6.006: Introduction to Algorithms

Jesse Yang

Fall 2020

§1 Lecture September 1, 2020

§1.1 Algorithmic Thinking

Example

An $n \times n$ magic square contains numbers 1 through n^2, and all rows, columns, and diagonals sum to the same number. Consider the problem of generating all magic squares for a given n.

Usually with algorithms, there is an obvious, brute-force, and inefficient method. In this case, the brute-force method is to think of the magic square as a permutation of the entries. This method is $n!$ because it requires checking all permutations to see if they satisfy the properties of a magic square. This method might work at smaller values but does not scale, i.e., it will be extremely inefficient for larger values of n. We simplify by considering

a	b	c
d	e	f
g	h	i

The total sum of all magic square entries for a 3×3 magic square is $1 + 2 + \cdots + 9 = 45$, so the common sum is $\frac{45}{3} = 15$.

Lemma

For all 3×3 magic squares, we must have $e = 5$.

Proof. By adding up four rows, columns, or diagonals, we get $60 = (a + e + i) + (c + e + g) + (b + e + h) + (d + e + f) = (a + b + c + d + e + f + g + h + i) + 3e$, meaning $60 = 45 + 3e$, so $e = 5$. □

Based on the result of our lemma, we must have $a+i=b+h=c+g=d+f=10$. Thus, these values must come in pairs $(1,9),(2,8),(3,7),(4,6)$. We can consider two configurations of where we place the $(1,9)$ which account for all cases by rotational symmetry:

1	b	c
d	**5**	f
g	h	**9**

a	**1**	c
d	**5**	f
g	**9**	i

We notice that the diagonal configuration does not work. Suppose for the sake of contradiction that we could make a magic square. Then we must have $c<5$ because $c+f+9=15$ and $f>1$ but since $a+b+c=15$ and $b<9$, $c>5$. Using the uniqueness of magic square entries and their common sum, we arrive at a contradiction. If we were to analyze the second configuration, we would end with exactly 8 3×3 magic squares given rotational symmetry, a drastic simplification from 9! brute-force cases.

Example

A tennis ladder consists of a small group of people who play each other. In particular, consider five players a,b,c,d,e who all play each other, resulting in a tournament graph. Matches are played between every pair of nodes, and an outcome is represented by a directed edge, where $a\to b$ if player a beats player b. Given an arbitrary tournament, order players so each player has beaten the player *immediately* below him or her.

For example, consider three players, where a has beaten b who has beaten c who has beaten a. Then there is an ordering (abc) where each player has beaten the player immediately below them. First we need to show that every arbitrary tournament has such an ordering.

Lemma

Every tournament graph has a Hamiltonian path.

Proof. We proceed by strong induction. For our base case, $n=1$, we simply return the player, which gives a valid ordering. For our inductive hypothesis, we assume that there exists a Hamiltonian path (valid ordering) for a tournament graph with $n-1$ nodes.

Then, considering our recursive case, choose an arbitrary player i from the n players. We recursively solve the problem for the remaining $n-1$ players. By our inductive hypothesis, there already exists an ordering of the remaining $n-1$ players. We can insert player i with the following algorithm:

1. Go down the list to find the *first* player i has beaten.

2. Insert i above that player in the ordering.

3. If i has not beaten any of the players in the ordering, i is placed at the bottom of the ordering.

In terms of recursive calls, we can proceed in reverse lexicographic order (but any arbitrary selections will work). By induction and our recursive divide and conquer algorithm for constructing orderings, we have established not only that every tournament graph has a Hamiltonian path, but also a method for how we may construct it. □

Example

Suppose we have a list of words from the Webster dictionary arranged in lexicographic order and would like to sort the words so that all anagrams are next to each other. For example, we would like to sort abed, acme, bade, bead, came, and mace → abed, bade, bead, acme, came, mace.

The brute-force method would be to check if v and w are anagrams for all $v \neq w$ in the list, and if so, to move w next to v. This would require $O(n^2)$ anagram checks and each check would be $O(p^2)$ if there are p letters max in a word, meaning the algorithm would be $O(n^2p^2)$. Assuming our words were small enough, p would be constant meaning our algorithm would be $O(n^2)$.

Instead, we could sort each word in lexicographic order, e.g. "abed" would be sorted to "abde" and "bead" would also be sorted to "abde". Given anagrams (even with repeated letters), they will have the same sorted letters in lexicographic order. It takes $O(p \log p)$ (technically constant) time to sort each word, meaning sorting all words takes $O(np \log p)$ (technically $O(n)$) time. We can then sort the entire list of words (looking at the sorted lexicographic orders) in $O(n \log n)$ time to get exactly what we want.

§2 Recitation September 2, 2020

Definition

A **problem** is a binary relation connecting problem inputs to correct outputs. A (deterministic) **algorithm** is a procedure that maps inputs to single outputs (a function). An algorithm **solves** a problem if for every problem input it returns a correct output.

Note: The goal of this class is not only to teach how to solve problems, but to teach how to communicate that a solution to a problem is both correct and efficient.

Example

Given the students in your recitation, return either the names of two students who share the same birthday and year, or state that no such pair exists.

Algorithm

Maintain an empty record of student names and birthdays. Go around and ask each student their name and birthday. After interviewing each student, check if their birthday is already in the list. If yes, return the names of the two students; otherwise, add their name and birthday to the record. If after interviewing all students no pair is found, state so.

Key Idea

We need to prove **correctness** and **efficiency** of an algorithm.

Definition

Correctness: does your algorithm return the right answer for every input?

Definition

Efficiency: How many resources does your algorithm use? e.g. time and space complexity

One program is said to be *more efficient* than another if it can solve the same problem input using fewer resources.

Key Idea

We can compare algorithms against each other in terms of **asymptotic performance** relative to problem **input size**.

Fact

To calculate the resources used by an algorithm, we analyze a **model of computation** which models how long a computer takes to perform basic operations. We use a w-bit **Word-RAM** model:

- **computer**: random access array of machine words called **memory**, with a processor that can perform basic binary operations on the memory.
- **machine word**: sequence of w bits representing an integer from the set $\{0, \ldots, 2^w - 1\}$.

We can use **asymptotic notation** to ignore constants that do not change with the size of the problem input.

Definition (O Notation)

O **Notation**: Non-negative function $g(n)$ is in $O(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

This definition upper bounds the **asymptotic growth**

Definition (Ω Notation)

Ω **notation**: Non-negative function $g(n)$ is $\Omega(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $c \cdot f(n) \leq g(n)$ for all $n \geq n_0$.

This definition lower bounds the asymptotic growth.

Definition (Θ Notation)

When one function both asymptotically upper bounds and asymptotically lower bounds another, we utilize Θ **notation** for a **tight bound** on $g(n)$:

$$\Theta(f(n)) = \{g(n) : \text{ there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \text{ for all } n \geq n_0\}$$

We often use shorthand to characterize the asymptotic growth (i.e. **asymptotic complexity**) of common functions.

Example

Order the following functions by asymptotic complexity:

$$f_1 = 2^n$$
$$f_2 = 6006^n$$
$$f_3 = 2^{6006^n}$$
$$f_4 = 6006^{2^n}$$
$$f_5 = 6006^{n^2}$$

We rewrite $f_2 = (2^{\log_2 6006})^n = (2^n)^{\log_2 6006}$, $f_4 = (2^{\log_2 6006})^{2^n}$, and $f_5 = (2^{\log_2 6006})^{n^2}$. Using this base change allows us to compare the asymptotic complexity of the exponents and thus get the order f_1, f_2, f_5, f_4, f_3.

Definition (Data Structure)

A **data structure** is a way to store data that supports a set of operations to interact with that data. The set of operations supported by a data structure is called an **interface**.

Many data structures might support the same interface, but could provide different performance for each operation.

Example

The most primitive data structure native to the Word-RAM is the **static array**. A static array is simply a contiguous sequence of words reserved in memory, supporting a static sequence interface.

§3 Lecture September 3, 2020

§3.1 Algorithm Analysis & Recurrences

Example (Celebrity Problem)

A *celebrity* is defined among a group of n people as a person in the group who knows nobody but is known to everyone else. In terms of a graph, we can think of every node besides the celebrity having a directed edge (representing "knows") towards the celebrity, while the celebrity has no directed edges outwards.

Suppose that there is exactly one celebrity in the group:

Algorithm

Consider the following recursive algorithm with base case: $n = 1$, the single person is a celebrity; recursive step: S: n person group; arbitrarily pick $A, B \in S$, and ask if A knows B (an asymmetric relation).

- if A knows B, we perform a recursive call on $S - \{A\}$.
- if A does not know B, we perform a recursive call on $S - \{B\}$.

Now assume that there is no assumption regarding the existence of a celebrity. On the final step, our previous algorithm would now fail if A knew B and B knew A. One of these individuals would be declared incorrectly to be a celebrity. Similarly, our previous algorithm would fail if A and B were two strangers (neither knew the other). Then, consider the following extended algorithm:

Algorithm (Extended)

Consider the following recursive algorithm with base case: $n = 1$, the single person is a celebrity; recursive step: S: n person group; arbitrarily pick $A, B \in S$, and ask if A knows B (an asymmetric relation).

- if A knows B, we perform a recursive call on $S - \{A\}$.
  - If the recursive call returns B
    - if B knows A, return NONE
    - If B doesn't know A, return B as celebrity
  - If the recursive call returns $C \neq B$
    - if C knows A, return NONE
    - if C does not know A
      - if A knows C, return C as celebrity
      - if A does not know C, return NONE
  - If we get NONE for celebrity in $S - \{A\}$, then S has no celebrity.
- if A does not know B, we perform a recursive call on $S - \{B\}$, we proceed with similar logic

Let T_n be the number of questions asked. Then $T_n = T_{n-1} + 3$ (since 3 is the most number of additional questions that may need to be asked) for $n \geq 3$. We can express $T_n = 3n - 4$ for $n \geq 2$ since $T_1 = 1, T_2 = 2$.

Example (Polish National Flag Problem)

Given white pegs and red pegs in a line in some order, using a minimum number of swaps, move the pegs so all the red pegs are to the left of all white pegs.

Algorithm

A reasonable algorithm would be to swap the leftmost white peg and the rightmost red peg. This swap effectively shrinks the size of the problem that we're looking at. Explicitly, search for leftmost white peg, and search for rightmost red peg

- If the right peg is to the left of the red peg, swap pegs and repeat
- else, the algorithm is done

Our algorithm originally considers an n size set of pegs where there is a white peg on the left and a red peg on the right. In the worst case, this swap yields an $n-2$ size set of pegs. This gives the recurrence $T_n = T_{n-2} + 1$, which implies that $T_n = n/2$, where T_n is the worst-case number of swaps to perform on n pegs.

§4 Recitation: September 4, 2020

§4.1 Recurrences

Definition (Recurrence Relation)

A **recurrence relation** is an equation that recursively defines a sequence or multi-dimensional array of values, given some starting values.

When we say we solve a recurrence relation, we find a closed form for the recurrence.

Example (Fibonacci Numbers)

Recurrence Relation: $f_n = f_{n-1} + f_{n-2},\ f_1 = f_2 = 1$

Closed Form: $f_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$

Fact

In this class, we write recurrences in order to analyze the runtime of algorithms that break problems into smaller subproblems. We let $T(n)$ denote the amount of work it takes to solve the subproblem of size n.

$$T(n) = \langle \text{recursive step} \rangle + \langle \text{work done at current step} \rangle$$

Fact

There are three primary methods for solving recurrences:
Substitution: Guess a solution and substitute to show the recurrence holds.
Recursion Tree: Draw a tree representing the recurrence and sum computation at nodes. This is a very general method.
Master Theorem: A general formula to solve a large class of recurrences. It is useful, but can also be hard to remember.

Example

Solve the following recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

Assume $T(1) = \Theta(1)$.

Example (via Substitution)

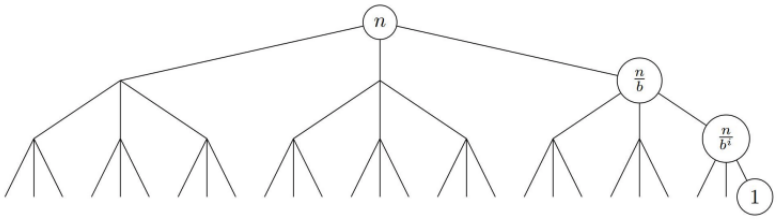
Guess: $T(n) = cn \log n$. Substituting gives:

$$\begin{aligned} cn \log n &= 2c(n/2) \log(n/2) + \Theta(n) \\ cn \log n &= cn \log n - cn \log 2 + \Theta(n) \\ cn \log 2 &= \Theta(n) \end{aligned}$$

Therefore $T(n) = \Theta(n \log n)$.

Recursion trees

We construct a tree. A general recursion tree might take the following form:



For example, if $T(n) = T(n-1) + O(n)$, at the n level, we have input n and do n work. We chain a tree downwards until we reach 1 (at the nth step), where at the k level, we have input k and do k work. The total amount of work is

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{(n)(n+1)}{2} = \Theta(n^2)$$

If we had an example like $T(n) = 3T(n-2) + \Theta(1)$, where we perform branching, e.g. the n node branches to 3 separate $n-2$ nodes. Each node contributes $\Theta(1)$ work and we step down $n/2$ levels, so the total work is

$$1 + 3 + 9 + \cdots + 3^{n/2} = \Theta(3^n)$$

Example (Recursion Tree)

Considering our example $T(n) = 2T(n/2) + \Theta(n)$, we branch out to two nodes from each node, and every level does n work. Then, we find the number of levels which is $\log n$ because we reduce our problem by 2 each time. Therefore, the total work done must be $n \log n$.

Definition (Master Theorem)

The Master Theorem provides a way to solve recurrence relations in which recursive calls decrease the problem size by a constant factor. Recurrences that take the Master Theorem generally take the form:

$$T(n) = aT(n/b) + f(n)$$

where $T(1) = \Theta(1)$, $a \geq 1$, $b > 1$. In the **special case** of the Master Theorem, we have

$$T(n) = aT(n/b) + n^c$$

where a is the "branching factor," b is the "problem size reduction factor," and n^c is the work done at each level. In general, the work at each level goes

$$n^c,\ a \cdot (n/b)^c,\ a^2 \cdot (n/b^2)^c,\ \ldots,\ a^{\log_b n} \cdot 1$$

where the final term equals $n^{\log_b a}$. So we compare c with $\log_b a$ and whichever has a larger value is the one that primarily controls the asymptotic complexity of the function.

Theorem 1 (Master Theorem Special Case)

Consider the **special case** of the Master Theorem:

$$T(n) = aT(n/b) + n^c$$

We have three cases:

case	solution	conditions	intuition
1	$T(n) = \Theta(n^{\log_b a})$	$c < \log_b a$	Work done at leaves dominates
2	$T(n) = \Theta(n^c \log n)$	$c = \log_b a$	Work balanced across the tree
3	$T(n) = \Theta(n^c)$	$c > \log_b a$	Work done at root dominates

Theorem 2 (Master Theorem General Case)

The **general case** $T(n) = aT(n/b) + f(n)$ also has three cases:

case	solution	conditions
1	$T(n) = \Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
2	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$	$f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$
3	$T(n) = \Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(n/b) < cf(n)$ for some constant $0 < c < 1$

Solving our recurrence via the Master Theorem, since $a = 2, b = 2, c = 1$, $\log_b a = c = 1$, we can apply the Special Case of the Master Theorem, Case 2. Thus, $T(n) = \Theta(n \log n)$. Continuing, we consider some more recurrence practice:

$$T(n) = T(n-1) + O(1)$$

Solution: $T(n) = O(n)$, length n chain, $O(1)$ work per node.

$$T(n) = T(n-1) + O(n)$$

Solution: $T(n) = O(n^2)$, length n chain, $O(k)$ work per node at height k.

$$T(n) = 2T(n-1) + O(1)$$

Solution: $T(n) = O(2^n)$, height n binary tree, $O(1)$ work per node.

$$T(n) = T(2n/3) + O(1)$$

Solution: $T(n) = O(\log n)$, length $\log_{3/2}(n)$ chain, $O(1)$ work per node.

