1)$, while we run over all n. So really, we are doing $O(n(d(v)+1)) = O(m+n)$, where the plus n comes from the constant work we are doing for each node. Note that if we have a non-DAG, then at some point the queue will be empty but we haven't processed all the vertices, at which point we can declare failure.

To end this, we can make a general statement about all directed graphs.

Theorem 5.5 ()

Every directed graph is a DAG of strongly connected components (SCC).

This gives us a way to represent a directed graph with a collection of DAGs.[3] An extension of topological sort is making a *BFS tree*, which partitions a graph into layers that represent the number of steps required to go from a source vertex to a node.

Algorithm 5.3 (BFS Tree)

To construct a BFS tree, we just need to slightly modify the original BFS code.

[3] In fact, this Kosaraju's algorithm, can be done in linear time, though it is highly nontrivial.

```
Require: Nodes V, adjacency list E
visited = set()
layers = {v : 0 | v ∈ V}
function BFS(start)
    layer ← 0
    toExplore ← queue()
    add (start, layer) to toExplore
    add start to visited
    while toExplore do
        curr, layer = pop from toExplore
        layers[curr] = layer
        for n ∈ neighbors of curr do
            if n ∉ visited then
                add n to visited
                add (n, layer + 1) to visited
            end if
        end for
    end while
end function
```

This is simply BFS with constant extra work so it is $O(n+m)$.

So, a BFS tree is really just another way to topologically sort. Note the following properties.

1. In a directed graph, no nodes can jump from layer i to layers $j > i+1$, since if it could, then it would be in layer $i+1$. However, nodes can jump from layer j back to any layer $i < j$, even skipping layers.
2. In a directed graph, going forward is the same as going back, so nodes can jump at most one layer forwards or backwards.

Exercise 5.3 (DPV 3.16)

Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph $G = (V, E)$ has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Note this is a directed acyclic graph (DAG). In order for a student to take a course w with prerequisite v, they must take v in an earlier semester. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum, under the assumption any number of courses can be taken in one semester. The running time of your algorithm should be $O(n+m)$, where n and m are the numbers of vertices and edges in G, respectively.

Proof.

For each vertex, we want to find the longest path leading to it: if there is a path leading to a node, then all of the courses in the path should be taken sequentially. Perform a topological sort of G's nodes and label them 1 through n. Then, we go through the nodes in the resulting topological order. For each vertex, we assign the minimum number of semesters required to take it: if there are no prerequisites, we assign 1, and if there are prerequisites, we assign 1 plus the maximum value assigned to its prerequisite nodes.
Implementation details: If the input is in adjacency list format, then we do not have access to the *incoming* edges to a node (its prerequisites). By exploring the entire graph with BFS calls, we can compute the list of incoming edges to every vertex in $O(n+m)$ time. These details are not required.

If the input is in adjacency matrix format, for each node it takes $O(n)$ time to find its incoming edges, so the total running time is $O(n^2)$.

Exercise 5.4 (DPV 3.22)

Give an efficient algorithm that takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

Proof.

We first build the DAG representation of the SCCs of G in $O(m + n)$ time, as described in lecture. This graph is a DAG where each SCC of G is represented by a single node. We return true if this DAG has exactly one node with no incoming edges (i.e., exactly one source node), and return false otherwise.

Correctness. Let u be a vertex in an SCC that is a source node in the DAG representation. If there is a path in G from a vertex v not in the SCC of u to u, then there must be an edge (corresponding to an edge in this path) into the SCC of u in the DAG, which contradicts that the node has no incoming edges. Thus, if there are two source SCCs in the DAG, no vertex of G can reach all vertices; in particular, no vertex can reach the vertices in both of the source SCCs. Thus we correctly return false if there are multiple source SCCs.

On the other hand, if there is a single source SCC in the DAG, we claim that every vertex in the SCC can reach every other vertex in G, in which case our algorithm correctly returns true. Every other SCC in the DAG is not a source, so it has an incoming edge.[a] Consider starting at an SCC in the DAG, picking an incoming edge to the SCC, and then repeating this process from the SCC from which the edge was leaving. This process stops when we reach an SCC without incoming edges. In this case there is exactly one source SCC, so this process will arrive at the single source SCC when starting from any SCC in the DAG. This implies there is a path from the unique source SCC to every other SCC in the DAG, and thus every vertex of the source SCC can reach all vertices in all SCCs of G; that is, all vertices of G.

Exercise 5.5 (DPV 3.19)

You are given a binary tree $T = (V, E)$ with designated root node with $n = |V|$ vertices which are the integers from 1 to n. In addition, there is an array $x[1..n]$ of values where $x[u]$ is the value of vertex $u \in V$. Define a new array $z[1..n]$ where, for each $u \in V$,

$$z[u] = \max\{x[v] \mid v \text{ is a descendant of } u\} \tag{65}$$

That is, $z[u]$ is the maximum x-value of all vertices in the subtree of T rooted at u. Note that, by definition, any node is a descendant of itself. Describe an $O(n)$-time algorithm which calculates the entire z-array.

Proof.

We propose the following recursive algorithm performs a *postorder* traversal of the tree and populates the values of the z-array in the process:

```
computeZ(u):
  maxVal = x[u]
  if u.left is not null:
    computeZ(u.left) # compute z for all descendants of u.left
```

[a] The following argument can be made formal with induction.

```
    maxVal = max(z[u.left], maxVal)
  if u.right is not null:
    computeZ(u.right) # computes z for all descendants of u.right
    maxVal = max(z[u.right], maxVal)
  z[u] = maxVal
```

We initially call `computeZ` on the root node of T. The algorithm takes $O(1)$ time per node, which is $O(n)$ overall[a]

Exercise 5.6 ()

Your data center needs to run a number of jobs (compute requests) numbered $1, 2, \ldots, n$. These are specified in a list of m tuples where (i, k) means that job i must be completed before job k can run. A given job may have multiple dependencies; for example, you might have constraints $(1, 4), (2, 4), (3, 4)$ that all of jobs $1, 2,$ and 3 must be completed before 4 can run.

1. Describe an $O(n + m)$ runtime algorithm that determines whether it is possible to execute all of the jobs, and if so, determines a valid order in which to execute the jobs one at a time. *Hint. How to relate SCCs to cycles?*
2. Suppose you have n identical servers (so that if there were no constraints you could simply run each job on a separate server). Suppose every job has the same runtime R. Describe an $O(n + m)$ runtime algorithm to compute the total runtime that will be necessary to run all of the jobs in a valid order.

Proof.

For this question we define a graph $G = (V, E)$ where there is a vertex for every job $1, \ldots, n$ and an edge from i to k for every listed dependency (where k depends on i).

1. We note that a sequence of jobs can be executed if and only if there is no circular dependency. In the language of graphs, this requires that G be free of cycles. To this end, it suffices to propose an algorithm that runs in $\mathcal{O}(m+n)$. We will use Kosaraju's SCC algorithm. We prove the following claim:

   $$\text{Any SCC with } \geqslant 2 \text{ vertices contain a cycle.}$$

   To see this, consider any distinct u, v in this SCC. Let $p_{u,v}$ and $p_{v,u}$ be the paths from u to v and backwards, respectively. Now concatenate the paths and get a walk that starts from u and ends at u. Note that each vertex appears at most once in $p_{u,v}$ and in $p_{v,u}$, so in the combined walk, it appears at most twice. Consider the set of vertices that are revisited in this walk — clearly, u is one of them and is the latest one to be revisited. There must exist a vertex w that was the *first* to be revisited. Then, the section of the walk between the two visits of w form a cycle by definition: it starts from w, ends at w, and does not repeat any other vertices. This proves the claim. And to go back to our problem, the following are equivalent:
   (a) jobs can be executed
   (b) no cycles in G
   (c) each SCC obtained from Kosaraju is a singleton
2. We first run the algorithm from part (a) to check if it is possible and to find a valid order of the jobs if so. Then define an array L of length n. We will compute $L[k]$ as the length of the longest dependency chain prior to k. Loop over the k jobs in topological order. For each, compute $L[k] = 0$ if k has no dependencies, or $L[k] = 1 + \max_{(i,k)} L[i]$ otherwise. Finally, return $\max_k L[k]$.

[a] This algorithm can also be described as a modified version of the so-called *depth-first search* (DFS) graph traversal algorithm, which is different from BFS.

5.4 Bipartite Graphs

Now we shall see a further application of BFS trees.

Definition 5.9 (Bipartite Graph)

A **bipartite graph** is an undirected graph $G(V, L)$ where we can partition $V = L \sqcup R$ such that for all $e = \{u, v\} \in E$, we have $u \in L, v \in R$.

We would like to devise some method to determine if an arbitrary graph is bipartite.

Theorem 5.6 ()

G is bipartite if and only if all cycles in G are even length.

Proof.

Proving ($\implies$) is quite easy since if we suppose G has an odd length cycle, then we start packing vertices of a cycle into L, R, but by the time we came back to the start, we are forced to pack it into the wrong partition!
The converse is quite hard to prove, and we'll take it at face value.

Now in practice, how would we determine if all cycles are even length? This is where BFS shines.