Example

$$T(n) = 2T(\frac{n}{2}) + O(\sqrt{n})$$

Solution: $T(n) \in \Theta(n)$ by case 1 of the Master Theorem, since $O(\sqrt{n}) \subseteq O(n) = O(n^{\log_2 2})$

Example

Assuming $T(a) < T(b)$ for all $a < b$,

$$T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + \Theta(n)$$

Solution: By monotonicity, $T(n) \leq 2T(\frac{n}{3}) + \Theta(n)$. Then $T(n) \in O(n)$ by case 3 of the Master Theorem, since $\Theta(n) \subset \Omega(n^{\log_3 2+\varepsilon})$ for some $\varepsilon > 0$ (e.g. $\varepsilon = 1/3$). On the other hand, if we ignore the recursive component, $T(n) \in \Omega(n)$. Combining the two gives $T(n) \in \Theta(n)$.

Example

$$T(n) = T(\sqrt{n}) + O(1)$$

Solution: $T(n) = O(\log \log n)$.
The 'tree' is simply a chain where the node at depth i does $O(1)$ work, so the running time is simply $O(d)$, where d is the depth of the tree. The input at depth i is $\sqrt{\sqrt{\cdots \sqrt{n}}} = n^{1/2^i}$, and the tree bottoms out when this quantity becomes constant–say, less than 2:

$$n^{1/2^i} < 2 \iff 2^{2^i} > n \iff 2^i > \log n \iff i > \log \log n$$

So the tree has depth $\log \log n$, and $T(n) = O(\log \log n)$.

A different strategy is to substitute $n = 2^m$ and $S(m) = T(2^m)$; the recurrence becomes $T(2^m) = T(2^{m/2}) + O(1)$, i.e., $S(m) = S(m/2) + O(1)$. This has solution $S(m) = O(\log m)$, meaning $T(n) = S(\log n) = O(\log \log n)$.

§5 Lecture September 8, 2020

The quintessential search problem is to search for x in an ordered array A of n elements which has complexity $O(\log n)$.

1. Compare x with $A[\frac{n}{2}]$ and recursive left/right
2. O(1) access to A

Example (Peak Finding)
Consider an array $A[1..n]$. Element $A[i]$ is a **peak** if it is *not smaller* than its neighbor(s).

Algorithm

We can consider the brute-force algorithm:

1. Scan the array from left to right
2. Compare each $A[i]$ with its neighbors
3. Exit when a peak is found.

This has complexity $\Theta(n)$ and thus is not too efficient.

Algorithm

Compare the middle element with nieghbors.

- If $A[n/2-1] > A[n/2]$, then search for a peak among $A[1] \dots A[n/2-1]$
- Else, if $A[n/2] < A[n/2+1]$, then search for a peak among $A[n/2] \dots A[n]$
- Else, $A[n/2]$ is a peak.

We have for our recursive complexity

$$T(n) = T(n/2) + \Theta(1),$$

and unraveling this recursion,

$$T(n) = \underbrace{\Theta(1) + \Theta(1) + \cdots + \Theta(1)}_{\log n} = \Theta(\log n).$$

A complexity of $O(\log n)$ is much better than $O(n)$, and in particular $\log n < 300$ for all n of relevant size.

Fact

Divide and conquer *divides* input into multiple disjoint parts, *conquers* each of the parts separately (using the recursive call), and sometimes *combines* results.

Example (2D Peak Finding)

Consider a 2D array. A[i,j] is a *2D peak* if it is not smaller than its (at most 4) neighbors.

First notice that such a peak always exists because any maximum element (in the finite array) is a peak.

Algorithm (Brute-Force)

We can find a maximum element and return its coordinates by scanning through each row.

The complexity of the brute-force algorithm is $O(nm)$ where there are n rows and m columns of the array.

Algorithm

We try to recycle the bisection search from earlier:

- Pick middle column ($j = m/2$)
- Find a *global* maximum $a = A[i, m/2]$ in that column (and quit if $m = 1$).
- Compare a to its neighbors: $b = A[i, m/2 - 1]$ and $c = A[i, m/2 + 1]$
- If $b > a$ then recurse on left columns, because there is a peak (of the original A) in the left columns. The left columns have a maximum, M_l, and M_l must be a peak ($M_l \geq b \geq a \geq$ all elements in middle column).
- Else, if $c > a$ then recurse on right columns.
- Else a is a 2D peak.

Our complexity yields

$$T(n, m) = T(n, m/2) + \Theta(n)$$

so

$$T(n, m) = \underbrace{\Theta(n) + \Theta(n) + \cdots + \Theta(n)}_{\log m} = \Theta(n \log m)$$

Example

The problem of sorting gives an input array $A[1 \ldots n]$ of numbers and an output permutation $B[1 \ldots n]$ of A such that

$$B[1] \leq B[2] \leq \cdots \leq B[n]$$

Insertion Sort inserts key $A[j]$ into the already sorted sub-array $A[1 \ldots j - 1]$ by swapping $A[j]$ with adjacent elements until it is in the correct position. This complexity is $O(n^2)$ but does not require much extra space.

Merge Sort says

1. If $n = 1$, you are done (there is nothing to sort).
2. Otherwise, recursively sort $A[1 \ldots n/2]$ and $A[n/2 + 1 \ldots n]$.
3. "Merge" the two sorted sub-arrays.

For the merge step, we look at the two minimums of the two sorted sub-arrays, pick the smaller one, and place it in the output array. The time is $\Theta(n)$ to merge a total of n elements (linear time). The recursion is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

giving $T(n) = \Theta(n \log n)$.

§6 Recitation September 9, 2020

Definition

A **data structure** is a way to store data that supports a set of operations to interact with that data. The set of operations supported by a data structure is called an **interface**.

Definition (Sequences/Sets)

Sequences maintain a collection of items in an **extrinsic order**, where the ordering is done by an external party and is relevant. By contrast, **sets** maintain a collection of items based on an **intrinsic property** involving what the items are, usually based on a unique key (but otherwise order doesn't matter).

Fact (Sequence Interface Operations)

There are container, static, and dynamic operations in sequence interfaces. Static operations do not change the size of the data set, while dynamic operations do.

Fact (Set Interface Operations)

In set interfaces, we have container, static, dynamic, and order operations. Order operations consider some specific ordering of the set based on key.

We discuss three data structures to implement the sequence interface.

Array Implementation

We can think of static arrays as a chunk of memory. To build this array takes n time (the container operation). Getting and setting (static operations) takes constant time. When inserting or deleting elements, we are forced to copy the whole array and make this modification, which takes n time.

Linked List Implementation
A linked list is like a sequence of nodes, and a singly-linked list has a pointer to the next node. This means that the insert_first operation can be completed by adding a node, pointing it to the first node, and having the head node point to it, which takes constant time. However, every individual node does not know its own index, so we sacrifice getting and setting specific nodes (which becomes n time).

Definition (Amortization)
One drawback of using static arrays is the cost of re-allocating space every time you want to grow or shrink the array. One easy way to support faster insertion would be to over-allocate additional space when you request space for an array. Then, inserting an item would be as simple as copying over the new value into the next empty slot, e.g. when asked for an array of size n, provide an array of size $2n$. Sometimes, appending to a Python list will take $O(1)$ time if there is space (for the first n appends, but sometimes it could take $O(n)$ time (every $n+1$st append, but the total work is $2n$ for n appends, so insertion will take $O(1)$ time per insertion *on average*, meaning the asymptotic running time is **amortized constant time**, because the cost of the operation is **amortized** (distributed) across many applications of the operation.

Dynamic Array Implementation
A dynamic array adds on the end of the array, as described above, which has an improvement where the insert last method would have amortized constant time.

Consider the following summary:

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

Set Implementations
One naive way to implement sets would be a static array, which is extremely inefficient.

Certain operations would be easier if our elements were *in order*, so we can use the idea of sorting by keys first.

§6.1 Sorting

Selection Sort: Having already sorted the largest items into some sub-array $A[i+1:]$, the algorithm repeatedly scans the array for the largest item not yet sorted (iterating through $A[:i+1]$ and swaps it with item $A[i]$. The runtime of this algorithm is n^2.

Insertion Sort: Having already sorted sub-array $A[:i]$, the algorithm repeatedly swaps the initial item $A[i]$ with items to its left until it is in front of all elements that its smaller than. The runtime of this algorithm is n^2.

Fact

Both insertion sort and selection sort are **in-place** algorithms, meaning they can each be implemented using at most a constant amount of additional space (they are space efficient). Insertion sort is **stable**, meaning that items having the same value will appear in the sort in the same order they appear in the input.

As an example of insertion sort being stable, $[2, 1, 1'] \to [1, 1', 2]$, and the 1 and $1'$ are in the same order they were originally. In contrast, selection sort, we get $[1', 1, 2]$, so the sort is not stable.

Merge Sort: an asymptotically faster algorithm for sorting large numbers of items. The algorithm recursively sorts the left and right half of the array, and then merges the two halves in linear time, with $T(n) = 2T(n/2) + \Theta(n)$ and $T(n) = \Theta(n \log n)$, a growth rate much closer to linear than quadratic.

Fact

Our merge sort uses a linear amount of storage space, so it is **not in-place**. However it *is* possible to do merge sort in-place (more complicated). Our implementation of merge sort is **not stable**, but it can be made stable with only a small modification.

Selection Sort: $O(n^2)$ comparison but only $O(n)$ *writes*.

Insertion Sort: worst-case $O(n^2)$, but best case $O(n)$ (good if already/close to sorted).

Merge Sort: $O(n \log n)$, generally an all-around decent algorithm.

Fact

The sorts we've discussed so far are all **comparison** sorts, meaning that we're sorting by comparing ($a < b$, $a = b$, or $a > b$). Comparison sorts are $\Omega(n \log n)$ in the worst-case. This is because, given n elements, there are $n!$ permutations, and we can at most half it each time.

Fact

With a sorted array, we can use binary search to find keys in $O(\log n)$. We sacrifice some time in building in order to speed up subsequent queries, a common technique called **pre-processing**. This makes more sense if we want to look up more things.

§7 Lecture September 10, 2020

Definition (Hashing)

The goal of **hashing** is to efficiently maintain a set of numbers. For example, given a set S, we may want to apply operations to it (a whole data structure).

Linked Lists have $O(1)$ inserts at the front but $O(n)$ deletes or finds.

Direct Access (Array) (e.g. A[i] where A[i] equals 1, if element i is in the array) allows elements to be accessed in $O(1)$, meaning insert, find, and delete can all be performed in $O(1)$. However, this is only efficient if N, the max candidate integer in S, is small enough. If S is the set of genes you hear about, then $N \geq 2^{472}$.

Hashing Functions map $h : U \to \{0, \ldots, m-1\}$, the universe to the set of m elements. Instead of inserting 1 at address x (as in a direct access array), we insert x at address $h(x)$. Eventually, there may be a collision, i.e. if $h(x_i) = h(x_j)$ for $i \neq j$. This is guaranteed if $|S| > |Array|$, frequent if $|S| \approx |Array|$, and not likely if $|S| << |Array|$.

We cannot simply avoid collisions, so we consider **chaining**, which stores an array of linked lists. Instead of inserting x_1 in a single position, we put it in a linked list. When a collision occurs, we insert the element in front of the linked list where the collision occurs.

To find a number, we must scan the whole list in a number's bucket. A length L list costs L comparisons, so if all numbers hash to the same bucket, the lookup cost is $\Omega(n)$.

Definition (Simple Uniform Hashing)

If we are optimistic, we assume numbers in S are equally likely to land in every bucket, independently of where other numbers in S land. We refer to this principle as **simple uniform hashing**.

If this assumption is accurate, we may perform average case anaylsis under SUHA (simple uniform hasing assumption). We have n keys in m buckets, so the average number of keys per bucket is the **load factor** $\alpha = \frac{n}{m}$. The expected worst-case time to find some key x is $1 + \alpha$. Then if $\alpha = O(1)$, we can find some key in $O(1)$ time.

In reality, x's are often nonrandom, so an idea is to pick a hash function $h(\cdot)$ whose values "look" random or are random "enough." A popular such function is the division hash function which maps

$$h(x) = x \pmod{m}$$

If $m = 2^r$, then $h(x)$ is the last r bits of x, which is a fast function, but not necessarily random (depending on keys). A very popular division hash function then is using $m = p$ prime.

Example

Given two sets of n numbers, $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, find $A \cap B$.

Algorithm 1: A brute-force algorithm would be to compare each $a_i \in A$ with each $b_j \in B$ which is $O(n^2)$.

Algorithm 2: Sort A and then search A for each $b_j \in B$, which is $O(n \log n)$ to sort A and then $O(n \log n)$ to search for each element, so the overall algorithm is $O(n \log n)$.

Algorithm 3: Hash each $a_i \in A$ in an array T of size n. Then, hash each $b_j \in B$ into the same T. Each collision belongs to $A \cap B$. It takes $O(n)$ time to hash A, and it also takes $O(n)$ time to hash B. Determining collisions takes $O(1)$ time, and collecting the set of collisions takes $O(n)$ time.

Generalizing, let S consist of 2-field elements, where a *key* is a unique identifier (like a word) and *data* is associated with this key (like meaning). In dynamic dictionaries, we can let S be the set of **pointers**, whose elements are key-data pairs. If x_i are the pointers, we can use key(x_i) to denote the corresponding key.

Open Addressing is a process that doesn't created linked lists: if a bucket is occupied, find another bucket (need $m \geq n$). For insert, we *probe* a sequence of buckets until we find an empty one, where h specifies a probe sequence for key x. For example, if a collision occurs when we try to hash $h(k, 1)$ we increment the second input, $1 \to 2$ until there is no collision, e.g., $h(k, 4)$ might be the first hash that has no collision. When deleting an element, we must leave it in the bucket, but mark it *deleted*.

§8 Recitation September 11, 2020

Definition (Direct Access Array)

We define a data structure called a **direct access array**, which is a normal static array that associates a semantic meaning with each array index location: specifically that any item x with key k will be stored at array index k.

Storing a set of n items with a direct access array, if the items have *unique* integer key in a bounded rage, we can store items in a direct access array where each array slot contains an item associated with a key, if it exists. Then a search algorithm can look in the array slot to respond to the search query in worst-case constant time.

However, this comes at the cost of storage space, because every possible key must have a slot available. We overcome this obstacle with **hashing**.

Fact

A possible solution to reduce space would be to store items in a smaller **dynamic direct access array**. To make this work, we need a function that maps item keys to different slots of the direct access array, $h(k) : \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$ for $u > m$ and m is linear with respect to n. We call such a function a **hash function**, or a **hash map**. The smaller direct access array is called a **hash table**, and $h(k)$ is the **hash** of integer key k.

If the hash function if injective over the keys, then we don't have to handle collisions and the runtime for searches is constant. However, if two items associated with keys k_1, k_2 hashing to the same index, $h(k_1) = h(k_2)$, the keys **collide**.

Definition (Chaining)

Chaining is a collision resolution strategy where colliding keys are stored separately from the original hash table. Each hash table index holds a pointer to a **chain**, a separate data structure that supports the dynamic set interface (like a linked list).

Consider the following hash functions:

Division Method (bad): The simplest mapping of an integer key domain of size u to a smaller one of size m is $h(k) = k \pmod m$. Ideally, the performance of our data structure would be **independent** of the keys we choose to store.

Universal Hashing (good): We can achieve good **expected** bounds on hash table performance by choosing our hash function **randomly** from a large family of hash functions. Here the expectation is over our choice of hash function, which is *independent* of input. One *family* of hash functions that performs well is:

$$\mathcal{H}(m, p) = \{h_{ab}(k) = (((ak + b \bmod p) \bmod m) \mid a, b \in \{0, \ldots, p-1\} \text{ and } a \neq 0\}$$

where p is a prime that is larger than the key domain u. A single hash function from this family is specified by choosing concrete values for a and b. This family of hash functions is **universal**: for any two keys, the probability that their hashes will collide when hashed

using a hash function chosen uniformly at random from the universal family, is no greater than $1/m$, i.e.

$$\Pr_{h\in\mathcal{H}}\{h(k_i) = h(k_j)\} \leq 1/m, \quad \forall\, k_i \neq k_j \in \{0,\ldots,u-1\}.$$

Then, we can upper bound the expected size of any chain, in expectation over our choice of hash function from the family, by summing up over the number of expected collisions. Letting X_{ij} be the indicator random variable representing the value 1 if keys k_i and k_j collide for a chosen hash function, and 0 otherwise, and letting $X_i = \sum_j X_{ij}$ be the random variable representing the number of items hashed to the index $h(k_i)$ (summed over all keys k_j), the expected number of keys hashed to the chain at index $h(k_i)$ is:

$$\begin{aligned}
\mathop{\mathbb{E}}_{h\in\mathcal{H}}\{X_i\} = \mathop{\mathbb{E}}_{h\in\mathcal{H}}\{\sum_j X_{ij}\} &= 1 + \sum_{j\neq i} \mathop{\mathbb{E}}_{h\in\mathcal{H}}\{X_{ij}\} \\
&= 1 + \sum_{j\neq i} (1) \Pr_{h\in\mathcal{H}}\{h(k_i) = h(k_j)\} + (0) \Pr_{h\in\mathcal{H}}\{h(k_i) \neq h(k_j)\} \\
&\leq 1 + \sum_{j\neq i} 1/m = 1 + (n-1)/m.
\end{aligned}$$

If the size of the hash table is at least linear in the number of items stored, i.e. $m = \Omega(n)$, then the expected size of any chain will be $1 + (n-1)/\Omega(n) = O(1)$ a constant. Thus, dynamic set operations will be performed in **expected constant time**, independent from the input keys. All together,

	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
Data Structure	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n

§9 Lecture September 15, 2020

Previously, we discussed merge sort which takes $O(n \log n)$ time. Today, we discuss linear sorting, which takes $O(n)$ time for integers in a polynomially bounded range i.e. $< n^c$ for some constant c.

Fact (Comparison Model)

Generally, a comparison sort can only do comparisons (for any comparison function) between keys to determine their relative order). The keys are black boxes.

We consider a lower bound for comparison sorting by just counting the number of comparisons (which serves as a lower bound on the running time of the whole algorithm). We represent our algorithm using a binary decision tree. For example, our root node might compare $A[i] < A[j]$ which returns True (and then compares $A[i] < A[k]$) or False (and then compares $A[k] < A[j]$). Internal nodes correspond to comparisons while leaf nodes correspond to some output (for a sorting algorithm, this is a permutation of the input in sorted order).

The worst case time is $\geq$ the length of the longest path in the binary decision tree (i.e. the height of the tree). Supposing we have L leaf nodes, our minimum height (shortest possible length of longest path) is $\Omega(\log L)$. The number of outputs for a sorting algorithm on n distinct keys is $n!$ (our output is a length n array in any order). Then, we must have $L \geq n!$ (each potential output must be in a leaf node).

Then, the minimum height of our binary decision tree is $\geq \Omega(\log(n!)) = \Omega(n \log n)$, since $n! \geq (\frac{n}{2})^{n/2}$, and thus $\log(n!) \geq \frac{n}{2} \log(\frac{n}{2}) = \Theta(n \log n)$. Thus, comparison sorting takes at least $\Omega(n \log n)$ time. This demonstrates that merge sort is optimal in the comparison model.

Algorithm 3 (Direct Access Array (DAA) Sort)

Suppose our input is n non-negative, unique integers whose values are all $< u$. We can

1. Make and initialize DAA of length u
2. Scan input array and for each key, mark corresponding index in DAA
3. Scan DAA and output indices corresponding to marked slots

As an example, consider $A = [5, 2, 7, 0, 4]$ with $u = 10$. Then, we can initialize a direct access array with indices 0 through 9 and mark the indices $0, 2, 4, 5, 7$, and scanning through the DAA, output the marked indices. The runtime of this algorithm is $O(u)$ to create the DAA, $O(n)$ to mark the slots, and $O(u)$ to scan the DAA, taking a total of $O(n + u)$ time.

If $u = \Theta(n)$, then the DAA sort takes $O(n)$ time, thus providing a linear time sorting algorithm. On the other hand, if $u \approx n^2$, our runtime would be $O(n^2)$, worse than merge sort. This leads to our next idea, tuple sort.

Algorithm 4 (Tuple Sort)

Map each key $k < n^2$ to tuple (a, b) where $a = \lfloor \frac{k}{n} \rfloor$ and $b = k\%n$, so $k = an + b$. Then, a, b are both $< n$.

As an example, consider $A = [17, 3, 24, 22, 12]$ with $n = 5$, which maps to $[(3, 2), (0, 3), (4, 4), (4, 2), (2, 2)]$. We can sort in two methods, by considering the most significant (first) digit first or least significant digit first.

Most significant digit first: (0,3) (2,2) (3,2) (4,4) (4,2) (which is a **stable sort** i.e. for equal valued keys, they appear in the same order in the output as they did in the input). Next, sorting by the least significant digit yields (2,2) (3,2) (4,2) (0,3) (4,4) (still a stable sort), which is not sorted.

Least significant digit first: (3,2) (4,2) (2,2) (0,3) (4,4) is the first output, and sorting by the most significant digit yields (0,3) (2,2) (3,2) (4,2) (4,4), which is in sorted order. This method guarantees that most significant digits are in order, and least significant digits are as well, because the sort is stable.

Algorithm 5 (Counting Sort)

We use a DAA but store multiple keys per slot (this is basically a combination of tuple sort and DAA sort).

For example, using the same example from above, we store values in DAA based on least significant digit (so some tuples will be in the same slot), and we make sure to put them in the slots in order to maintain stability. The running time is $O(u)$ to initialize the DAA, $O(n)$ time to scan the input array, and $O(u)$ time to scan the DAA, which takes $O(n + u)$ time. In this case, $u = O(n)$, which gives us a linear time sort.

Algorithm 6 (Radix Sort)

Given an input of n non-negative integers whose values are all $< n^c$ for some value of c,

1. Create tuple $(k_1, k_2, \ldots, k_c)$ for each original key k:

$$k = k_1 \cdot n^{c-1} + k_2 \cdot n^{c-2} + \cdots + k_{c-1} \cdot n + k_c$$

   where each $k_i < n$.

2. Apply Tuple Sort (using the least significant digit first) where each individual sort is a counting sort.

3. Each counting sort takes $O(n)$ time and we need c counting sorts

Therefore, the total runtime of the algorithm is $O(cn)$, and if c is constant, then this is $O(n)$.

§10 Recitation September 16, 2020

A quick review of Linear Sorting:
Counting Sort: insert into DAA with chains and read out items in order

Tuple Sort: repeatedly sort items in importance order using some *stable* auxiliary sort, from least to most important key

Radix Sort: if $u = O(n^c)$, write each number as a c-digit base n number and use tuple sort with auxiliary counting sort
Consider a comparison of sorts:

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k-proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$
Radix Sort	$n + n \log_n(u)$	N	Y	$O(n)$ when $u = O(n^c)$

§11 Lecture September 17, 2020

We begin our discussion of *binary trees*. Each node in a binary tree stores an *item* and has a *parent* pointer as well as a *left* and *right* child (it is helpful to think about this with a diagram). The root node doesn't have a parent, and the leaf nodes don't have any children. If a node doesn't have a parent or children, their parent or children (whichever they don't have) are referred to as *null*.

Definition (Root, Ancestors, Descendants, Subtree, Depth, Height, Traversal Order)

To assist our understanding of binary trees, we consider the following definitions.

- The **root** of the tree is the only node in the tree lacking a parent. All other nodes in the tree can reach the root by traversing parent pointers.
- The **ancestors** of a node x are all nodes "above" x, i.e. those that can be reached via parent pointers.
- The **descendants** of a node x are all nodes "below" x, i.e. those that can be reached via children pointers.
- The **subtree** of a node x is the tree rooted at x containing x and all its descendants
- The **depth** of a node x is the number of ancestors it has i.e. the number of edges on a path from x to the root.
- The **height** h of a node x is the number of edges in the longest downward path from x. The height of a tree is equivalently the height of the root.
- The **traversal order** (also known as *in-order* traversal order) of nodes satisfies the property that for each node x, all nodes in the left subtree are before x and all nodes in the right subtree are after x.

Algorithm (Determining Traversal Order)

Iter(x):
  If $x.left$:
    Iter($x.left$)
  Output x
  If $x.right$:
    Iter($x.right$)

The runtime of this algorithm is $O(n)$ where n is the number of nodes in the tree (each node is parsed through Iter once).

Algorithm (subtree_first, subtree_last)

Consider subtree_first(x) which returns the first node in traversal order of x's subtree. In general, this can be accomplished by going as far left as possible i.e. setting $x = x.left$ (the left node), and returning the last node seen. We could similarly define subtree_last(x) which returns the last node in traversal order of x's subtree, which goes right until it no longer can.

The runtime of these methods are $O(h)$, where h is the height of the tree.

Algorithm (successor)

The method successor(x) returns the node immediately after x in traversal order, i.e. its **successor**. if $x.right$ is not null, we return the first node in x's right subtree, by calling subtree_first($x.right$).

Otherwise, we walk up the tree ($x = x.parent$) until we go up the left branch of parent, i.e. $x == x.parent.left$, or there is no parent (arrived at the root), in which case we return NONE.

The runtime of this method is $O(h)$, where h is the height of the tree. We can also define a predecessor() method, which is symmetric to the successor() method, returning the **predecessor**, the immediately preceding node in traversal order.

Fact (Dynamic Operations)

There are three dynamic operations we would like to implement (in each case, we preserve the traversal order of original nodes relative to each other):

- insert_after(existing, new): insert new node into tree such that the new node is immediately after the existing node in traversal order.
- insert_before(existing, new): insert new node into tree such that the new node is immediately before the existing node in traversal order.
- delete(x): remove x from the tree such that the traversal order of remaining nodes is preserved.

For insert_after(existing, new), if existing.right == NONE, we make the new node the right child of the existing node. If existing.right is not null, we make new the left child of subtree_first(existing.right). This also takes $O(h)$ time, stemming from the case where existing.right is not null. We can define insert_before(existing, new) symmetrically.

For delete(x), if x is a leaf node of the tree, we can just remove the node (detach it from its parent) without affecting the remaining traversal order. If x is not a leaf node, on the other hand, we can swap x with its predecessors ($x.left$) *or* successors ($x.right$) (one or the other) until it is a leaf node. This causes x be out of place in traversal order, but the other nodes still have the same relative order. Once x is a leaf node, we can delete it without altering relative traversal order.

Algorithmically, we could write this as swapping x's key with its predecessor/successor's key, and then recursing on the predecessor/successor. The runtime of this algorithm is $O(h)$ because we go down at most h levels, doing constant work at each level.

§12 Recitation September 18, 2020

Definition (Binary Tree)

A **binary tree** is a tree (a connected graph with no cycles) of binary nodes: a linked node container, similar to a linked list node, having a constant number of fields:

- a pointer to an item stored at the node,
- a pointer to a parent node (possibly None),
- a pointer to a left child node (possibly None), and
- a pointer to a right child node (possibly None)

We call the nodes "binary" based on the number of children the node has. We cover traversal order, which is mentioned in class (see above). Next, we cover *tree navigation* which consists of the subtree methods we covered in class.

Fact

To use a Binary Tree to implement a Set interface, we use the traversal order of the tree to store the items sorted in increasing key order. This property is often called the **Binary Search Tree Property**, where keys in a node's left subtree are less than the key stored at the node, and keys in the node's right subtree are greater than the key stored at the node. Then finding the node containing a query key (or determining that no node contains the key) can be done by walking down the tree, recursing on the appropriate side.

Definition

An **augmentation** is stored information about the *subtree*. We augment a tree using a **subtree property**, a property stored at every node that gives information about the subtree, allowing us to determine the property at the root in $O(1)$.

For example, suppose we are looking for the *maximum* item in a tree. The property of each of the leaves is just the numbers themselves (because there are no subtrees). We can proceed upwards along the tree to determine the subtree properties at each node.

Subtree properties can (and need to be) calculated in $O(1)$ from children and self, allowing us to update our tree in time relative to the height of the tree. If we were to insert a leaf node, we would need to perform *maintenance*, by recursively updating up the tree, only affecting ancestors (only fixing $O(h)$ nodes).

§13 Lecture September 22, 2020

Recall that the property of a binary search tree (BST) is that all elements in the left subtree of a node are less than the item at the node, and all elements in the right subtree of the node are greater than the item at the node.

Fact

Searches in binary search trees are performed.. well in a binary search manner. At a given node, we proceed to the left child if the value we are seeking is less; otherwise we proceed to the right child if the value we are seeking is greater (until we either reach a nonexistent child or the value we are searching for). Insertion can be performed by running a search and then adding in an element in the resultant nonexistent child position.

To write the *in-order traversal* order, we start from the very left, and at each step, pop up, and then completely explore the right subtree of the pop up node, and then pop up again, and repeat the process. In a binary search tree, this in-order traversal is sorted.

Algorithm (BST Sort)

Each insert performs in $O(h)$, so n inserts runs in $O(nh)$. However, the worst case would be a string in reverse, which ends up with $O(n)$ height, meaning the algorithm would run in $O(n^2)$.

To improve this, we can consider a more efficient tree, an AVL tree.

Definition (AVL Property)

An AVL tree contains a property containing the height of a subtree at each node (the height of a null node is -1), and the **AVL Property** allows a skew (difference between heights of children of the same node) of at most 1, i.e., left and right child differ in height by at most ± 1.

In a complete binary tree (all nodes have two children except for the leaves) of height h, the number of nodes is

$$N_h = 2^{h+1} - 1$$

where $h = O(\log N_h)$.

AVL Jenga

Given a complete binary tree, we can remove nodes from the bottom of the tree while maintaining skew. The goal is to find the fewest nodes with the same original height.

Representing the worst case for number of nodes left as N_h (every node has its children differ by ± 1), we can write

$$N_h = N_{h-1} + N_{h-2} + 1$$

where the left node has height $h-1$ (without loss of generality), the right node has height $h-2$, and we must factor in the root node. Then,

$$N_h > 2N_{h-2}$$

meaning

$$N_h > 2^{h/2}$$

which implies that

$$h < 2 \log_2 N_h,$$

so the height of an AVL tree is $O(\log n)$.

Algorithm (AVL Insert)

If we try to perform a normal insert method, heights change and may result in a skew greater in magnitude than ± 1. So, we insert and search for the lowest node that violates the AVL property.