Algorithm 5.4 (Determine Bipartite On All Cycles of Even Length)

The general idea is we first run BFS on the graph starting at $s \in V$, which divides it up into layers $L_1, \dots, L_l$ representing the shortest path from s. Then for each layer $L_i \subset V$, we check if there are connections between two vertices $x, y \in L_i$. If there are connections, then this is not bipartite. If there are none, then this is bipartite since we can then color it.

```
Require: Nodes V, adjacency list E
visited = set()
layers = {v : 0 | v ∈ V}
function BFS(start)
    layer ← 0
    toExplore ← queue()
    add (start, layer) to toExplore
    add start to visited
    while toExplore do
        curr, layer = pop from toExplore
        layers[curr] = layer
        for n ∈ neighbors of curr do
            if n ∉ visited then
                add n to visited
                add (n, layer + 1) to visited
            end if
        end for
    end while
end function

function BIPARTITE(V, E)
    BFS(v) for some v ∈ V
    for (u, v) ∈ E do
        if layers[u] == layers[v] then
            return False
        end if
    end for
    return True
end function
```

Therefore, we run BFS, which is $O(n+m)$, and then to compare the edges, it is $O(m)$.

Bipartiteness is actually a special case of *coloring problems*. Given a graph with k colors, can I color it so that every neighbor has a different color than the original node? It may seem like at first glance that we can do the same method and look at the layers again, but it turns out that 3-coloring is hard. More specifically it is an NP-complete problem, which colloquially means that there isn't much of a better way than a brute-force solution. However, it turns out that according to the *4 color theorem*, any map can be colored with 4 colors.

5.5 Strongly Connected Graphs

Now how do we find out if a directed graph is strongly connected? The straightforward solution would be to take each vertex $v \in V$, run BFS to find the set of vertices reachable from v, and do this for every vertex. The total running time is $O(n(n+m))$, which is quadratic. Note that for an undirected graph this is trivial since we just run DFS/BFS once.

Theorem 5.7 ()

G is strongly connected if and only if for any $v \in V$,
1. all of V is reachable from v.
2. v is reachable from any $s \in V$

Algorithm 5.5 (Determine if Graph is Strongly Connected)

Using the theorem above, we can run BFS/DFS twice: one on the original graph and one on the reversed graph, consisting of all edges directed in the opposite direction.

Require: Nodes V, Adjacency list E
1: **function** StronglyConnected($s \in V$)
2: visited $\leftarrow$ set()
3: BFS(s)
4: **if** visited != V **then**
5: **return** False
6: **end if**
7: visited $\leftarrow$ set()
8: reverse all edges in E
9: BFS(s)
10: **if** visited != V **then**
11: **return** False
12: **end if**
13: **return** True
14: **end function**

The running time is just running BFS twice plus the time to reverse the edges, so it is $O(n+m)$.

5.6 Shortest Path

In the shortest path, you are given a *weighted* (positive integer) directed graph and your goal is to find a path from s to t with the smallest length. This is where we use Dijkstra's. What we can do to brute force is just replace a edge with length k to k edges of length 1, and we run BFS on this. However, this is not efficient since the weights can be unbounded. This is where we introduce Dijkstra. The following is how it is introduced in class.

Algorithm 5.6 (General Dijkstra)

We can keep a temporarily list π of the shortest path we have found so far, and a permanent list `dist` keeping track of all paths we know are for sure the shortest path.

Require: Graph $G(E, V)$
1: $S \leftarrow \{s\}$ ▷set of nodes that we know for sure is shortest
2: dist[s] = 0 and the rest very large numbers ▷our final list
3: $\pi[v] = w_{sv}$ for all $v \in V$ ▷initialize list with neighboring nodes from start s
4: **function** Dijkstra(s)
5: $u = \text{argmin}_{v \notin S} \pi[v]$ ▷Find node having minimum accum weight so far
6: add u to S ▷This node must be shortest so add to S
7: dist$[u] = \pi[u]$ ▷Now we can update its shortest path in dist
8: **for** $v \notin S$ **do** ▷Look at all neighbors of u and update those not in
9: $\pi[v] \leftarrow \min\{\pi[v], d[u] + w_{uv}\}$ ▷S since those in S are all guaranteed to be shortest
10: **end for**
11: **end function**

The problem is line 5 above. We don't want this linear search time since it makes the whole thing quadratic, so rather than a list, we can implement a heap, resulting in the code below.

Algorithm 5.7 (Dijkstra's Algorithm)

The general idea is to run a graph traversal like BFS but when you reach a new vertex v, you can store the accumulated time it took to get to v and store for all neighbors the accumulated time it will take to get to each of those neighbors. If it is less than what we have currently, then we have found a shorter path and we should update this.

Require: Nodes V, Edges E

```
function DIJKSTRA(s)
    dist ← list of size |V| with +∞      ▷Initialize list of big nums representing shortest distances
    dist[s] ← 0                          ▷The starting node has dist 0
    predecessors ← {v : None | v ∈ V}    ▷predecessors of each node for path tracking
    toExplore ← minheap()                ▷A priority queue of (weight, node)
    add (0, s) to toExplore              ▷You want to explore this first
    while toExplore do
        (curr_dist, curr_node) ← pop from toExplore      ▷pop from toExplore
        if curr_dist > dist[curr_node] then              ▷If this distance is greater than what
            continue                                     ▷I already have then not worth exploring
        end if
        for neighbor, weight ∈ E[curr_node] do           ▷Look at each neighbor
            new_dist ← curr_dist + weight                ▷The distance to getting to neighbor from now
            if new_dist < dist[neighbor] then            ▷If this new dist is shorter than what we have
                dist[neighbor] = new_dist                ▷Update best distance
                predecessors[neighbor] = curr_node       ▷Update its predecessor
                push (new_dist, neighbor) onto toExplore ▷Should prob explore from here
            end if
        end for
    end while
    return distances, predecessors
end function
```

You essentially push n times and pop m times, and the time per push and pop is $\log_2(n)$. Therefore, the total time to push is $n\log(n)$ and to pop is $m\log(n)$, making the total runtime $O(log(n)(n+m))$.

The first example gotten in class ignores the distances and just attempts to modify the distances in the heap itself (through the decrease key operation). This takes $2\log_2(n)$, but if we use a heap with d children, we can modify the runtime to $d\log_d(n)$. Therefore, the total runtime with tunable parameter d is

$$O\big((m+nd)\log_d(n)\big) \tag{66}$$

which can be minimized if we set $d = m/n$, so $O(m\log_{m/n} n)$, where for dense graphs m/n is large and so it can behave roughly in linear time $\Theta(m)$.

Exercise 5.7 ()

Let $G = (V, E)$ be a weighted strongly connected directed graph with positive edge weights. Let v_0 be a specific vertex. Describe an algorithm that computes the ***cost*** of the shortest walk between every pair of vertices of G, with the restriction that each of these walks must pass through v_0 (that is, for every distinct pair $u, v \in V$, among all walks from u to v that pass through v_0, compute the cost of the shortest walk). Describe the algorithm, analyze its runtime complexity, and briefly explain (not a formal proof) why it is correct. Try to give an algorithm that runs in $O(|E|\log(|V|) + |V|^2)$ time. As usual, you may use any algorithm as described in lecture without restating it or arguing for its correctness.

Proof.

The high level idea is to decompose any qualifying $u \to v$ walk into the combination of two paths $u \to v_0 \to v$, where we try to minimize the cost of both subpaths. It's easy to compute the minimum cost of $v_0 \to v$ for all v: running Dijkstra once over the graph suffices. The first half, $u \to v_0$, is the nuisance since we need to calculate this quantity for every $u \in V$. Solution? Observe that the destination node v_0 is fixed! We flip the direction, define a "reverse graph" G^{-1} where each edge carries its original weight but points in the other direction. Then, any cheapest $v_0 \to u$ path in G^{-1} would correspond to the cheapest $u \to v_0$ path in G, with matching total costs.

Exercise 5.8 ()

Let $G = (V, E)$ be a directed, weighted graph with $|V| = n$ and $|E| = O(n)$ (that is, the graph is sparse). Let s be a vertex in V. How quickly can the cost of the following shortest paths be computed under the given conditions? Just note the runtime and be prepared to explain. All of these can be solved using a single call to a shortest-path algorithm if provided the correct input graph (not necessarily the given one).

1. Compute the shortest path distance from some s to all other vertices in G under the condition that the weight of every edge is a positive integer ≤ 10.
2. Compute the shortest path distance *to* a target t from all possible source vertices s in a graph with positive edge weights.

Proof.

Listed.

1. Since all weights are integer and uniformly bounded, we convert G into an unweighted graph and apply BFS. Construct unweighted $G' = (V', E')$ as follows: for each directed edge $(u \to v \in E$, put a series of dummy nodes between u, v in G' so that the distance from u to v in G' is precisely the integer weight $w(u, v)$ of $u \to v$ in E. Now G has at most $10n$ nodes and $10n$ edges. So BFS runs in $O(|V'| + |E'|) = O(n)$.
2. Construct the reversed graph G^{-1} and run Dijkstra(G^{-1}, t). This finishes in $\mathcal{O}((m+n) \log n) = O(n \log n)$ time since G is sparse.

Exercise 5.9 ()

Let $G = (V, E)$ be an undirected, weighted graph with non-negative edge weights. Let vertices $s, t \in V$ be given. Describe an algorithm that efficiently solves the following questions.

1. Find the shortest/cheapest $s - t$ walk with an even number of edges.
2. Find the shortest/cheapest $u - v$ walk with a number of edges of form $6k + 1, k \in \mathbb{N}$.

Proof.

Listed.

1. The key observation is that as we travel on G, the number of edges we have travelled along alternates between being odd and even. Furthermore, the very same vertex may correspond to both even and odd: for example if we walked along $u \to v \to w \to u$, then initially we travelled for 0 edges, but upon return we travelled a total of 3 edges. We need a way to distinguish them. The solution? Duplicate each vertex into two categories: "odd" and "even."
   We construct a new graph $G' = (V', E')$ by duplicating every vertex $v \in V$, labeling one of them as v_{odd} and the other v_{even}. For each edge $(u, v) \in E$, add two edges $(u_{\text{odd}}, v_{\text{even}})$ and $(u_{\text{even}}, v_{\text{odd}})$ to E', both with the same as $(u, v) \in E$.
   Clearly, $|V'| = 2|V|$ and $|E'| = 2|E|$. What would edges look like in G'? By construction, the

two endpoints of an edge in G' have different subscripts, one with "odd," the other "even." This agrees with our previous observation on the original G that as we walk along the graph, the distance we have so far travelled alternates between even and odd. It follows that, starting from s_{even}, a vertex $v_{\text{even}} \in V'$ (resp. v_{odd}) is only reachable via even (resp. odd) number of edges. On the other hand, also notice that there is a natural correspondence between edges in G' and G: $(u_{\text{odd}}, v_{\text{even}}) \in E'$ corresponds to $(u, v) \in E$. This means a *path* in G' naturally corresponds to a walk in G, e.g.:

$$u_{\text{even}} \to v_{\text{odd}} \to w_{\text{even}} \to u_{\text{odd}} \to t_{\text{even}} \qquad \text{corresponds to} \qquad u \to v \to w \to u \to t.$$

Combining both observations above, there exists an $s - t$ walk in G with an even number of edges if and only if there is a path in G' from s_{even} to t_{even}. The rest is simple: run a pathfinding algorithm on G'. The weights are non-negative, so we use Dijkstra's algorithm.
Total runtime? Time to construct G' involves $|V'| = 2|V|$ vertices and $|E'| = 2|E|$ edges. This is dominated by running Dijkstra on G', which takes $O((|V'|+|E'|)\log|V'|) = O((|V|+|E|)\log|V|)$ time. Finally, transforming the path in G' back to a walk in G takes linear time w.r.t. the path length (one step for each edge), which is bounded by $O(|E'|)$. So overall most work is dominated by Dijkstra's algorithm and the overall algorithm runs in $O((|V| + |E|)\log|V|)$.

2. Same idea but make 6 copies of the graph.

Exercise 5.10 ()

Suppose that in addition to having edge costs $\{l_e : e \in E\}$, a graph also has vertex costs $\{c_v : v \in V\}$. Now define the cost of a path to be the sum of its edge lengths, *plus* the costs of all vertices on the path. Give an efficient algorithm for finding the minimum cost path from s to t. You may assume edge costs and vertex costs are all nonnegative.

Proof.

Using the generic approach, we can use $\text{cost}_u(v) = \text{cost}(u) + w(u, v) + c_v$ to solve this problem. Alternatively, for each edge (u, v) we can update its weight to $w(u, v) + c_v$ and run Dijkstra on this updated graph, which gives an equivalent mathematical formulation.

5.7 Negative Weighted Graphs

Now let's extend this problem to find the shortest path in negative weighted graphs. Before we think of a solution, we must make sure that there is no cycle that has a negative accumulated path. Otherwise, this problem becomes ill-defined, so we first assume that such a shortest path exists.

At first glance, we may just think of adding $\min(v)$, the minimum value to every node so that this now just becomes a regular positive graph and run BFS on it. However, this does not work since we are not adding a constant number over all paths (it is proportional to the number of nodes in the path).

Another way we can think of is just run Dijkstra. However, if it is looking at two paths. We can have $s \xrightarrow{2} b$ and $s \xrightarrow{5} a \xrightarrow{-4} b$. Dijkstra will immediately go to b thinking that it is the shortest path, since that's how far it see. So we need to look far into the future. Therefore, after an arbitrarily long path length, you could get a negative length that just kills your accumulator.

We use the Bellman equations, which could be solved using dynamic programming like we've seen before.

Theorem 5.8 (Bellman Equations)

We write the **Bellman equations**.

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{67}$$

with $d[s] = 0$ for the starting vertex. The solution has a unique solution that finds the shortest path from s to any $v \in V$.

Proof.

Note that $d[w] + l_{wv}$ is the length from some path from $s \mapsto v$ that goes through w. The minimum of it must be the shortest path over all $w \in V$. Suppose the shortest path goes through fixed x. If there exists a shorter path from $s \mapsto x$, then replace $d[x]$ by this shortest path. Therefore,

$$d[v] = d[x] + l_{xv} \leq d[w] + l_{wv} \implies d[v] = \min_w \{d[w] + l_{wv}\} \tag{68}$$

To prove uniqueness, suppose there are some other solutions π where $\pi[v] \neq d[v]$ for some v. But this cannot be the case by definition since $d[v] <= \pi[v]$ for all v.

Theorem 5.9 ()

Given the shortest paths, we can lay out this graph like a tree where $l_{ab} = l_{aa_1} + l_{a_1a_2} + \ldots + l_{a_lb}$.

So how do we actually implement this?

Algorithm 5.8 (Shortest Path in Possibly Negative Weighted Graph)

Require: Nodes V, Edges E
1: **function** ShortPath(V, E)
2: res $\leftarrow$ list(0) of large numbers of size $|V|$.
3: res[s] = 0
4: predecessors $\leftarrow \{v : None | v \in V\}$
5: **while** $\exists (u, v)$ s.t. res[v] > res[u] + l_{uv} **do**
6: res[v] $\leftarrow$ res[u] + l_{uv}
7: predecessor[v] $\leftarrow$ u
8: **end while**
9: **end function**

This is guaranteed to converge and stop after a finite number of steps since at every iteration, a path will either

1. get updated from infinity to a path length
2. get reduced from a path length to a shorter path length

And we will have to reach the shortest path length at which point we can't reduce it further.[a]
Computing the runtime is a bit tricky, since we can look at the same edge twice since minimum paths may have been updated in the middle. Therefore this list `res` may reduce very slowly. For example, let the length of each edge $|l_e| \leq L$. Then in the worst case, res[s] can be initialized to $(n-1)L$ representing the max path across all nodes, and we can decrease by 1 in each step. So over all nodes, we can decrease so that each res[s] becomes $-(n-1)L$, meaning that we are doing on the order of $2n^2L$ iterations. This is too slow, especially for non-distributed settings.

A better way is to not be so random about how we choose the (u, v) in the while loop. Notice how we can lay out the shortest paths like a tree, so we can work in layers. The next algorithm implements this.

Algorithm 5.9 (Bellman-Ford Algorithm)

We think of going in rounds indexed by t, and at every round, we are iterating through all the nodes and updating the shortest path of v using the shortest path of w included in all in-neighbors of v. At most, we will need to update this at most n times, which will guarantee convergence.

Require: Nodes V, Edges E

```
π ← list(0) of large numbers of size |V|.
π[s] = 0
function BellmanFord(x)
    for t = 1, ..., n − 1 do
        for v ∈ V do
            π^(t)[v] ← min_w{π^(k−1)[v], π^(k−1)[w] + l_wv}
        end for
    end for
end function
```

The runtime is easier to see.

1. The step in the inner loop looks over the set of nodes of size $\text{indeg}(v)$.
2. Looping over all the nodes in the inner for loop means that we are going over all edges, so $O(m)$.
3. The outer for loop goes through $n - 1$ times, so the total runtime is $O(nm)$.

At first glance, this problem seems like it isn't too different from Dijkstra, but there is a 50-year conjecture that this cannot be improved to linear time.

Exercise 5.11 ()

Let $G = (V, E)$ be a directed graph with real-valued edge weights, where each vertex is colored in either red or green. Find the shortest/cheapest $s - t$ walk such that, not counting s, the walk visits red vertices for an even number of times and green vertices at least thrice. (Duplicates allowed and will be counted more than once.)

Proof.

Similar to the last problem in the previous recitation, the key insight lies in constructing a directed graph $G' = (V', E')$ that captures some additional structures. Based on the constraints, as we walk along a path in G, there are two things we need to take care of:

- The number of (not necessarily distinct) red vertices we have walked past, and whether this number even or odd (this is called the *parity* of that number), and
- The number of (not necessarily distinct) distinct green vertices we have walked past.

To encode all of the information above, each vertex in G' will be represented by a "state", or a tuple (v, p, g) where

- $v \in V$ corresponds to an original vertex in G,
- $p \in \{0, 1\}$ (or "even", "odd") represents the parity of the count of red vertices (not necessarily distinct) visited so far, and
- $g \in \{0, 1, 2, 3+\}$ represents the number of times green vertices (not necessarily distinct) have been visited.

Now we will need to consider the conditions under which each of the tuple variable updates. For

[a] This algorithm is also called *policy iteration* in reinforcement learning and is analogous to gradient descent.

example, every time we visit a red vertex, the value p should alternate, and every time we visit a green vertex, the value g should increase until it becomes 3+. Formally, the state transitions (i.e. edges in E') can be formulated as follows. For each edge (u, v) in the original graph G, depending on the colors of u and v, we add the following edges, all with the same weight as (u, v), to E':

(v red) For every state (u, p, g) [a total of 8 such thates because $p \in \{0, 1\}$ and $g \in \{0, 1, 2, 3+\}$], add an edge to the corresponding state $(v, 1 - p, g)$. In other words, we flip the parity because we visited one more red vertex, but this does not affect the value of g.

(v green)

- For every state (u, p, g) with $g \in \{0, 1, 2\}$, add a (directed) edge to $(v, p, g + 1)$ because our green counter increases given v is green. (Define $2 + 1$ to be "3+.")
- For states of form $(u, p, 3+)$, add a (directed) edge to $(v, p, 3+)$ because we still fall under the "$g \geqslant 3$" category after visiting an additional green vertex.