We could have what's called a *right-right* case, where the lowest node violating the AVL property has two *right-heavy* children on the right, in which case we perform a left rotated on the node. So, if we originally had a node n with right child $r1$ with right child $r2$, we rotate left so that $r1$ is a node with left child n and right child $r2$.

We could also have what's called a *right-left* case, where the lowest violated node is *right-heavy* and it's right child is *left-heavy* (let these be n, r, l). Then, we right rotate r and then left rotate on n.

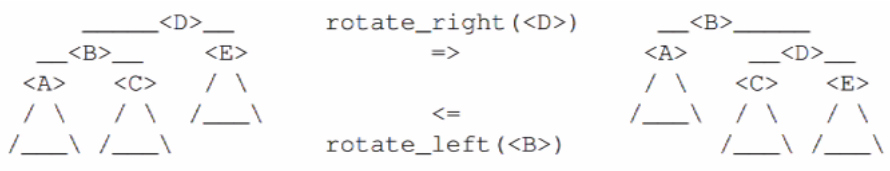
§14 Recitation September 23, 2020

Fact

The ultimate goal is to keep a binary tree balanced: a tree on n nodes is balanced if its height is $O(\log n)$. Every node of an AVL Tree is **height-balanced** (i.e., satisfies the **AVL Property** where the left and right subtrees of a height-balanced node differ in height by at most 1). A tree is height-balanced if every node is height-balanced.

Definition (Rotations)

We can change the structure of a tree using a local operation called a **rotation**. A rotation takes a subtree that locally looks like one of the following two configurations and modifies the connections between nodes in $O(1)$ time to transform it into the other configuration.



This operation preserves the traversal order of the tree. Suppose we perform an insertion or deletion. Given a subtree whose root has skew 2 and every other node in its subtree is height-balanced, we can restore balance to the subtree in at most two rotations. Thus, to rebalance the entire tree, it suffices to walk from the leaf to the root, rebalancing each node along the way, performing at most $O(\log n)$ rotations.

Fact

In general, the idea behind augmentation is to store additional information at each node so that information can be queried quickly in the future. To augment the nodes of a binary tree with a subtree property $P(\langle X \rangle)$, you need to

- clearly define what property of $\langle X \rangle$'s subtree corresponds to $P(\langle X \rangle)$
- show how to compute $P(\langle X \rangle)$ in $O(1)$ time from the augmentations of $\langle X \rangle$'s children.

Example

Maintain a sequence of n bits that supports two operations, each in $O(\log n)$ time:

- flip(i): flip the bit at index i
- count_ones_upto(i): return the number of bits in the prefix up to index i that are one

We can maintain a sequence tree storing the bits as items, augmenting each node A with A.subtree_ones, the number of 1 bits in its subtree. We can maintain this augmentation in $O(1)$ from the augmentations stored at its children. To implement flip(i), we can find the ith node using subtree_node_at(i), flip the node, and update the augmentation upwards in $O(\log n)$.

To implement the second method, we let subtree_count_ones_upto(A, i) return the number of 1 bits in the subtree of node A that are at most index i within A's subtree. Then count_ones_upto(i) is symantically equivalent to subtree_count_ones_upto$(T.root, i)$. Since each recursive call makes potential use of stored subtree_ones() and at most one recursive call on a child, the operation takes $O(\log n)$ time.

This problem is referred to as a *prefix* problem (everything up to i, i.e. $[: i]$) (there are also *suffix* problems $[i :]$).

§15 Lecture September 24, 2020

Definition (Priority Queues)

A **priority queue** consists of a set S of elements each associated with a key. Some methods we may want to apply include insert(S,x), return max(S), extract_max(S), increase_key(S,x,k) (set the priority of x to k).

When considering binary heaps, we can visualize the array as a nearly complete binary tree.

- max_heap Property: the key of a node is $\geq$ the keys of its children. We can visualize as an array because the root is always in index 1 of the array, and then we proceed by level, numbering nodes with indices from left to right in each level.
- We can then return parent(i) in constant time as $\lfloor \frac{i-1}{2} \rfloor$. Similarly, left$(i) = 2i + 1$ while right$(i) = 2i + 2$ (assuming array index starts with root at 0).

Algorithm (max_heapify())

max_heapify: corrects a single violation of the heap in a subtree at its root (subtrees of roots are max heaps, but the root violates the max heap property). We can do this by repeatedly swapping the item in the root node with the bigger child until it doesn't violate the max heap property, assuming that trees rooted at left and right childern are max_heaps.

We can perform this algorithm either going up or down. The up method is used for insertion (appending to end of array) while the down method is used for deletion.

Algorithm (build_max_heap())

Build_max_heap(A): turns an unordered array (in the form of a heap) into a max heap. Notice that all indices greater than $\frac{n}{2}$ are going to be at the bottom (leaves that are themselves already max heaps), and we can use them to build up. Then, for $i = n/2$ down to 1, we apply max_heapify(A,i), treating i as a root node.

The total amount of work can be calculated by the max number of swaps at each level, which yields

$$\frac{n}{4}(1) + \frac{n}{8}(2) + \frac{n}{16}(3) + \cdots + 1(\log n - 1)$$

Setting $n = 2^k$ gives

$$2^k \left(\underbrace{\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \cdots + \frac{k-1}{2^k}}_{O(1)} \right)$$

meaning the runtime is $O(n)$. This procedure does not affect the *asymptotic* efficiency of heap sort, but it is *practically* more efficient to initially insert n items into an empty heap.

Algorithm (Heap Sort)

We can build a max heap from an unordered array in $O(n)$. Then, we can find the max element $A[1]$, swap elements $A[n]$ and $A[1]$, discard node n from the heap (decrementing the heap size variable), and we can run max_heapify once on the new root node (since its children are still max heaps). We perform this iteration until the list is empty.

The complexity of this sort is $O(n \log n)$ (and notice this makes sense since it is still a comparison sort).

§16 Recitation September 25, 2020

In priority queues, you are most concerned with determining the max element at any point efficiently.

Idea: If we are able to efficiently insert and delete elements, we can sort elements in a set simply by inserting all of our elements into a priority queue and repeatedly deleting the max element.

Fact

Both selection sort and insertion sort are examples of priority queue sorts.

Definition

A **binary heap** is a data structure that is a **complete binary tree** (every potential node to the left of a node and above the node should come before the node). Elements are stored in order based on a heap property: at every node, the key of the item at that node is either $\geq$ or $\leq$ that of its children, and consequently every node in its subtree.

Binary heaps take advantage of the logarithmic heights of complete binary trees.

§17 Lecture September 29, 2020

Definition (Graph)

A **graph** $G = (V, E)$ considers of V a set of **vertices** ($|V| = n$) and $E \subset V \times V$ a set of **edges** (pairs of vertices) ($|E| = m$).

There are both undirected and directed graphs (where edge sets are either not ordered or ordered).

Definition (Degree)

In an undirected graph, **degree** of a node x is the number of nodes connected to x.

$$degree_x = k$$

if there are k nodes y such that $\{x, y\} \in E$.

$$\sum_{x \in V} degree_x = 2m$$

In a directed graph, we consider instead outdegree and indegree.

$$\sum_{x \in V} outdegree_x = \sum_{x \in V} indegree_x = m$$

Graphs can model lots of things, such as friendship, powergrids, maps, rubik's cube, etc.

Example

We can represent graphs with adjacency lists or adjacency matrices.

Adjacency lists are essentially an array of linked lists (each linked list contains the neighbors of a given node). A graph is *sparse* when $|E| = O(n)$, in which case an adjacency list is more economical (and more popular). However, for a *dense* graph $|E| = O(n^2)$, an adjacency matrix would have a faster edge check.

Definition (Subgraphs, Paths, Connected, Distance)

Consider the basic graph notions:

- A **subgraph** of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that

$$V' \subseteq V \text{ and } E' \subseteq E$$

- A **path** in G is a sequence of vertices $P = s_0, s_1, \ldots, s_k$ such that for all i, $(s_i, s_{i+1}) \in E$. P is **simple** if and only if none of its vertices are equal. If there is a path P from s_0 to s_k, s_0 is **connected** to s_k, with path **length** k (counting edges).
- G is **connected** if each pair of vertices in V are connected.
- A **connected component** of G is a maximal (vertex-wise) connected subgraph of G.
- The **distance** between s and t is the length of the minimum length s-to-t path.

Example

Some basic graph problems include

- Find the distance from s to every other vertex of G
- Find whether G is connected
- Find the connected component to which a vertex s belongs
- Find the connected components of G

Assuming no edge weights, these problems can all be tackled with the Breadth First Search algorithm.

Algorithm 7 (Breadth First Search (from s))

Initially $i = 0$, s is marked 0 and all other vertices are unmarked (or marked "∞")

1. Mark $i + 1$ for all unmarked neighbors of vertices marked i. If none, stop.
2. Increase the counter by 1 ($i = i + 1$) and go back to Step 1.

Upon termination, each vertex is marked with its distance from s.

Lemma

If t is marked i, then there is a length-i path from s to t.

Proof. **Base Case:** $i = 0$. In this case, the lemma is true because s is the only vertex marked 0 and there is a length-0 path from s to s.

Inductive Step: t is marked $i+1$ implies that there exists $P \equiv s, x_1, \ldots, x_i$ (by the inductive hypothesis) and $\{x_i, t\} \in E$. Then, $s, x_1, \ldots, x_i, t$ is a path with length $i+1$. □

Now, we show that there is no shorter path than the distance marked.

Proof. Assume for the sake of contradiction, that there is i, some minimum integer with t a vertex marked i but whose distance from s is $d < i$. Then there is a path $P \equiv s, x_1, \ldots, x_{d-1}, t$, meaning the distance of x_{d-1} is $d-1$ so x_{d-1} is marked $d-1$ (since t is the closest counterexample). Then, t is marked d, so t is marked $< i$, leading to a contradiction. □

Therefore, BFS finds min distances (from s). Consider the complexity:

- Marking node x: $O(1)$ (if it is unmarked)
- Insert x in i-marked list: $O(1)$
- Processing x: $O(degree_x)$

Each node is marked only once, and each node is marked, inserted, and processed once, so the complexity of BFS is $\Theta(n+m)$. This process yields an n-node **tree**, i.e. an n-node connected graph with no cycles (because each node is marked only once). In fact, this is a spanning tree.

Definition (Spanning Tree)

A **spanning tree** is a subgraph (V', E') that is a **tree** with $V' = V$.

§18 Lecture October 6, 2020

We can think of depth first search, DFS as walking on a graph with a pen (to number nodes and mark edges) and a flag (representing your center of activity).

Algorithm 8 (Depth First Search)

Essentially, we always traverse unmarked edges to reach an unnumbered node, and if at any point we have no valid options, we move back along our path until we again have valid options.

- Mark all edges "unused"
- Increase the counter
- If the counter has no unused edges, check a later step to see if it is time to halt
- Choose an unused edge, mark it, and traverse it (if it is not numbered, otherwise we backtrack and mark the node complete)
- If at the initial node, we halt

Lemma

If you number a node, you'll eventually backtrack from it.

Proof. The graph is finite yo - QED by Silvio. □

Theorem 9

If the graph is connected, you will number all its vertices.

Proof. Assume for the sake of contradiction that we have an unnumbered node. We also have a numbered node, namely the starting node. Since the graph is connected, there exists a path from the numbered node to the unnumbered node. We label the first unnumbered node along this path. Then, there is a final numbered node in the path i which has an unmarked edge to the first unnumbered node. However, by our lemma, we would have backtracked from i, but then we ought to have traversed the unmarked edge, leading to a contradiction. □

We discuss complexity intuitively. For each edge in our graph, we mark it once $O(1)$, traverse it twice $O(1)$, and backtrack once $O(1)$, leading to a total $O(m)$. For each node in our graph, we number once $O(1)$, bring center of activity once $O(1)$, and remove center of activity once $O(1)$, for a total $O(n)$. Then, the total complexity is $O(n+m)$.

Fact

A DFS tree consists only of tree edges, the edges that reach new nodes. Back edges must lead to an ancestor node in the tree. The numbering is unnecessary, only the DFS is.

Example

Directed DFS performs in the same way but simply uses directed edges rather than undirected edges.

We have the same lemma that if you number a node, you'll eventually backtrack from it. We also have an essentially identical theorem that you will number all the vertices reachable from S, with the same proof by contradiction.

Fact

In a directed DFS tree, tree edges point to children and back edges point to ancestors. We can also have forward edges to lower numbered non-ancestors or higher numbered descendants in our original graph.

Example (Topological Sort)

A topological sort is a numbering of the vertices of a directed acyclic graph (DAG) such that all edges go from low to high, i.e., if $u \to v$ then $TS(u) < TS(v)$.

Example (Topological Reverse)

A topological reverse is a numbering of the vertices of a directed acyclic graph (DAG) such that all edges go from high to low (opposite of topological sort).

Each of these would take $O(n!m)$ to brute-force. Instead of numbering in the order things are discovered in our DFS, we number in the order that we backtrack, i.e., number nodes based on when you backtrack from them. This gives us a topological reverse; we can redefine to a topological sort by applying

$$TS(x) = n - TR(x)$$

The complexity of this algorithm is $O(n + m)$. Furthermore, any DAG has a topological sort, because we can always use our DFS algorithm with our new numbering system.

§19 Recitation October 7, 2020

In this class, a *path* can actually repeat vertices (a small distinction). One way we can represent a graph is with an adjacency matrix. A drawback of this representation is determining whether a graph contains a given edge.

We could also hash each vertex to its neighbors in a hash table.

Algorithm (Breadth First Search)

A **breadth-first search** (BFS) from s discovers the **level sets** of s: level L_i is the set of vertices reachable from s via a *shortest* path of length i (not reachable via a path of shorter length).

This is a single-source shortest path (SSSP). Every node in BFS has a parent pointer, indicating which parent it came from in the BFS (the parent pointer of our source node is itself). We also have a level set, and while our previous level is not empty, we iterate through the nodes in the previous level and check their adjacent nodes, and append if they are not already labeled.

We return the set of parent pointers, because it tell us exactly how we performed our BFS. Using these parent pointers, we can exactly determine the shortest path to a node (and determine whether or not it exists).

Example

Given an unweighted graph $G = (V, E)$, find a a shortest path from s to t having an *odd* number of edges.

We use a concept called *graph duplication* (involving some kind of state), in this case considering if we are on an odd number of path nodes or even number of path nodes. We construct $G' = (V', E')$, where for every vertex u in V, we can construct u_E and u_O in V'. For every edge (u, v), we can construct (u_E, v_O) and (u_O, v_E) in E'. Then, the shortest path with an odd number of edges from s to t is the shortest path from s_E to t_O. We can run BFS on s_E to find such a path in $O(|V| + |E|)$ time.

§20 Lecture October 8, 2020

Fact (Length of Path)

In general, we can consider edges as having any integer length. Then, the length of a path $P = v_0, v_1, \ldots, v_n$ is

$$l(P) = \sum_{i=0}^{n-1} l(v_i, v_{i+1})$$

Fact (Dijkstra's Algorithm)

Dijkstra's Algorithm finds distances of all nodes reachable from s when $l(e)$ ranges in $0, 1, 2, \ldots$. We denote the distance from s to another vertex t, $\delta(t)$ is the length of the shortest path from s to t.

- Lengths: $l(e)$ is the length of edge e, $l(P)$ is the length of path P
- Labels: $\lambda(v)$ means that there *is* a path from s to v of length $\lambda(v)$ (there could be a shorter path).
- Distances: $\delta(v)$ represents the distance (shortest) from s to v.

Definition (Improvement/Relaxation)

If there is an edge from u to v of length l, and $\lambda(u) + l < \lambda(v)$, then we can set $\lambda(v) = \lambda(u) + l$. This **improvement (relaxation)** is correct, because there does exist such a path.

Since all edge lengths are nonnegative, the minimum length path from S to S is 0. A general strategy for Dijkstra's Algorithm is that if you find *any* edge improvement, then relax that edge.

- Relax until you cannot relax any longer
- At that point, all labels are actually distances

Algorithm (Dijkstra's Algorithm)

Let T be a set of *temporarily* labeled vertices and P be a set of *permanently* labeled vertices. We set $\lambda(s) = 0$, T to $\{s\}$ and P to $\emptyset$. While $T \neq \emptyset$:

- Choose $x \in T$ with minimum label
- For all edges from x, perform the relaxation if possible (and add vertex to T if relevant)
- Pull out $\{x\}$, i.e. $T = T \backslash \{x\}$ and set $P = P \cup \{x\}$

We can argue for correctness (the shortest path from s to T is composed of $sP \cdots PT$) by contradiction.

Considering complexity of the nodes, we find min node in $O(n)$ (n times) for $O(n^2)$, and the edge cost is each $O(1)$ but there are m for $O(m)$, so the complexity of Dijkstra's is $O(n^2)$.

An alternative complexity is $O(m \log m)$ which involves using heaps (as we need to continuously insert and pull the min).

§21 Recitation October 9, 2020

Algorithm (Depth First Search)

A depth-first search also finds all vertices reachable from s, but does so by searching undiscovered vertices as deep as possible before exploring other branches. Just as with BFS, DFS returns a set of parent pointers for vertices reachable from s in the order the search discovered them, together forming a *DFS tree*.

DFS is called on each vertex at most once, and the amount of work done by DFS is $O(|E|)$ (proportional to the outdegree for each vertex), so DFS runs in $O(|V| + |E|)$.

Fact

We can perform full graph exploration by introducing a *supernode*, an auxiliary vertex with an outgoing edge to every vertex, and then we may run BFS or DFS from the supernode to examine each of the connected components of the graph.

We could also iterate through the nodes, but this wouldn't work asymptotically well if our graph was dense and not quite disconnected.

Definition (DFS Edge Classification)

Consider a graph edge from vertex u to v.

- We call an edge a **tree edge** if the edge is part of the DFS tree, i.e., the parent pointer of v points to u.
- We call an edge a **back edge** if u is a descendant of v.
- We call an edge a **forward edge** if v is a descendant of u.
- We call an edge a **cross edge** if neither u nor v is a descendant of the other.

Notice that descendants and tree edges are defined within the context of the *DFS* tree, whereas edges in general are defined within the context of the whole graph.

We can identify back edges computationally by checking in our DFS if adjacent vertices are ancestors of another.

Definition (Topological Sort)

A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of the vertices such that for each edge (u, v) in E, vertex u appears before vertex v in the ordering.

Fact (Topological Reverse)

If the graph is acyclic, the order of vertices backtracked from returned by DFS is the *reverse* of a topological sort order.

Fact (Explicit Graph Construction)

If a graph is not explicitly given, or if modification or graph duplication is being used, the construction process for the graph must be explicitly stated within a solution.

§22 Recitation October 14, 2020

Definition (Weighted Graphs)

A **weighted graph** is a graph $G = (V, E)$ together with a **weight function** $w : E \to \mathbb{R}$, mapping edges to real-valued weights. In practice, edge weights will often be stored as values in an adjacency matrix.

In this class, we assume that a weight function w can be stored using $O(|E|)$ space, and can return the weight of an edge in constant time.

Example (Weighted Shortest Path Problem)

A *weighted shortest path* problem asks for a lowest weight path to every vertex v in a graph from an input source vertex s, or an indication that no lowest weight path exists from s to v (e.g. negative weight *cycle*). If all edges weights are positive and equal to each other, we could simply run BFS.

Fact

If a path contains a negative weight cycle, the shortest path is undefined with weight $-\infty$. If no path exists, the shortest path is undefined with weight ∞.

Weighted Single Source Shortest Path Algorithms

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$

Definition (Relaxation)

As a general algorithmic paradigm, a **relaxation** algorithm searches for a solution to an optimization problem by starting with a solution that is not optimal, then iteratively improving the solution until it becomes an optimal solution to the original problem.

For shortest path, if $\delta(s, v)$ is the true shortest path length, we have an estimate $d(s, v)$ which is initialized to ∞ at first (except $d(s, s) = 0$), and then repeatedly relaxed until $d(s, v) = \delta(s, v)$ for all (s, v).

Fact

If at any point $d(s, u) + w(u, v) < d(s, v)$, we can relax the edge by setting $d(s, u) + w(u, v) = d(s, v)$ by the triangle inequality.

Lemma (Safety Lemma)

Relaxing an edge maintains $d(s, v) \geq \delta(s, v)$ for all $v \in V$.

At any point $d(s, v)$ is either infinite or the weight of some path from s to v (so it cannot be smaller than a shortest path). This is true at initialization and holds inductively.

Lemma (Termination Lemma)

If no edge can be relaxed, then $d(s, v) \leq \delta(s, v)$ for all $v \in V$.

If $\delta(s, v) < d(s, v)$, there is a shorter path from s to v so we can relax an edge in this path.

Fact

Together, the Safety Lemma and Termination Lemma imply that when no edges can be relaxed, i.e. at termination, $d(s, v) = \delta(s, v)$.

Algorithm (DAG Relaxation)

Relaxing each outgoing edge from every vertex in sorted topological order gives the shortest weight paths.

Relaxing in topological order guarantees that any edge we walk through has already been relaxed (edges are relaxed in the order they appear in the path). Since depth-first search runs in linear time and the loops relax each edge one time, DAG Relaxation performs in $O(|V| + |E|)$.

§23 Lecture October 15, 2020

Fact

Recall that Dijkstra yields the shortest paths from s in digraphs with *non-negative* edge weights.

Dijkstra's relaxes by assigning labels to each node (not necessarily a distance) and taking the shortest path from a marked node (one that necessarily has its label equal its minimum distance) and marking the new node it reaches with a distance. There are no repeats, and the algorithm is efficient because $l \geq 0$.

Algorithm 10 (Bellman-Ford)

Bellman-Ford finds the shortest paths from s in digraphs with *arbitrary* edge weights (an edge and a path can have positive, negative, or 0 length) provided there are *no* negative cycles.

For such digraphs as required for Bellman-Ford, we have the following properties

- Since no cycles can have negative length, we know that distances can be set by *simple* paths (a cycle contributes nonnegative length)
- There are only a finite number of simple paths
- Distances are well-defined
- Any shortest path sets the distance of each of its nodes (if there was a smaller distance, we would not be considering the shortest path)

Unfortunately, we can't proceed in the same manner as with Dijkstra's, since edges don't necessarily have nonnegative length. We initialize from a starting point s (or a supernode, in case we have unconnected components), setting all other distances to ∞ and fix any ordering of edges $(e_1, e_2, \ldots, e_m)$ and iterate through them. Until no improvement is found, we execute a pass:

Algorithm (Pass (Bellman-Ford))
For $i = 1$ to m, if $e_i = u \to v$ has the property that $\lambda(v) > \lambda(u) + l(e_i)$

- Set $\lambda(v)$ to $\lambda(u) + l(e_i)$
- Set the predecessor of v to u

The cost of one pass is $O(m)$ since we iterate through our m edges. If we perform n passes, the complexity of Bellman-Ford finds distances from s in $O(n \cdot m) = O(|V| \cdot |E|)$ time.

Correctness

- After pass k, for all nodes x whose shortest path from s has k edges, $\lambda(x) = \delta(x)$.
- For all nodes x, the shortest path from s to x has $\leq n-1$ edges (it must be simple)
- After $n-1$ passes, the temporary label λ of every node coincides with its permanent distance δ

If our graph has negative cycles, we can still run Bellman-Ford, but some labels will continue to become smaller if we continue to execute passes (but this *proves* that there are negative cycles, since the algorithm would terminate otherwise). In fact, a single additional pass detects the presence of a negative cycle (since the shortest edge-length of a negative cycle is at most $n-1$).

By sticking to the same (arbitrary) order for n passes, Bellman-Ford, in $O(n \cdot m)$ time,

- Detects whether a digraph with arbitrary edge weights has a negative cycle and
- If no such cycle exists, then it finds the shortest paths to a given node reachable from s

§24 Recitation October 16, 2020

Lemma
At the end of relaxation round i of Bellman-Ford, $d(s, v) = \delta(s, v)$ for any vertex v that has a shortest path from s to v which traverses at most i edges.

We can prove this lemma utilizing induction.

Fact

Bellman-Ford actually takes $O(|V| + |E|)$ time to loop over the entire adjacency list structure, even for vertices adjacent to no others, but if the graph contains isolated vertices not in S, we can remove them to make the runtime $O(|E|)$, since V would be lowered to $O(|E|)$.

Algorithm 11 (Bellman-Ford with Graph Duplication - Unofficial Material)

We can also perform Bellman-Ford with graph duplication:

- Idea - Use graph duplication: make multiple copies (or levels) of the graph
- $|V| + 1$ levels: vertex v_k in level k represents reaching vertex v from s using $\leq k$ edges.
- If edges only increase in level, resulting graph is a DAG!

We construct a new DAG $G' = (V', E')$ from $G = (V, E)$, where G' has $|V|(|V| + 1)$ vertices v_k for all $v \in V$ and $k \in \{0, \ldots, |V|\}$. Also, G' has $|V|(|V| + |E|)$ edges, with V edges (v_{k-1}, v_k) of weight zero for each v and V edges (v_{k-1}, v_k) of weight $w(u, v)$ for each $(u, v) \in E$.

Lemma

Then, $\delta(s_0, v_k) = \delta_k(s, v)$ for all $v \in V$ and $k \in \{0, \ldots, |V|\}$

Fact

Ordinary Bellman-Ford does not give the shortest paths of a certain length or return the actual negative weight cycle.

§25 Lecture October 20, 2020

We have discussed two algorithms for determining single source shortest paths (all targets) in general graphs (Dijkstra with non-negative edge weights and Bellman-Ford). Today, we discuss the single source single target problem, where we only care about the shortest path from a source to our target.

Fact (Dijkstra Review)

In Dijkstra's Algorithm,

- We have non-negative edge weights
- We have a distance array d, where $d[v]$ stores the shortest path distance from s to v (so far)
- We maintain a predecessor array Π, where $\Pi[v]$ stores the predecessor of v on the shortest path from s to v (so far)
- We have a priority queue Q, where keys are vertices, and the priority of v is $d[v]$

Algorithm (Dijkstra's)

We first initialize $d[s] = 0$, $d[u \neq s] = \infty$ and $\Pi[s] = s$ (special case), $\Pi[u \neq s] = None$. We also initialize Q for all vertices in the graph.

while ($Q \neq$ empty)

- $u = \text{Extract\_min}(Q)$
- for each vertex $v \in Adj[u]$, $\text{Relax}(u, v, w(u, v))$, which, if $d[u] + w(u, v)$ is less than $d[v]$, relaxes $d[v]$, sets $\Pi[v] = u$, and decreases the priority of v in Q to $d[u] + w(u, v)$ (the new distance).

When u is extracted, $d[u]$ is the correct shortest path distance from s to u. The runtime of this algorithm is $O(m \cdot T_{decrease-key} + n \cdot T_{extract-min}$. If we use a binary heap (min heap), then the cost of $T_{decrease-key}$ and $T_{extract-min}$ are both $O(\log n)$, giving us a runtime of $O((m + n) \log n)$.

We can actually do better by using a Fibonacci heap, which allows us to perform $T_{decrease-key}$ in $O(1)$ amortized time and $T_{extract-min}$ in $O(\log n)$ amortized, giving us a runtime of $O(m + n \log n)$. In particular, if $m = \omega(n \log n)$, then the Fibonacci heap provides a faster algorithm than the binary heap.

Example

We consider instead, if we want to use Dijkstra's Algorithm to solve a single source single target problem.

One method we could perform is simply stop our above algorithm when $u = t$, i.e., when our target node is the one being extracted (this favors closer nodes).

Algorithm (Bi-directional Search)

We alternate (one step at a time) between performing a forward search from s and a backward search from t (following the reverse of edges). The algorithm terminates once the forward and backward frontiers (vertices extracted from respective queues) intersect.

We initialize arrays

- d_f: distances for forward search and d_b: distances for backward search
- Π_f: predecessors of vertices in forward search and Π_b: predecessor of vertices in backward search (edges reversed)
- Q_f: forward search priority queue and Q_b: backward search priority queue

The termination criteria is that some vertex u has been extracted from both Q_f and Q_b.

We could claim that the shortest path from s to t goes through u. This can be extracted by performing $\Pi_f[\Pi_f[u]] \dots$ until we get to s and $\Pi_b[\Pi_b[u]] \dots$ until we get to t, but this is wrong. We can view as a counterexample a case where there is a more weighted path from s to t that has less edges than the actual shortest path (the first u extracted from both would be in the more weighted path).

Fix: We look at *all* vertices u that have been processed by at least one search and compute

$$\min_u d_f[u] + d_b[u]$$

which will give our shortest path.

Correctness: We argue that no shorter path goes through vertices not processed by either search. Assume, for the sake of contradiction, that there was such a path. Suppose that y is some vertex on the shortest path that is not processed by either search. Then, we can assume the path looks like

$$s \to \cdots \to x \to y \to z \to \cdots \to t$$

where instead, vertex u caused us to terminate:

$$s \to \cdots \to u \to \cdots \to t$$

If the path from $s \to \cdots \to x \to y$ is shorter than $s \to \cdots \to u$, then $d_f[y] < d_f[u]$, but then we would have processed y before u, so it is not shorter; similarly, the path from $y \to z \to \cdots \to t$ is not shorter than the path from $u \to \cdots \to t$, meaning the path from s to t through y is not shorter than the path through u, leading to a contradiction.

§26 Recitation October 21, 2020

We can think of Dijkstra's as constant flowing water, flowing down a path, which takes advantage of the nonnegative edge weights. By our water-flowing paradigm, we relax via our priority queue data structure, which is equivalent to the order of vertices the water reaches.

Fact

Letting T_i, T_e, T_d be our insertion, extract_min, and decrease_key runtimes, our total runtime is

$$T_{Dijkstra} = O(|V| \cdot T_i + |V| \cdot T_e + |E| \cdot T_d)$$

Using a *dictionary* priority queue, $T_i = O(1)$, $T_e = O(|V|)$, and $T_d = O(1)$, giving us a runtime of

$$T_{Dict} = O(|V|^2 + |E|)$$

which is good for *dense* graphs because this would be $O(|E|)$.

Using a *binary min heap*, we can implement insertion and extract-min in $O(\log n)$ time, giving us

$$T_{Heap} = O((|V| + |E|) \log |V|)$$

which is $O(|V| \log |V|)$ for *sparse* graphs.

A *Fibonacci Heap* which supports amortized $O(1)$ insertion and decrease-key, along with $O(\log n)$ minimum extraction,

$$T_{FibHeap} = O(|V| \log |V| + |E|).$$

This is a *theoretical* runtime that we don't often use in practice but quote when discussing Dijkstra's runtime. They are also asymptotically efficient, but have a large constant factor and are not necessarily efficient in practice. If we can figure out whether our graph is more sparse or dense to begin with, we can practically choose to use a dictionary or binary min heap.

Fact

Graph duplication is useful when dealing with state problems (we move to a duplicated graph when we change our state).

§27 Lecture October 22, 2020

Today, we discuss *All Pairs Shortest Paths* (APSP).