All of our observations on the Recitation #2 graph modeling problem still hold: if we have a path in G', we can uniquely recover a well-defined walk in G. Initially, we want to start from state $(s, 0, 0)$ because we start from vertex $s \in V$ and, per the problem, the starting point does not contribute to the red and green count. Our goal is to reach the state $(t, 0, 3+)$, which means (i) we arrive at t, and along the course we have (ii) visited an even number of red vertices and (iii) green vertices $\geqslant 3$ times. This is exactly what we want.

How about the runtime? The construction of G' involves defining $8|V|$ vertices since p has 2 possible values and g has 4, and we need to construct one state for each pair of p and g. Similarly, for each (u, v), regardless of the color of v, in both cases we add a total of 8 edges. Therefore $|E'| = 8|E|$. Since G, G' are directed graphs with real-valued weights, we need to run Bellman-Ford, which takes $\mathcal{O}(|V||E|)$. Like shown before, other costs (e.g. the one to recover a walk in G from a path in G') are linear and hence dominated by the pathfinding runtime. So the final complexity is $\mathcal{O}(|V||E|)$.

5.8 All Pairs Shortest Paths

Now what if we want to find the minimum distance between all $u, v \in V$? We can just use $|V|$ Dijkstras or Bellman-Fords to get the appropriate runtimes of $O(EV + V^2 \log V)$ or $O(EV^2)$, respectively, but for negative weighted graphs, there is a way to do this in $O(V^3)$.[4]

5.9 Minimum Spanning Trees

Definition 5.10 (Spanning Tree)

Given an undirected graph $G(V, E)$, a **spanning tree** is a subgraph $T(V, E' \subset E)$ that is

1. a tree, and
2. spans the graph, i.e. is connected

[4]Note that if the graph is sparse, then $|E| < |V|$ and just running $|V|$ Bellman Fords may be optimal.

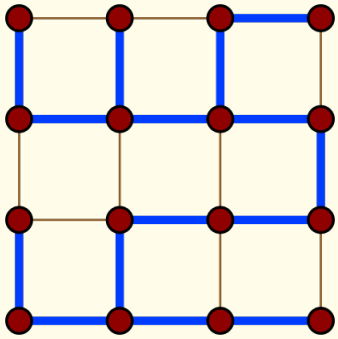


Figure 11: A spanning tree on a 4×4 grid graph.

Note that an unconnected graph will never have a spanning tree, but what about a connected graph?

Theorem 5.10 (Spanning Trees of Connected Graphs)

A connected graph will always have at least one spanning tree, not necessarily unique.

Given a connected undirected weighted graph, we may want to find the **minimum spanning tree (MST)**, i.e. the spanning tree with edges E' such that the sum of the weights of all $e \in E'$ is minimized.[5] How do we do this? There are two well-known algorithms to solve this. Prim's and Kruskal's algorithm.

5.9.1 Prim's Algorithm with Cuts

Let's try to apply what we already know: Dijkstra. If we run Dijkstra on the graph starting at $s \in V$, we can get the shortest path from s to every other node in the graph. This will give us a tree, but it may not be minimum.

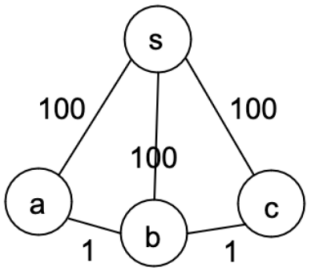


Figure 12: If we run Dijkstra on s, then our output will be a tree of cost 300, even when the actual MST can be of cost 102 starting from a.

It may seem like this is just a problem of where we start, but even this is not the case.

[5] An application of this is when we generally want to make sparse graphs. In a datacenter, wires can be expensive, so how I can minimize the length of wires to buy to construct a spanning subgraph?

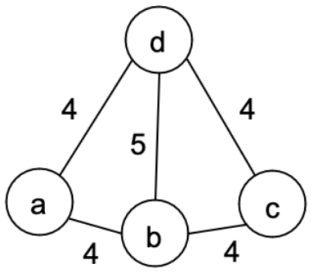


Figure 13: No matter where we start from, we will never output the MST. The MST has cost 12. If we start from b or d, we will get a tree of cost 13. If we start from a or c, we will get a tree of cost 16.

Definition 5.11 (Cuts)

Given graph $G(V, E)$, a **cut** is a partitioning of V into $(S, V \setminus S)$. Furthermore, let $\text{Cut}(S)$ be the number of edges with exactly one endpoint in S and the other in $V \setminus S$.

Theorem 5.11 (Cycles and Cuts)

Given cycle $C \subset E$ in a graph and a cut $S \subset V$,

$$|C \cap \text{Cut}(S)| \tag{69}$$

is even. We can intuit this by visualizing the cycle as a long piece of looped string and a cut is a circle. The intersection between this circle and the string must be even since every time the cycle crosses through the cut, it must return back across the cut to the initial point.

Now time for a bizarre theorem.

Theorem 5.12 (Cut Property of MSTs)

For all cuts $S \subset V$ of an undirected graph, the minimum cost edge in $\text{Cut}(S)$ belongs to the MST. Furthermore, the converse is true: if we take all cuts and find all their minimum cost edges, these edges is precisely the MST! Therefore, an edge $e \in$ MST iff e is a min-cost edge for some cut.

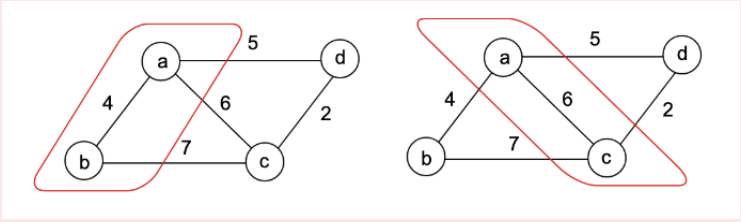


Figure 14: In the left cut, the edges are $(a, d), (b, c), (a, c)$. The minimum weight is 5 on the (a, d) edge, so it must be in the MST. For the right cut, (c, d) must be in the MST.

The final part is that if we have all edge costs different, then we will have a *unique* MST.

Proof.

We use a greedy approach and prove by contradiction. Suppose that this is not true, i.e. there exists a cut S with minimum cost edge e, and $e \notin$ MST. Then, there exists some other edge $e' \in \text{Cut}(S)$

that is in the MST, since the MST is spanning and it must cross over to connect the whole graph. Well if we just put e in and take e' out, we will still have a spanning tree since it connects the left spanning tree to the right spanning tree, and we now have a cheaper tree. So the original cannot be the MST in the first place.
To prove the converse, consider some edge e in the MST and we must prove that it is the minimum cost edge in some cut. Note that if we take e out, then it divides the MST into two connected components, and we can just define the cut as these subsets of nodes. So this is in $\text{Cut}(S)$ for some $S \subset V$. We can also prove that this is minimal since if it wasn't the minimum cost edge for some cut, we could have taken it out and inserted a cheaper edge e' to begin with, getting a cheaper spanning tree.

We can just brute force this logic into an algorithm by going through all possible cuts and adding the minimum cost edge to our MST set. It is clear that a cut is defined by a subset of S, so really the number of cuts a graph can have is $2^{|S|-1}$, which is exponential in n. However, the minimum spanning tree isn't exponential since it must have $n-1$ edges, so there must be many cuts with the same minimum edge.

One way is to start with one vertex a that contains the minimum cost edge (a, b) across all edges. This edge must be minimal and must be in the MST. Then we can look at the cut $S = \{a, b\}$ and look at that cut. We keep doing this, keeping track of the set of edges we need to look at after adding a new node to our cut. So the number of cuts I consider is equal to the number of edges in the spanning tree.

Algorithm 5.10 (Prim's Algorithm to Find MST)

It turns out that we can modify Dijkstra to solve it.

Require: Graph $G(E, V)$
function PRIM(s)
  $S \leftarrow \{s\}$ ▷Our initial cut
  $\pi[v] \leftarrow$ list of size $|V|$ of $+\infty$ ▷$\pi[u]$ is min cost of getting from u into the S
  $\pi[v] = w_{sv}$ for all $v \in V$ ▷initialize list with neighboring nodes from start s
  while $S \neq V$ **do**
    $u = \text{argmin}_{v \notin S} \pi[v]$ ▷Find node having minimum cost to reach from S
    $S \leftarrow S \cup \{u\}$ ▷Adding this node to S to expand our cut
    for $u \notin S$ **do** ▷Since we expanded S, our min reach distances
      $\pi[u] \leftarrow \min\{\pi[u], w_{wu}\}$ ▷must be updated. It can only get shorter
    end for ▷through a path from new u, so compare them
  end while
end function

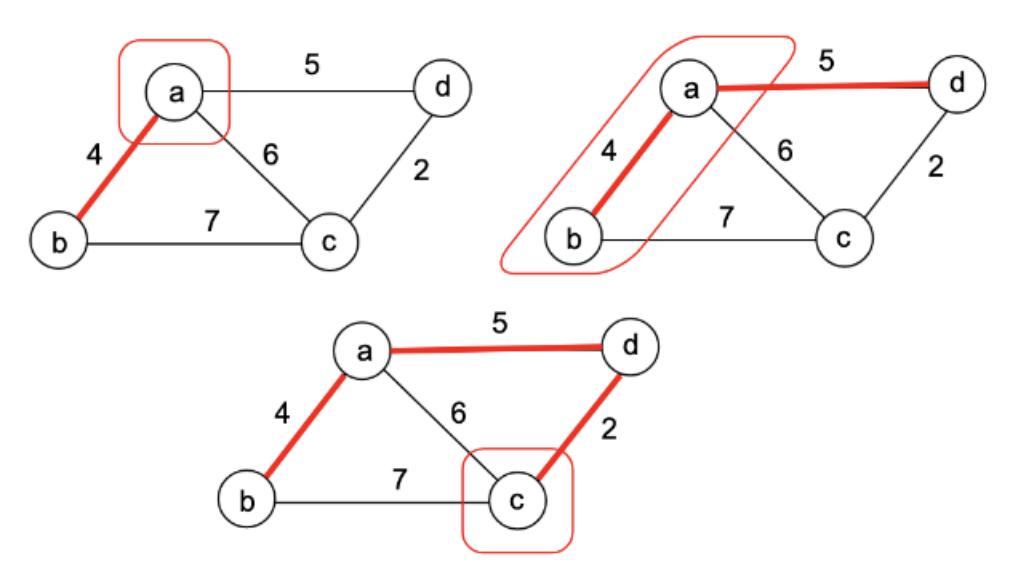


Figure 15: Step by step process of the method we mention above.