Example (All Pairs Shortest Paths Problem)

Given as input a weighted graph $G(V, E)$ with weight function w, we would like to output the shortest path distance $\delta(u, v)$ between u and v for all $u, v \in V$.

If we were to run Dijkstra n times, from each vertex (only if edge weights are non-negative), then our runtime would be

$$O(n(m + n \log n)) = O(n^2 \log n + mn).$$

In general, we could run Bellman-Ford n times with a runtime of

$$O(n(mn)) = O(n^2 m).$$

We can lower bound this problem with $\Theta(n^2)$, since as a bare minimum, the size of our output is $\Theta(n^2)$ (considering the number of pairs of vertices).

Fact

We develop an APSP algorithm that works for general weights and takes $O(n^2 \log n + mn)$ time.

The idea here is to take our original graph G (general weights), and modify edge weights such that shortest paths between vertices are preserved, constructing a new graph G' (same vertices and edges) with only non-negative edges. This allows us to

1. run Dijkstra's n times on G'
2. convert shortest path distances in G' to shortest path distances in G

We claim that we can do the second step in $O(n(n + m))$. Namely, for each vertex v ($O(n)$), we can get the shortest path tree of v (in G') and run BFS ($O(n + m)$) on this tree using our original edge weights.

Idea: If the smallest weight in G is x, then add $-x$ to all edges to make them non-negative to create G'.

- However, this does not preserve shortest weights, because paths with more edges are penalized more

Idea: For some vertex v, add weight h to all outgoing edges of v and subtract weight h from all incoming edges of v.

Lemma

We claim that this preserves shortest paths.

Proof. Suppose we have some path

$$\pi : s \to \cdots \to t$$

with weight $w(\pi)$.

Case 1
If v is not on the path, then

$$w'(\pi) = w(\pi).$$

Case 2
If v is in the middle of the path, assuming there are no negative cycles, our paths are simple paths so

$$w'(\pi) = w(\pi) + h - h = w(\pi).$$

Case 3
If $v = s$, all paths starting at v change by $+h$, so the shortest path is still the same.

Case 4
If $v = t$, all paths ending at v change by $-h$, so the shortest path is still the same. □

Algorithm (Johnson's Algorithm)

Since we can perform such a process for any vertex without changing shortest paths, we can perform the process over all vertices. We define a *potential* function h, where vertex v has potential $h(v)$ for all $v \in V$.

We construct G' by adding $h(v)$ to all v's outgoing edges and subtracting $h(v)$ from all v's incoming edges (for all $v \in V$).

Transforming an edge from $x \to y$ with weight $w(x, y)$ in G makes our new edge from $x \to y$ in G' have weight

$$w'(x, y) = w(x, y) + h(x) - h(y).$$

We can prove that shortest paths are preserved. Consider some path

$$\pi : v_0 v_1 \cdots v_k$$

where the weight in G is

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The weight in G' is

$$w'(\pi) = \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i))$$

$$= w(\pi) + \sum_{i=1}^{k} (h(v_{i-1}) - h(v_i)) = w(\pi) + h(v_0) - h(v_k).$$

Since $h(v_0)$ and $h(v_k)$ are constants, the shortest paths between v_0 and v_k are still the same.

Fact

To choose our potential function h, we want it to satisfy for all $(u, v) \in E$,

$$w'(u, v) = w(u, v) + h(u) - h(v) \geq 0.$$

This is equivalent to satisfying the Triangle Inequality

$$h(v) \leq h(u) + w(u, v).$$

In particular, shortest paths in graphs satisfy the triangle inequality (assuming these shortest path distances are finite). Suppose we have some source vertex s where we computed $\delta(s, v)$ for all $v \in V$ (we have to construct a supernode s with weight 0 edges to all vertices in G, in case our graph has disconnected components).

We can run SSSP from s using Bellman Ford. If there is a negative cycle, we can abort. Otherwise, we can make G' by reweighting each $(u, v) \in E$ with

$$w'(u, v) = w(u, v) + \delta_s(s, u) - \delta_s(s, v).$$

Then, we can run Dijkstra's from every vertex in G' that gives $\delta'(u, v)$, the shortest path distance in G' between u and v for all $u, v \in V$. Then, we convert $\delta'(u, v)$ back to $\delta(u, v)$ which gives the shortest path distances in G:

$$\delta'(u, v) = \delta(u, v) + h(u) - h(v)$$

meaning

$$\delta(u, v) = \delta'(u, v) - h(u) + h(v) = \delta'(u, v) - \delta_s(s, u) + \delta_s(s, v).$$

The total runtime of this algorithm is $O(n^2 \log n + mn)$.

§28 Recitation October 23, 2020

The *All Pairs Shortest Paths* problem seeks to find any minimum weight path between any two vertices in a weighted graph.

A straight-forward method is to solve an SSSP problem $|V|$ time. For a *sparse* ($|E| = O(|V|)$) graph, this takes $O(|V|^2)$ time, so this algorithm is optimal for graphs that are both *unweighted* and *sparse*.

Algorithm 12 (Johnson's Algorithm)

The idea behind Johnson's Algorithm is to reduce the APSP problem on a graph with *arbitrary edge weights* to the APSP problem on a graph with *non-negative edge weights*. The algorithm does this by re-weighting the edges in the original graph to non-negative values in such a way that shortest paths in the re-weighted graph are also shortest paths in the original graph. Then finding shortest paths in the re-weighted graph using Dijkstra $|V|$ times will solve the problem (and we convert weights back to the original graph).

Johsnon's assigns each vertex v a real number $h(v)$, and changes the weight of each edge (a, b) from

$$w(a, b) \mapsto w'(a, b) = w(a, b) + h(a) - h(b)$$

to form a new weight graph $G' = (V, E, w')$.

Lemma

A shortest path $(v_1, v_2, \ldots, v_k)$ in G' is also a shortest path in G from v_1 to v_k.

We showed in class that the weight of any path from v_1 to v_k changes by a constant factor from G to G', so the shortest paths will be the same in each.

Fact

Johnson's defines h in the following way: add a new node x to G with a directed edge from x to v for each vertex $v \in V$ to construct graph G^*, letting $h(v) = \delta(x, v)$. This assignment of h ensures $w'(a, b) \geq 0$ for every edge (a, b).

We can find these distances from the added node x using Bellman-Ford (which also determines negative cycles, in which case Johnson's can terminate)

Lemma

If $h(v) = \delta(x, v)$ and $h(v)$ is finite, then $w'(a, b) = w(a, b) + h(a) - h(b) \geq 0$ for every edge $(a, b) \in E$.

This claim is equivalent to claiming $\delta(x, b) \leq w(a, b) + \delta(x, a)$ for every edge $(a, b) \in E$, which is true by definition of minimum weight (exactly a restatement of the triangle inequality).

Fact (Johnson's Algorithm Time Complexity)

Johnson's takes $O(|V|\,|E|)$ time to run Bellman-Ford and $O(|V|\,(|V|\log|V|+|E|))$ time to run Dijkstra $|V|$ times, so this algorithm runs in $O(|V|^2\log|V|+|V|\,|E|)$ time, which is asymptotically better than $O(|V|^2\,|E|)$.

§29 Lecture October 27, 2020

We begin our discussion of *Dynamic Programming* (DP). We can consider an example like the Fibonacci sequence for $\text{fib}(n)$:

- if $n \leq 2$ return $f = 1$
- else: return $f = \text{fib}(n-1) + \text{fib}(n-2)$

This runtime is exponential, because it recomputes the same Fibonacci numbers many times. We can fix this using *memoization*, for example, creating a dictionary to story Fibonacci values (which we can just call if we already know).

Fact

A fundamental property of DP analysis is that total runtime is # suproblems $\times$ time per subproblem (assuming recursive problems perform in constant time).

In our Fibonacci example with memoization, our runtime is $\Theta(n)\times\Theta(1)=\Theta(n)$.

Example (Coin Row Problem)

Given a row of coins

$$C_0, C_1, C_2, \ldots, C_{n-1}$$

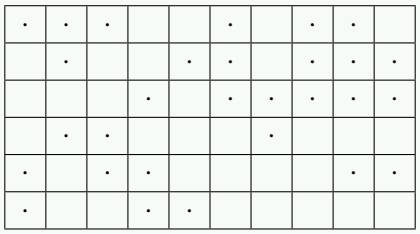
we want to the pick the maximum amount of money subject to the constraint that no two coins adjacent in a row can be picked up.

Let $M(n)$ be the maximum amount of money that can be picked up from n coins. If we choose C_0, then $M(n) = C_0 + M(n-2)$. If we don't choose C_0, $M(n) = M(n-1)$. Then , let

$$M(n) = \max\{C_0 + M(n-2), M(n-1)\}.$$

Example (Robotic Coin Collection)

Next consider the coin collection:



We want to pick up as many coins ($\cdot$) as possible, moving from bottom left corner to top right corner with movements that only go upwards and to the right.

We let $C_{ij} = 1$ if there is a coin at (i, j), otherwise we let it be 0. We let $S(i, j)$ be the largest number of coins the robot can pick up until and including (i, j). Assuming there are $n + 1$ columns and $m + 1$ rows (0-indexed), we want to maximize $S(n, m)$.

When reaching (i, j), we can either do so from $(i-1, j)$ or $(i, j-1)$ (assuming such points are within the bounds of the collection). Then,

$$S(i, j) = C_{ij} + \max\{S(i - 1, j), S(i, j - 1)\}$$

where $S(0, 0) = C_{00}$, $S(i, 0) = S(i - 1, 0) + C_{i0}$ for $i \geq 1$, and $S(0, j) = S(0, j - 1) + C_{0j}$ for $j \geq 1$. The complexity of this algorithm is $\Theta(nm) \cdot \Theta(1) = \Theta(nm)$.

§30 Lecture November 3, 2020

Fact

Problem sets will ask for 5 explicit easy steps:

1. Define subproblems
2. Guess
3. Relate subproblems and recurrence
4. Recurse and memoize or iterative bottom-up (acyclic order)
5. Solve original problem

Example (Maximum Value Contiguous Subsequence)

Given an array $A[0:n-1]$ of n numbers, we seek the maximum sum of a contiguous subsequence.

We could brute force this method in $O(n^3)$ by checking every single contiguous subsequence (we could also reduce this to $O(n^2)$ just by keeping track of the current sum). Using Dynamic Programming,

(1) $M(j)$: max sum over all windows ending with j.

(2) $M(j) = \max\{M(j-1) + A[j], A[j]\}$ with $M[0] = 0$.

(3) See above.

(4) We can perform calls in the order $0, 1, \ldots n$.

(5) The maximum continguous sum is $\max_{l \in [0,n-1]} M(l)$.

The complexity is given by

$$\#\text{subproblems} \times \text{time/subproblem} = n \times \Theta(1) = \Theta(n).$$

Taking the max of all $M(l)$ values is also $\Theta(n)$.

Example

Consider an array of $words[0:n-1]$. Define $badness(i,j)$ for a line of $words[i,j]$ as ∞ if the total length is $>$ page width, otherwise

$$(\text{page width} - \text{total length})^3.$$

We split $words[0:n-1]$ into lines in order to minimize the sum of all $badness$.

(1) Subproblem: min badness for suffix $words[i:]$. The number of subproblems is $\Theta(n)$.

(2) Guessing: where to end the first line $i:j$, with $n - i = O(n)$ choices.

(3) Recurrence: $DP[i] = \min\{badness(i,j) + DP[j] \mid j \in [i+1, n]\}$ with $DP[n] = 0$.

- $DP[n-1] = badness(n-1, n)$
- $DP[n-2] = \min\{badness(n-2, n-1) + DP[n-1], badness(n-2, n) + DP[n]\}$

(4) Recurse and memoize

(5) $DP[0]$ is the final solution

§31 Recitation November 4, 2020

Fact

Dynamic Programming generalizes Divide and Conquer type recurrences when subproblem dependencies form a directed acyclic graph instead of a tree.

Fact (SRT BOT)

We can solve a problem recursively with

1. Subproblem definition (subproblem $x \in X$)
   - Describe the meaning of a subproblem in words, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. Relate subproblem solutions recursively, $x(i) = f(x(j), \dots)$ for one or more $j < i$
3. Topological order to argue relation is acyclic and subproblems form a DAG
4. Base cases
   - State solutions for bases cases (cases that don't apply recursively)
5. Original problem
   - Show how to compute solution to original problem from solutions to subproblems
6. Time analysis

Definition

A **top down** approach evaluates the recursion starting from roots, while a **bottom up** approach calculates each subproblem according to a topological sort order (DFS finishing order) of the DAG of subproblem dependencies.

Example

Suppose we want to find the nth Fibonacci number.

A recursive call would be inefficient and recompute values multiple times, but in reality,

there are only n subproblems. We can instead approach this problem with dynamic programming:

(1) Subproblems: $x(i) = i$th Fibonacci number

(2) Relation: $x(i) = x(i-1) + x(i-2)$

(3) Topological order: since $x(i)$ depends on strictly smaller subproblems, we can construct a DAG

(4) Base cases: $x(0) = x(1) = 1$

(5) Original/Solution: $x(n)$, using top down or bottom up

(6) Time analysis: the number of subproblems is $O(n)$ and the work/subproblem is $O(1)$, so the complexity is $O(n)$.

Fact

Top down is a recursive approach while bottom up essentially unrolls the recursion and is an iterative approach.

§32 Lecture November 5, 2020

Example (Arithmetic Parenthesization)

Given an arithmetic expression containing n integers $A = \{a_1, a_2, \ldots, a_n\}$ with a_i and a_{i+1} separated by a binary operator $f_i \in \{+, \times\}$, we would like to determine where to place parentheses to maximize the evaluated expression.

A naive approach would be to check all ways to parenthesize, of which there are $O(4^n)$ different ways. We can instead use dynamic programming, where our subproblems are *substrings* that we both maximize and minimize (to account for the case where we have negatives with large magnitude that could multiply).

- Let $X(i, j, +1)$ be the max result in the substring from i to j and $X(i, j, -1)$ be the min result in the substring from i to j.
- We guess which f_i is executed last.
- We can recurse:

  $$X(i, j, +1) = \max\{f_k(X(i, k, s_1), X(k+1, j, s_2)) \mid i \le k \le j-1, s_1 = \pm 1, s_2 = \pm 1\}$$

  and

  $$X(i, j, -1) = \min\{f_k(X(i, k, s_1), X(k+1, j, s_2)) \mid i \le k \le j-1, s_1 = \pm 1, s_2 = \pm 1\}$$

Our base cases are

$$X(i, i, s) = a_i \text{ for } s = \pm 1.$$

- Subproblems only depend on other subproblems with smaller substrings, i.e., smaller values of $j - i$.
- The answer to the original problem is $X(1, n, +1)$. We can retrieve the best parenthesization by storing 2 pointers, one to each of the smaller subproblems that gave the "best" result.
- The number of subproblems is $2 \cdot \binom{n}{2} = \Theta(n^2)$, and the non-recursive work per subproblem is $O(n)$, so the total runtime is $O(n^3)$.

Example (Piano Fingering)

Given a sequence of single notes $t_1, \ldots t_n$ and F fingers labeled $1, \ldots, F$. We define a difficulty metric $d(t, f, t', f')$ which measures the difficulty of playing note t with finger f followed by note t' with finger f'. Ideally, we would like to minimize the overall transition difficulty,

$$\sum_{i=1}^{n} d(t_i, f_i, t_{i+1}, f_{i+1})$$

while optimizing f_i's.

We assume that d is given to us.