We are really going through two loops. We add to the cut S n times and for each time we add, we must compute the argmin, which is also $O(n)$, so our total time complexity of $O(n^2)$.

However, this is not efficient, so we can introduce a minheap to get the argmin step faster.

Algorithm 5.11 (Prim's Algorithm with MinHeap)

Require: Nodes V, Edges E

```
function PRIM(V, E)
    mst ← []                                          ▷Initialize mst array to return
    s ← 0                                             ▷Choose any starting node
    visited ← set()                                   ▷Our expanding set of cuts.
    edges ← minheap()                                 ▷The set of low-weight edges that we can explore from
    add (weight, s, next_node) for edges in E         ▷Look for edges from s to expand our cut from.
    while |visited| < |V| do                          ▷Until we have visited all cuts,
        weight, frm, to ← pop from edges              ▷Get the cheapest edge to explore
        if to ∉ visited then                          ▷If this isn't already in our cut,
            add to → visited                          ▷Add it to our cut. From cut
            add (frm, to, weight) to mst              ▷property, this must be added to mst
            for next_to, next_weight ∈ E[to] do       ▷After expanding, add newly discovered
                if next_to ∉ visited then             ▷edges for future exploration
                    push (next_weight, to, next_to) to edges
                end if
            end for
        end if
    end while
    return mst                                        ▷of form (from, to, weight)
end function
```

You essentially push n times and pop m times, and the time per push and pop is $\log_2(n)$. Therefore,

the total time to push is $n\log(n)$ and to pop is $m\log(n)$, making the total runtime $O((n+m)log(n)) = O(m\log n)$.

This can be sped up even faster if we use Fibonacci heaps or assume extra structure on the graph.

5.9.2 Kruskal's Algorithm

If we were to try and construct this algorithm from scratch, we may take a greedy approach by incrementally adding the minimum cost edge from your cut. However, there is one thing to check: have we entered a cycle? Checking whether the next added node a completes a cycle in $S \cup \{a\}$ is nontrivial.

Theorem 5.13 (Cycle Property)

For all cycles C, the max cost edge of C does not belong to MST.

Therefore, you can take S and either add to it using the cut property or delete candidates from it using the cycle property. What is the best order to do this in? Kruskal's algorithm answers this question, which takes a greedy approach.

Algorithm 5.12 (General Kruskal's Algorithm)

The general idea is that we sort $e \in E$ in increasing cost, and for each $e \in E$, we use either the cut or cycle property to decide whether e goes in or out.

Require: Graph $G(V, E)$
1: **function** KRUSKAL(V, E)
2: sort E in increasing cost
3: $T \leftarrow \{\}$
4: **for** $e \in E$ **do**
5: **if** $T \cup \{e\}$ does not have cycle **then**
6: $T \leftarrow T \cup \{e\}$ ▷Cut property
7: **else** ▷Cycle property
8: continue ▷discard e since from sorting, this edge is heaviest in cycle
9: **end if**
10: **end for**
11: **end function**

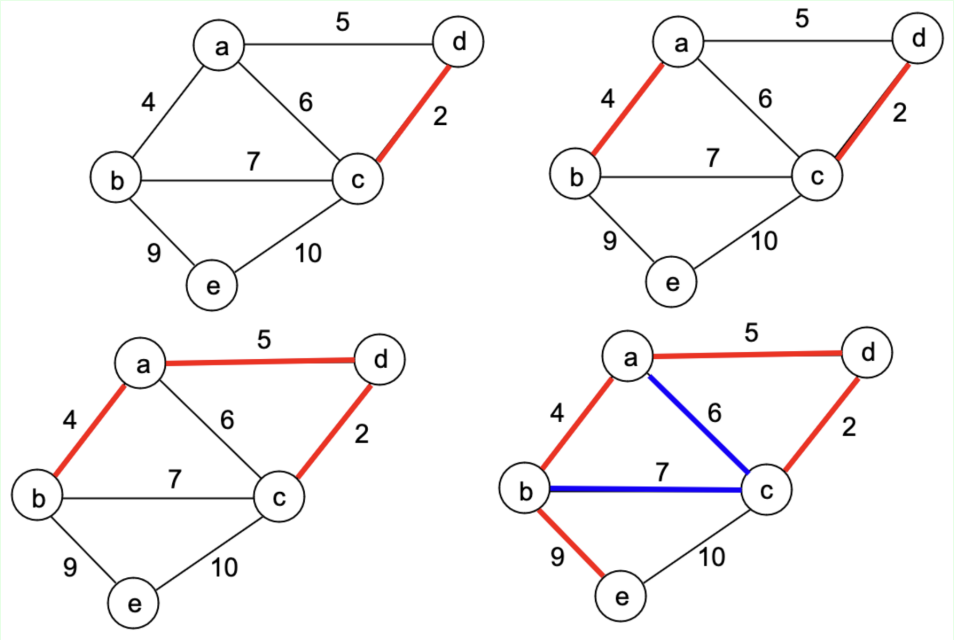


Figure 16: Kruskal's algorithm. In the last step, we see that the next minimum cost edge of 6 and 7 forms a cycle, so we add the edge of length 9.

The sorting of edges take $O(m \log m)$ time, and after sorting, we iterate through all the edges and apply m find-union algorithm, which each take at most $O(\log n)$ time. Therefore, the overall complexity is $O(m \log m + m \log n)$. However, the value of m can be at most $O(n^2)$, so the two logarithms are essentially the same, arriving at the final runtime of $O(m \log n)$.

The way to prove that this is correct is to show that every step you do is correct, known as *structural induction*, either because of one of the two properties. Say that so far, we have some edges in V which forms a partition of $V = \sqcup_i T_i$ of disjoint trees (can be trees, one edge, or just single nodes). We are looking at the next biggest edge $e = (a, b)$. There are two possibilities.

1. If a, b are both in a single T_i, then this forms a cycle and can be thrown away since this is the max cost edge in the cycle by the cycle property.
2. If a, b connect T_i and T_j for $i \neq j$, then this edge is in $\text{Cut}(T_i)$ and is the minimum cost edge since the rest of the edges in $\text{Cut}(T_i)$ come next in the sorted E. Therefore this must be included by the cut property.

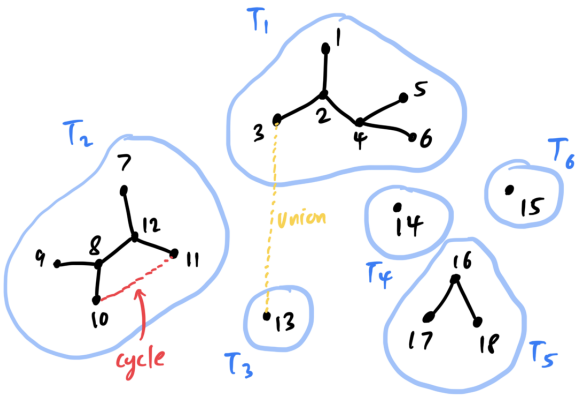


Figure 17: Each T_i is a component formed by the edges chosen so far. For example, $T_1 = \{1, 2, 3, 4, 5, 6\}, \ldots$. We can either discard an edge (red) or include an edge (yellow).

The only bottleneck in here is line 5, where we check if e does not complete a cycle in T. It will obviously not be efficient to do BFS, construct a tree, and see if there is a loop by checking if two points in the same layer are connected. It may help to decompose this algorithm into two steps: what is being stored and what is being checked?

Note that from our visual, we are really just keeping a set of these points that each make a subtree and connecting them together. How do we efficiently search for which cluster a point is a part of and efficiently merge two clusters? We could use a hashmap but this wouldn't work. We need something like a doubly linked list.

Algorithm 5.13 (Kruskal's Algorithm)

The implementation uses the Union-Find data structure. For clarity, we will not elaborate it but will show the full pseudocode.

Require: Nodes V. Edges $E = \{(u, v, w)\}$ where $u, v \in V$ and $w > 0$ is a weight.

```
function KRUSKAL(V, E)
    n ← |V|
    sort E in increasing order of weights.                ▷Needed for Kruskal
    parent ← [0, ..., n − 1]                              ▷Initialize the disjoint cluster each node is in
    rank ← list of 0s of size n
    function FIND(x)
        if parent[x] ≠ x then
            parent[x] ← Find(parent[x])                   ▷Path compression
        end if
        return parent[x]
    end function
    function UNION(x, y)
        px, py = find(x), find(y)
        if px = py then
            return False
        end if
        if rank[px] < rank[py] then
            parent[px] ← py
        else if rank[px] > rank[py] then
            parent[py] ← px
        else
            parent[py] ← px
            rank[px] ← rank[px] + 1
        end if
        return True
    end function
    mst ← []
    for u, v, weight ∈ edges do
        if union(u, v) then
            add (u, v, weight) to mst
        end if
        if len(mst) = n - 1 then
            break
        end if
    end for
    return mst
end function
```

To analyze the runtime of this, we define the function.

Definition 5.12 (Ackerman Function)

The **Ackerman function** is one of the fastest growing functions known. It is practically infinity.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases} \tag{70}$$

The inverse Ackerman function therefore grows extremely slowly.

If we optimize the steps in Kruskal's algorithm, we can get its runtime to

$$O((m+n)\log^*(n)) \tag{71}$$

which is practically linear.

5.9.3 Applications

Here is a way to cluster data, a surprising way to apply MSTs. It is the most widely used application, especially in data science. The problem is that given n data points $x_i \in \mathbb{R}^d$ and an integer k, we want to partition the points into k groups $\mathbf{C} = (C_1, \dots, C_k)$ where $\mathbf{x} = \sqcup_i C_i$. You want to distances between the points within a group to be small and the distances between groups to be large. We can think of finding the objective which takes every pair of clusters and computes the minimum distance between these clusters, and we want to maximize this distance over all pairs of clusters.

Algorithm 5.14 (Single Linkage/Hierarchical Clustering)

The general idea is to take this dense graph, find the MST, and cut off the largest edges from this MST, which will give you k components. This is the answer. Or really, you can use Kruskal's algorithm and terminate earlier when T has k sets/components. Note that as we add edges as we construct our MST, we are merging two clusters into one. So that all you are doing is finding the next pair of closest points and merging the clusters that they are a part of.

Require: Nodes $V = \{v_i\} \subset \mathbb{R}^n$
1: **function** CLUSTER(V)
2: Run Kruskal and at each iteration, check if you have K clusters.
3: If so, terminate and return the `parent` list.
4: **end function**

Theorem 5.14 ()

The algorithm above minimizes the objective function.

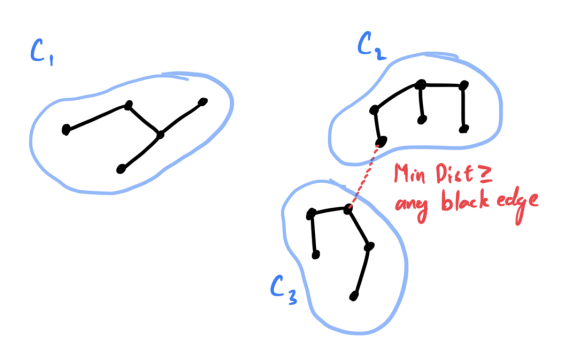
$$\underset{\mathbf{C}}{\operatorname{argmax}} \min_{p \in C_i, q \in C_j} \{d(p,q)\} \tag{72}$$

Proof.

Let $\mathbf{C}^*$ be the MST clustering. We claim that for any other clustering $\mathbf{C} = \{C_1, \dots, C_k\}$,

$$\text{mindist}(\mathbf{C}) \leq \text{mindist}(\mathbf{C}^*) \tag{73}$$

Assume that this was not the case, so $\text{mindist}(\mathbf{C}) > \text{mindist}(\mathbf{C}^*)$ and therefore there exists a $p, q \in C_i, C_j$ such that $d(p,q) = \text{mindist}(\mathbf{C}) > \text{mindist}(\mathbf{C}^*)$. Since this is a different clustering, p, q must have been in the same cluster C_i^*. But note that since Kruskal adds edges in increasing length, all edges within a cluster must have length less edges that go across two clusters.



So $d(p, q)$ must be the less than the length of all edges within a cluster in $\mathbf{C}^*$. But all within-cluster edges must be smaller than mindist($\mathbf{C}^*$), meaning that mindist($\mathbf{C}^*$) $> d(p, q)$, contradicting the fact that is is greater, and we are done.

If you define the distance between two clusters to be the distance between the centroids (mean point), then this is called *average linkage* (min avg distance). If we define the cluster distance as the maximum distance between two points, then it is called *complete linkage* (min max distance). Kruskal's algorithm only worked for the single linkage case but may not work for these additional definitions. This is why there is usually a whole suite of clustering algorithms for a particular problem and we just find out which one fits the data the best. Furthermore, we have done *bottom-up clustering*, where we took individual points to make clusters. In *top-down clustering*, we take the whole set and cut it up into clusters.

6 Dynamic Programming

Let's take a look at a motivating example.

Example 6.1 (Computing Fibonacci Numbers)

To compute the Nth Fibonacci number, we can use a recursive method.

Algorithm 10

Require: N

```
function RecFib(N)
    if N = 0 then
        return 0
    else if N = 1 then
        return 1
    else
        return RecFib(N − 1) + RecFib(N − 2)
    end if
end function
```

This is exponential, in fact $O(\varphi^N)$ where φ is the golden ratio. The reason is that we are repeatedly computing the same subproblem (e.g. RecFib($N - 2$ is computed twice)), leading to inefficiency. It would be great if we could store these intermediate values rather than recomputing them. This introduces us to the concept of memoization.

Definition 6.1 (Memoization)

Memoization refers to storing intermediate values for reuse rather than computing them again (e.g. in future recursive calls).
Therefore, the term **dynamic programming** just means *to deliberately evaluate and memoize in the correct order.*

Algorithm 6.1 (Redursive Memoized Fibonacci)

This leads us to a memoized version of computing Fibonacci Numbers, which is linear runtime. In fact, we can have constant space complexity since we don't need more than the last 2 previous Fibonacci numbers to compute the next one.

Algorithm 11 Memoized Fibonacci

Require: N
  Initialize memo array $F[0..N]$ with -1
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  function MemFib(N)
    if $N < 0$ **then**
      return 0
    else if $F[N] \neq -1$ **then**
      return $F[N]$
    else
      $F[N] \leftarrow$ MemFib$(N-1)$ + MemFib$(N-2)$
      return $F[N]$
    end if
  end function

The runtime is computed, by taking the number of distinct problems (i.e. the number of calls to MemFib with distinct inputs) multiplied by the time per subproblem (constant since lookup is constant and adding is constant). Note that this assumes that arbitrary arithmetic operations take constant time, but this is not true if we look at the bit complexity, which can scale quite fast as these numbers grow.

Note that this does not really explicitly show the order in which the memoized list is being filled. It is implicit but hard to see in the recursive calls. Therefore, it may help to write it iteratively.

Algorithm 6.2 (Iterative Memoized Fibonacci)

In here, we can explicitly see that the nth Fibonacci number is explicitly dependent on the $n-2$ and $n-1$.

Algorithm 12 Iterative Fibonacci

Require: N
  Initialize array $F[0..N]$
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  for $i \leftarrow 2$ to N **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
  end for
  return $F[N]$

6.1 Longest Increasing Subsequence

Definition 6.2 (Longest Increasing Subsequence)

Given a sequence of numbers $A = \{a_1, a_2, ..., a_n\}$, a **longest increasing subsequence** is a subsequence $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$ of A such that:

1. $1 \leq i_1 < i_2 < ... < i_k \leq n$ (maintains original order)
2. $a_{i_1} < a_{i_2} < ... < a_{i_k}$ (strictly increasing)
3. k is maximized (longest such subsequence)

Example 6.2 ()

For the sequence $A = \{3, 10, 2, 1, 20, 4, 25\}$:

- $\{3, 10, 20, 25\}$ is an increasing subsequence of length 4
- $\{2, 4, 25\}$ is an increasing subsequence of length 3
- $\{3, 10, 20, 25\}$ is a longest increasing subsequence as no increasing subsequence of length 5 or greater exists

For the actual problem of calculating the length of the LIS, dynamic programming gives us an efficient approach. First, let's do a brute force algorithm.

Algorithm 6.3 (Recursive Brute Force LIS)

At each step, we consider whether to include the current element in our subsequence. We can only include it if it's larger than the previous element we chose, maintaining the increasing property. We explore both possibilities (including and excluding) recursively to find the longest possible subsequence.

Algorithm 13 Recursive Brute Force Longest Increasing Subsequence

```
Require: Array A[1..n]
  function LIS(A, i, prev)          ▷ i is current position, prev is last element we took
    if i = n + 1 then                                   ▷ If we've processed all elements
      return 0                                          ▷ Return 0 as we can't add more elements
    end if
    // First choice: skip current element
    skip ← LIS(A, i + 1, prev)                          ▷ Keep same prev, move to next element
    // Second choice: try to take current element
    take ← 0                                            ▷ Initialize take option to 0 in case we can't take it
    if A[i] > prev then                                 ▷ Only if current element maintains increasing sequence
      take ← 1+ LIS(A, i + 1, A[i])                     ▷ Add 1 for taking current element
    end if                                              ▷ Recursively find best sequence starting at i+1 with A[i] as previous
    // Return best option between taking and skipping
    return max(skip, take)                              ▷ Choose the better of our two options
  end function
  // Initial call with sentinel value to allow taking any first element
  return LIS(A, 1, −∞)                                  ▷ Start at first element, no previous restrictions
```

The runtime can be used by the recurrence relation. At every call, we may at most have to compute 2 calls on the subarray not including the current element, and we compute the max of them which takes $O(1)$, so

$$T(N) = 2T(N-1) + O(1) \implies T(N) = O(2^N) \tag{74}$$

which is not good.

Now let's move to our DP solution.