- For our subproblems, we could try a suffix of notes, where $X(i)$ is the minimum total difficulty for playing notes $t_i, \ldots, t_n$. We can guess which finger f to use for t_i, in which case, our recurrence would be given by

  $$X(i) = \min\{X(i+1) + d(t_i, f, t_{i+1}, ?) \mid 1 \leq f \leq F\}$$

  but this doesn't work. *Instead*, let our subproblem be $X(i, f)$: the minimum total difficulty for playing notes $t_i, \ldots, t_n$ where finger f is used to play t_i.
- We guess which finger f' is used for playing t_{i+1}.
- Our recurrence is given by

  $$X(i, f) = \min\{X(i+1, f') + d(t_i, f, t_{i+1}, f') \mid 1 \leq f' \leq F\}$$

  with base case

  $$X(n, f) = 0 \text{ for } 1 \leq f \leq F.$$

- Every suffix starting at i depends on the suffix starting at $i + 1$, so our order can be given by decreasing order of i and any order for f (iteratively, for $i = n$ to 1 and for $f = 1$ to F would work).

- To solve our original problem, we find

$$\min_{1 \le f \le F} X(1, f).$$

- The number of subproblems is $O(nF)$ and the non-recursive work per subproblem is $O(F)$, giving us a total runtime of $O(nF^2)$.

§33 Recitation November 6, 2020

Example (Edit Distance)

A plagiarism detector needs to detect the similarity between two texts, string A and string B. One measure of similarity is called *edit distance*, the minimum number of *edits* that will transform string A into string B. An edit may be one of three operations: delete a character of A, replace a character of A with another letter, and insert a character between two characters of A (or at an end). Describe a $O(|A|\,|B|)$ time algorithm to compute the edit distance between A and B.

- For subproblems, we modify A until its last character matches B. We let the subproblem be $x(i, j)$: the minimum number of edits to transform prefix up to $A(i)$ to prefix up to $B(j)$ (we could also formulate this with suffixes).
- For our relation, if $A(i) = B(j)$, match (we assume everything that comes before is correct). Otherwise, we edit to make the last element of A equal to $B(j)$. Our edit is either an insertion, replace, or deletion. Deletion removes $A(i)$, Insertion adds $B(j)$ to the end of A and removes it and $B(j)$, and Replace changes $A(i)$ to $B(j)$ and removes both $A(i)$ and $B(j)$.

$$x(i,j) = \begin{cases} x(i-1, j-1) & \text{if } A(i) = B(i) \\ 1 + \min(\underbrace{x(i-1,j)}_{\text{deletion}}, \underbrace{x(i,j-1)}_{\text{insertion}}, \underbrace{x(i-1,j-1)}_{\text{replace}}) & \text{otherwise} \end{cases}$$

- Each subproblem $x(i, j)$ depends on strictly smaller i and j, so our graph of subproblems is acyclic, giving us a valid topological order.
- For our base case, $x(i, 0) = i$ and $x(0, j) = j$ (you need many insertions or deletions).
- To solve the original problem, we can solve subproblems via recursive top down or iterative bottom up. The solution to the original problem can be given by $x(|A|, |B|)$. We can also store parent pointers to reconstruct edits transforming A to B.
- The number of subproblems is $O(|A|\,|B|)$, and the work per subproblem is $O(1)$, so the total runtime is $O(|A|\,|B|)$.

§34 Lecture November 10, 2020

Fact (Dynamic Programming SSSP (DAG))

In this lecture, we revisit shortest paths, letting $n = |V|$ and $m = |E|$. Recall that single source shortest paths take as input $G(V, E)$ with weight function w and source vertex s and output $\delta(s, v)$, the shortest path distance from s to v, for all $v \in V$.

If there were no negative weights, we could run Dijkstra's in $O(m + n \log n)$, and if there were negative weights, we could run Bellman Ford in $O(mn)$. The key procedure was relaxing (u, v) if

$$\delta(s, u) + w(u, v) < \delta(s, v).$$

Then, consider approaching this problem with dynamic programming:

- Subproblems: $X(v) = \delta(s, v)$ for all $v \in V$
- Guess: what is the last edge on the shortest path from s to v?
- Recurrence:
$$X(v) = \min\{X(u) + w(u, v) \mid u \in Adj^-(v)\}$$
where $Adj^{-1}(v)$ represent the set of incoming neighbors of v
- Order: the dependence graph (of subproblems) *is* the original graph; we can get a topological order only if our input graph is a DAG.
- Base Case:
$$X(s) = 0 \text{ and } X(v) = \infty \text{ if } v \neq s \text{ and } Adj^-(v) = \emptyset.$$
- Original Problem: we can return $X(v)$ for all $v \in V$ (and we could also keep track of parent pointers to the "best' subproblem for all subproblems)
- Time: the number of subproblems is n and the time per subproblem is $O(1 + indegree(v))$, so the total runtime is
$$\sum_{v \in V} O(1 + indegree(v)) = O(n + m).$$
This algorithm is just DAG Relaxation.

Fact (Dynamic Programming SSSP)

What about *general graphs*? For general graphs, we can't use the above dynamic programming solution, because the dependency structure (of subproblems) could contain cycles.

Instead, consider

- Subproblems: $X(v, k)$: shortest path distance from s to v using $\leq k$ edges
- Guess: what is the last edge on the shortest path from s to v?
- Recurrence:

$$X(v, k) = \min\{\{X(u, k-1) + w(u, v) \mid u \in Adj^{-}(v)\} \cup X(v, k-1)\}$$

- Order: we can evaluate our subproblems in increasing values of k.
- Base Cases:

$$X(s, 0) = 0 \text{ and } X(v, 0) = \infty \text{ for all } v \in V.$$

- Original Problem: assuming there are no negative weight cycles, we return

$$X(v, n-1) \text{ for all } v \in V.$$

  We can check for negative weight cycles if

$$X(v, n) < X(v, n-1) \text{ for some } v \in V.$$

  We can return such a negative weight cycle by following parent pointers from v.

- Time: the number of subproblems is $n \cdot (n+1) = \Theta(n^2)$ and the time per subproblem is $O(1 + indegree(v))$, so the total runtime is

$$\sum_{k=0}^{n} \sum_{v \in V} O(1 + indegree(v)) = \sum_{k=0}^{n} O(n + m) = O(n^2 + mn),$$

  which is $O(mn)$ when only considering vertices reachable from s (in which case $m = \Omega(n)$). This algorithm is Bellman-Ford! As presented, this takes $\Theta(n^2)$ space. However, if we keep only subproblems for current and previous values of k, then this takes $\Theta(n)$ space to store subproblems.

Fact (All Pairs Shortest Paths)

Recall that the All Pairs Shortest Paths problem takes as input $G(V, E)$ with weight function w and outputs $\delta(u, v)$ for all $u, v \in V$. In graph problems, we could solve with Bellman-Ford n times in $O(mn^2)$ or Johnson's Algorithm in $O(mn + n^2 \log n)$.

We could also check for negative weight cycles using Bellman-Ford.

Algorithm 13 (Floyd-Warshall)

We present a "simpler" algorithm, Floyd-Warshall, which computes APSP problems in $O(n^3)$, which runs in the same time as Johnson's Algorithm if $m = \Theta(n^2)$.

One possible *attempt* is allowing subproblems to be $X(u, v, k)$ which stores the shortest path distance from u to v using $\leq k$ edges, but this would require $O(mn^2)$ time. Instead, we consider

- Subproblems: number vertices $1, 2, \ldots, n$ and let $Z(u, v, k)$ be the shortest path distance from u to v using only vertices in $\{1, \ldots, k\} \cup \{u, v\}$.
- Guess: the shortest path from u to v using vertices in $\{1, \ldots, k\} \cup \{u, v\}$ either uses vertex k or does not.
- Recurrence:

$$Z(u, v, k) = \min\{\underbrace{Z(u, k, k-1) + Z(k, v, k-1)}_{\text{uses } k}, \underbrace{Z(u, v, k-1)}_{\text{doesn't use } k}\}.$$

- Base Cases:

$$Z(u, u, 0) = 0 \text{ and } Z(u, v, 0) = w(u, v) \text{ if } (u, v) \in E \text{ (else } \infty).$$

- Order: we can solve our subproblems in increasing order of k.
- Original Problem: we return

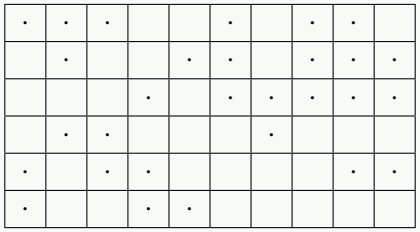
$$Z(u, v, n) \text{ for all } u, v \in V.$$

- Time: the number of subproblems is $\Theta(n^3)$ and the time per subproblem is $O(1)$, giving a total runtime of $\Theta(n^3)$.

§35 Lecture November 12, 2020

Example (Coin Collection)

Consider again, the coin collecting robot



which picks up as many coins ($\cdot$) as possible, moving from bottom left corner to top right corner with movements that only go upwards and to the right.

Today, we consider Mauricio's approach to DP, which considers two questions

(1) What decisions does the robot need to make? (go up or right)

(2) What information does it need to make its decision? (what is the best strategy if I go up or right)

Example (Rod Cutting)

Given a rod of length L, where every piece of length l has value $v(l)$, cut the rod in some way to maximize the total value.

As an example, consider $L = 7$ with the values

l	0	1	2	3	4	5	6	7
$v(l)$	0	1	10	13	18	20	31	32

which can be maximized if we cut the rod into pieces of length $2, 2, 3$ yielding a total value of $10 + 10 + 13 = 33$.

(1) What decisions do we need to make? (where to make the first cut)

(2) What information do we need? (what is the maximum value we can get from a rod of length $L - l$)

Considering these questions allows us to proceed with DP:

- Subproblems: $X(l)$ is the maximum value when the rod is of length l, for $l = 0, \ldots, L$

- Relate: we can relate

$$x(l) = \max_{1 \le l' \le l} \{v(l') + X(l - l')\}$$

- Base: $X(0) = 0$
- Order: we can calculate in increasing value of l
- Original: to solve our original problem, we can call $X(L)$
- Runtime: the number of subproblems is $L+1$ and the time per subproblem is $O(L)$, so the total runtime is $O(L^2)$ – the size of the input is L, so $O(L^2)$ is polynomial on the size of the input

Example (Subset Sum)

Given an input $A = \{a_1, a_2, \ldots, a_n\}$ and a target T, is there a susbet $A' \subseteq A$ that sums up to T:

$$\sum_{a \in A'} a = T.$$

Consider

(1) What decisions do we need to make? (is $a_1 \in A'$)

(2) What information do we need? (can we make T or $T - a_1$ from $\{a_2, \ldots, a_n\}$)

Again, we consider DP:

- Subproblems: $X(i, t)$ is whether or not we can add up to t with a subset from $\{a_i, \ldots, a_n\}$, where $i = 1, \ldots, n$ and $t = 0, \ldots, T$
- Relate: we can relate

$$X(i, t) = X(i + 1, t) \text{ or } \underbrace{X(i + 1, t - a_i)}_{\text{if } a_i \le t}$$

- Base: $X(i, 0)$ is True and $X(n, t)$ is True if and only if $t = 0$ or $a_n = t$ (otherwise False)
- Order: we can calculate in decreasing value of i
- Original: we can compute $X(1, T)$ to solve the original problem
- Runtime: the number of subproblems is $n(T + 1)$ and the time per subproblem is $O(1)$, so the runtime of this algorithm is $O(nT)$ – the size of the input is $n + 1$ (n values and one target), but we can have $T \sim 2^w$

Definition (Pseudo Polynomial Time)

We get **pseudo polynomial time** when the runtime of an algorithm is polynomial on the size of input and integers.

For example, the subset sum example has a pseudo polynomial dynamic programming runtime.

Definition ((Strongly) Polynomial Time)

We get **(strongly) polynomial time** when the runtime of an algorithm is polynomial on the size of input and independent of integers (assumption that operations take $O(1)$).

For example, the rod cutting example has a (strong) polynomial dynamic programming runtime.

Definition ((Weakly) Polynomial Time)

We get **(weakly) polynomial time** when the runtime of an algorithm is polynomial on the size of input in *bits* (w, the word size).

If possible, we want (strongly) polynomial time runtimes, followed by (weakly) polynomial time runtimes, followed by pseudo polynomial time runtimes.

§36 Recitation November 13, 2020

Example (Treasureship!)

Treasureship is played by placing 2×1 ships within a $2 \times n$ rectangular grid, horizontally or vertically, occupying exactly 2 grid squares, and each grid square may only be occupied by a single ship. Each grid square has a positive or negative integer value, representing how much treasure may be acquired or lost at that square. You may place as many ships on the board as you like, with the score of a placement of ships being the value sum of all grid squares covered by ships. Design an efficient dynamic-programming algorithm to determine a placement of ships that will maximize your total score.

- **Subproblems**: the game board has n columns of height 2 (alternatively 2 rows of width n); let $v(x, y)$ denote the grid value at row y column x, for $y \in \{1, 2\}$ and $x \in \{1, \ldots, n\}$
- We guess how to cover the right-most square(s) in an optimal placement; we can either: not cover, place a ship to cover vertically, or place a ship to cover horizontally (after choosing an option, the remainder of the board may not be a rectangle)

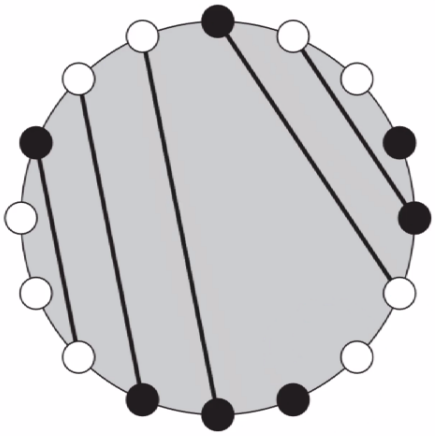
- The right side of the board can have equal top row and bottom row, or 1 or 2 additional in one of the rows; there exists an optimal placement where no two ships are aligned horizontally on top of each other (otherwise we could cover instead by two vertical ships next to each other), so we only need to consider differences of 0,1, or -1
- Let $s(i, j)$ represent the game board subset containing columns 1 to i of row 1 and columns 1 to $i + j$ of row 2, for $j \in \{0, +1, -1\}$ (acting as a state variable) and $x(i, j)$ be the maximum score, only placing ships on board subset $s(i, j)$ for $i \in \{0, \ldots, n\}$, $j \in \{0, +1, -1\}$, giving $O(n)$ subproblems
- **Relate:** If $j = \pm 1$, we can cover the right-most square with a horizontal ship or leave empty, or if $j = 0$, we can cover column i with a vertical ship or not cover *one* of the right-most squares (not covering any is covered in $j = \pm 1$ cases.

$$x(i,j) = \begin{cases} \max\{v(i,1) + v(i-1,1) + x(i-2,+1), x(i-1,0)\} & \text{if } j = -1 \\ \max\{v(i+1,2) + v(i,2) + x(i,-1), x(i,0)\} & \text{if } j = +1 \\ \max\{v(i,1) + v(i,2) + x(i-1,0), x(i,-1), x(i-1,+1)\} & \text{if } j = 0 \end{cases}$$

  It is extremely helpful to think about this with a diagram.
- **Order**: the size of the board is always decreasing, so the subproblems are acyclic.
- **Base:** $s(i, j)$ contains $2i + j$ grid squares, and $x(i, j) = 0$ if $2i + j < 2$ (can't place a ship if fewer than 2 squares)
- **Original**: our solution is $x(n, 0)$, the maximum considering all grid squares; we can also store parent pointers to reconstruct ship locations
- **Time**: the number of subproblems is $O(n)$, and the work per subproblem is $O(1)$, so the runtime is $O(n)$.