Algorithm 6.4 (Dynamic Programming LIS)

The key insight is that LIS[i] (the length of the LIS within the input array ending at i, inclusive) depends on all previous LIS[j] where j < i and A[j] < A[i]. For each position i, we can extend any previous increasing subsequence that ends with a smaller value. In other words, we are solving

$$\text{LIS}[i] = 1 + \max\{\text{LIS}[j] \mid j < i \text{ and } A[i] > A[j]\} \tag{75}$$

Algorithm 14 Dynamic Programming Longest Increasing Subsequence

```
Require: Array A[1..n]
  function DPLIS(A)
    // Initialize LIS array - each single element is an increasing sequence
    Initialize array LIS[1..n] with 1          ▷Base case: each element forms LIS of length 1
    for i ← 2 to n do                          ▷Consider each element as end of sequence
      for j ← 1 to i − 1 do                    ▷Look at all previous elements
        if A[i] > A[j] then                    ▷Can we extend sequence ending at j?
          LIS[i] ← max(LIS[i], LIS[j] + 1)     ▷Take best possible extension
        end if
      end for
    end for
    return LIS[-1]                             ▷Find the maximum value in LIS array
  end function
```

The runtime is $O(n^2)$ since we have two nested loops, and the space complexity is $O(n)$ since we store one value for each position in the array. Note that we could also have filled this DP array backwards by considering all arrays that start at $A[i]$.

Example 6.3 (DPLIS for Small Array)

For array $A = [3, 1, 4, 1, 5]$:

- Initially: $LIS = [1, 1, 1, 1, 1]$
- After processing $i = 2$: $LIS = [1, 1, 1, 1, 1]$
- After $i = 3$: $LIS = [1, 1, 2, 1, 1]$ (4 can extend sequence from 3)
- After $i = 4$: $LIS = [1, 1, 2, 2, 1]$
- After $i = 5$: $LIS = [1, 1, 2, 2, 3]$ (5 can extend sequence from 4)

Final answer is 3, corresponding to subsequence $[3, 4, 5]$

6.2 0/1 Knapsack

Another application of DP is in the following problem.

Definition 6.3 (0/1 Knapsack)

Given a knapsack with maximum weight capacity W, along with a set of n items, each with:

- Weight/Cost c_i
- Value v_i

You want to know the size of the subset of items that maximizes total value while keeping total weight/Cost $\leq W$. The constraint is that each item can be picked at most once (hence 0/1).

Example 6.4 ()

For $n = 3$ items and capacity $W = 4$:
- Item 1: $(c_1 = 2, v_1 = 3)$
- Item 2: $(c_2 = 1, v_2 = 2)$
- Item 3: $(c_3 = 3, v_3 = 4)$

Optimal solution: Take items 1 and 2
- Total weight: $2 + 1 = 3 \leq 4$
- Total value: $3 + 2 = 5$ (maximum possible)

The most natural way to approach this would be greedy, but this does not exactly work.

Example 6.5 (Counter-Example for Greedy Knapsack)

Let's consider a knapsack with capacity $W = 10$ and:
- Values $V = [100, 48, 60, 11]$
- Weights $C = [10, 6, 4, 1]$

Then the value/weight ratios would be

$$V/C = [10, 8, 15, 11] \tag{76}$$

and so we would choose to gain 60 for cost 4, then gain 11 for cost of 1. We do not have enough to buy any more and have a cost of 71, when we could have gotten a cost of 108 by buying 60 and 48 for a total of 10.

Just like as always, we just solve this using recursive brute force and then apply optimization with DP.

Algorithm 6.5 (Recursive Brute Force Knapsack)

The key idea is that for each item, we have two choices: either include it (if we have enough capacity) or exclude it. We try both possibilities recursively to find the maximum value possible. In here, i represents the current item we're considering and r is the remaining weight we can still use.

Algorithm 15 Recursive Knapsack

Require: Values $V[1..n]$, Weights $W[1..n]$, Capacity C
```
V, W
function KNAPSACK(i, r)
    if i = n + 1 or r = 0 then                          ▷Base case
        return 0                                        ▷Either we've considered all items or filled the knapsack
    end if                                              ▷No more value can be added
    skip ← KNAPSACK(i + 1, r)                           ▷1st Op: Skip current item and maintain same r
    take ← 0                                            ▷2nd Op: Try to include curr item
    if W[i] ≤ r then                                    ▷Only try taking if item's weight fits in remaining r
        take ← V[i]+ KNAPSACK(i + 1, r − W[i])          ▷Add current item's value + best value from remaining items
    end if                                              ▷Subtract current item's weight from remaining r
    return max(take, skip)                              ▷Return best possible value between two choices
end function
// Start considering from first item with full r
return KNAPSACK(1, C)
```

Now let's apply memoization.

Algorithm 6.6 (Dynamic Programming Knapsack)

The key insight is that $K[i, r]$ (best value possible using items up to i exclusive with remaining capacity r) can be built from $K[i-1, r]$ and $K[i-1, r-W[i]]$ using the formula

$$K[i,r] = \begin{cases} K[i-1,r] & \text{if } W[i] > r \text{ (can't take item i)} \\ \max(K[i-1,r], V[i] + K[i-1, r-W[i]]) & \text{if } W[i] \leq r \text{ (can take item i)} \end{cases} \tag{77}$$

In here, i represents the items 1..i we're considering and r represents the remaining capacity.

Algorithm 16 Dynamic Programming Knapsack

Require: Values $V[1..n]$, Weights $W[1..n]$, Capacity C
V, W
function DPKnapsack(C)
Create table $K[0..n, 0..C]$ ▷K[i,r] = max value using items V[:i] with remaining r
Initialize all entries to 0
for $i \leftarrow 1$ to n **do** ▷Consider each item
for $r \leftarrow 0$ to C **do** ▷Consider each possible remaining capacity
$K[i,r] \leftarrow K[i-1,r]$ ▷Default: inherit value from excluding item i
if $r \geq W[i]$ **then** ▷If current item fits in remaining capacity
take $\leftarrow V[i] + K[i-1, r-W[i]]$ ▷Value of item i + best value with remaining r
$K[i,r] \leftarrow \max(K[i,r]$, take) ▷Take better of including or excluding
end if
end for
end for
return $K[n, C]$ ▷Best value possible using all items
end function

The memory complexity is obviously $\Theta(nC)$. The number of subproblems is nC, and the processing for each step is constant time (possibly addition and max), so $O(1)$. Therefore the total runtime is $O(nC)$ also. Note that if we compare this in terms of the bit runtime, then this is $O(n \log_2 C)$, which is psuedopolynomial since C is described by $\log_2 C$ bits. However, the n will scale linearly since it is the size of the array and not the size of each integer in V, W.

Note that if C can be very big, and this can be problematic.

6.3 Line Breaking

Definition 6.4 (Line Breaking)

Line breaking is used whenever you compile a tex document. Given

- a sequence of words $w_1, w_2, ..., w_n$ where w_i has length l_i
- a maximum line length L
- each line must contain whole words in order, separated by spaces

Our goal is to break words into lines to minimize the sum of squares of empty spaces at the end of each line, i.e. our cost function for a sequence of words $W[i : j+1]$ is

$$\left(L - \sum_{k=i}^{j} w_k\right)^2 \tag{78}$$

If we used the absolute value, there exists a greedy solution.

Example 6.6 (Line Breaking Example)

Given $L = 20$ and $W = [12, 8, 9]$ (note that spaces won't count in length in this problem), we have 3 possible arrangements.

1. Option 1: First two words on first line
   - Line 1: [12 + 8 = 20] (0 spaces remain)
   - Line 2: [9] (11 spaces remain)
   - Total cost $= 0^2 + 11^2 = 121$
2. Option 2: Words one per line
   - Line 1: [12] (8 spaces remain)
   - Line 2: [8] (12 spaces remain)
   - Line 3: [9] (11 spaces remain)
   - Total cost $= 8^2 + 12^2 + 11^2 = 64 + 144 + 121 = 329$
3. Option 3: Word 2 and 3 together
   - Line 1: [12] (8 spaces remain)
   - Line 2: [8 + 9 = 17] (3 spaces remain)
   - Total cost $= 8^2 + 3^2 = 64 + 9 = 73$

Therefore, Option 3 is optimal with cost 73. This also demonstrates that the greedy strategy, which gives Option 1, will not work.

We start off with recursive brute force. Given any word w_i, we can either end the line there $w_i|$ or add another word $w_i w_{i+1}$.

Algorithm 6.7 (Recursive Brute Force Line Breaking)

The key idea is at each position i, we try placing different numbers of words on the current line and recursively solve for the rest. For each word i, we try all possible ways to break the line starting at word i. Let $MinCost(i)$ be the minimum cost to arrange words $[i..n]$. Then:

$$MinCost(i) = \begin{cases} 0 & \text{if } i > n \\ \min_{i \le j \le n} \left\{ (L - \sum_{k=i}^{j} W[k])^2 + MinCost(j+1) \right\} & \text{if else} \end{cases} \tag{79}$$

where $(L - \sum_{k=i}^{j} W[k])^2$ is the cost to put the words $i...j$ on a new line and then ending. So whenever we add the new ith word at the end of the line, we are looking at all words j at which we can break the line, and taking the minimum cost given this line break. The line break from $j + 1$ should be computed as well (this is what we will store in our DP array later).