Example (Wafer Power)

Consider electronic circuit design for highly-parallel computing. Evenly-spaced along the perimeter of a circular wafer sits n ports for either a power source or a computing unit. Each computing unit needs energy from a power source, transferred between ports via a wire etched into the top surface of the wager. However, if a computing unit is connected to a power source that is too close, power can overload and destroy the circuit. Further, no two etched wires may cross each other. Evaluate the effectiveness of different design; given an arrangement of power sources and computing units, give an $O(n^3)$-time to connect as many computing units to power sources by etching non-crossing wires between them, in order to maximize the number of powered computing units, where wires may not connect two adjacent ports along the perimeter.



- **Subproblem**: Let $(a_1, \ldots, a_n)$ be the ports cyclically ordered counterclockwise around the wafer, where a_1 and a_n are adjacent. let a_i be True if the port is a computing unit, and False if it is a power source. Let $x(i, j)$ be the maximum number of matchings, restricting to ports a_k for all $k \in \{i, \ldots, j\}$ for $i \in \{1, \ldots, n\}$ and $j \in \{i-1, \ldots, n\}$, and $j - i + 1$ is the number of ports in a substring (allowing $j = i - 1$ as an empty substring)
- We guess which port to match with the first port in the substring (either it does not match and we try to match the rest, or it matches to a port in the middle and we try to match each side independently)
- **Relate:** Let $m(i, j) = 1$ if $a_i \neq a_j$ (type of node) and $m(i, j) = 0$ otherwise (ports of opposite type match)

$$x(1, n) = \max\{x(2, n)\} \cup \{m(1, t) + x(2, t-1) + x(t+1, n) \mid t \in \{3, \ldots, n-1\}\}$$

$$x(i, j) = \max\{x(i+1, j)\} \cup \{m(i, t) + x(i+1, t-1) + x(t+1, j) \mid t \in \{i+2, \ldots, j\}\}$$

- **Order**: subproblems depend on strictly smaller $j - i$, so the dependency is acyclic
- **Base**: $x(i, j) = 0$ for any $j - i + 1 \in \{0, 1, 2\}$ (no match within 0,1, or 2 adjacent ports)
- **Original**: the solution to the original problem is $x(1, n)$ and we can store parent pointers to reconstruct matching (e.g., chose of t or no match at each step).
- **Time**: the number of subproblems is $O(n^2)$ and the work per subproblem is $O(n)$, taking $O(n^3)$ runtime

§37 Lecture November 17, 2020

Definition (Complexity Classes)

Complexity classes are collections of problems:

- $P = \{\text{problems solvable in (weakly) polynomial time}\}$, e.g., $n^{O(1)}$
- $Exp = \{\text{problems solvable in exponential time}\}$, e.g. $2^{n^{O(1)}}$
- $R = \{\text{problems solvable in finite time}\}$

Fact (Computability Theory)

Computability Theory asks whether a problem can be solved algorithmically.

Example (Halting Problem)

Consider the problem that is not in R, i.e., not solvable in finite time: given the description of a computer program PROG, does PROG halt on all inputs?

Definition (Decision Problem)

A **decision problem** determines whether or not something is the case; it can be posed as a yes-no question of the input values. Most decision problems are not in R.

We can show by counting that the set of all decision problems is much bigger than the set of all programs. Every program computes exactly one decision problem, and there can be multiple programs that compute the same decision problem. We can consider a program as Python code, which corresponds in totality to a finite binary string, corresponding to some $n \in \mathbb{N}$.

We can represent decision problems as: input $\to$ {yes, no}. Thus decision problems can be viewed as a function from $\mathbb{N} \to \{1, 0\}$. Therefore, every decision problem can be represented as an *infinite binary string*, where each input in $\mathbb{N}$ corresponds to 0 or 1. While the set of finite strings is countable, the set of infinite strings is uncountable.

Definition (NP Time)

NP refers to non-deterministic polynomial time. Consider

$$NP = \{\text{decision problems solvable in poly. time via non-deterministic machine}\}.$$

A non-deterministic machine essentially just guesses the answer correctly for every input.

A non-deterministic machine will find a "yes" path if it exists.

Definition (NP Time)

We can also consider via verifiers

$$NP = \{\text{decision problems whose "yes" instances have certification that can be verified in polynomial time}\}$$

Example

What do we do if we cannot find an efficient algorithm for a problem?

Fact

The 3-color problem is whether a graph's vertices can be colored with 3 colors such that its edges are bichromatic. If $P \neq NP$, then 3-color $\notin P$. 3-color is "as hard" as any other NP problem and is considered NP-hard.

Definition

We say that A can be reduced to B, i.e., $A \leq_p B$ if an input A can be transformed to an input B, which can be solved to yield yes/no, which can then solve A.

Definition (NP-hard)

A problem is NP-hard if every problem in NP is reducible to it.

Definition (NP-complete)

NP-complete problems refer to problems that are both in NP and NP-hard. In a sense, NP-complete problems are the hardest problems in NP.

§38 Recitation November 18, 2020

Fact

If doing one call, in practicality, top-down might be more efficient, but if you have multiple queries, you may want to run a single bottom-up and then call the multiple values later.

Example (0-1 Knapsack)

Given a knapsack with size S, we want to fill it with items i of size s_i and value v_i. We want to output a subset of items (may take 0 or 1 of each) with $\sum s_i \leq S$ maximizing the value of $\sum v_i$.

- Subproblems: We guess if the last item is in an optimal knapsack; if yes, get value v_i and pack remaining space $S - s_i$ using remaining items. If no, try to sum S using remaining items. Let $x(i, j)$ be the maximum value by packing a knapsack of size j using items 1 to i.
- Relate:
$$x(i,j) = \max\{\underbrace{v_i + x(i-1, j-s_i)}_{\text{if } j \geq s_i}, x(i-1,j)\}$$
for $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, S\}$
- Topo: $x(i, j)$ depends on strictly smaller i, so acyclic
- Base: $x(i, 0) = 0$ and $x(0, j) = 0$ for all i, j
- Original: Evaluate $x(n, S)$
- Time: There are $O(nS)$ subproblems and each takes $O(1)$ time, for $O(nS)$ overall work

We can also solve unbounded knapsack (any amount of any items) by not decreasing i but only decreasing $i + j$. We could also solve the subproblem $x(j)$, which is the maximum value by packing a knapsack of size j using the provided items (try taking every single item - it doesn't matter how many times we take each item).

§39 Lecture November 19, 2020

Fact

We could think of a *proof* as something (a string or message) that the prover finds and sends to the verifier. Then, NP proofs are efficient.

Example

For example, we could consider whether two graphs are isomorphic. We could simply send a permutation of vertices, and for each originally edge, the new edge (using the new vertices) ought to exist.

Fact (Proof System)

We can think about a *proof system* in a different way

- True statements are provable (completeness)
- False statements are *not* provable (soundness)

Definition (Interactive Proofs)

Interactive proofs think about proofs not as syntactic objects but interactive processes.

§40 Recitation November 20, 2020

Definition (Decision Problem)

A **decision problem** assigns inputs to No (0) or Yes (1). Inputs are either *No instances* or *Yes instances*.

Fact

An algorithm/program is constant length code (working on a word-RAM with $\Omega(\log n)$-bit words) to solve a problem, i.e., it produces correct output for every input and the length of the code is independent of the instance size. A problem is *decidable* if there exists a program to solve the problem in finite time.

Fact

A program is a finite string of bits, while a problem is a function $p : \mathbb{N} \to \{0, 1\}$, i.e. an infinite string of bits; the number of programs is $|\mathbb{N}|$ which is countably infinite, while the number of problems is $|\mathbb{R}|$, which is uncountably infinite.

Fact (Decidable Problem Classes)

We can consider the decidable problem classes:

- R is the set of problems decidable in finite time
- EXP is the set of problems decidable in exponential time $2^{n^{O(1)}}$ (most problems are here)
- P is the set of problems decidable in polynomial time $n^{O(1)}$ (efficient algorithms)

These sets are distinct with $P \subsetneq EXP \subsetneq R$.

Fact

P is the set of decision problems for which there is an algorithm A such that for every instance I of size n, A on I runs in poly(n) time and solves I correctly.

Fact

NP is the set of decision problems for which there is an algorithm V, a "verifier", that takes as input an instance I of the problem, and a "certificate" bit string of length polynomial in the size of I, so that:

- V always runs in polynomial time in the size of I,
- if I is a YES-instance, then there is some certificate c so that V on input (I, c) returns YES, and
- if I is a NO-instance, then no matter what c is given to V together with I, V will always output NO on (I, c).

Certificate can be thought of as proof that I is a YES-instance. If I is actually a NO-instance then no proof should work.

Fact

It is true that $P \subset NP$ (if you can solve the problem, the solution is a certificate), but it is questioned whether $P = NP$ or $NP = EXP$.

Example (Reductions)

Suppose we want to solve problem A; one way to solve is to convert A into a problem B and solve using an algorithm for B and then use it to compute the solution to A. This is called a **reduction** from problem A to problem B ($A \to B$), and B is at least as hard as A, i.e., $A \leq B$.

Definition

Problem A is **NP-Hard** if every problem in NP is polynomially reducible to A, i.e., A is at least as hard as (can be used to solve) every problem in NP ($X \leq A$ for $X \in NP$).

Definition

A problem is **NP-Complete** if it is both NP and NP-Hard. All NP-Complete problems are equivalent, i.e. reducible to each other.

§41 Lecture December 1, 2020

We discuss parallel computing, but I was too lazy to take notes.

§42 Recitation December 2, 2020

We cover material from the Fall 2019 6.006 Quiz 3 in preparation for the quiz.

Example (Relic Rescue)

Archaeologist Montana Smith has discovered an ancient tomb housing an immovable treasure chest containing n valuable relics. The chest is known to be (r, g)-booby-trapped:

- if strictly more than $r < n$ relics are removed from the chest, or
- if strictly more than $g > n$ grams of total relic weight is removed from the chest,

the entire vault will collapse. For each relic i, Montana has measured its weight w_i in positive integer grams and appraised its estimated museum value v_i in positive integer dollars. Given this relic information and positive integers r and g, describe an $O(rng)$-time algorithm to determine the maximum total value of relics Montana can remove without causing the vault to collapse.

Assuming relics are labeled $1, \ldots, n$, we let $X(i, j, k)$ be the maximum value by removing $\leq j$ relics with $\leq k$ weight from relics $\{1, \ldots, i\}$. Our guess and relation step consider whether we remove the ith item or not.

$$X(i, j, k) = \max\{\underbrace{v_i + X(i-1, j-1, j-w_i)}_{\text{if } k \geq w_i}, X(i-1, j, j)\}.$$

Example (Road Trip)

Joy Ryder is planning a road trip that will last k days. She has already determined her driving route: she will visit $n + 1 > k$ cities, $(c_0, \ldots, c_n)$ in that order, starting in city c_0 and ending in city c_n. Joy wants to split her driving as evenly as possible among the k days. Given the driving distance d_i between cities c_{i-1} and c_i for every $i \in \{1, \ldots, n\}$, describe an $O(kn^2)$-time algorithm to determine which $k - 1$ cities Joy should stay in overnight between days, so as to minimize the maximum total distance she drives on any day of her trip.

We let $X(i, j)$ be the minimum maximum daily distance to go from c_0 to c_i in j days. We can relate by guessing the last city stayed in overnight:

$$X(i, j) = \min\left\{\max(X(i', j-1), \sum_{i'+1}^{i} d_j) \mid i' \in \{0, \ldots, i\}\right\}$$

We let $X(0, j) = 0$ for $j \geq 0$ and $X(i, 0) = \infty$ for $i > 0$.

Example (Cooking Constraints)

TV Chef Rordon Gamsey is participating in a cooking contest to prepare a known three-course meal. Each course has a provided recipe containing an ordered ingredient list: the sequence of ingredients that will be needed to make the course, listed in the order that they will be used. To make the contest more interesting, all ingredients will be stored in a personal pantry, and Gamsey may use at most one ingredient from the pantry at a time. Given ingredient lists L_1, L_2, and L_3 for the three courses respectively, Gamsey could cook the three courses, one after the other, by taking $|L_1| + |L_2| + |L_3|$ trips to the pantry. However, if the lists have ingredients in common, it will be possible to take fewer trips to the pantry by cooking the courses in parallel, using ingredients in more than one course at a time. Describe an $O(|L_1|\,|L_2|\,|L_3|)$-time algorithm to determine the minimum number of pantry trips that Gamsey must take to make the three-course meal.

We let $X(i, j, k)$ be the minimum number of trips to use ingredients $L_1[0 : i]$, $L_2[0 : j]$, and $L_3[0 : k]$. For our relation, we can guess the last ingredient retrieved (either i, j, or

k). We define $f(a,b) = 1$ if $a = b$ and 0 otherwise. Then,

$$X(i,j,k) = 1 + \min\{\underbrace{X(i-1, j - f(L_1[i], L_2[j]), k - f(L_1[i], L_3[k])}_{\text{if } i>0}, \dots\}$$

and symmetrically for the other cases (if we remove $L_2[j]$ or $L_3[k]$). For our base case,

$$X(0,0,0) = 0.$$

§43 Lecture December 8, 2020

For the last lecture, we first discuss some algorithmic puzzles.

Example (Reversal Sort)

Suppose we have a list of numbers in increasing order, which we want to switch this to decreasing order using only pairwise swaps of numbers in positions i and $i+2$ for any i

First, notice that the total amount of numbers in the list n must be odd (every number must stay in a position of the same parity, and we must somehow ultimately swap the first and last positions). Assuming n is odd, we can independently sort

$$1, 3, \dots, n$$

and

$$2, 4, \dots, n-1.$$

by performing adjacent swaps in each of these lists. Thus, we can perform any sorting algorithm (like bubble sort) which swaps adjacent elements.

Example (Lemonade Stand Placement)

Suppose we have a coordinate grid with n houses, each in location (x_i, y_i) and want to place a lemonade stand at (x, y) such that

$$\sum_{i=1}^{n} |x_i - x| + |y_i - y|$$

is minimized.

Notice that we can minimize this quantity by minimizing the sum of the $|x_i - x|$ and the $|y_i - y|$ separately. In particular, we can look at x, assuming $x_1, x_2, \dots, x_n$ are in non-decreasing order. We claim that for arbitrary n, choosing x to be the median of the list minimizes the cost. We can write the cost function as

$$(|x_1 - x| + |x_n - x|) + (|x_2 - x| + |x_{n-1} - x|) + \cdots$$

by pairing up endpoints. Notice that the last term will either be one or two, depending on the parity of n. Each of these expressions is minimized when x is between x_i and x_{n-i}, so we can simultaneously minimize all expressions by letting x be the median of $x_1, x_2, \ldots, x_n$, thus minimizing their sum.

Example (Robot Coin Collection)

Suppose we have a grid filled with coins, robots moving from the top left to the bottom right by moving only down and right. We want to know the least number of robots we need to collect all coins.

We can solve the maximum number of coins collectable by one robot using Dynamic Programming, but using a greedy algorithm in this case is not necessarily optimal; for example, having a robot take the maximum number of coins possible may require more robots to take all the coins overall than otherwise.

We can notice that there are coins that cannot be picked up by the same robot, for example if one is down and to the left of another. We can lower bound the number of robots we need by finding the size of the largest maximal disjoint sets of coins (where coins are disjoint if they can't possibly be picked up by the same robot).

In particular, we can perform a *peeling* algorithm, where we peel off the bottommost layer. In particular, for the largest maximal disjoint sets, we could consider having a single robot take away the bottommost coin in each set.