Algorithm 17 Recursive Line Breaking

Require: Word lengths $W[1..n]$, Line length L

```
W
function MinCost(i)                                  ▷Returns min cost for words[i..n]
    if i = n + 1 then                                ▷Base case: no words left
        return 0
    end if
    min_cost ← ∞
    lineLen ← 0
    for j ← i to n do                                ▷Try placing words i through j on current line
        lineLen ← lineLen +W[j]
        if lineLen ≤ L then                          ▷If these words fit on the line
            spaces ← L− lineLen                      ▷Extra spaces at end of line
            cost ← spaces^2+ MinCost(j + 1)          ▷Cost of this line + rest
            min_cost ← min(min_cost, cost)
        end if
    end for
    return min_cost
end function
return MinCost(1)                                    ▷Start with first word
```

Example 6.7 (Recursive Brute Force Line Breaking)

For example, with $L = 20$ and $W = [12, 8, 9]$:

- At $i = 1$: Try
  - 12 alone (8 spaces) + solve for [8,9]
  - 12,8 together (0 spaces) + solve for [9]
- At $i = 2$: Try
  - 8 alone (12 spaces) + solve for [9]
  - 8,9 together (3 spaces) + solve for []
- At $i = 3$: Try
  - 9 alone (11 spaces) + solve for []

Algorithm 6.8 (Dynamic Programming Line Breaking)

The key insight is that $DP[i]$ represents the minimum cost of optimally arranging words $[i..n]$. For each word i, we try placing words $[i..j]$ on a line and add the cost of optimally arranging the remaining words $[j + 1..n]$. We do the same logic but rather than computing it we just retreive it from the DP array.

$$DP[i] = \begin{cases} 0 & \text{if } i > n \\ \min_{i \le j \le n} \left\{ (L - \sum_{k=i}^{j} W[k])^2 + DP[j+1] \right\} & \text{if else} \end{cases} \tag{80}$$

Algorithm 18 Dynamic Programming Line Breaking

Require: Word lengths $W[1..n]$, Line length L
W
function LineBreak(L)
Create array $DP[0..n]$ ▷DP[i] = min cost for words[i..n]
Initialize all entries to ∞
$DP[n+1] \leftarrow 0$ ▷Base case: no words left
for $i \leftarrow n$ downto 1 **do** ▷Consider each starting word
lineLen $\leftarrow 0$
for $j \leftarrow i$ to n **do** ▷Try placing words i through j on a line
lineLen $\leftarrow$ lineLen $+W[j]$
if lineLen $\leq L$ **then** ▷If these words fit on the line
spaces $\leftarrow L-$ lineLen ▷Extra spaces at end
cost $\leftarrow$ spaces$^2 + DP[j+1]$ ▷Cost of this line + rest
$DP[i] \leftarrow \min(DP[i], \text{cost})$ ▷Update if better
end if
end for
end for
return $DP[1]$ ▷Cost of optimally arranging all words
end function

The memory complexity is clearly $O(n)$. To add each element i, we must iterate over all the possible j's, making this $O(n^2)$ total iterations. However, computing the cost is also $O(n)$, making the total runtime $O(n^3)$. However, if we also store another DP array `sums`, which can be computed in linear time and stores

$$\texttt{sums}[i] = \sum_{k=i}^{n} w_k \implies \sum_{k=i}^{j} = \sum_{k=i}^{n} w_k - a \sum_{k=j+1}^{n} w_k = \texttt{sums[i]} - \texttt{sums[j+1]} \tag{81}$$

which can be accessed and computed in $O(1)$ time, bringing us down to $O(n^2)$.

Example 6.8 (DP Line Breaking)

For example, with $L = 20$ and $W = [12, 8, 9]$:
- $DP[4] = 0$ (base case)
- $DP[3] = 11^2 = 121$ (only option for last word)
- $DP[2] = \min(12^2 + 121, 3^2) = \min(265, 9) = 9$ (alone or with word 3)
- $DP[1] = \min(8^2 + 9, 0^2 + 121) = \min(73, 121) = 73$ (alone or with word 2)

6.4 Bellman Ford Revisited

Recall the Bellman equations that we must solve using DP. That is, given some $s \in V$,

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{82}$$

with $d[s] = 0$. Note that the problem was not well defined if there are negative cycles in the graph, since we can just loop an infinite number of times. However, we can modify the bellman equations to get a better sense.

Theorem 6.1 (Modified Bellman Equations)

For paths of length at most i edges, the Bellman equations become:

$$d(v,i) = \begin{cases} 0 & \text{if } v = s \\ \min\{d(v,i-1), \min_{(u,v)\in E}\{d(u,i-1) + w_{uv}\}\} & \text{otherwise} \end{cases} \tag{83}$$

where $d(v,i)$ represents the shortest path from source s to vertex v using at most i edges, and w_{uv} represents the weight of edge (u,v). Note that the inner minimum takes the minimum over all paths with the final edge connecting from some other node u to target v, and the outer minimum compares this minimum path to what we already have to see if it's an improvement.

This indicates that we should use a 2D DP array to memoize. This allows you to have a more flexible representation in case there are negative cycles since we are also limiting the number of edges a path could have.

Algorithm 6.9 (2D DP Bellman-Ford)

The implementation of the modified equations gives us the following algorithm.

Require: Nodes V, Edges E, source s
$d \leftarrow$ 2D array of size $|V| \times |V|$ initialized to ∞
$d[s,0] \leftarrow 0$ ▷Base case: Can reach source with 0 edges
for $i = 1, \dots, n-1$ **do**
  for $v \in V$ **do**
    $d[v,i] \leftarrow d[v,i-1]$ ▷Keep best path seen so far
    for $(u,v) \in E$ **do**
      $d[v,i] \leftarrow \min(d[v,i], d[u,i-1] + l_{uv})$
    end for
  end for
end for

Note that this algorithm still has time complexity $O(nm)$ because the outer loop runs $n-1$ times and for each iteration, we examine each edge once. The space complexity is $O(n^2)$, and finally, note the important properties.

1. *Monotonicity*: For all vertices v and indices i:

$$d(v,i) \leq d(v,i-1) \tag{84}$$

   This is because any path using $\leq (i-1)$ edges is also a valid path using $\leq i$ edges.
2. *Convergence*: The algorithm will stabilize after at most $n-1$ iterations since:
   - Any shortest path without negative cycles can use at most $n-1$ edges
   - Therefore, $d(v,n-1) = d(v,n)$ for all v if no negative cycles exist
3. *Negative Cycle Detection*: A negative cycle exists if and only if:

$$\exists v \in V : d(v,n) < d(v,n-1) \tag{85}$$

   This is because any improvement after $n-1$ edges must use a negative cycle.

7 Hashing and Probabilistic Algorithms

One of the most important applications of hashing and probabilistic algorithms is in cryptography. For example, when you generate a RSA keypair, you must choose two 128-bit prime numbers which serve

as your private key. State of the art methods simply generate a random 128-bit prime number perform *independent* randomized probabilistic tests with error rate ϵ. Given that this is independent, simply running this tests k times reduces our error exponentially to ϵ^k, practically guaranteeing primality. In Bitcoin and Ethereum wallets, hardened child key derivation functions in hierarchical determinstic wallets hash subsets of the parent key to generate both the child keys and their seeds.[6]

7.1 Hashing

Assume we want to map keys from some **universe** U into a smaller set $[N] = \{1, \dots, N\}$ of N bins. The algorithm will have to handle some dataset $S \subset U$ of $|S| = M$ keys.[7] We want to store S with efficient performance in

1. `Find(x)`, which returns true if $x \in S$
2. `Insert(x)`
3. `Delete(x)`

This is essentially a set. If one is new to hashing, we could try to use a balanced binary search tree or a heap (implemented as a red-black tree or some other variant), which can do all three operations in $O(\log(N))$ time. If we had access to a randomized function $h : U \to [N]$, with the property that for a fixed $x \in U$, $h(x) \sim \text{Multinomial}(N)$, then by linearity of expectation[8]

$$\mathbb{P}(h(x) = h(y)) = \mathbb{E}[\mathbb{1}_{h(x)=h(y)}] = \frac{N}{N^2} = \frac{1}{N} \tag{86}$$

This is great, but this is also random, which means that we are not guaranteed to map to the same bin across time. If we add time-invariance, this is pretty much a hash function, which is really deterministic, but we like to call it psuedo-random.

So what if there are collisions? They are inevitable anyways even with a completely random function, but minimizing them will give us the best runtime performance. There are a two main ways to approach this.

1. At each bin, store the head of a linked list, BST, or another hash table. You would incur an additional linear, logarithmic, or constant cost of traversing this data structure for each operation.
2. Look for another bin. We can just look at $h(x) + 1$ or $h(h(x))$ (or really any deterministic function $f(h(x))$) if $h(x)$ is occupied.

This is what we have to work with here, so we would like to modify our assumptions. Therefore, we can achieve good performance by reducing the collisions and by improving how we deal with collisions. We will focus on the first part.

[6] For an implementation of both, look at my blockchain implementation here

[7] For example, think of all possible strings as U and all 256-bit numbers as $N = 2^{256}$. S in this case S may be the set of all addresses on a blockchain. This is SHA256, and to date there is no known hash collisions, though there are a few for SHA1.

[8] Note that this is not $\mathbb{P}_{x,y}$ since x, y are not random variables. They are fixed, and $h(x), h(y)$ are the random variables.

7.2 Modulo Operations

7.3 Primality Testing

8 Linear Programming

9 Streaming Algorithms

Algorithm 9.1 (Boyer-Moore Majority Vote Algorithm)

Algorithm 19

Require:

function FUNC(x)
end function

Algorithm 9.2 (Track Median From Data Stream)

Algorithm 20

Require:

function FUNC(x)
end function